Systems Programming

Lecture 10: const, arrays and function pointers

Stuart James

stuart.a.james@durham.ac.uk


Recap

https://PollEv.com/stuartjames

Recap - Pointers

- Pointers
- Pointer Assignment
- Pointers as Arguments
- Pointer Arithmetic
- Pointers to Pointers
- Dangers of Pointers

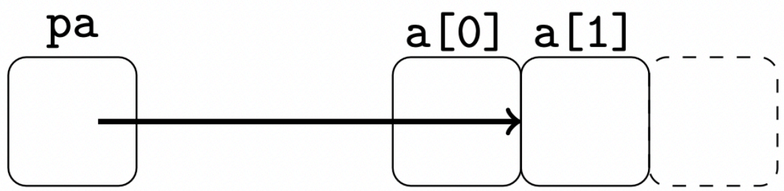
Breaking things is easy!

Recap - Pointer Arithmetic

```
int a[10];
int *pa;
```

This pair of statements are equivalent (+1 translates to +4 bytes (1 int)):

```
pa = &a[1];
pa = (a+1);
```



Recap - Arrays and Strings

```
char a[] = "Hello worlds";
char b[13];
char *c;
c = a;
```


Recap - Arrays and Strings

```
char a[] = "Hello worlds";
char b[13];
char *c;
c = a;
```

- This will set pointer `c` to the same address as `a`
  - `a` is essentially a pointer!
- We can use `strcpy(b,a);`
  - first argument is the destination - need to `#include <string.h>`


`const`

What is const?

- const is a keyword in C used to define variables whose values should not be altered after initialisation.
- It ensures code safety by preventing accidental changes to these values, making your code more predictable and maintainable.

Why Use const?

- Safety: Protects critical data from accidental modification.
- Optimization: Helps compilers optimize code by knowing certain values are immutable.
- Readability: Indicates to other developers that certain values are not meant to change.


`const`

```
const int max_value = 100;
```

- In this example, max_value is declared as a const int, meaning it can't be modified once assigned.
- Any attempt to modify max_value will result in a compiler error.

Common Use Cases:

- Constants: Defining fixed values, e.g., const float PI = 3.14;
- Array Sizes: Fixed-size arrays, e.g., const int ARRAY_SIZE = 10;
- Function Parameters: Ensures that arguments passed to a function are not modified within it.


`const` in Functions

Using const for Function Parameters:

If a function parameter is not supposed to be modified, declare it as const. Example:

```
void print_value(const int value) {
    printf("%d\n", value);
}
```


`const` in Functions

const Return Types:

- Return a const type to indicate the returned value should not be altered.
- Useful in certain design patterns where immutability is critical for consistency.

```
const int get_fixed_value() {
    return 42;
}
```


`const` and Pointers

The `const` keyword is used differently when pointers are involved.

- These two declarations are equivalent:

```
const int *ptr_a;
int const *ptr_a;
```


`const` Pointers

The `const` keyword is used differently when pointers are involved.

- However, are these equivalent?

```
int const *ptr_a;
int *const ptr_b;
```


`const` Pointers

- No, these two are **Not** equivalent:

```
int const *ptr_a;
int *const ptr_b;
```

- In the first example, the `int` (i.e. `*ptr_a`) is `const`.
  - We cannot do `*ptr_a = 123`.
- In the second example, the pointer itself is `const`.
  - We can change `*ptr_b` just fine, but you cannot change (using pointer arithmetic, e.g. `ptr_b++`) the pointer itself.


`const` Pointers: `int const *ptr_a;`


`const` Pointers: `int const *ptr_a;`

In [17]:

```c
#include<stdio.h>

int main(){
    int x = 10;
    int y = 20;

    int const *ptr_a = &x;  // ptr_a points to x

//     *ptr_a = 15;  // Error: Cannot modify the value of x through ptr_a because it's a pointer to a const int.

    ptr_a = &y;   // Allowed: ptr_a can point to a different int (y)

}
```


`const` Pointers: `int *const ptr_a;`


`const` Pointers: `int *const ptr_a;`

In [18]:

```c
#include<stdio.h>

int main(){
    int x = 10;
    int y = 20;

    int *const ptr_b = &x;  // ptr_b is a constant pointer to x

    *ptr_b = 15;   // Allowed: You can modify the value of x through ptr_b

//     ptr_b = &y;    // Error: Cannot change ptr_b to point to y because it's a constant pointer.

}
```


Arrays

- We can also have multi-dimensional arrays in C, e.g.

```
int matrix[2][3] = {{1,2,3},{4,5,6}};
```

- Now `matrix[0][1]==2`.
- We can have more than 2-dimensional arrays:

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};
```


Multi-Dimensional Arrays

- We can have more than 2-dimensional arrays:

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};
```

The elements of arr3d will be allocated in memory in the order

`arr3d[0][0][0]`, `arr3d[0][0][1]`, `arr3d[0][0][2]`,

`arr3d[0][0][3]`, `arr3d[0][1][0]`, `arr3d[0][1][1]`, etc.


Multi-Dimensional Arrays

- How would we print the values in a multi-dimensional array?


Multi-Dimensional Arrays

- How would we print the values in a multi-dimensional array?

In [19]:

```c
#include<stdio.h>

int main(){
    int arr3d[3][2][4] = {
        {{1, 2, 3, 4}, {5, 6, 7, 8}},
        {{9, 10, 11, 12}, {13, 14, 15, 16}},
        {{17, 18, 19, 20}, {21, 22, 23, 24}}
    };
        for(int j=0;j <2;j++)
    for(int i=0;i < 3;i++)

            for(int k=0;k < 4;k++){
                printf("the value at arr[%d][%d][%d]: ",i,j,k);
                printf("%d\n",arr3d[i][j][k]);
            }
}
```

```
the value at arr[0][0][0]: 1
the value at arr[0][0][1]: 2
the value at arr[0][0][2]: 3
the value at arr[0][0][3]: 4
the value at arr[1][0][0]: 9
the value at arr[1][0][1]: 10
the value at arr[1][0][2]: 11
the value at arr[1][0][3]: 12
the value at arr[2][0][0]: 17
the value at arr[2][0][1]: 18
the value at arr[2][0][2]: 19
the value at arr[2][0][3]: 20
```


Multi-Dimensional Arrays

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};
```

- `&arr3d[i][j][k]` is the same as `&arr3d[0][0][0]+(i*2*4)+j*4+k`


Multi-Dimensional Arrays

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};
```

- `&arr3d[i][j][k]` is the same as `&arr3d[0][0][0]+(i*2*4)+j*4+k`

In [20]:

```c
#include<stdio.h>

int main(){
    int arr3d[3][2][4] = {
        {{1, 2, 3, 4}, {5, 6, 7, 8}},
        {{9, 10, 11, 12}, {13, 14, 15, 16}},
        {{17, 18, 19, 20}, {21, 22, 23, 24}}
    };
    int i = 1;
    int j = 2;
    int k = 3;
    printf("%d\n", arr3d[i][j][k]);
    printf("%d\n", arr3d[0][0][0]+(i*2*4)+j*4+k);

}
```

```
20
20
```


Multi-Dimensional Arrays

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};
```

How would you create a variable to store these?

- What is the type of `arr3d[0][0][0]`?
- What is the type of `arr3d[0][0]`?
- What is the type of `arr3d[0]`?
- What is the type of `arr3d`?


In [21]:

```c
#include <stdio.h>

int main(){

    int arr3d[2][3][4] = {
        {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
        {{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}
    };

    //What are the types of the following?
    //How would you create a variable to store them?
    int zerod = arr3d[0][0][0];
    int *oned = arr3d[0][0];
    int (*twod)[4] = arr3d[0];
    int (*threed)[3][4] = arr3d;

    // Print the variables
    printf("\nVariables:\n");
    printf("zerod = %d\n", zerod);
    printf("oned = %d\n", *oned);
    printf("twod[0][0] = %d\n", twod[0][0]);
    printf("threed[0][0][0] = %d\n", threed[0][0][0]);
}
```

```
Variables:
zerod = 1
oned = 1
twod[0][0] = 1
threed[0][0][0] = 1
```

Setting data


In [22]:

```c
#include <stdio.h>

int main(void) {
    int arr[2][3][4];

    // Naïve version — triple for loop
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 3; j++)
            for (int k = 0; k < 4; k++)
                arr[i][j][k] = 0;

    printf("arr[1][2][3] = %d\n", arr[1][2][3]);
}
```

```
arr[1][2][3] = 0
```


Setting data: memset


Setting data: memset

In [23]:

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

int main(void) {
    int arr[2][3][4];

    // Equivalent version
    memset(arr, 0, sizeof(arr));   // fill all bytes with 0

    printf("arr[1][2][3] = %d\n", arr[1][2][3]);
}
```

```
arr[1][2][3] = 0
```


Setting data: memset

```
void *memset(void *ptr, int value, size_t n);
```

- Fills the first n bytes of memory at ptr with the byte value value.
- Common use cases:
  - Zero-initialising arrays, structs, or buffers.
  - Clearing sensitive data before freeing memory.
  - Resetting large regions of contiguous memory quickly.


Setting data: memset

Setting data: memset

In [24]:

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

int main(void) {
    int arr[2][3][4];

    // Equivalent version
    memset(arr, 1, sizeof(arr));

    printf("arr[1][2][3] = %d\n", arr[1][2][3]);
}
```

```
arr[1][2][3] = 16843009
```


Setting data: memset

In [24]:

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

int main(void) {
    int arr[2][3][4];

    // Equivalent version
    memset(arr, 1, sizeof(arr));

    printf("arr[1][2][3] = %d\n", arr[1][2][3]);
}
```

```
arr[1][2][3] = 16843009
```

⚠️ Non-zero patterns can yield unexpected integer values (e.g., 0x01010101 = 16843009

Copying data


Copying data

In [25]:

```c
#include <stdio.h>

int main(void) {
    int src[2][3][4];
    int dst[2][3][4];

    // Initialise source with sequential values
    for (int i = 0, n = 1; i < 2; i++)
        for (int j = 0; j < 3; j++)
            for (int k = 0; k < 4; k++)
                src[i][j][k] = n++;

    // Copy source to Dest
    for (int i = 0, n = 1; i < 2; i++)
        for (int j = 0; j < 3; j++)
            for (int k = 0; k < 4; k++)
                dst[i][j][k] = src[i][j][k];

    printf("src[1][2][3] = %d\n", src[1][2][3]); // prints 24
    printf("dst[1][2][3] = %d\n", dst[1][2][3]); // prints 24
}
```

```
src[1][2][3] = 24
dst[1][2][3] = 24
```


Copying data: memcpy


Copying data: memcpy

In [26]:

```c
#include <stdio.h>
#include <string.h> // Declared in string.h

int main(void) {
    int src[2][3][4];
    int dst[2][3][4];

    // Initialise source with sequential values
    for (int i = 0, n = 1; i < 2; i++)
        for (int j = 0; j < 3; j++)
            for (int k = 0; k < 4; k++)
                src[i][j][k] = n++;

    // Copy source to Dest
    memcpy(dst, src, sizeof(src));

    printf("src[1][2][3] = %d\n", src[1][2][3]); // prints 24
    printf("dst[1][2][3] = %d\n", dst[1][2][3]); // prints 24
}
```

```
src[1][2][3] = 24
dst[1][2][3] = 24
```


Copying data: memcpy

```
void *memcpy(void *dest, const void *src, size_t n);
```

- Copies n bytes from src to dest — here, the whole 3D array: n = sizeof(src) → 2 × 3 × 4 × sizeof(int) bytes.
- Operates on raw memory, not types or array dimensions.


Some other memory functions

```
memcmp(const void *a, const void *b, size_t n);
```

Compares the first n bytes of two memory blocks. Returns <0, 0, or >0 depending on lexical byte order.

```
memchr(const void *ptr, int value, size_t n)
```

Scans memory for the first occurrence of a byte value. Useful for binary search within raw buffers.

```
aligned_alloc(size_t alignment, size_t size); // Note C11
```

Allocates memory aligned to a given power-of-two boundary (e.g. 16-byte for SIMD).


Pointers and Arrays

For further fun with pointers and arrays, take look at:

https://www.oreilly.com/library/view/understanding-and-using/9781449344535/ch04.html


Function Pointers

It's possible to take the address of a function, too.

- Similarly to arrays, **functions decay to pointers when their names are used**.

So if we wanted the address of `strcpy`, we could just use `strcpy` or `&strcpy`.


Function Pointers

It's possible to take the address of a function, too.

- Similarly to arrays, **functions decay to pointers when their names are used**.

So if we wanted the address of `strcpy`, we could just use `strcpy` or `&strcpy`.

In [27]:

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

int main(){
    char src[] = "This is a string.", dst[18];
    strcpy(dst, src); //strcpy is basically a pointer
    printf("%s", dst);
    return 0;
}
```

```
This is a string.
```


Function Pointers

There's syntax for declaring variables whose type is a function pointer.

- This is an ordinary function declaration:

```
char *strcpy(char *dst, const char *src);
```

- We can now have a function pointer:

```
char *(*strcpy_ptr)(char *dst, const char *src);
```


In [28]:

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

char *(*strcpy_ptr)(char *dst, const char *src);

int main(){
    char src[] = "This is a string.", dst[18];

    //strcpy_ptr = strcpy;
    //strcpy_ptr = &strcpy;
    strcpy_ptr = &strcpy[0];
    strcpy_ptr(dst, src);
    printf("%s", dst);
    return 0;
}
```

```
/var/folders/4j/xg6vmdqs44z51nqdy93ncv_h0000gn/T/tmp30uu3k7k.c:11:19: error: subscript of pointer to function type 'char *(c
har *, const char *)'
   11 |     strcpy_ptr = &strcpy[0];
      |                   ^~~~~~
1 error generated.
[C kernel] GCC exited with code 1, the executable will not be executed
```

Function Pointers

```
char *(*strcpy_ptr)(char *dst, const char *src);

strcpy_ptr =  strcpy;
strcpy_ptr = &strcpy;    // This works too
strcpy_ptr = &strcpy[0]; // But not this, for obvious reasons
```


Function Pointers

```
char *(*strcpy_ptr)(char *dst, const char *src);
```

- Note the parentheses around `*strcpy_ptr` in the declaration.
  - These separate the `*` indicating return type (`char *`) from the `*` indicating the pointer variable (`*strcpy_ptr` — pointer to function).
- As you might expect, a pointer to a pointer to a function has two asterisks inside of the parentheses:

```
char *(**strcpy_ptr_ptr)(char *, const char *) = &strcpy_ptr;
```


Function Pointers

Things can get a bit too complicated:

- A function pointer can even be the return value of a function.

```
char *(*get_strcpy_ptr(void))(char *dst, const char *src);
```

- This is basically the declaration of a function that returns a function pointer.

Since function pointers can get confusing, many use `typedefs` to abstract them:

```
typedef char *(*strcpy_funcptr)(char *, const char *);

strcpy_funcptr strcpy_ptr = strcpy;
strcpy_funcptr get_strcpy_ptr(void);
```


Function Pointers

What is important is, just as we have pointers to variables, we can also have pointers to functions!


Function Pointers

What is important is, just as we have pointers to variables, we can also have pointers to functions!

In [29]:

```c
#include<stdio.h>
void hello_function(int times);

int main(){
    void (*func_ptr)(int);
    func_ptr=hello_function;
    func_ptr(3);
    return 0;
}

void hello_function(int times){
    for(int i=0;i<times;i++)
        printf("Hello, Function Pointer!\n");
}
```

```
Hello, Function Pointer!
Hello, Function Pointer!
Hello, Function Pointer!
```


Realistic Example - Using `qsort()`

`stdlib.h` contains an implementation of the quicksort algorithm:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void *, const void *))
```

- `void *base` is a pointer to the array.
- `size_t nmemb` is the number of elements in the array.
- `size_t size` is the size of each element.
- `int (*compare)(const void *, const void *)` is a function pointer composed of two arguments and returns:
  - `0` when the arguments have the same value,
  - < 0 when `arg1` comes before `arg2`,
  - and > 0 when `arg1` comes after `arg2`.


In [30]:

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

int compare (const void *, const void *);

int main() {
    int arr[] = {52, 14, 50, 48, 13};
    int num, width, i;
    num = sizeof(arr)/sizeof(arr[0]);
    width = sizeof(arr[0]);

    qsort(arr, num, width, compare);
    // we could have used &compare
    for (i = 0; i < 5; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
int compare (const void *arg1, const void *arg2) {
    return *(int *)arg1 - *(int *)arg2;
}
```

```
13 14 48 50 52
```

Other uses of function pointers?

Q: Can anyone name any common examples?


Other uses of function pointers?

Q: Can anyone name any common examples?

- Interface libaries e.g. Gnome/Gtk+/Glib? accept call backs (i.e. function pointers)
- Game loops: Render, Physics or Event
- Timers
- Threads


Summary

- `const`
- Pointers and Arrays
- Multi-Dimensional Arrays
- Function Pointers
- Function Pointers real use in `qsort()`