Pointers

Raul P. Pelaez

September 25, 2025

Contents

1 Introduction **1**

2 Memory, addresses, and pointers **2**
2.1 Key new concepts . . . 3

3 The `auto` keyword **4**

4 Pointer to pointer **4**

5 The null pointer **6**

6 Passing by value or by pointer **6**
6.1 Key new concepts . . . 7

7 Const-ness **7**
7.1 Key new concepts . . . 8

8 Casting **8**

9 The stack and the heap **9**
9.0.1 The `alloc` family . . . 11
9.1 Key new concepts . . . 12

10 Exercises **13**
10.1 Pointer shenanigans . . . 13

1 Introduction

Today, we will learn about the concept of pointers, a fundamental aspect of C programming that allows to manipulate memory addresses directly.

Info

Pointer
A variable that stores the memory address of another variable.
Example

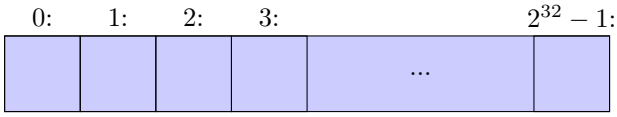
```c
int main(){
  int x = 5;       // Declare an integer variable
  int* ptr = &x;   // Declare a pointer variable that stores the address of x
  printf("Value of x: %d\n", *ptr); // Dereference the pointer to get the value of x
  printf("Address of x: %p\n", (void*)&x); // Print the address of x
  printf("Address stored in ptr: %p\n", (void*)ptr); // Print the address stored in ptr
  return 0;
}
```

Output

```
Value of x: 5
Address of x: 0x16d531fac
Address stored in ptr: 0x16d531fac
```

2 Memory, addresses, and pointers

A computer's memory is a sequence of bytes. We can number the bytes from 0 to the last one (if your computer has 4GB of memory, the last byte will be number $2^{32} - 1$). We call each of these numbers an address. In C we have direct access to these addresses by requesting chunks of it from the operative system. We can represent the memory as a sequence of boxes, each box representing a byte. For example, the memory of a computer with 4GB of memory would look like this:



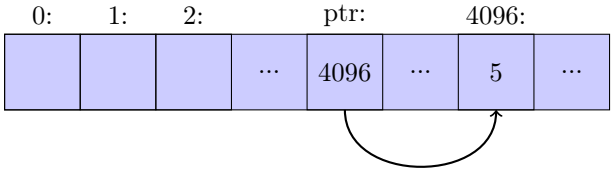
Everything we put in memory has an address. For example:

```c
int x = 5;
```

This will reserve an "int-sized" piece of memory for `x` somewhere and store the value 5 in it. We can access this memory by using the variable name x. We can also store and manipulate addresses. An object that holds an address is called a pointer. For example, the type needed to store the address of an int is called a "pointer to int" and the notation is `int*`.

```c
int* ptr = &x;
//This is equivalent, but you should think about the construction "int*" as a type name.
//int *ptr = &x;
```

Prepending an object's name with `&` reads as "the address of this object".



`ptr` (which lives in some address in memory itself) stores the address of `x`. We can use this address (imagine it is stored at location 4096) to access the value stored in `x` (5).

We can access the value stored in an address by prepending the address with `*`. For example:

```c
int y = *ptr;
```

This is called *dereferencing* an address/pointer.

Advice

As you see, the different operators/keywords of the language have wildly different meanings depending on context (which can be really subtle). This happens a lot in C, so beware!

Finally, we can also store pointers to variables. Like this

```c
int main(){
  int original = 5;
  int* alias = &original; // Reads as "alias points to to original"
  *alias = 10; // Dereference alias and set the value to 10
  printf("original: %d\n", original);
  return 0;
}
```

Output

```
original: 10
```

Info

Dereferencing and the `[]` operator

If you have an array, such as the one we created with `int arr[10];` or the underlying memory of a malloc'd vector, you can access the elements of the array by using the `[]` operator.

```c
int main(){
  int arr[10];
  arr[7] = 5;
  printf("%d\n", *(&arr[0] + 7));
  // This is equivalent to arr[7]
  //printf("%d\n", *(arr + 7)); // In C, arrays decay to pointers automatically
  return 0;
}
```

Output

```
5
```

Advanced

Does pointer arithmetics imply that we can access and modify the value stored in any address of the RAM?

Yes! unless that address is protected by the operative system, which includes sensitive regions of the memory (like the kernel's memory) and memory reserved by other programs. Tricking the OS into letting you inspect forbidden addresses is a common way to exploit vulnerabilities in software.

2.1 Key new concepts

Info

The memory model

The computer memory is a sequence of bytes, each byte has an address.

Info

Pointers

A pointer is a variable that stores an address.

Info

The `&` and `*` operators
The `&` symbol returns the address of a variable (a pointer to it). The `*` symbol returns the value stored in an address (derreferences) or denotes a pointer when decorating a type.

3 The `auto` keyword

The latest revision of the C language (C23) has incorporated the `auto` keyword, which allows the compiler to deduce the type of a variable from its initializer. This is similar to how `auto` works in C++. For example:

```c
int main(){
  auto x = 5; // The compiler deduces that x is an int
  auto y = 3.14; // The compiler deduces that y is a double
  auto z = &x; // The compiler deduces that z is an int*
  printf("x: %d, y: %f, *z: %d\n", x, y, *z);
  return 0;
}
```

Output

```
x: 5, y: 3.140000, *z: 5
```

4 Pointer to pointer

We can have pointers to pointers. For example, say we want to encode a 2D array (a matrix). We can do this by having a pointer to a pointer. For example:

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
  int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
  };
  printf("matrix[1][2] = %d\n", matrix[1][2]); // Prints 7
  return 0;
}
```

Output

```
matrix[1][2] = 7
```

Insight

In practice, it is usually a bad idea to encode a matrix (or a higher order tensor) as an array of arrays. This is because the memory layout of an array of arrays is not contiguous, which can lead to poor performance due to cache misses. It is usually better to encode a matrix as a single array and compute the indices manually. For example:

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
  int rows = 3;
  int cols = 4;
  int matrix[3 * 4]; // Single array to store the matrix
  // Initialize the matrix
  for(int i = 0; i < rows; i++){
    for(int j = 0; j < cols; j++){
      matrix[i * cols + j] = i * cols + j + 1;
    }
  }
  printf("matrix[1][2] = %d\n", matrix[1 * cols + 2]); // Prints 7
  printf("Distance in memory between first two rows: %ld bytes\n",
         (char*)&matrix[cols] - (char*)&matrix[0]);
  return 0;
}
```

Output

```
matrix[1][2] = 7
Distance in memory between first two rows: 16 bytes
```

In another example, we might want to have a function that modifies a pointer passed to it:

```c
#include <stdio.h>
#include <stdlib.h>
void find_largest(int* arr, int size, int** largest) {
  if(size <= 0) {
    *largest = NULL; // If the array is empty, set largest to NULL
    return;
  }
  *largest = &arr[0]; // Initialize largest to point to the first element
  for(int i = 1; i < size; i++) {
    if(arr[i] > **largest) {
      *largest = &arr[i]; // Update largest to point to the new largest element
    }
  }
}
int main(){
  int arr[] = {1, 3, 2, 5, 4};
  int* largest;
  find_largest(arr, 5, &largest); // Pass the address
  printf("Largest element is %d\n", *largest);
  printf("Largest element is at element %ld\n", largest - arr);
  return 0;
}
```

Output

```
Largest element is 5
Largest element is at element 3
```

After all, a pointer is just another variable, and as such we can store is address in another pointer.

Therefore, the contraption `int** ptr` is a pointer to a pointer to an int. In other words, `ptr` is a variable that stores the address of a pointer (which is a variable that stores an address) to an integer. I know it is a bit mind-bending at first, which is why I am repeating it in different ways.

5 The null pointer

There is a special address reserved for the meaning "no address". This is called the null pointer. In C we call this `NULL`. For example:

```
void foo(int* ptr) {
  // It is a good practice to check for nullptr when a function receives a pointer
  if (ptr == NULL) {
    //if(!ptr){ // nullptr casts to false automatically
    printf("The pointer is null\n");
    exit(1);
  }
}
```

6 Passing by value or by pointer

This is such an important concept I am giving it a separate section. See these three versions of a function:

```
void increment_val(int x/*This is called passing by value*/) {
  x++;
}

void increment_ptr(int* x /*This is called passing by pointer*/) {
  // Receiving a pointer allows us to modify the original variable
  // But forces us to check for NULL
  if(x == NULL) {
    printf("The pointer is null\n");
    return;
  }
  (*x)++;
}

int main(){
  int x = 5;
  increment_val(x);
  printf("%d\n", x); // This will print 5, we did not modify the original x
  increment_ptr(&x);
  printf("%d\n", x); // This will print 6, we modified the original x
  return 0;
}
```

Output

```
5
6
```

We can have a function that receives a copy of the variable (passing by value), and a function that receives a pointer to the variable. The last version can be used to modify the original variable. The first version can't. One has to be careful when designing an interface that receives a pointer with the intention of modifying the original variable. The fact that a variable will be modified might not be obvious to the reader. You should use `const` pointers when the function is not supposed to modify the original variable. For example:

```
void process_val(const int* x);
```

The reader and the compiler know that the function will not modify the memory pointed to by `x`.

Advanced

Passing values in other languages
In C++, there exists the concept of *passing by reference*, with it, we can pass a variable to a function without copying it, and without using pointers. For example:

```
//Only in C++, not in C
void increment_ref(int& x /*This is called passing by reference*/) {
  x++;
}
```

Other languages have different ways of passing variables to functions. For example, in Python, everything is passed by reference (but immutable types behave like they are passed by value). In Java, primitive types are passed by value, and objects are passed by reference (but the reference itself is passed by value). In Rust, everything is passed by value, but we can use references to avoid copying large objects.

6.1 Key new concepts

Info

Passing by value
The function receives a copy of the variable.

Info

Passing by pointer
The function receives a pointer to the variable, and can modify the original variable. Always check for NULL pointers.

7 Const-ness

There are two levels of const-ness in C: `const` and `constexpr`. The first one means that the value or object can't be modified. The second one means that the value or object is constant at compile time.

Const

Sometimes we want a value or object that is constant, meaning that we can't modify it. We can declare a variable as constant by using the `const` keyword.

Example

```
const int x = 5;
//x = 10; // This would give a compilation error
int y = 10;
const int* ptr_y = &y; // This is a pointer to a constant int
// *ptr_y = 5; // This would give a compilation error
```

Constexpr

The `constexpr` keyword is used to declare that a value is constant at compile time. This is similar to how a preprocessor macro works, but it has many added benefits (type safety, scope,...). Be aware, though, that `constexpr` is a recent addition to the language, being only available in C23.

Example

`constexpr` is a safer alternative to a preprocessor macro (`#define`) and is recommended in modern C.

```
#define PI 3.14159265359
constexpr double pi = 3.14159265359; //Always prefer this
```

`constexpr` allows us to use the full power of the C language to define constants (type safety, scope,...) , while preprocessor macros are just text substitutions. This keyword is a very recent addition to the language (C23), and some compilers might not support it yet.

Advice

Why use const and constexpr?
Using `const` and `constexpr` helps to prevent accidental modifications of values that are not meant to be modified. This can help to catch bugs at compile time, and also makes the code easier to read and understand. The reader of the code, when encountering a `const` variable, knows that this variable will not be modified, increasing his confidence when reasoning about the code. On the other hand, the compiler might use this guarantee to perform some aggressive optimizations.

7.1 Key new concepts

Info

The `const` **keyword**
A variable declared as `const` can't be modified. This includes references, pointers, etc. Always mark variables as `const` if they are not meant to be modified.

Info

The `constexpr` keyword
`constexpr` roughly means "to be evaluated at compile time". It can be used to define objects. Always prefer this to preprocessor macros. Always mark variables as `constexpr` when possible.

8 Casting

Casting is the process of converting a variable from one type to another. In C, we can cast a variable by using the syntax `(new_type)variable`. For example:

```c
int main(){
  int x = 5;
  double y = (double)x; // Cast int to double
  printf("x: %d, y: %f\n", x, y);
  int z = (int)7.999; // Cast double to int (truncates)
  printf("z: %d\n", z);
  return 0;
}
```

Output

```
x: 5, y: 5.000000
z: 7
```

We can also cast pointers. For example:

```c
int main(){
  int x = 5;
  void* ptr = &x; // Cast int* to void*
  int* int_ptr = (int*)ptr; // Cast void* back to int*
  printf("Value of x: %d\n", *int_ptr);
  return 0;
}
```

Output

```
Value of x: 5
```

Info

The pointer to void type: `void*`
A pointer to void (`void*`) is a special type of pointer that can point to any type of object. It is often used in functions that need to accept pointers to different types of objects. Or for APIs that need to be type-agnostic.
When using a void pointer, you need to cast it back to the original type before dereferencing it. A void pointer is simply the value of an address in memory, without any assumption about the type of object stored starting at that address.
As you can imagine, void pointers are a really easy way to shoot yourself in the foot. They are a consequence of the lack of type safety in C, and some detractors of the language use their existence (and necessity) as an argument against C.

What happens if we cast a pointer to an incompatible type? For example, casting a pointer to an int to a pointer to a float and dereferencing it?

```c
int main(){
  int x = 1178658487;
  float* float_ptr = (float*)&x; // Cast int* to float*
  printf("Value of x as float: %f\n", *float_ptr); // Dereference float*
  return 0;
}
```

Output

```
Value of x as float: 12345.678711
```

This is called type punning, and it is a way to interpret the same memory as different types. This can be useful in some situations, but it can also lead to undefined behavior if the types are not compatible. In this case, both `int` and `float` are 4 bytes in size, so the cast works. But if instead of float we had used double (which is 8 bytes), we would have read beyond the bounds of the original variable, leading to undefined behavior.

9 The stack and the heap

The most relevant memory regions in a C program are the stack and the heap.

- The stack is a region set aside by the compiler to store things that are "known at compile time", for instance the local variables of a function. The stack is fast and has a limited size (a few megabytes).
- The heap (also known as free store or dynamic memory) consists of the rest of the memory that the OS gives us access to (the RAM). We can request memory from the heap at runtime using the `malloc` family of functions. The heap is slower than the stack and has a much larger size.

Advanced

Other memory regions
Besides the stack and the heap, there are other memory regions in a C program:
- Static storage: A region that stores global variables and static local variables. This region is allocated when the program starts and deallocated when the program ends. It has a fixed size and is not very relevant for most programs.
- Register memory: A region that stores variables that are frequently used. This region is very fast, but has a very limited size (hundreds of bytes). The compiler decides which variables to store in register memory.

Info

Allocating on the stack:
Besides local variables (like defining an `int`), we can also store things in the stack by using the `[]` operator on a type. This is a fixed-size array that is allocated in the stack.
Example

```c
#include <stdio.h>
int main(){
  int stack_arr[10]; // This is an array of 10 ints, it is stored in the stack
  for(int i = 0; i < 10; i++){
    stack_arr[i] = i;
  }
  for(int i = 0; i < 10; i++){
    printf("%d ", stack_arr[i]);
  }
  printf("\n");
  return 0;
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

Note that we cannot resize stack arrays, and their size must be known at compile time. Additionally, stack arrays are automatically freed when they go out of scope (for example, when the function they are defined in returns). In this sense, stack arrays are akin to local variables, like `int x;`.

Advanced

In practice the situation is a bit more complicated. Built in types are usually promoted to an even faster memory region called register memory.

Info

Allocating on the heap: `malloc`
We request memory from the heap at runtime using the `malloc` family of functions.
Example

```c
#include <stdio.h>
#include <stdlib.h> // For malloc and free
int main(){
   // This is an array of 10 ints, it is stored in the heap
  int* heap_arr = (int*)malloc(10 * sizeof(int));
  for(int i = 0; i < 10; i++){
    heap_arr[i] = i;
  }
  for(int i = 0; i < 10; i++){
    printf("%d ", heap_arr[i]);
  }
  printf("\n");
  // Always free memory allocated with malloc
  free(heap_arr);
  return 0;
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

Note that the examples above are functionally equivalent (they both create an array of 10 integers and

print them). In practice, we should prefer stack arrays when possible, as they are faster and safer (automatically freed). We should use heap arrays when we do not know the size at compile time or when their size is too large to fit in the stack.

Warning

Memory leaks
Always free memory allocated with `malloc` using `free` when you are done with it. Failing to do so results in whats called a *memory leak*, which means that the memory is still allocated but we have lost the pointer to it, so we can't free it anymore. This is perhaps the most common source of bugs in C programs. Some situations dealing with heap memory are nuanced and loosing track of a pointer is easy.

Example

```c
void leak_memory() {
  int* leak = (int*)malloc(100 * sizeof(int));
  // do something
}
int main(){
  leak_memory();
  // No way to free the memory allocated in leak_memory
  return 0;
}
```

The function `leak_memory` stores a pointer to a malloc'd array in a local variable. When the function returns, the pointer is lost, and we have no way to free the memory anymore. That section of memory will be marked as used by the OS until the program ends, a moment when the OS will reclaim all the memory.

9.0.1 The `alloc` family

Besides `malloc`, there are other functions to allocate memory in the heap:

- `calloc`: Similar to `malloc`, but it initializes the memory to zero.
- `realloc`: Resizes a previously allocated block of memory. It can be used to grow or shrink the memory block. If the OS can't resize the block in place, it will allocate a new block, copy the data, and free the old block. Note that `realloc` does not zero-initialize the new memory if the block is grown.

Example

```c
#include <stdio.h>
#include <stdlib.h> // For malloc and free
int main(){
    int stack_var; // This is a local variable, it is stored in the stack
    int c_arr[10]; // This is an array of 10 ints, it is stored in the stack
    //c_arr cannot be resized, and its size must be known at compile time
    c_arr[0] = 5;
    for(int i = 0; i < 10; i++){
      printf("%d ", c_arr[i]);
    }
    printf("\n");
    // This is an array of 10 ints, it is stored in the heap and initialized to zero
    int* heap_arr = (int*)calloc(10 * sizeof(int));
    for(int i = 0; i < 10; i++){
      heap_arr[i] = i;
    }
    for(int i = 0; i < 10; i++){
      printf("%d ", heap_arr[i]);
    }
    printf("\n");
    //heap_arr can be resized at runtime using realloc
```

```
    // Resize the heap array to 20 ints
    heap_arr = (int*)realloc(heap_arr, 20 * sizeof(int));
    free(heap_arr); // Always free memory allocated with malloc
    return 0;
}
```

Insight

Assuming that `malloc` initializes the memory to zero is a common source of bugs. `malloc` will not modify the memory it reserves in any way. The memory addresses will contain whatever was there before. Try running the following code multiple times:

```
#include <stdio.h>
#include <stdlib.h> // For malloc and free
int main(){
    int* heap_arr = (int*)malloc(10 * sizeof(int));
    for(int i = 0; i < 10; i++){
      printf("%d ", heap_arr[i]);
    }
    printf("\n");
    free(heap_arr); // Always free memory allocated with malloc
    return 0;
 }
```

Output

```
644893 644893 644893 644893 644893 644893 644893 644893 644893 644893
```

You will see different values each time you run the program. The bugs that arise from reading uninitialized memory can cause are a kind of undefined behavior (meaning that *anything* can happen as far as the C standard is concerned). In practice, these bugs are characterized by the program appearing to be *non-deterministic*, like crashing only some times. Or in so-called *heissenbugs*, which disappear when you try to debug them. In 99% of the cases, the error comes from assuming that some memory is initialized to zero when it is not.

9.1 Key new concepts

Info

The heap
A region of memory that we can request from the operative system at runtime.
The heap is slower than the stack and has a much larger size. The syntax `int* arr = (int*)malloc(10 * sizeof(int));` creates an array of 10 integers in the heap. Always free memory allocated with malloc using `free(arr);` when you are done with it.

Info

The stack
A region of memory set aside by the compiler to store local variables, which have size known at compile-time.
The stack is fast and has a limited size.
The syntax `int arr[10];` creates an array of 10 integers in the stack.

10 Exercises

10.1 Pointer shenanigans

Goal

Lets do some exercises to work the memory layout of C programs and the use of pointers.

Milestone

Pointer basics
Write a function that swaps the values of two int variables. Demonstrate its use in a main function.

hint

- You will need to make use of passing by pointer.

Advanced

Can you do it without using a temporary variable?

Milestone

Memory layout 1
Think about this code:

```c
#include <stdio.h>

int main(){
  int a = 1;
  int b = 2;
  printf("Address of a: %p\n", (void*)&a);
  printf("Address of b: %p\n", (void*)&b);
  printf("Distance between a and b: %ld bytes\n", (char*)&b - (char*)&a);

  return 0;
}
```

Output

```
Address of a: 0x16f9d1fac
Address of b: 0x16f9d1fa8
Distance between a and b: -4 bytes
```

Answer the following questions:

- Why is the distance between the two variables 4 bytes?
- What would happen if we declared the variables as `double` instead of `int`?
- What if we declared them as `char`?
- What if we declare one as `int` and the other as `double`?

Insight

Note that the order of the variables in memory, as well as their location, is not guaranteed to be the same in every execution of the program. The compiler and the OS can rearrange the variables in memory for optimization purposes.

Milestone

Memory layout 2

```c
#include <assert.h>

#define Type double
int main(){
  Type a;
  size_t addr = (size_t)&a; // Get the address of a as an integer
  assert(addr%sizeof(Type) == 0);
  return 0;
}
```

This code will always work, no matter what type we use for `Type`. Can you explain why?

hint

- Read about memory alignment.

Milestone

Memory layout 3

Use the `sizeof` operator to determine the size of some types.
What happens when you print the `sizeof` of a malloc'd heap vector?

```c
int* heap_arr = (int*)malloc(1000 * sizeof(int));
printf("Size of heap_arr: %zu\n", sizeof(heap_arr));
free(heap_arr);
```

Do the same for an stack `array` with 1000 elements. Can you explain what you see?

Milestone

Casting 1

Define an integer variable. Then:

- Store this variable as a `float` using a C-style cast.
- Reinterpret the integer variable as a `float` using a pointer cast and dereferencing.
- Make it so that the resulting `float` stores the value "2.0" when reinterpreting the integer.

Milestone

Casting 2

Run this code several times:

```c
#include <stdio.h>

int main(){
  int x = 0;
  double* d_ptr = (double*)&x; // Cast int* to float*
  printf("Value of x as double: %g\n", *d_ptr); // Dereference as double
  return 0;
}
```

Chances are that you will see a different value than this, and that the value will change each time you run the program. Almost never you will see a 0. On the other hand, changing the `double` to `float` will invariably print `0`. Can you explain why?

Milestone

String concat

Info

C encodes strings as arrays of characters ending in a null character (`'\0'`). For example, the string "hello" is encoded as the array `{'h', 'e', 'l', 'l', 'o', '\0'}`. C offers a convenient shortcut to define strings using double quotes: `char* str = "hello";`. This is equivalent to defining the array manually. We can only use this syntax (defining a pointer but not allocating memory) for string literals. Strings are special in this way.

Write a function that concatenates two strings and returns the result as a new string. Make sure it works by demonstrating its use in a main function with some known inputs and outputs.

hint

- There are several ways for your function to communicate the result back to the caller. You can return a pointer to the new string, or you can pass a pointer to a pointer as an argument and modify it inside the function (output parameter).

Milestone

Malloc fun

Write a program that will ask the user for numbers until the user inputs a negative number. The program should store the numbers in a heap array. When the user inputs a negative number, the program should print all the numbers, their sum, their average, and then free the memory.
Make an effort to make this code readable by writing functions for the different tasks (like reading input, resizing the array, computing the sum...).

Learning goal

By writing functions instead of placing everything in a long main function, I am forcing you to practice several concepts:

- Code design/architecture
- Working with pointers and passing by pointer.

We are also opening the door to things like unit testing, as we can then test each functionality independently (is this function summing correctly? is this function resizing the array correctly?, etc).

hint

- You can use `scanf` to read user input from the terminal.

Milestone

Big boy malloc

Transform your previous code into a full fledged "project", following all the good practices that we have seen thus far, namely:

- Separate the functions and the main function into different files (such as `main.c`, `library.c` and `library.h`).
- Make sure you are checking for errors (like failed mallocs, invalid user input, etc).
- Use CMake to coordinate your build.
- Create a sensible file structure for your project (such as `src/` for source files, `include/` for header files, etc).
- Write some unit tests for your functions (even if just some basic mock ones), add the tests to your CMake build.
- Add an environment file to your project with its dependencies (just use the one in c-template-project as an example).

Learning goal

This looks like A LOT of work, and it will certainly take you a while to do it now. You will struggle and face every error in existence. After you have done this process a few times, it will become natural to you, and really easy to translate to other frameworks and languages. After many years, applying all of the above (and many more) when starting a new project takes me less than 5 minutes.
This kind of milestone is particularly important for you now because it covers everything we have seen in class thus far. If you are able to comfortably do this one, chances are you will do pretty well in exams.

Milestone

auto all the things

Replace as many types as possible in the following short code with the `auto` keyword. Make sure the code still compiles and works as expected.

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

int main(){
  float x = 5.0;
  assert(sizeof(x) == 4);
  int* arr = (int*)malloc(10 * sizeof(int));
  for(int i = 0; i < 10; i++){
    arr[i] = i;
  }
  for(int i = 0; i < 10; i++){
    printf("%d ", arr[i]);
  }
  printf("\n");
  free(arr);
  char *str = "Hello, world!";
  for(int i = 0; str[i] != '\0'; i++){
    printf("%c", str[i]);
  }
  printf("\n");

  double d_arr[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  for(int i = 0; i < 5; i++){
    printf("%g ", d_arr[i]);
  }
  printf("\n");
  return 0;
}
```

Advanced milestone

Privilege escalation

While only being allowed to add code inside the inject function, make it so that the root message is printed

```
#include <cstdio>
#include <string.h>

void root_privileged_print(const char* message, long user){
  if (user==0xbada55){
    print("Root says: %s\n", message);
  }
  if(user == 0xdeadbeef){
    print("Users do not have access to this function\n");
  }
}

void injection(){

}

int main(){
  long user = 0xdeadbeef;
  injection();
  root_privileged_print("ALL YOUR BASE ARE BELONG TO US", user);
  return 0;
}
```

This is a hard, lateral-thinking, problem and you might not even be able to get it working in your machine due to the compiler optimizations. The resulting code is absolutely terrible and should never be used in a real program. It will unavoidably have to break the rules of the language and rely on undefined behavior. That being an option is both the power and the danger of C.
Many times, even with the intended solution, the program will crash.
Can you explain why it crashes sometimes?

hint

- Disable compiler optimizations by using the -O0 flag.
- You will need to abuse undefined behavior and the stack layout to solve it.
- The final program will work sometimes even with optimizations enabled if you decorate `user` with the `volatile` keyword.