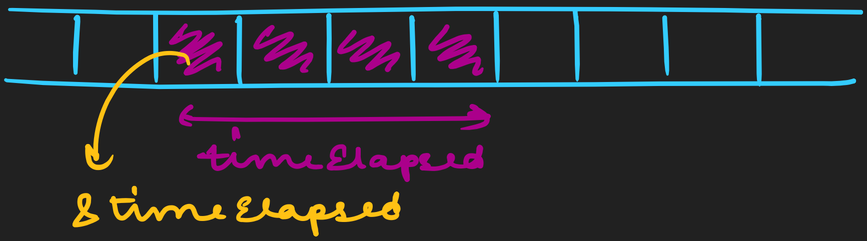
The new address (&) operator.

1. As discussed previously, data is stored in memory, different types of data are stored differently. We can store and use data by using variables. We can find the address of the data in memory by using the & (address operator, not and) operator.
2. In this example, we print the address of the variable timeElapsed. Note that we used %ld instead of %d to print the address because memory addresses are usually very big numbers representing the starting byte of the data in memory.
```
            int timeElapsed = 10000;
            printf("%ld\n", &timeElapsed);            // 140737247519948
```
3. &timeElapsed is the starting point of the variable, and since it is an int, the next 4 bytes together are the actual variable timeElapsed.

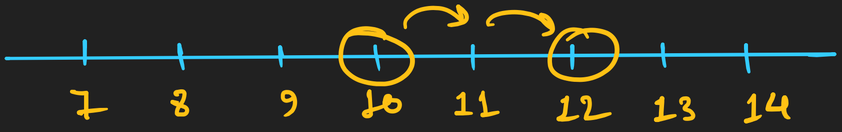


What are pointers?

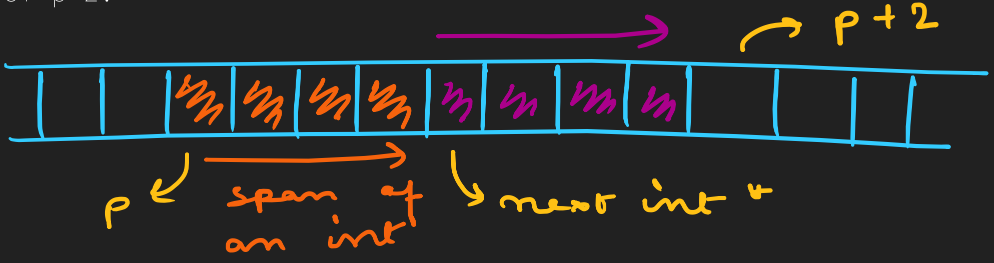
1. Pointers in C are data types which store memory addresses. We could have also used long int, but there are a few advantages of using pointers instead of long integers. Pointer arithmetic is one such example, we will look at it in a while.
2. Pointers come in different flavours, int pointers, float pointers, char pointers, depending on the value stored in the address.
3. To store the address of an integer variable, we can use int *, for char we have char * and so on. Note that pointer types are just data types followed by an asterisk. int -> int *, double -> double *, although we have not yet learned about structs, pointers also work for structs, struct Square -> struct Square *
4. Pointers are called pointers because they point at a byte in the memory (if it wasn’t obvious already).
5. We can use the %p format specifier to print the pointers in hexadecimal (they look better that way because they are big numbers).
```
            int timeElapsed = 10000;
            int *timePointer = &timeElapsed;
            printf("%ld\n", timePointer);             // 140735564207916
            printf("%p\n", timePointer);              // 0x7fff8d4fd72c
```
   Note that the address in both of the examples is different, because there is no fixed place to store a variable, it changes every time we execute the code. Another note: we don’t use the & operator before timePointer in the printf functions this time, because we have already stored the address in the variable timePointer. Can you guess what would happen if we did? Would the code even run?

Pointer arithmetic.

1. For integer arithmetic, whenever we write n+2, we expect the operation to return the integer 2 integers ahead of n. If n were 10, then n+2 would return 12.



2. Similarly in pointer arithmetic, whenever we write p+2, the operation returns the pointer 2 pointers ahead of p. If p were an int * and p = 100, what would've been the value of p+2?



3. So, if p = 100, p+2 should be 100 + 2*4 = 108 and not 102! We can visualize this by thinking p+2 would be 2 ints ahead of p, and if 1 int is 4 bytes long, then 2 ints would be 8 bytes long, thus p+2 would actually be (long)p+8.
4. The code snippet tests the expression. The only things to notice are the explicit type conversions, int to pointer conversions are kind of illegal (and unsafe too). Some compilers may leave you with just a warning while some may throw compile time errors, so make sure to mention explicitly whenever you are trying to convert between int and pointers.

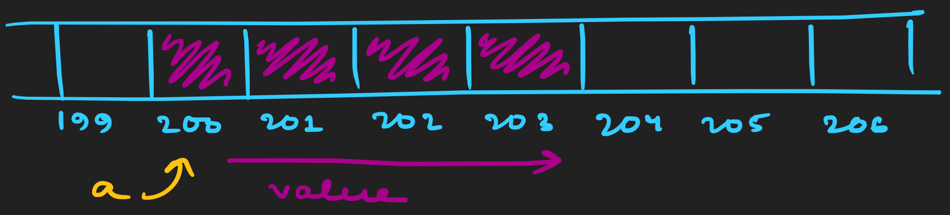
```
int *p = (int *)100;
if (p+2 == (int *)((long)p+8))) {
      printf("That's correct!\n");
} else {
      printf("Oops, you messed up.\n");
}
```

5. The same logic also holds true for subtraction, p-2 would be 2 ints behind p. Fortunately, there are no pointer multiplication or division, just addition and subtraction.

The dereferencing (*) operator.

1. Now we know that pointers point to memory address and all that, and that we can use the & operator to get the address of data, but how do we know what the value at a memory address is? That is where we use the dereferencing operator (*).
2. When * is used before a pointer, it returns the value stored at the address.

```
int *a = (int *)200;
int value = *a;
```



3. It is easy to confuse the dereferencing operator, it may be multiplication, pointer type or dereferencing operator depending on the context.
4. If there is an asterisk between two expressions (expression are like 1*0 or less simpler ones like (a+b)*(b-c) or more complex ones like (*a)*(*b)), then it is the multiplication operator.
5. But if it follows a type name (like int *), then it is a pointer type.
6. Finally, if it comes before a pointer, it is dereferencing the pointer.
7. Bonus challenge: Can you figure out what the following code snippet means? (Specifically, the int *c expression, hint: be careful of operator precedence and different meanings of *)

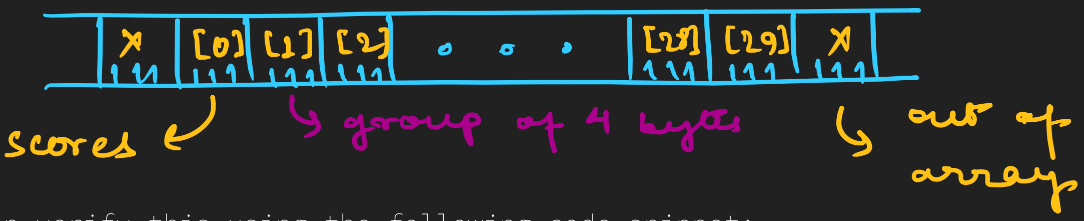
```
int *a = (int *)100;
int *b = (int *)200;
int *c = (int *)(*a+2**(b+3));
```

Arrays.

1. An array is a group of data of the same type stored in contiguous memory location.
2. For instance, if we want to store scores of 30 students in a class, we can create an array to store 30 float values. We can do this by:
   ```
   float scores[30];
   ```
3. In this example, we are telling the size of the array when we are writing the code, this is called hard coding; hard coded details don't change during the execution of the program. When we hard code the length of an array like this, it is called static allocation. We can also set the length of an array during execution; it is called dynamic allocation and we will look at it some other day.

Arrays as pointers.

1. How does an array look like in the memory? More specifically, how is scores represented in the memory?
2. scores is actually a pointer. Since it is a float array, scores is a float pointer, and it points to the start of the array.



We can verify this using the following code snippet:
```
if (scores == &scores[0]) {
      printf("They are the same thing.\n");
} else {
      printf("They are different?\n");
}
```
3. You may already know how to iterate (go through) over all elements of an array using a for loop. But since scores is only the starting point and not the entire 30 blocks of float, C won't mind if we access the scores[50], unlike in languages like python which throw an error if we try to access elements outside of the array. So, it is the responsibility of the programmer to not access any index outside of the array (unless you added a feature which requires you to do so).
4. When we use the sizeof statement with a statically allocated array, it returns the total size of the array and not the size of the pointer. It is because when we specify the size of array during static allocation, C knows what the size of the array is and so does sizeof. But we should not rely on sizeof to find size of arrays, because it won't work outside of the scope where the array was declared anyway. (What does it even mean? We will understand it when we learn about scopes and know C better in general.)

If arrays are pointers, can we use pointer arithmetic with arrays?

1. Absolutely yes! In fact, we can have the same behaviour as array indexing using pointer arithmetic.

2. If we want to access the 20th score, scores[19] and *(scores+19) will be the same. The first one is just easier. If scores is a pointer to the first element of the array, then scores+19 will point to the 20th element of the array and *(scores+19) will dereference the pointer to get the value.
3. So, the general rule is that array[n] is the same as *(array+n).

Write a C program to take the marks obtained and total marks of 5 students from the user and store them in two arrays (one array for marks obtained, another for total marks). Finally, print the percentages obtained by all 5 students. Use pointer arithmetic instead of array indexing for this one.

```c
#include <stdio.h>

int main() {
	float obtained[5], total[5];
	printf("Enter marks obtained and total marks:\n");
	for (int i = 0; i < 5; ++i) {
		printf("Student %d: ", i+1);
		scanf("%f %f", obtained+i, total+i);
		// we did not use & operator because obtained+i
		// and total+i are already addresses
	}
	printf("Percentages obtained:\n");
	for (int i = 0; i < 5; ++i) {
		float percentage = *(obtained+i) * 100.0 / *(total+i);
		printf("Student %d: %.2f%%\n", i+1, percentage);
		// %.2f tells printf to print only 2 places after point
		// we used %% to print %, this is similar to
		// writing \\ to print \ but for format specifiers
	}
	return 0;
}
```

What to do at home?

1. Try the above problem on your own. Do mistakes, fix on your own, in that way you can learn more than what you do in class.
2. Bonus challenge: Try the same problem but you cannot use more than one array to store obtained and total marks.