Pointers

In the previous section, we learned about the Heap memory structure. Pointers are our way of accessing and manipulating the Heap. This lesson will show you how!

We'll cover the following

Pointers represent one of the more powerful features of the C language, but also one of the most feared. Some of this fear and apprehension stems from the ridiculously obtuse ways that pointers **may** be used in C. Often tutorials and courses on C approach teaching pointers by asking the student to decipher bizarre looking pointer syntax combinations that truthfully are rarely used in practice. You may have seen bizarre looking pointer syntax, (and you may see it again), but typically, pointers do not have to be used in a horribly complicated way. Typically pointers are pretty straightforward.

The bottom line is, pointers allow you to work with dynamically allocated memory blocks. You will use pointers to deal with variables that have been allocated on the **heap**. (You can use pointers to deal with stack variables, but in most cases this just isn't necessary).

The basic idea is, a **pointer** is a special data type in C, that contains an **address** to a location in memory. Think of a pointer as an arrow that points to the location of a block of memory that in turn contains actual data of interest. It's not unlike the concept of a phone number, or a house address.

The purpose of pointers is to allow you to manually, directly access a block of memory. Pointers are used a lot for **strings** and **structs**. It's not difficult to imagine that passing the **address** of a large block of memory (such as a struct that contains many things) to a function, is more efficient than making a copy of it and passing in that copy, only to delete that copy when your function is done with it. This is known as **passing by reference** versus **passing by value**. There is a code example below that illustrates this more clearly.

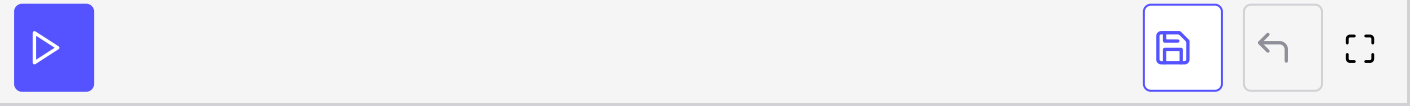
below that illustrates this more clearly.

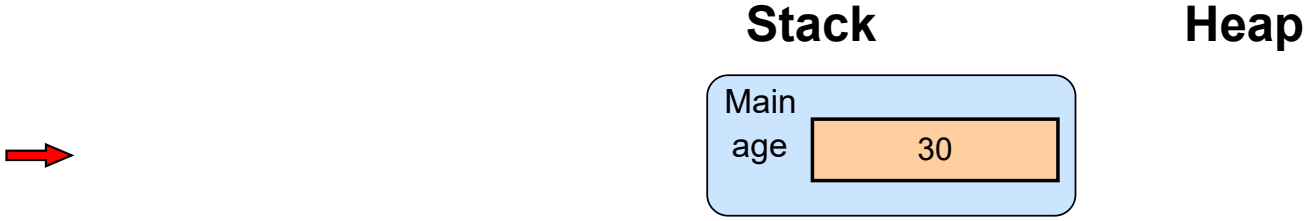
Declaring a pointer

Here is how to declare a pointer:

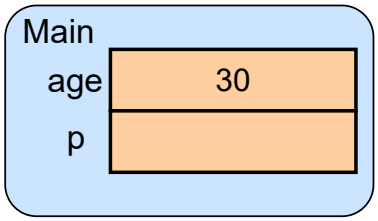
```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int age = 30;
  int *p;
  p = &age;
  printf("age=%d\n", age);
  printf("p=%p\n", p);
  printf("*p=%d\n", *p);
  printf("sizeof(p)=%ld\n", sizeof(p));
  *p = 40;
  printf("*p=%d\n", *p);
  printf("age=%d\n", age);
  return 0;
}
```

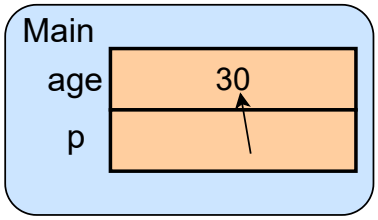


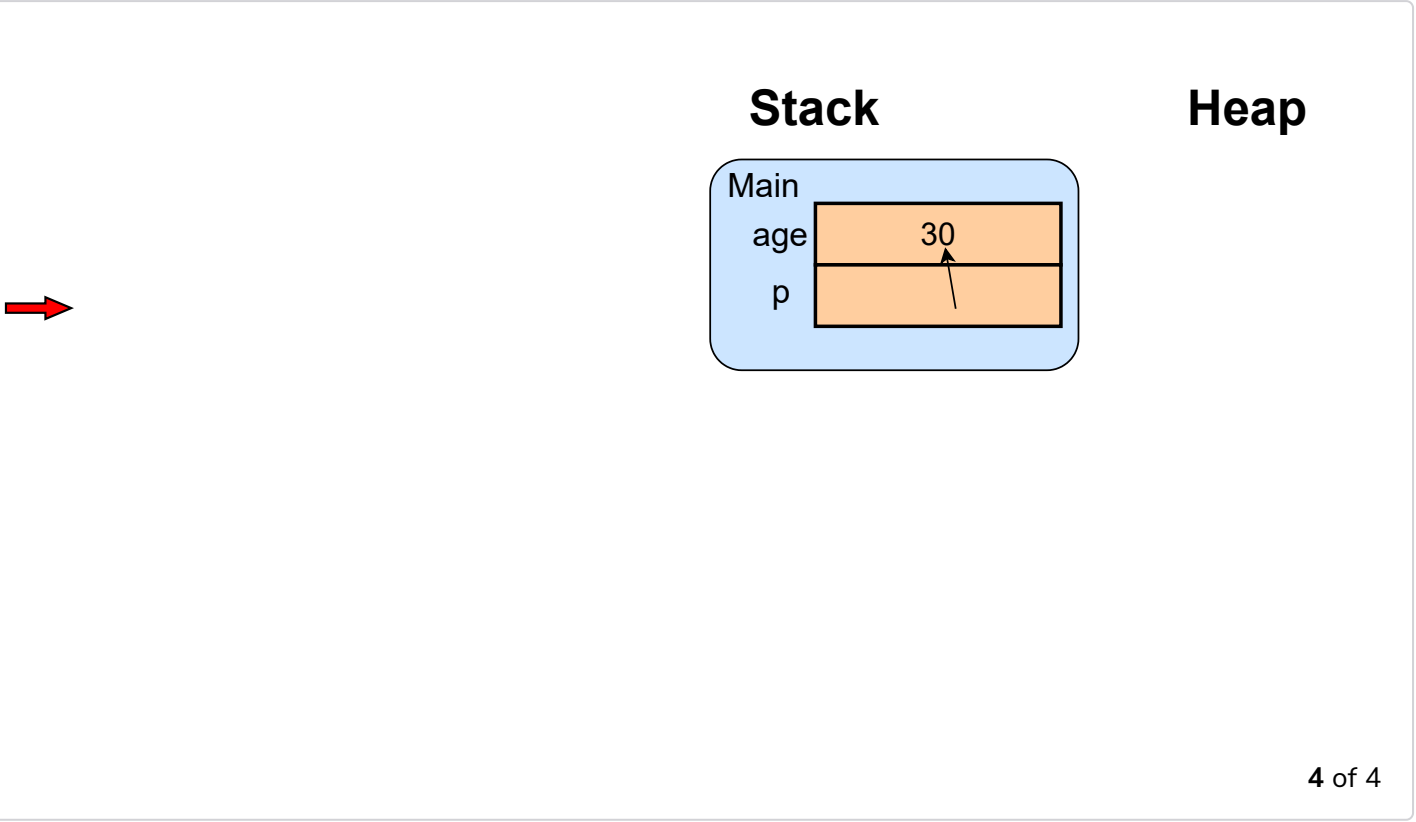


Here, we declare an integer variable `age` and initialize it to **30**.



Here, we declare a variable `p` whose type is a **`pointer to an int`**.





On line 5 we declare an `int` variable `age` and initialize it to `30`. On line 6 we declare a variable `p` whose type is a **pointer to an int**. The star * syntax is how to declare the type as a pointer **to** a particular other type. Usually you need to declare pointers as pointing to a particular type, but the exception is a so-called **void** pointer, which is a pointer to an unknown type. Don't worry about void pointers for now.

So we have a pointer variable `p` that is a pointer to an `int`. So far, it doesn't point anywhere... we have only declared that we want a space in memory to hold a pointer to an int. On line 7 is where we assign a value to our pointer `p`. We assign to `p` the **address** (the location) of the other variable `age`. So now at this point in the program, we have the following (which we can see in the output):

- the value of `age` is the integer `30`
- the value of `p` is the address of the `age` variable
- the integer that `p` points to has a value of `30`

On line 2 of the output we see that the value of the `p` pointer is a long strange looking string of letters and numbers. This is in fact a hexidecimal (base 16) **memory address**.

Dereferencing a Pointer

On line 10, we see how to access the **value** that a pointer points to, by using the star * syntax. So `p` is the address of an `int`, whereas `*p` is the value of the `int` that `p` points to. Accessing the value that a pointer points to is called **dereferencing** a pointer.

Finally, on line 11 (and on the last line of the output) we can see that our pointer `p` is 8 bytes. Remember, there are 8 bits in a byte, so a 4-byte pointer can hold up to 32 bits, or 2^{32} distinct values, i.e. 2^{32} distinct addresses in memory. So 32-bit pointers allow us to access up to 4 gigabytes (4 GB) of memory. If you want to use more than 4 GB of memory you will need bigger pointers! On 64-bit systems, pointers are 8 bytes, which allows for 2^{64} distinct addresses in memory. It will be a **long time** before your computer has that much RAM (16 exabytes in principle, which is 1 billion gigabytes or 1 million terabytes).

Take note of what happens in the code example above on line 12 of the code listing, and the values output on lines 5 and 6 of the output. On line 12 of the code listing, we use pointer dereferencing to set the value of the variable **pointed to by p** to `40`. Since `p` points to the `age` variable, we are setting the value of the address in memory corresponding to the `age` variable to `40`.

Another powerful use of pointers is that they allow us to communicate with arrays. We'll go into more detail in the next lesson.