Pointers and Arrays

Rob Hackman

Winter 2025

University of Alberta

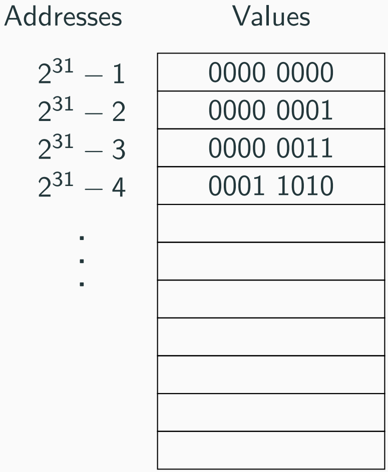
Table of Contents

What is a pointer?

Another data type we have in C is that of a *pointer*.

A pointer is a datatype that stores a memory address — the address of data instead of the value.

The value a pointer stores is the address of some data you're interested in, you can then use the pointer to read, or write, to the data at that location.



Declaring Pointers

When declaring a variable you can declare a pointer with an asterisk (*) character.

The asterisk is part of your type, you must also declare the type you wish this variable to be a pointer to.

```c
int main() {
  int x = 10;
  int *ptr;
}
```

In this example `ptr` has the type “pointer to an `int`”

Assigning Pointers

You shouldn't assign pointers to arbitrary numbers — otherwise you'll have a pointer to a random piece of your memory.

The address-of operator (&) can be applied to a piece of data — it evaluates to the address of that data.

Now `ptr` stores the address of `x` — we would say `ptr` is a *pointer to x*.

```c
int main() {
  int x = 10;
  int *ptr = &x;
}
```

Dereferencing Pointers

A pointer by itself isn't particularly useful — I'm not actually interested in the address of x.

However, the dereference operator (unary *) allows us to "follow" a pointer to the data it points at.

```c
int main() {
  int x = 10;
  int *ptr = &x;
  printf("%p\n", ptr);
  printf("%d\n", *ptr);
}
```

Dereferencing Pointers - aliasing

Dereferencing a pointer is more than an expression that evaluates to the *value* at that memory address however.

Dereferencing a pointer acts as a complete *alias* for the data at that location — that means you can assign to it as well.

What value is printed when we print x?

```c
int main() {
  int x = 10;
  int *ptr = &x;
  printf("%p\n", ptr);
  printf("%d\n", *ptr);
  *ptr = -1;
  printf("%d\n", x);
}
```

The times2 function

Now, let's rewrite the `times2` function below to make it behave as we want.

```c
void times2(int x) {
  x = x*2;
}

int main() {
  x = 10;
  times2(x);
  printf("%d\n", x);
}
```

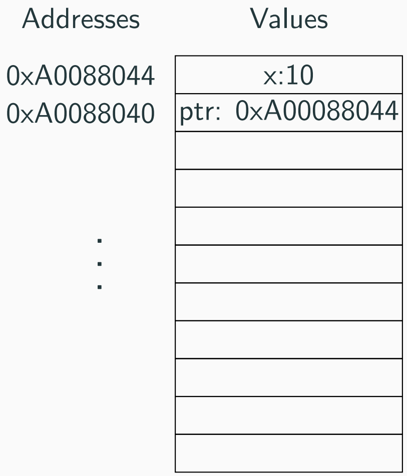
The times2 function

Now, let's rewrite the `times2` function below to make it behave as we want.

```
void times2(int *p) { // !!
  *p = *p*2; // !!
}

int main() {
  x = 10;
  times2(&x); // !!
  printf("%d\n", x);
}
```

The times2 function - example memory layout

```c
void times2(int *p) {
  *p = *p*2;
}

int main() {
  x = 10;
  times2(&x);
  printf("%d\n", x);
}
```



Words

Bytes are grouped together in words — typical word size is 4 or 8 bytes. We'll assume 4-byte words for this class.

In the immediately previous memory diagram and most future memory diagrams we'll only show words, not individual bytes.

That means each box we'll show is actually 4 bytes (each with their own address), and the addresses between each boxes will change by 4 instead of 1.

Hexadecimal

Hexadecimal is often used when displaying memory addresses — it is the base-16 number system

Since digits in base-16 can go from 0 to 15, letters a-f are used to represent digits 10-15

It is also very easy to convert between hexadecimal and binary — if you can interpret a 4-bit binary number you can translate between binary and hex.

Examples:

$$0000_2 = 0_{16}$$
$$1010_2 = A_{16}$$
$$1101\ 0011 = D3_{16}$$
$$0101\ 1111 = 5F_{16}$$

Table of Contents

scanf

We can finally (mostly) understand how input is read in C.

The function `scanf` from the `stdio` library can be used to read from standard input.

Much like `printf`, `scanf` expects a formatted string passed as the first argument, subsequently it expects the addresses you'd like to read user input into.

Reading an int from standard input

The formatted string `"%d"` tells scanf we'd like to read an int from the user.

As we've seen, if we want `scanf` to be able to mutate our variable it must take the address of it — so we pass the address of x.

```c
#include <stdio.h>

int main() {
  int x;
  scanf("%d", &x);
  printf("%d\n", x*2);
}
```

Reading an int from standard input

The formatted string `"%d"` tells scanf we'd like to read an int from the user.

As we've seen, if we want `scanf` to be able to mutate our variable it must take the address of it — so we pass the address of `x`.

What if the user doesn't type an integer?

```
#include <stdio.h>

int main() {
  int x;
  scanf("%d", &x);
  printf("%d\n", x*2);
}
```

scanf — handling errors

`scanf` returns the number of successfully read items.

In our case it should be 1 — if not, we know the user didn't give us an integer

Let's update our program so it only prints out when an integer was successfully read in

scanf — handling errors

`scanf` returns the number of successfully read items.

In our case it should be 1 — if not, we know the user didn't give us an integer

Let's update our program so it only prints out when an integer was successfully read in

```c
#include <stdio.h>

int main() {
  int x;
  int ret = scanf("%d", &x);
  if (ret == 1) {
    printf("%d\n", x*2);
  }
}
```

Reading an Integer

Let's write a program that reads exactly one integer from standard input and prints out twice its value.

If the user enters non-integer input we should ask them to please enter an integer, until they do so.

Reading an Integer

Let’s write a program that reads exactly one integer from standard input and prints out twice its value.

If the user enters non-integer input we should ask them to please enter an integer, until they do so.

Let’s test this program, what happens when you do enter an integer? What about when you dont?

```c
#include <stdio.h>

int main() {
  int x;
  while (1) {
    printf("Enter int: ");
    if (1==scanf("%d",&x)) {
      printf("%d\n", x*2);
      break;
    }
  }
}
```

Reading an Integer

When we don't enter an integer first our program loops forever, even if we type an integer after — what's happening?

Reading an Integer

When we don't enter an integer first our program loops forever, even if we type an integer after — what's happening?

Since `scanf` was unable to read our desired item (an integer) it failed to do so, but as such it doesn't remove any data from standard input.

```
#include <stdio.h>

int main() {
  int x;
  while (1) {
    printf("Enter int: ");
    if (1==scanf("%d",&x)) {
      printf("%d\n", x*2);
      break;
    }
  }
}
```

Reading an Integer

When we don't enter an integer first our program loops forever, even if we type an integer after — what's happening?

Since `scanf` was unable to read our desired item (an integer) it failed to do so, but as such it doesn't remove any data from standard input.

When we try to read each subsquent time `scanf` sees the same non-digit character at the front of the stream and fails.

```c
#include <stdio.h>

int main() {
  int x;
  while (1) {
    printf("Enter int: ");
    if (1==scanf("%d",&x)) {
      printf("%d\n", x*2);
      break;
    }
  }
}
```

Fixing our integer reading

If we want to fix our program so it always reads exactly one `int`, we'll have to remove the offending characters from our stream.

How should we remove the problem characters from our stream? How do we know how much to read?

Fixing our integer reading

If we want to fix our program so it always reads exactly one `int`, we'll have to remove the offending characters from our stream.

How should we remove the problem characters from our stream? How do we know how much to read?

Solution: try to read an integer, if we fail remove exactly *one* character, since we know at least the very first character was not part of an integer. Repeat this process until the read succeeds.

Fixed integer read

```
#include <stdio.h>

int main() {
  int x;
  while (1) {
    if (1==scanf("%d", &x)) {
      printf("%d\n", x*2);
      break;
    }
    char c;
    scanf("%c", &c);
  }
}
```

Reading all integers from stream

What if we want to read *all* integers from the stream?

We could use the same program but just remove the `break` statement.

How would our program ever end? When do we want it to end?

Reading all integers from stream

What if we want to read *all* integers from the stream?

We could use the same program but just remove the `break` statement.

How would our program ever end? When do we want it to end?

When the input stream is exhausted — remember what typing ctrl+d does!

Checking EOF

We an check if we've received an EOF character from a file (or input stream) with the `feof` function.

The name `stdin` is defined in `stdio.h` as our standard input stream.

The function call `feof(stdin)` will return true when an attempted read of `stdin` received the EOF character — that means until you try and read and fail because of EOF it won't return true.

Using feof

```c
#include <stdio.h>

int main() {
  int x;
  int rc;
  while (!feof(stdin)) {
    if (1 == scanf("%d", &x)) {
      printf("%d\n", x*2);
      continue;
    }
    char c;
    scanf("%c", &c);
  }
}
```

Using feof — knowledge check

Knowledge Check: What would happen if we hadn't switched the `break` to `continue`? What would happen if we just deleted `break`?

Some More I/O tools

- `EOF` — constant defined in `stdio`, used by I/O funtions to represent receiving the end of file character.
- `int getchar()` — function that reads and returns one character from standard input, returns `EOF` if end of file reached.

Practice Problem

Write a C program that mimics the behaviour of the `wc` tool we've used when learning about the shell. For now only worry about the behaviour of `wc` when it receives no command-line arguments.

Table of Contents

Collections of Data

In Python we had many ways to store collections of data — `Lists` being one of the most commonly used.

We don't have anything quite like `Lists` in C by default.

What we do have are `arrays`.

Arrays

Arrays allow us to store a collection of sequential data, and they have a few important properties.

- Are a fixed size, specified when they're allocated
- All the items in an array are the same data type
- Is guaranteed to be contiguous in memory[1]

[1]Well, at least the memory as your program sees it...

Declaring an Array

We can declare an array on the stack — however its size must be known at compile-time.

Much like an asterisk is used to denote a pointer we use [] to declare an array. However the brackets come after the identifier.

Additionally [] is also our index operator.

```
int main() {
  int arr[10];
  int x = 5;
  for (int i=0; i<10; ++i) {
    arr[i] = i*2;
  }
}
```

The `sizeof` operator

The `sizeof` operator evalutes to the size (in bytes) of its operand.

```c
int main() {
  int x;
  char c;
  int arr[10];
  printf("%ld\n", sizeof(x));
  printf("%ld\n", sizeof(c));
  printf("%ld\n", sizeof(arr));
}
```

We can see the size of our array is 40 bytes — because it stores 10 integers that are each 4 bytes.

Iterating over a local statically-allocated array

Couple things to notice here

- We can initialize our array with comma-separated values inside curly braces, called an initializer list
- We've calculated the length of the array by dividing its size in bytes by the size in bytes of the type it holds.

```c
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  unsigned int len = sizeof(arr)/sizeof(arr[0]);
  for (unsigned int i=0; i<len; ++i) {
    printf("%d\n", arr[i]);
  }
}
```

Iterating over a local statically-allocated array

Note: We can also omit the length of the array we're declaring when assigning it to an initializer list. The compiler determines the size of the array based on the number of elements in the initializer list.

```c
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  unsigned int len = sizeof(arr)/sizeof(arr[0]);
  for (unsigned int i=0; i<len; ++i) {
    printf("%d\n", arr[i]);
  }
}
```

Arrays and Functions

Consider the output of the following program — is something odd?

```
void printArray(int paramArr[]) {
  unsigned int len = sizeof(paramArr)/sizeof(paramArr[0]);
  for (unsigned int i=0; i<len; ++i) {
    printf("%d\n", paramArr[i]);
  }
}

int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  printArray(arr);
}
```

Arrays and Functions

Consider the output of the following program — is something odd?

```c
void printArray(int paramArr[]) {
  unsigned int len = sizeof(paramArr)/sizeof(paramArr[0]);
  for (unsigned int i=0; i<len; ++i) {
    printf("%d\n", paramArr[i]);
  }
}

int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  printArray(arr);
}
```

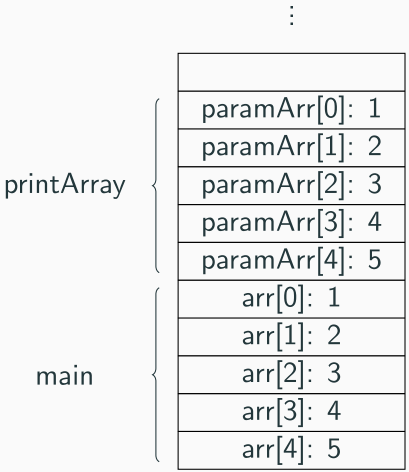
It only prints 1 and 2 — why?

Arrays as parameters

Consider what is involved in copying an array

For this array of five `ints` we would have to copy 40 bytes into the functions stackframe.

What about for larger arrays?



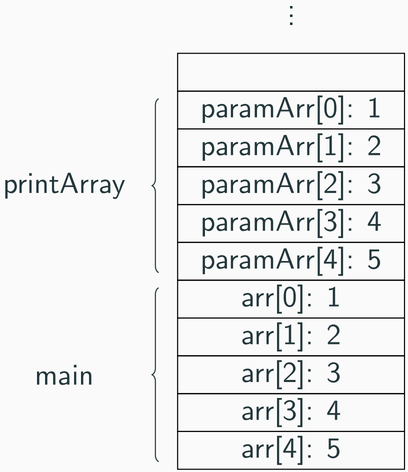
Arrays as parameters

Consider what is involved in copying an array

For this array of five `ints` we would have to copy 40 bytes into the functions stackframe.

What about for larger arrays?

This is not what C does!

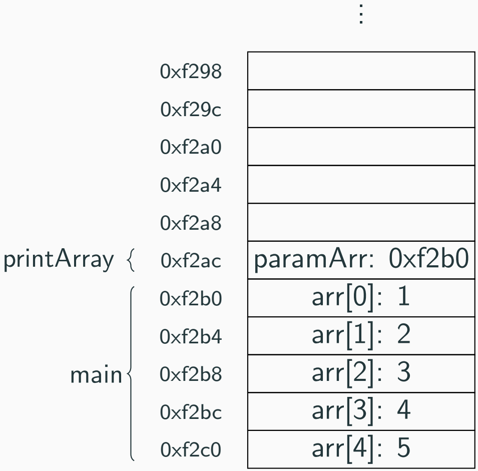


Array Decay

In order to avoid copying an entire array, C copies only the address of the first element of the array.

This means, even when you define your function to take an array parameter it really takes a pointer!

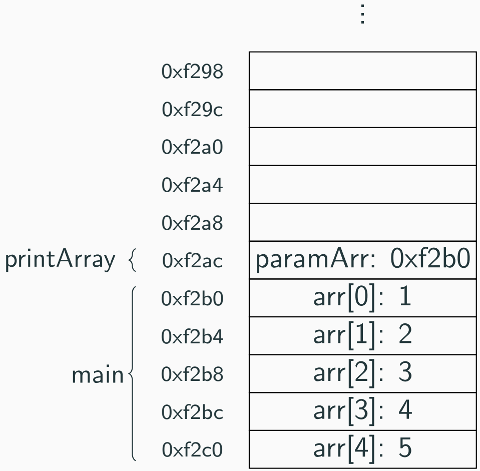
We call this *array decay*, as the array is “decaying” into the pointer to its first element when passed as an argument.



Array Decay

Array decay also means that our function can mutate the caller's array!

As we've seen pointers allow mutating data's memory, since the pointer is the address of that data.



Arrays as Parameters

Array decay means we cannot use `sizeof` to determine the length of an array that is a parameter.

Functions that operate over array parameters must also be given the length — or have some other indication of the end of the array[1].

```
void printArray(int arr[], unsigned int size) {
  for (unsigned int i = 0; i < size; ++i) {
    printf("%d\n", arr[i]);
  }
}
```

[1] More on this soon

Arrays as Parameters

Another consequence of array decay is that these two functions are identical.

```
void printArray(int arr[], unsigned int size) {
  for (unsigned int i = 0; i < size; ++i) {
    printf("%d\n", arr[i]);
  }
}

void printArray(int *arr, unsigned int size) {
  for (unsigned int i = 0; i < size; ++i) {
    printf("%d\n", arr[i]);
  }
}
```

Arrays are not pointers!

Note: While `int *` and `int []` become the same type **when used as function parameters.** That does *not* mean that arrays and pointers are the same thing.

Arrays are contiguous blocks of memory — pointers are pieces of data that store a memory address.

Arrays are not pointers!

```c
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int *p = arr;
  printf("sizeof(arr): %lu\n", sizeof(arr));
  printf("sizeof(p): %lu\n", sizeof(p));
  printf("p: %p\n", p);
  printf("&p: %p\n", &p);
  printf("arr: %p\n", arr);
  printf("&arr: %p\n", &arr); // !!!
}
```

Arrays are not pointers!

```c
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int *p = arr;
  printf("sizeof(arr): %lu\n", sizeof(arr));
  printf("sizeof(p): %lu\n", sizeof(p));
  printf("p: %p\n", p);
  printf("&p: %p\n", &p);
  printf("arr: %p\n", arr);
  printf("&arr: %p\n", &arr); // !!!
}
```

When used in an expression an array identifier evalutes to the address of its first element.

Arrays are not pointers!

```c
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int *p = arr;
  printf("sizeof(arr): %lu\n", sizeof(arr));
  printf("sizeof(p): %lu\n", sizeof(p));
  printf("p: %p\n", p);
  printf("&p: %p\n", &p);
  printf("arr: %p\n", arr);
  printf("&arr: %p\n", &arr); // !!!
}
```

When used in an expression an array identifier evalutes to the address of its first element.

Unlike a pointer, that address is not stored anywhere, it just *is* the address of the array, so asking for the address of an array produces the same value.

Arrays as Parameters - Mutation

Since arrays are being passed as pointers, the caller's array can be mutated by the function's code.

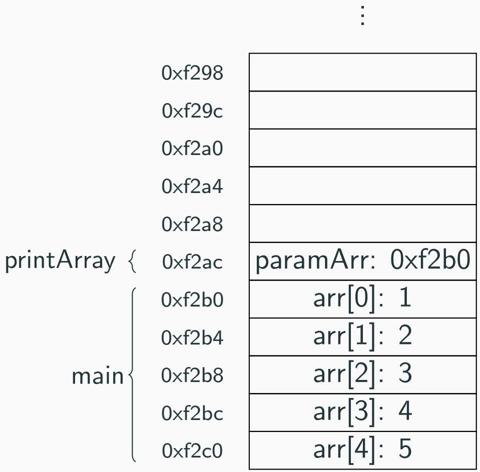
```
void doubleArray(int arr[], unsigned int size) {
  for (unsigned int i = 0; i < size; ++i) {
    arr[i] = arr[i]*2;
  }
}

void main() {
  int arr[5] = {1, 2, 3, 4, 5};
  doubleArray(arr, 5);
  for (unsigned int i = 0; i < 5; ++i) {
    printf("%d\n", arr[i]);
  }
}
```

Array Storage

Why, in this diagram, is the first element of `arr` not stored at the bottom of the stack?

Seemingly, the array is stored reverse of what we might expect. Though we have to consider it is only the stack that grows towards 0, the rest of our memory grows towards the largest address.

To answer this question, we consider what `arr[0]` really means.



Pointer Arithmetic

Pointer decay implies that the index operator can be applied to pointers as well as arrays.

```
int *p = ...;
p[0] = 10;
```

So what then does `p[0]` mean?

Pointer Arithmetic

Pointer decay implies that the index operator can be applied to pointers as well as arrays.

```
int *p = ...;
p[0] = 10;
```

So what then does `p[0]` mean?

```
T *x = ...;
x[i] -> *(x+i)
```

Pointer Arithmetic

We can see this for ourselves:

```c
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  for (unsigned int i = 0; i < 5; ++i) {
    printf("%d\n", *(arr+i));
  }
}
```

Pointer Arithmetic

We can see this for ourselves:

```
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  for (unsigned int i = 0; i < 5; ++i) {
    printf("%d\n", *(arr+i));
  }
}
```

But `arr` isn't a pointer, it's an array! The same concept as array decay is taking place here. More generally, when used in expressions an array evaluates to the address of its first element. So in the expression `*(arr+i)`, `arr` evaluates to the address of its first element.

Pointer Arithmetic

But the index doesn't correspond to how much we would have to add to the address to resolve to the address of the next item! In our array of integers the address of each subsequent item is 4 bytes ahead, not 1.

When performing arithmetic on pointers, C performs *pointer arithmetic*. In general, for any arbitrary type T if we have a pointer to type T as such:

```
T *x = ...;
x + i; // Actually performs x + sizeof(T)*i
```

Pointer Arithmetic in Action

We can see pointer arithmetic in action by printing out the results of adding our indices to the address of our first element.

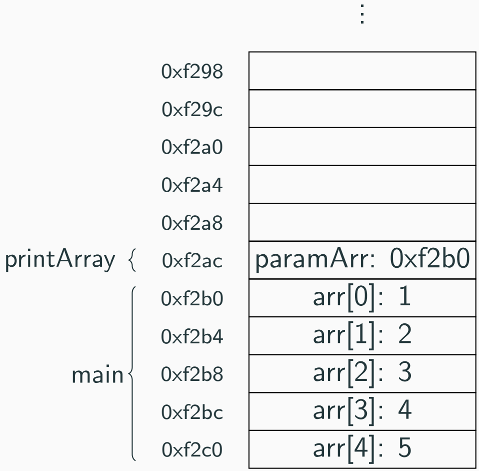
The output of this program shoes that everytime we add 1 to our address we're actually adding 4 (the size of an int, which our array stores).

```
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  char cArr[5] = {'a', 'b', 'c', 'd', 'e'};
  for (unsigned int i = 0; i < 5; ++i) {
    printf("%p: %d\n", arr+i, *(arr+i));
    printf("%p: %c\n", cArr+i, *(cArr+i));
  }
}
```

Array Storage and Pointer Arithemtic

So our indices into an array are really offsets from the start of the array.

As our offsets are added to the start of the array, the start of the array must be the lowest address with the end being the highest.



Practice Question

Practice Question: Write a function `argmax` that returns the *index* of the maximal item in an array of integers. What parameters does your function need to have?

Practice Question: Write a function `replace` which takes in an array and is parameterized by two integers `tar` and `repl` and replaces every instance of `tar` in the given array with `repl`.

Table of Contents

Strings in C

We’ve mentioned there is no `string` type in C — so what do we have?

We have arrays of character, we will use arrays of characters to implement our basic strings (often called C strings).

String literals and storage

String literals we've already seen — but if strings are arrays of characters where are these literals stored?

String literals are stored often in the text *or* data segment of your program[1].

Except when explicitly allocated on the stack by storing a string literal in a stack-allocated array.

[1]This is implementation dependant, meaning your compiler chooses.

String literals and storage

These two variable definitions are quite different.

```c
int main() {
  char myS[] = "On Stack";
  char *s = "Text/Data";
  myS[0] = 'X';
  s[0] = 'X'; // Crashes
}
```

String literals and storage

These two variable definitions are quite different.

The array `myS` is stored on the stack, and as such all of the characters that make up the string are on the stack — we are able to modify our stack so assigning `myS[0] = ’X’;` is perfectly fine.

```c
int main() {
  char myS[] = "On Stack";
  char *s = "Text/Data";
  myS[0] = 'X';
  s[0] = 'X'; // Crashes
}
```

String literals and storage

These two variable definitions are quite different.

The *pointer* `s` is stored on the stack, the data it points at is stored in the text or data section. Wherever it is stored it is marked *read-only*, so attempting to modify that memory with `s[0] = 'X';` crashes our program.

```c
int main() {
  char myS[] = "On Stack";
  char *s = "Text/Data";
  myS[0] = 'X';
  s[0] = 'X'; // Crashes
}
```

`const` type modifier

You can specify data that should never change with the `const` type modifier.

Modifying a const variable produces a compilation error.

Anything you don't intend to change mark const!

```
int main() {
  const int x = 5;
  x = 10; // Error!
}
```

String literals and pointers

Pointers to string literals should always be pointers to `const char`, so that one does not modify a portion of memory that does not belong to them.

```
int main() {
  char myS[] = "On Stack";
  const char *s = "Text/Data";
  myS[0] = 'X';
  s[0] = 'X'; // Compilation error! Better!
}
```

String literals and pointers

Pointers to string literals should always be pointers to `const char`, so that one does not modify a portion of memory that does not belong to them.

```
int main() {
  char myS[] = "On Stack";
  const char *s = "Text/Data";
  myS[0] = 'X';
  s[0] = 'X'; // Compilation error! Better!
}
```

We read the type of `s` here as "pointer to a constant character" — this is different than a "constant pointer to a character"

Using `const`

- px is a pointer to a constant `int` — cannot change the `int` that px points at *through* px.
- py is a constant pointer to an `int` — can change the `int` that py points at, but cannot change py iteslf. i.e. py will always point at `y`'s address.
- pz is also a pointer to a constant `int` — same behaviour as px.

```c
int main() {
  int x = 0;
  int y = 10;
  int z = 20;
  const int *px = &x;
  int * const py = &y;
  int const *pz = &y;
  *px = 5; // Illegal
  *py = 7; // Legal
  *pz = 10; // Illegal
  px = &y; // Legal
  py = &x; // Illegal
  pz = &x; // Legal
}
```

More about `const`

`const` can be applied to any types, not just pointers.

`const` variables cannot be left uninitialized — since they can't be changed they must have an initial value.

Anytime you have data you don't intend to change you should mark it as `const`

Example of `const`!

```c
int contains(const int *arr, const int targ,
                             const int len) {
  for (unsigned int i = 0; i < len; ++i) {
    if (arr[i] = targ) return 1;
  }
  return 0;
}

int main() {
  const int SIZE = 5;
  int arr[SIZE] = {1, 2, 3, 4, 5};
  contains(arr, 3, SIZE);
}
```

The String Library

While it doesn't provide a special type for strings, C does have a standard string library which provides several useful functions.

One such function is the strlen function:

```
int main() {
  const char *p = "abcd";
  printf("%d\n", strlen(p));
}
```

The String Library

While it doesn't provide a special type for strings, C does have a standard `string` library which provides several useful functions.

One such function is the `strlen` function:

```
int main() {
  const char *p = "abcd";
  printf("%d\n", strlen(p));
}
```

But how can the function `strlen` calculate the length of the string, it receives only a pointer to a character array...

Strings in C

How many characters are actually in our string "abcd"?

Since we can place the whole string in a stack array we can calculate the number of characters using sizeof.

```c
int main() {
  char arr[] = "abcd";
  printf("%lu\n", sizeof(arr)/sizeof(char));
}
```

Strings in C

How many characters are actually in our string `"abcd"`?

Since we can place the whole string in a stack array we can calculate the number of characters using `sizeof`.

```c
int main() {
  char arr[] = "abcd";
  printf("%lu\n", sizeof(arr)/sizeof(char));
}
```

Prints 5 — but we only typed 4 characters in the literal

Strings in C

A string in C is a null-terminated array of characters.

If an array of characters does not end in the null terminator character (ASCII character 0), then it is not a valid C string.

```c
int main() {
  char arr[] = "abcd";
  char same[] = {'a', 'b', 'c', 'd', '\0'};
  printf("%lu\n", sizeof(arr)/sizeof(char));
}
```

Strings in C

A string in C is a null-terminated array of characters.

If an array of characters does not end in the null terminator character (ASCII character 0), then it is not a valid C string.

```c
int main() {
  char arr[] = "abcd";
  char same[] = {'a', 'b', 'c', 'd', '\0'};
  printf("%lu\n", sizeof(arr)/sizeof(char));
}
```

Knowledge Check: how does `strlen` work without being provided a size parameter?

strlen sample implementation

Here is one way one could implement strlen

```c
unsigned int strlen(const char *s) {
  unsigned int i = 0;
  while (*s != '\0') {
    ++i; ++s;
  }
  return i;
}
```

Note: Do you understand this code? It combines several of the topics we've just discussed.

Practice Question

Practice Question: Write the function str_replace as described below.

```
int str_replace(char *s, char find, char repl);
```

str_replace replaces every instance of find in string s with the character repl and returns the number of instances replaced.

Useful string Functions

You should read up on the functions available in `string.h`, here are a few particularly useful ones:

- `int strcmp(const char *s1, const char *s2)` — compares two strings lexicographically, returns 0 if strings are equal, < 0 if $s1 < s2$, and > 0 if $s1 > s2$.
- `char *strchr(const char *s, int c)` — searches for character `c` in the string `s`. If it finds it returns a pointer to the first occurrence of the character in the string, otherwise returns NULL.
- `char *strcat(char *dest, const char *src)` — concatenates `src` on to the end of `dest`. Note: the array pointed at by `dest` must be large enough to have the characters of `src` appended on, and also `dest` and `src` must not overlap.