Pointers about pointers

What are pointers?

They contain addresses

Thank you for your time ☺

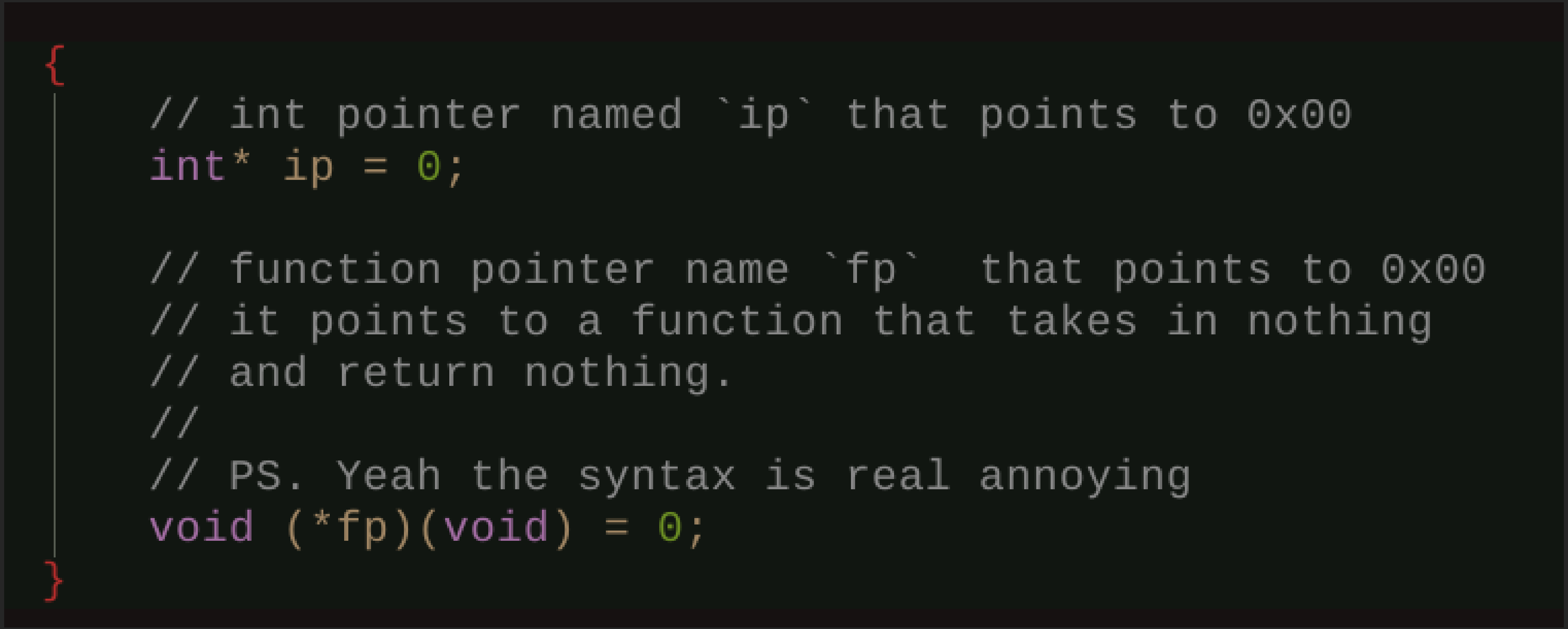
Pointers for function callbacks

What is the difference between a pointer and a function pointer?

They both contain addresses

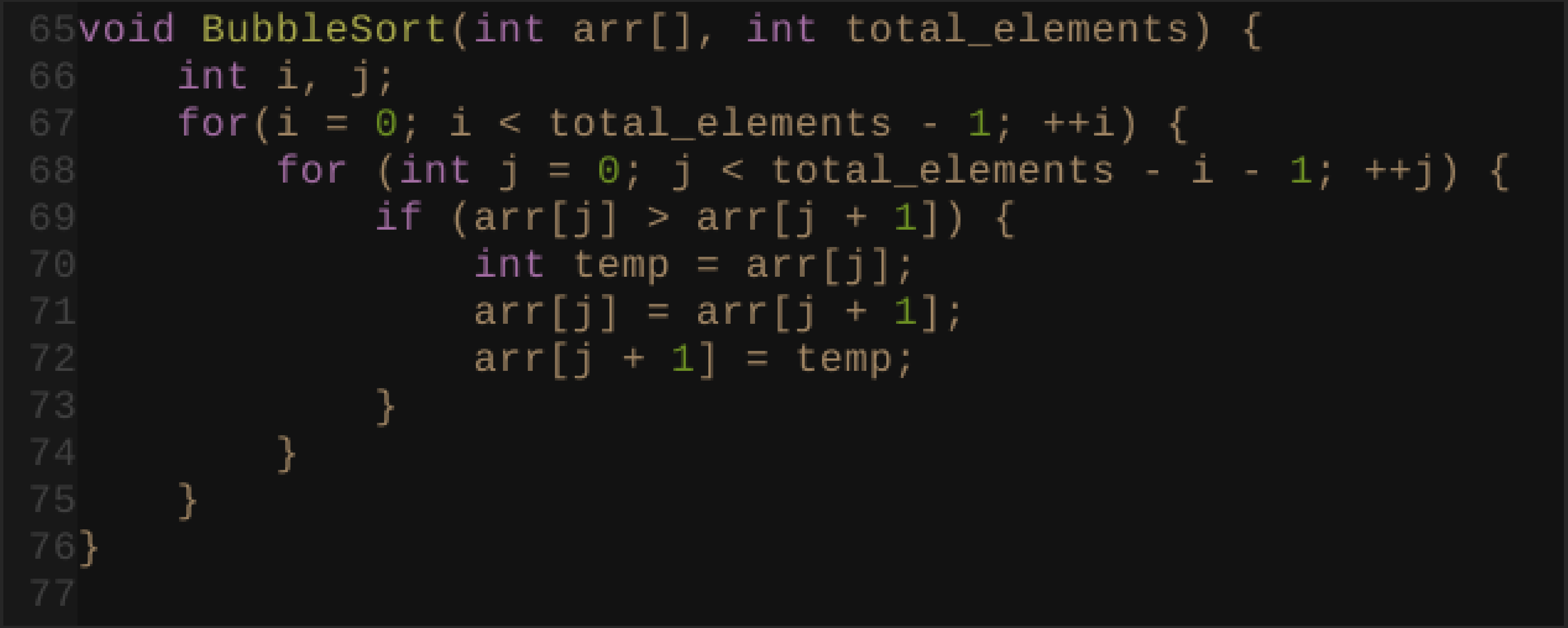
Thank you for your time ☺

Recall what a pointer is



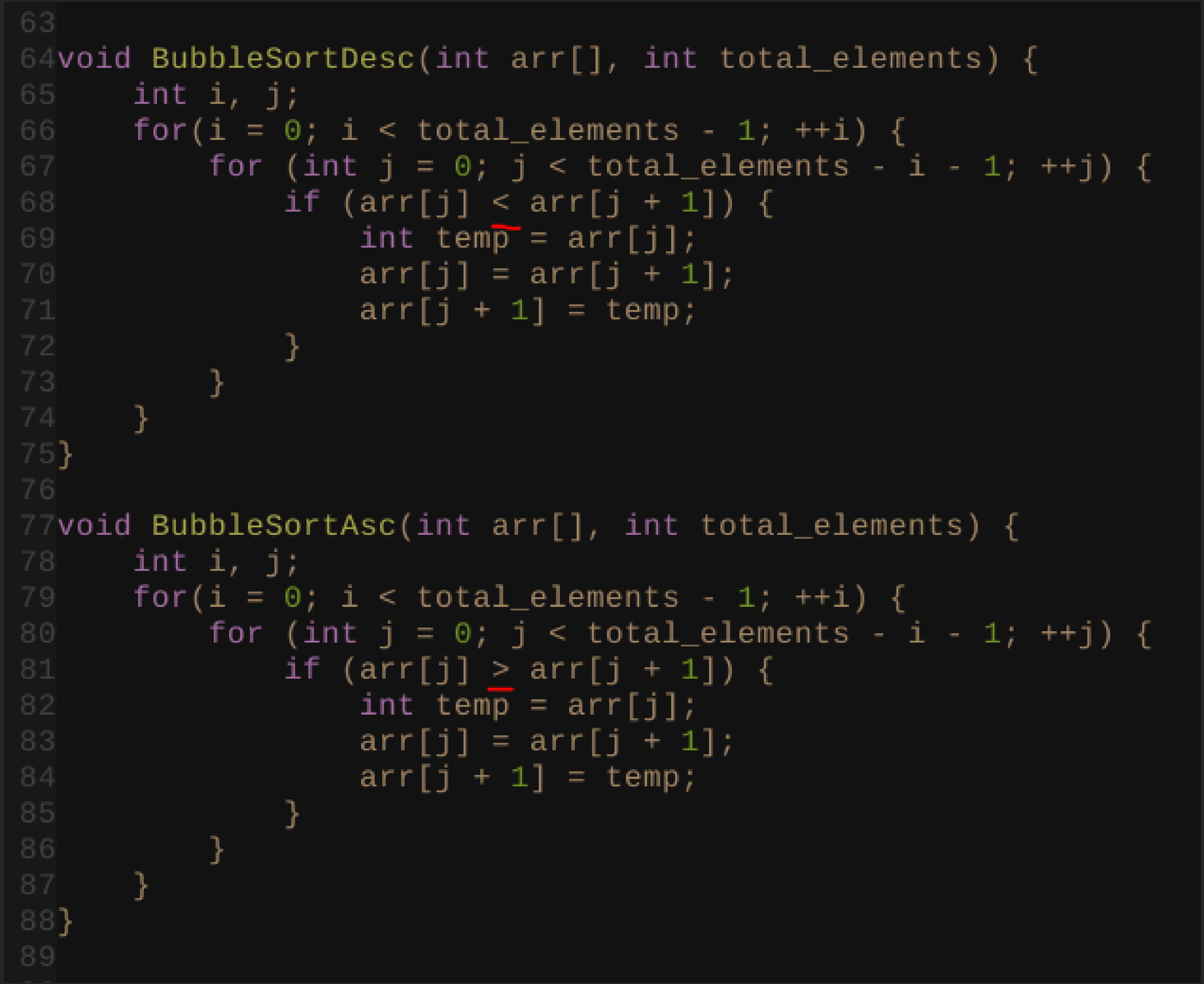
```
{
    // int pointer named `ip` that points to 0x00
    int* ip = 0;

    // function pointer name `fp`  that points to 0x00
    // it points to a function that takes in nothing
    // and return nothing.
    //
    // PS. Yeah the syntax is real annoying
    void (*fp)(void) = 0;
}
```

Case Study: Sort function



```
void BubbleSort(int arr[], int total_elements) {
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (int j = 0; j < total_elements - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

```

What if I want to sort in descending order?

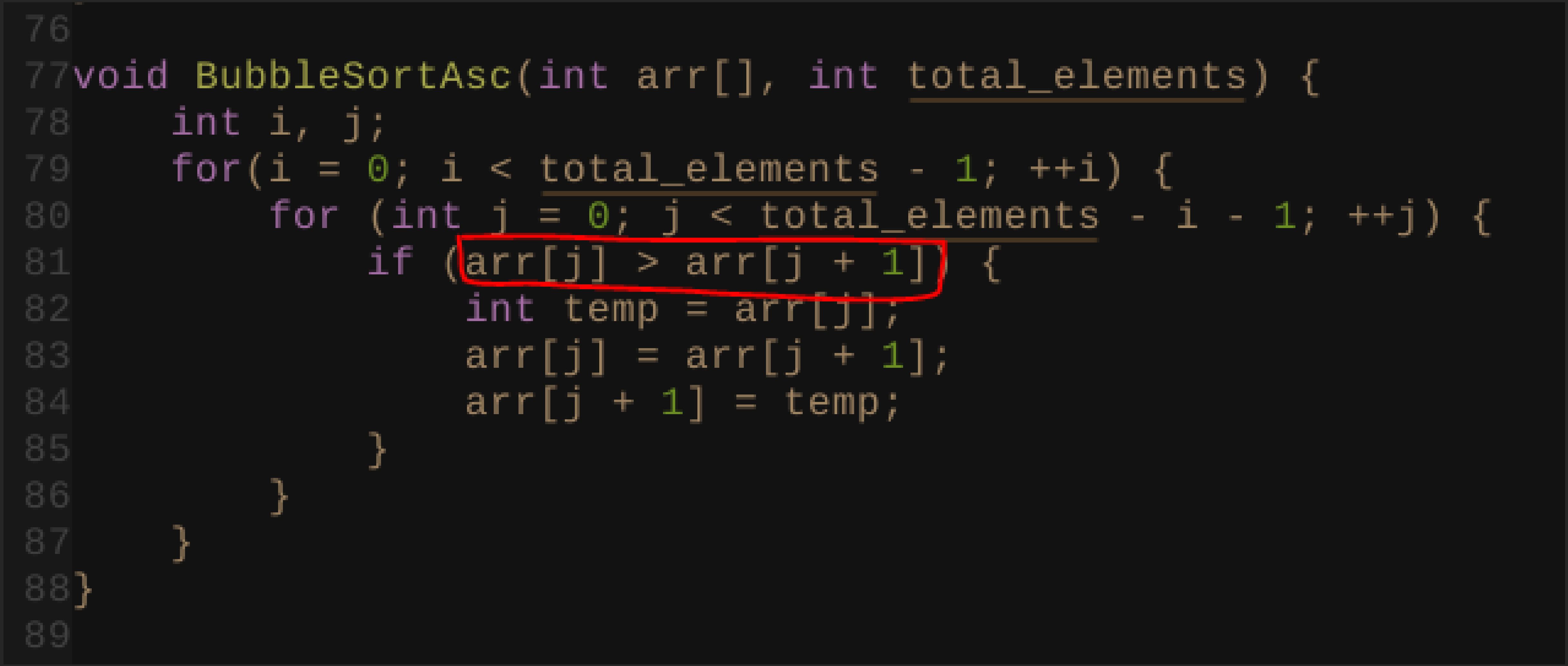


```
void BubbleSortDesc(int arr[], int total_elements) {
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (int j = 0; j < total_elements - i - 1; ++j) {
            if (arr[j] < arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void BubbleSortAsc(int arr[], int total_elements) {
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (int j = 0; j < total_elements - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

What if I want to sort in a different way which I will think of later?

SO YOU ARE TELLING ME
YOU WILL DISTURB ME EVERYTIME
YOU WANT A NEW WAY TO SORT?
imgflip.com

Realize that the ‘way to compare’ needs to be a variable

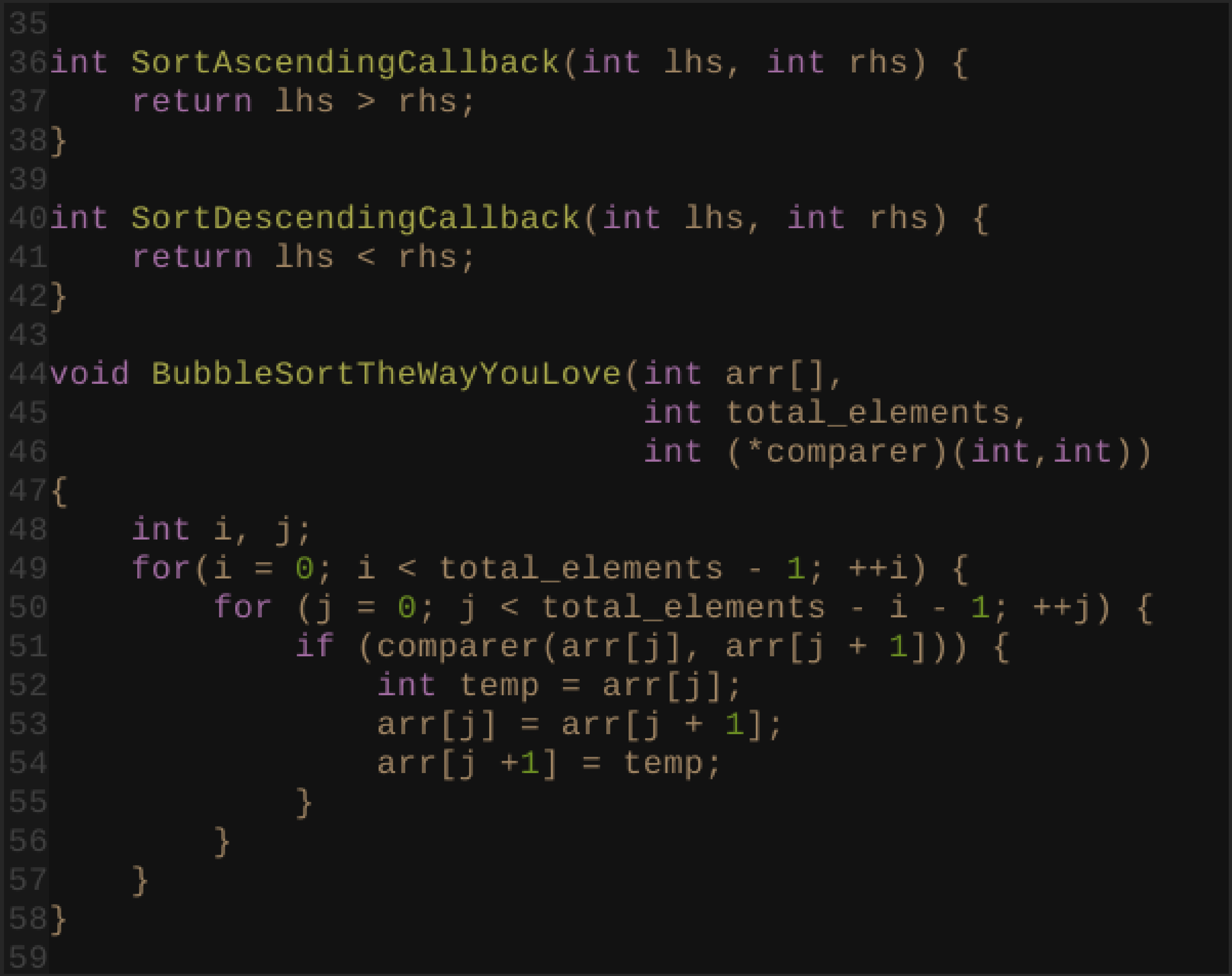


```
void BubbleSortAsc(int arr[], int total_elements) {
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (int j = 0; j < total_elements - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

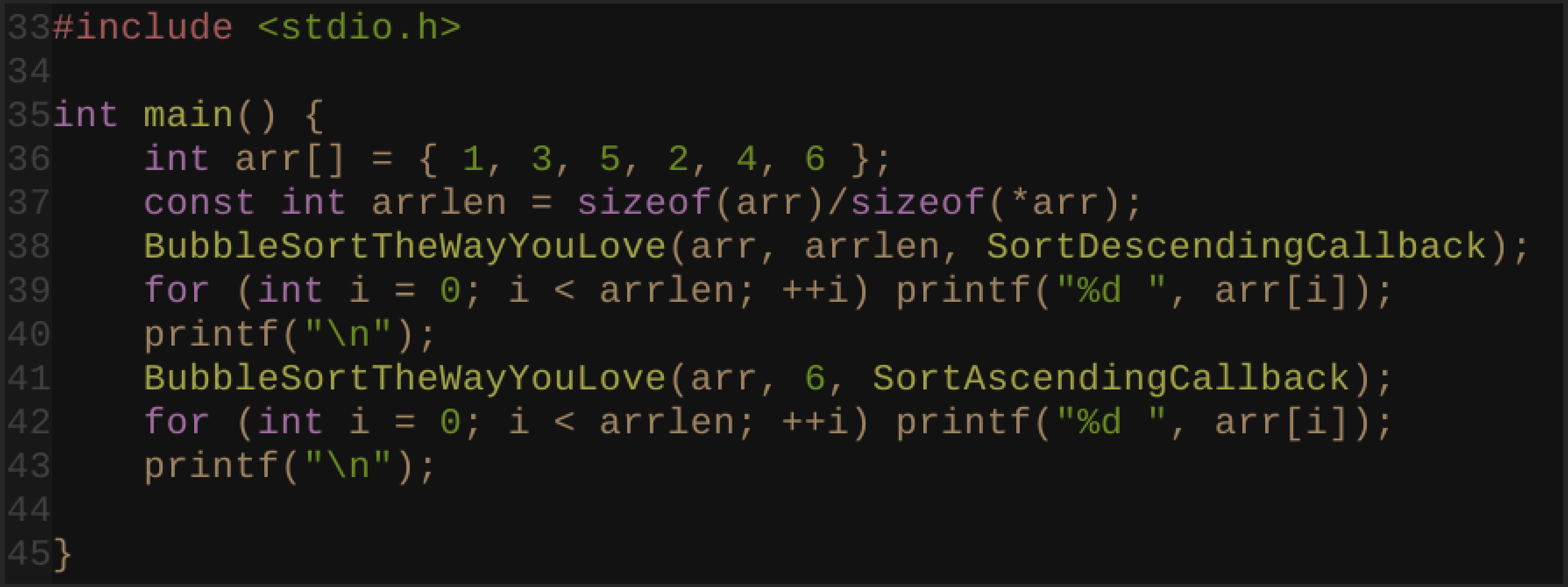
We need a variable that will help us compare two integer values and return a boolean value

But what is the type of such a variable?

A function?



```
int SortAscendingCallback(int lhs, int rhs) {
    return lhs > rhs;
}

int SortDescendingCallback(int lhs, int rhs) {
    return lhs < rhs;
}

void BubbleSortTheWayYouLove(int arr[],
                             int total_elements,
                             int (*comparer)(int,int))
{
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (j = 0; j < total_elements - i - 1; ++j) {
            if (comparer(arr[j], arr[j + 1])) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j +1] = temp;
            }
        }
    }
}

```

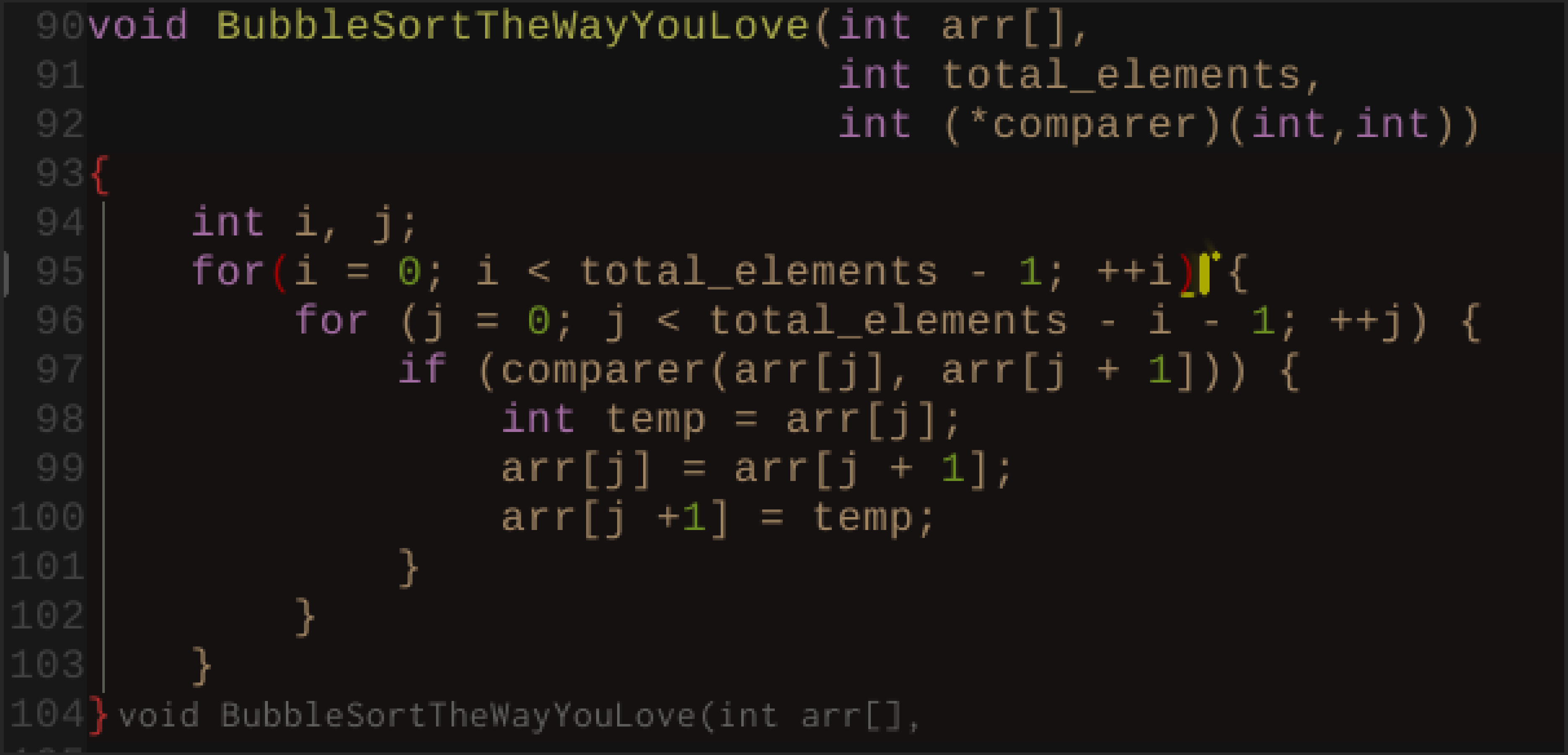


```c
#include <stdio.h>

int main() {
    int arr[] = { 1, 3, 5, 2, 4, 6 };
    const int arrlen = sizeof(arr)/sizeof(*arr);
    BubbleSortTheWayYouLove(arr, arrlen, SortDescendingCallback);
    for (int i = 0; i < arrlen; ++i) printf("%d ", arr[i]);
    printf("\n");
    BubbleSortTheWayYouLove(arr, 6, SortAscendingCallback);
    for (int i = 0; i < arrlen; ++i) printf("%d ", arr[i]);
    printf("\n");

}
```

Now, we are exposing the ‘way to compare’ as a variable for users in input!

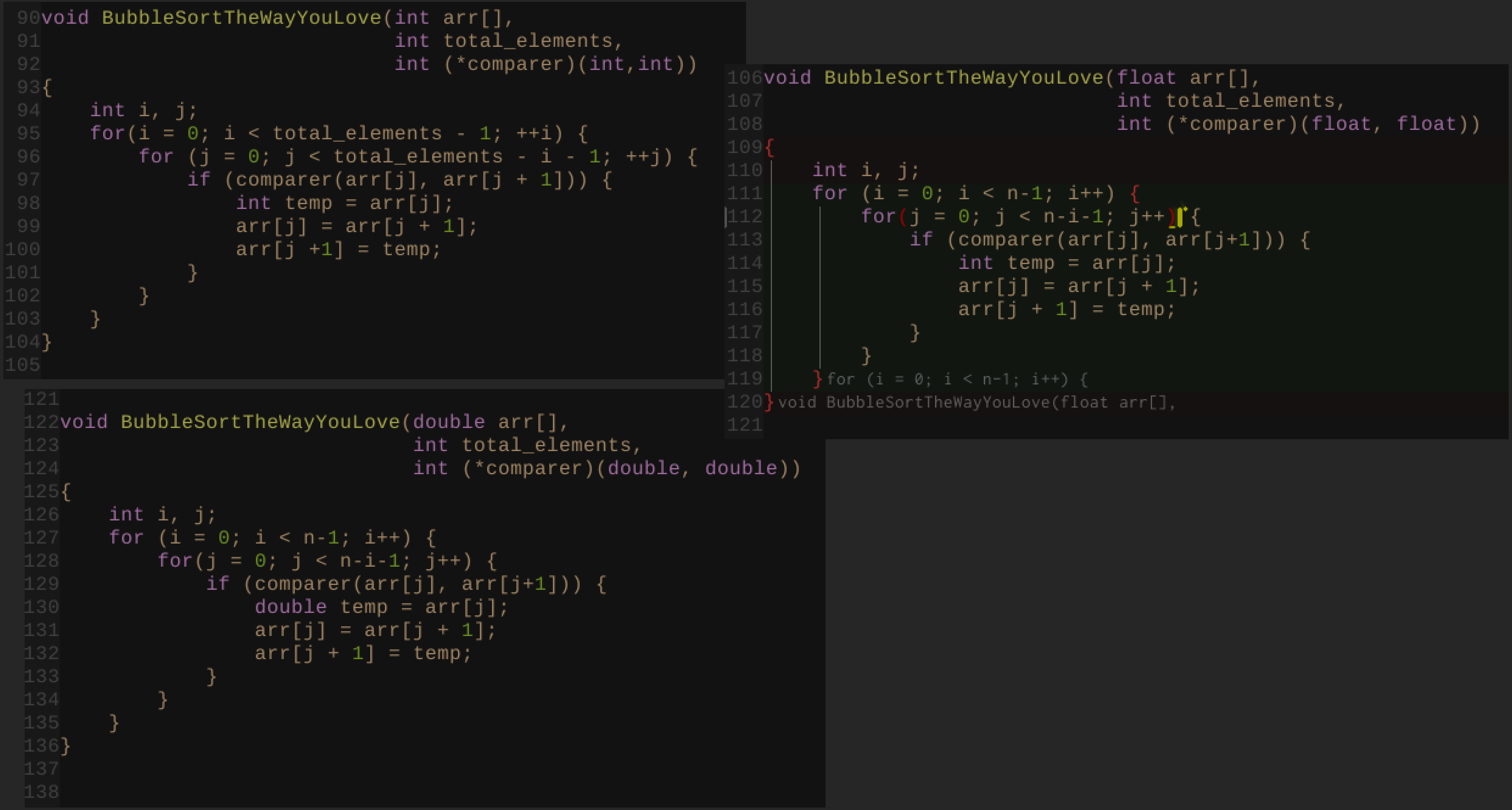
However, there is a problem.

Our code is restricted to int



```
void BubbleSortTheWayYouLove(int arr[],
                             int total_elements,
                             int (*comparer)(int,int))
{
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (j = 0; j < total_elements - i - 1; ++j) {
            if (comparer(arr[j], arr[j + 1])) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j +1] = temp;
            }
        }
    }
} void BubbleSortTheWayYouLove(int arr[],
```

Does this mean that for each type, I have to create a new function? O_o



```c
void BubbleSortTheWayYouLove(int arr[],
                             int total_elements,
                             int (*comparer)(int,int))
{
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (j = 0; j < total_elements - i - 1; ++j) {
            if (comparer(arr[j], arr[j + 1])) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j +1] = temp;
            }
        }
    }
}

```

```c
void BubbleSortTheWayYouLove(float arr[],
                             int total_elements,
                             int (*comparer)(float, float))
{
    int i, j;
    for (i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if (comparer(arr[j], arr[j+1])) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    } for (i = 0; i < n-1; i++) {
} void BubbleSortTheWayYouLove(float arr[],

```

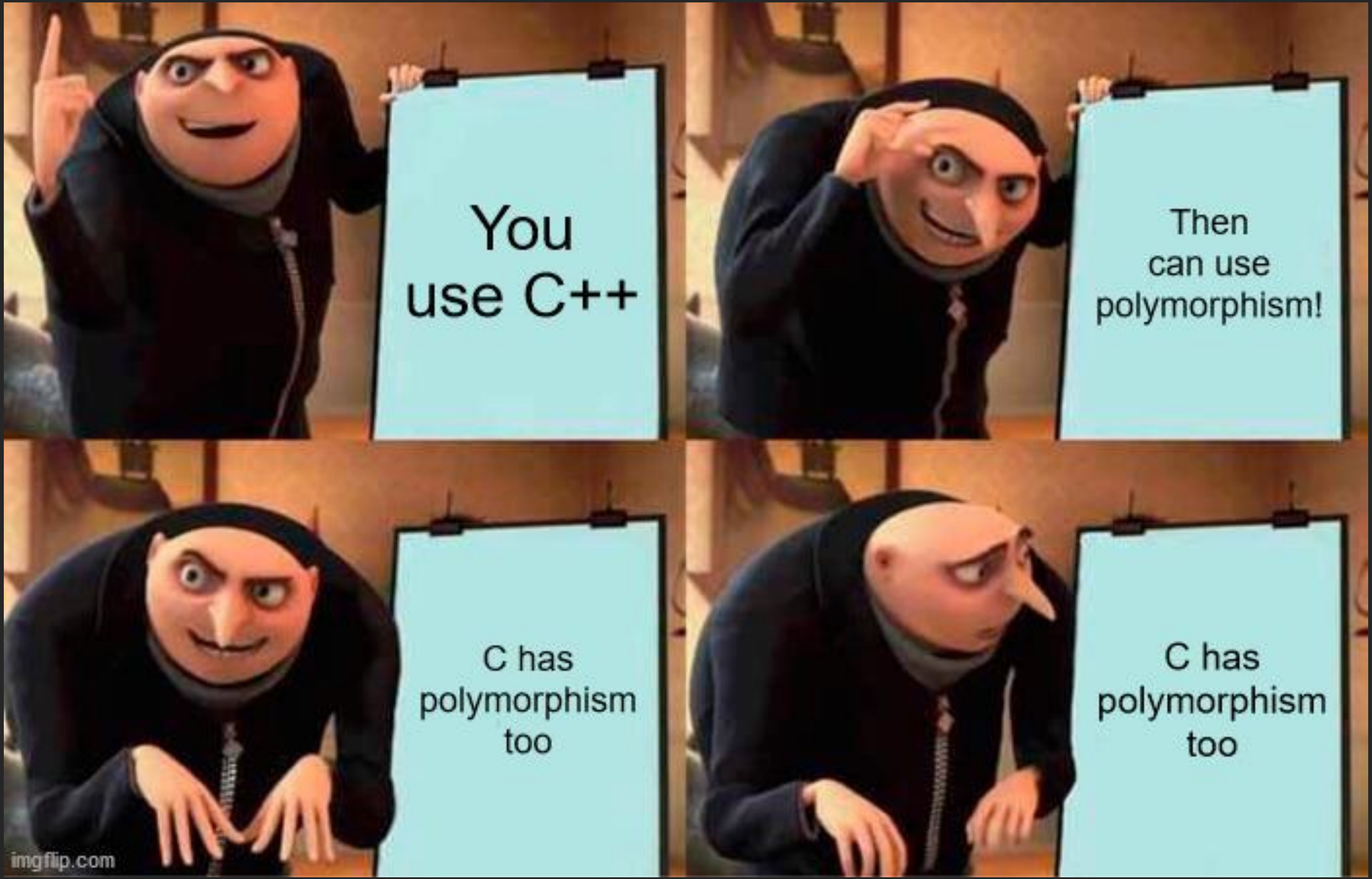
```c

void BubbleSortTheWayYouLove(double arr[],
                             int total_elements,
                             int (*comparer)(double, double))
{
    int i, j;
    for (i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if (comparer(arr[j], arr[j+1])) {
                double temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}


```

Is there a way to make it more

GENERIC?

C-Style ‘Generics’

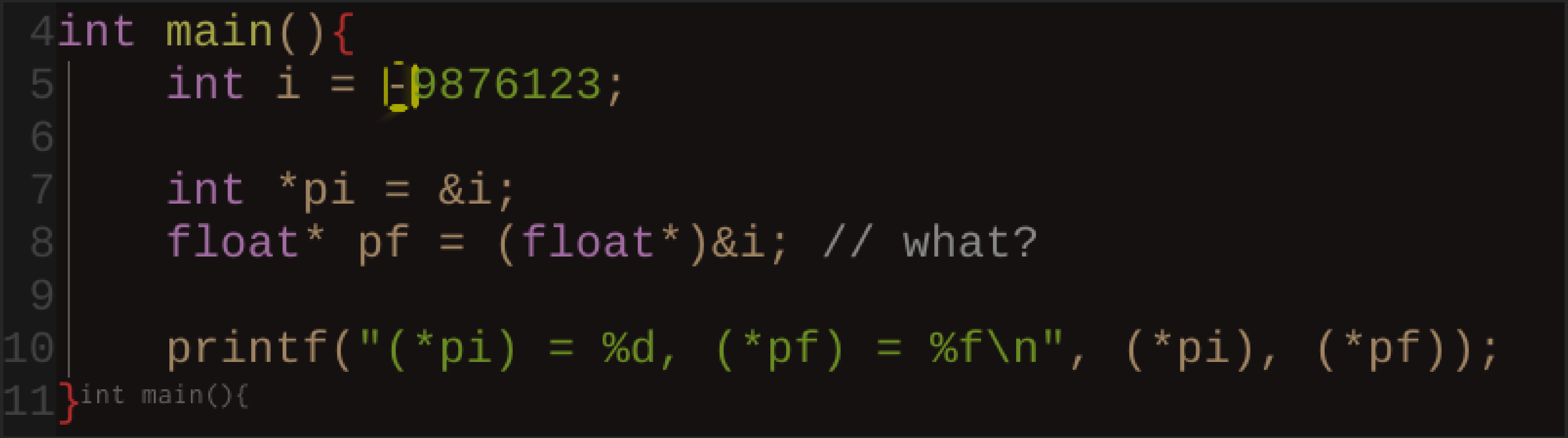
And ‘polymorphism’


You use C++
Then can use polymorphism!
C has polymorphism too
C has polymorphism too
imgflip.com

Polymorphism is a way for a type to be represented as other types.

In plain-C,

pointers are a way to do that

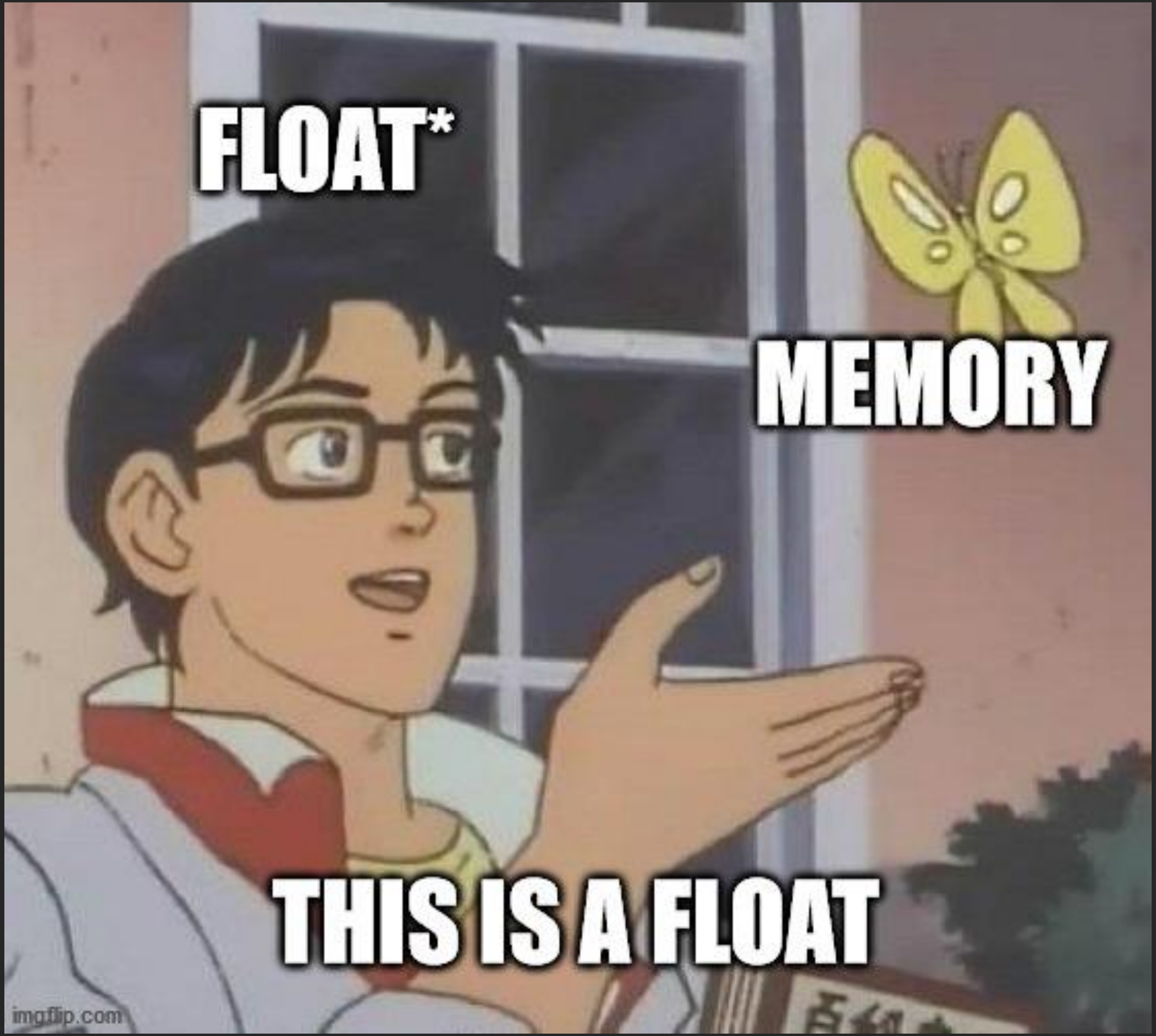


```c
int main(){
    int i = -9876123;

    int *pi = &i;
    float* pf = (float*)&i; // what?

    printf("(*pi) = %d, (*pf) = %f\n", (*pi), (*pf));
}int main(){
```

Disclaimer: okay that example is a little ‘dangerous’, but it’s to illustrate what pointers are and what you can do with them. Please don’t write code like that unless you REALLY know EXACTLY what you are doing ☺

This means that `i` can be read as an integer or a float (or anything really)

...Polymorphism?


FLOAT*
MEMORY
THIS IS A FLOAT
imgflip.com

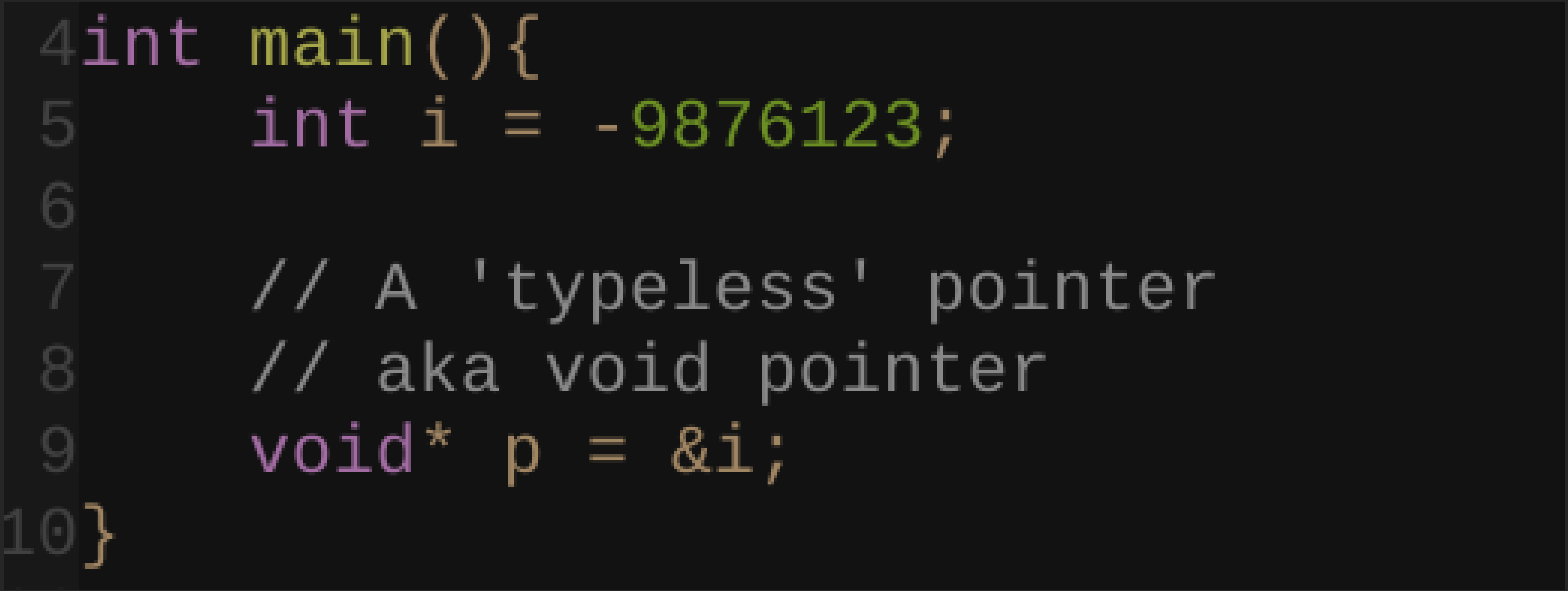
A pointer doesn’t really care about what it’s pointing to. It will just treat the memory it’s pointing to as a certain type.

(e.g. an int pointer will treat the memory it’s pointing at as an int. It doesn’t care if it REALLY is an int or not.)

Remember that in raw memory, the concept of ‘types’ does not exist

Speaking of types, there are also typeless pointers.

These simply hold an address.



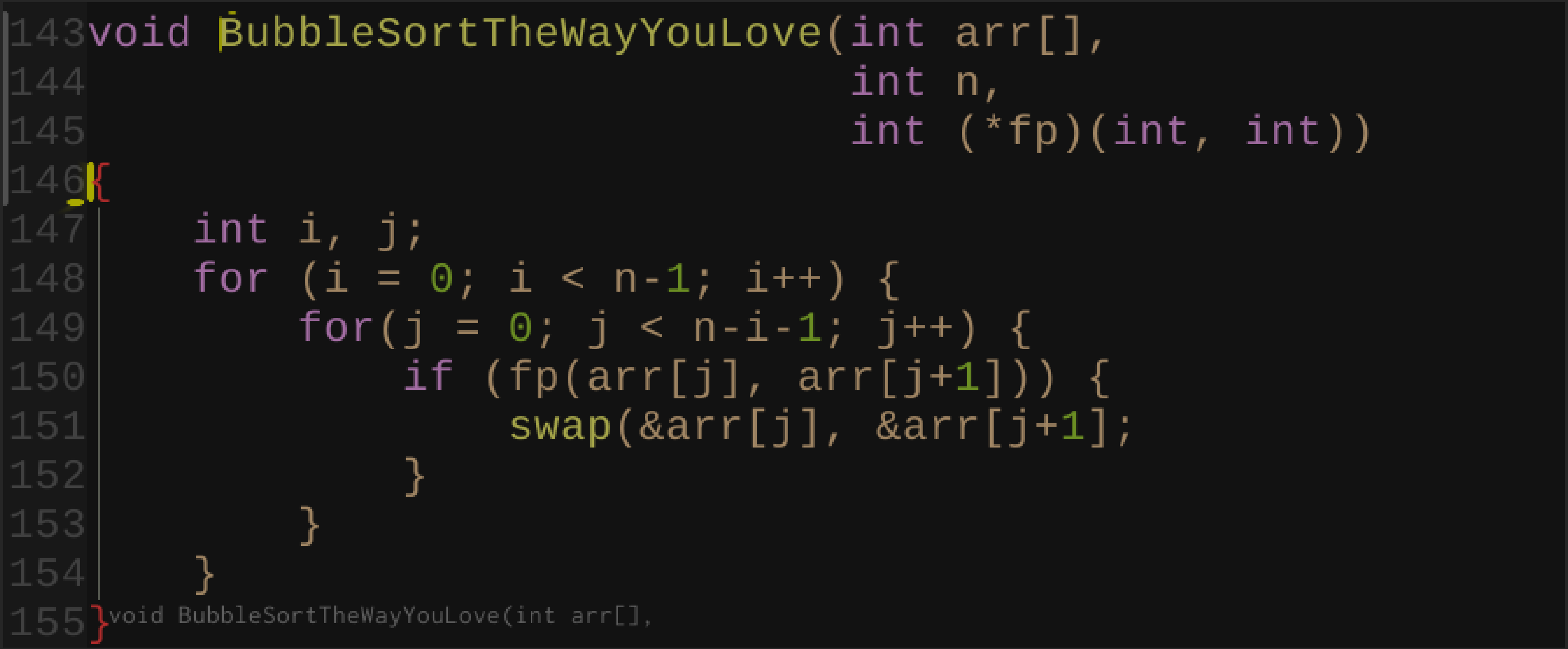
```c
int main(){
    int i = -9876123;

    // A 'typeless' pointer
    // aka void pointer
    void* p = &i;
}
```

Since they have no type
(and void is not a real type),
you can’t dereference them.

NOT SURE IF I'M JUST LOST
OR I JUST DON'T
GET WHERE THIS IS GOING
imgflip.com

The final ingredient:
Casting between pointer types.

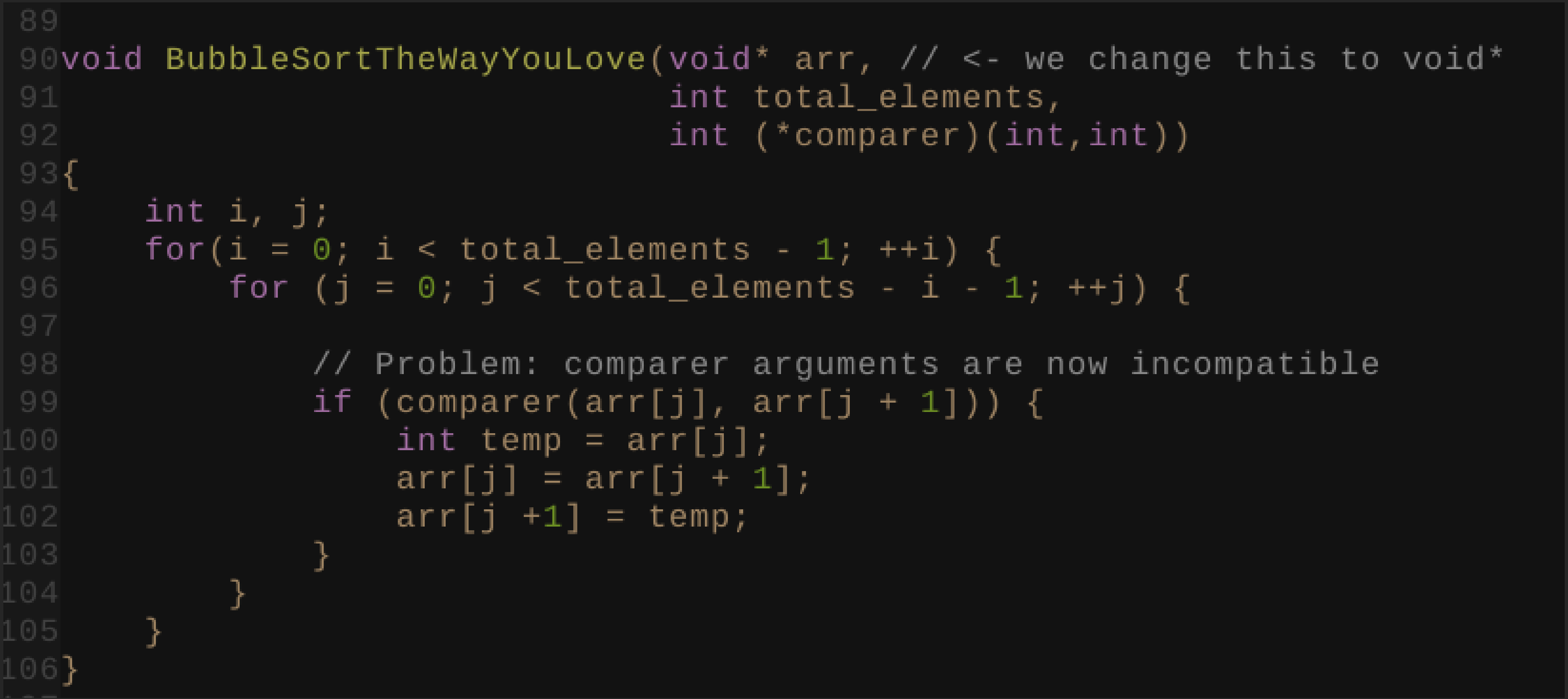
Back to our example



```c
void BubbleSortTheWayYouLove(int arr[],
                             int n,
                             int (*fp)(int, int))
{
    int i, j;
    for (i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if (fp(arr[j], arr[j+1])) {
                swap(&arr[j], &arr[j+1];
            }
        }
    }
}void BubbleSortTheWayYouLove(int arr[],
```

Instead of taking in an array of int, we want to take an array of ANY type.

Since arrays degenerate into pointers when passed to a function anyway, we simply need the address.



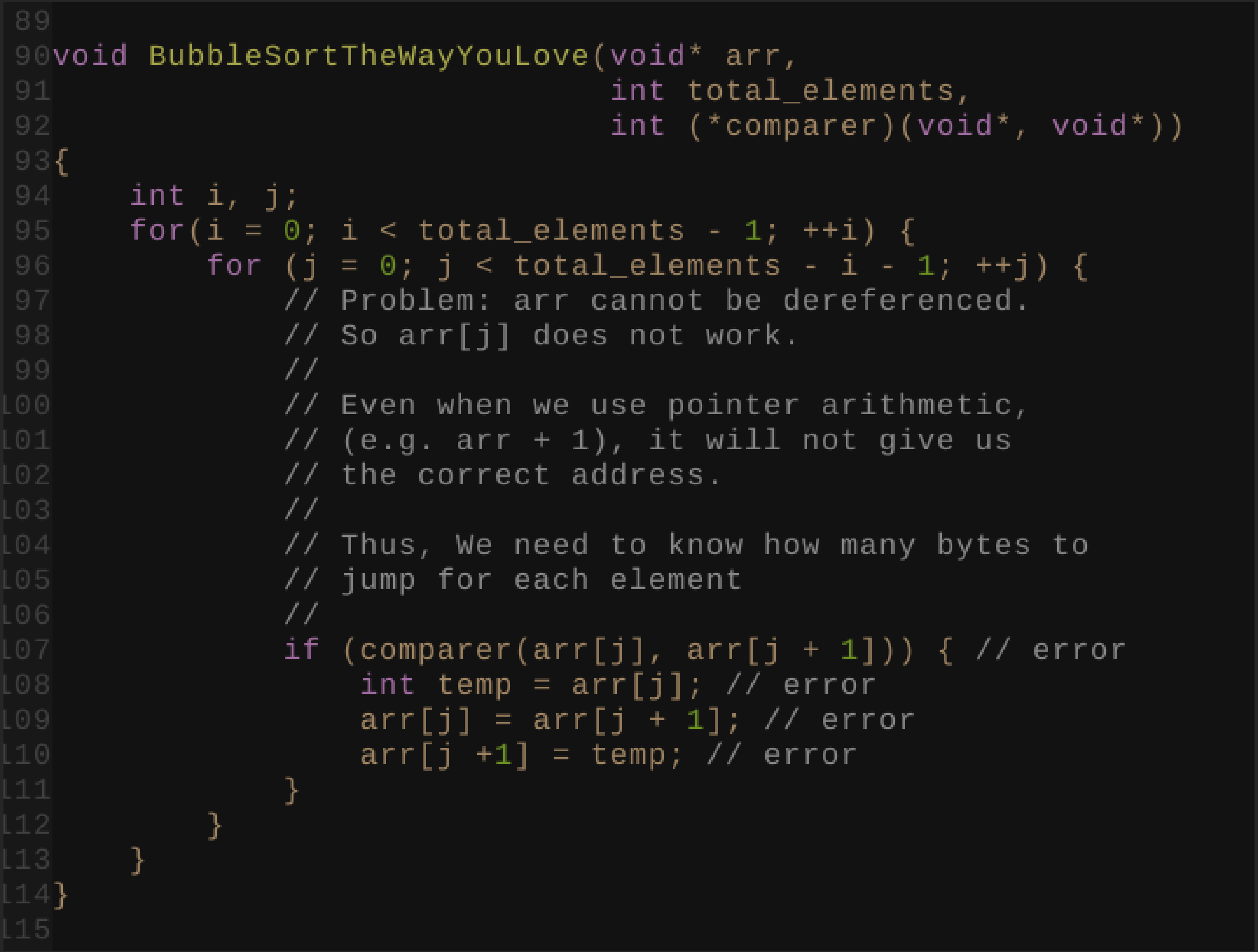
```c
void BubbleSortTheWayYouLove(void* arr, // <- we change this to void*
                             int total_elements,
                             int (*comparer)(int,int))
{
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (j = 0; j < total_elements - i - 1; ++j) {

            // Problem: comparer arguments are now incompatible
            if (comparer(arr[j], arr[j + 1])) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j +1] = temp;
            }
        }
    }
}
```

Problem

The comparer function pointer is takes in integers. But we don’t know whether we are dealing with int anymore.

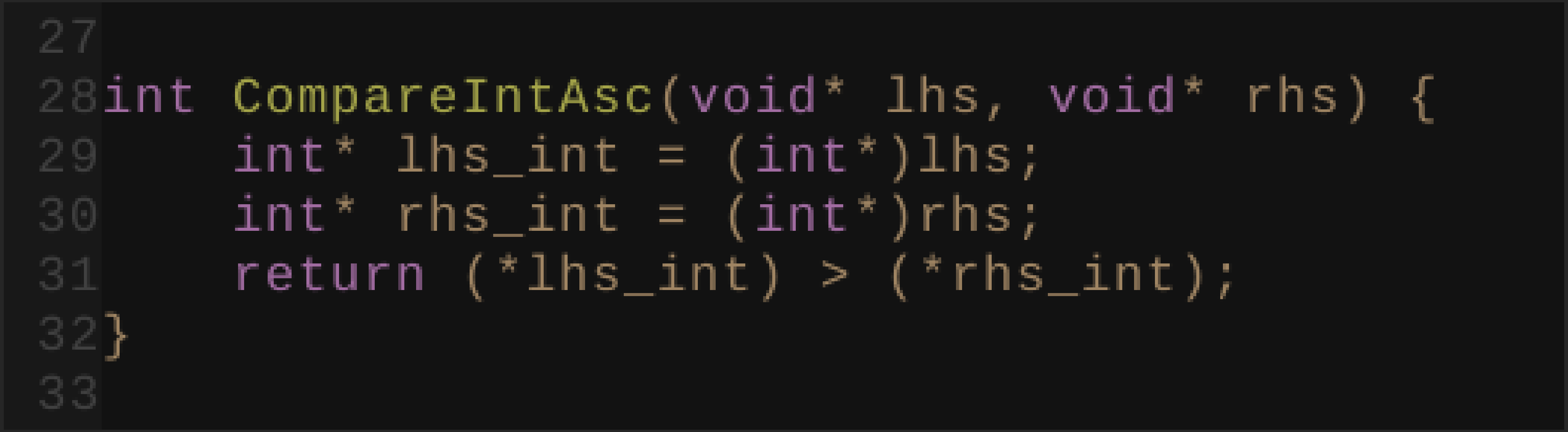
Solution

Only the users knows what type the arguments should be, so we let them decide. We will pass the address to the memory, and they will cast it to the type they need.



```c

void BubbleSortTheWayYouLove(void* arr,
                             int total_elements,
                             int (*comparer)(void*, void*))
{
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (j = 0; j < total_elements - i - 1; ++j) {
            // Problem: arr cannot be dereferenced.
            // So arr[j] does not work.
            //
            // Even when we use pointer arithmetic,
            // (e.g. arr + 1), it will not give us
            // the correct address.
            //
            // Thus, We need to know how many bytes to
            // jump for each element
            //
            if (comparer(arr[j], arr[j + 1])) { // error
                int temp = arr[j]; // error
                arr[j] = arr[j + 1]; // error
                arr[j +1] = temp; // error
            }
        }
    }
}

```

The comparer will then need to adjust taking in the typeless memory address.



```

int CompareIntAsc(void* lhs, void* rhs) {
    int* lhs_int = (int*)lhs;
    int* rhs_int = (int*)rhs;
    return (*lhs_int) > (*rhs_int);
}

```

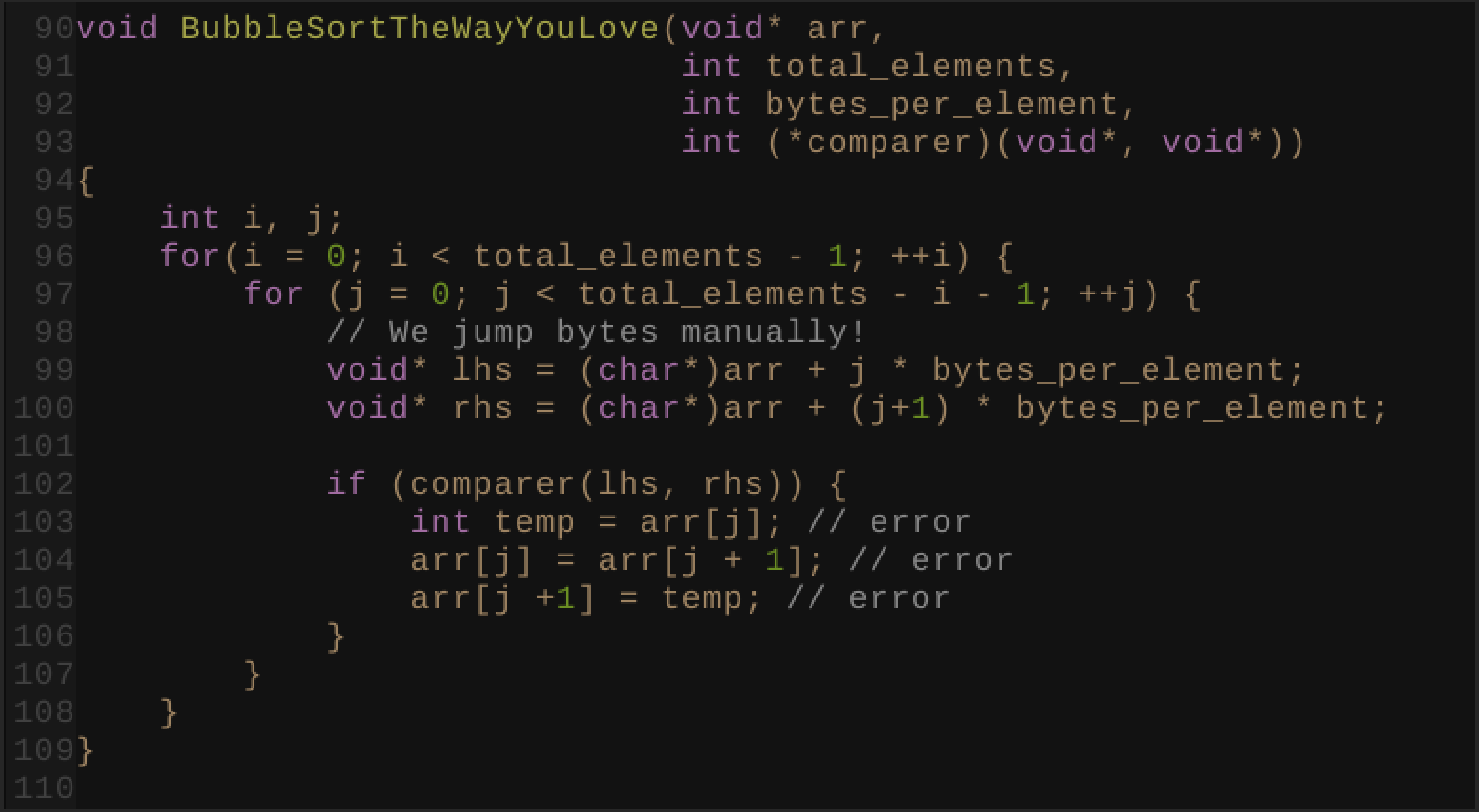
Problem

Since we lost the type information, we don’t know how much bytes to ‘jump’ for each element in the array.

Solution

No choice, we ask the user for the bytes to jump for each element.

Then, to traverse the array, we manually jump that amount of bytes



```
void BubbleSortTheWayYouLove(void* arr,
                             int total_elements,
                             int bytes_per_element,
                             int (*comparer)(void*, void*))
{
    int i, j;
    for(i = 0; i < total_elements - 1; ++i) {
        for (j = 0; j < total_elements - i - 1; ++j) {
            // We jump bytes manually!
            void* lhs = (char*)arr + j * bytes_per_element;
            void* rhs = (char*)arr + (j+1) * bytes_per_element;

            if (comparer(lhs, rhs)) {
                int temp = arr[j]; // error
                arr[j] = arr[j + 1]; // error
                arr[j +1] = temp; // error
            }
        }
    }
}

```

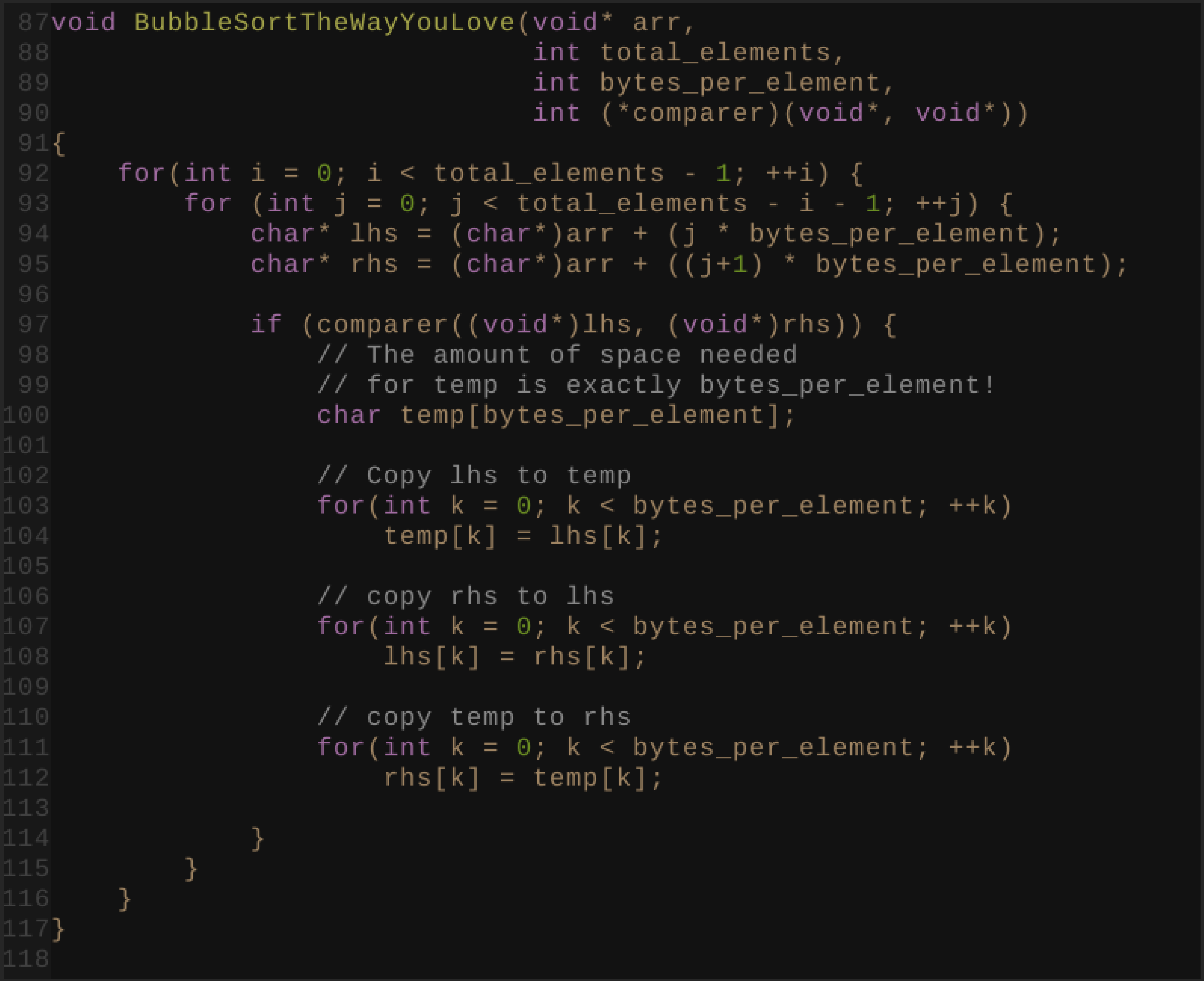
Problem

The swapping algorithm does not work anymore because it used to rely on type

(using the [] and = operator)

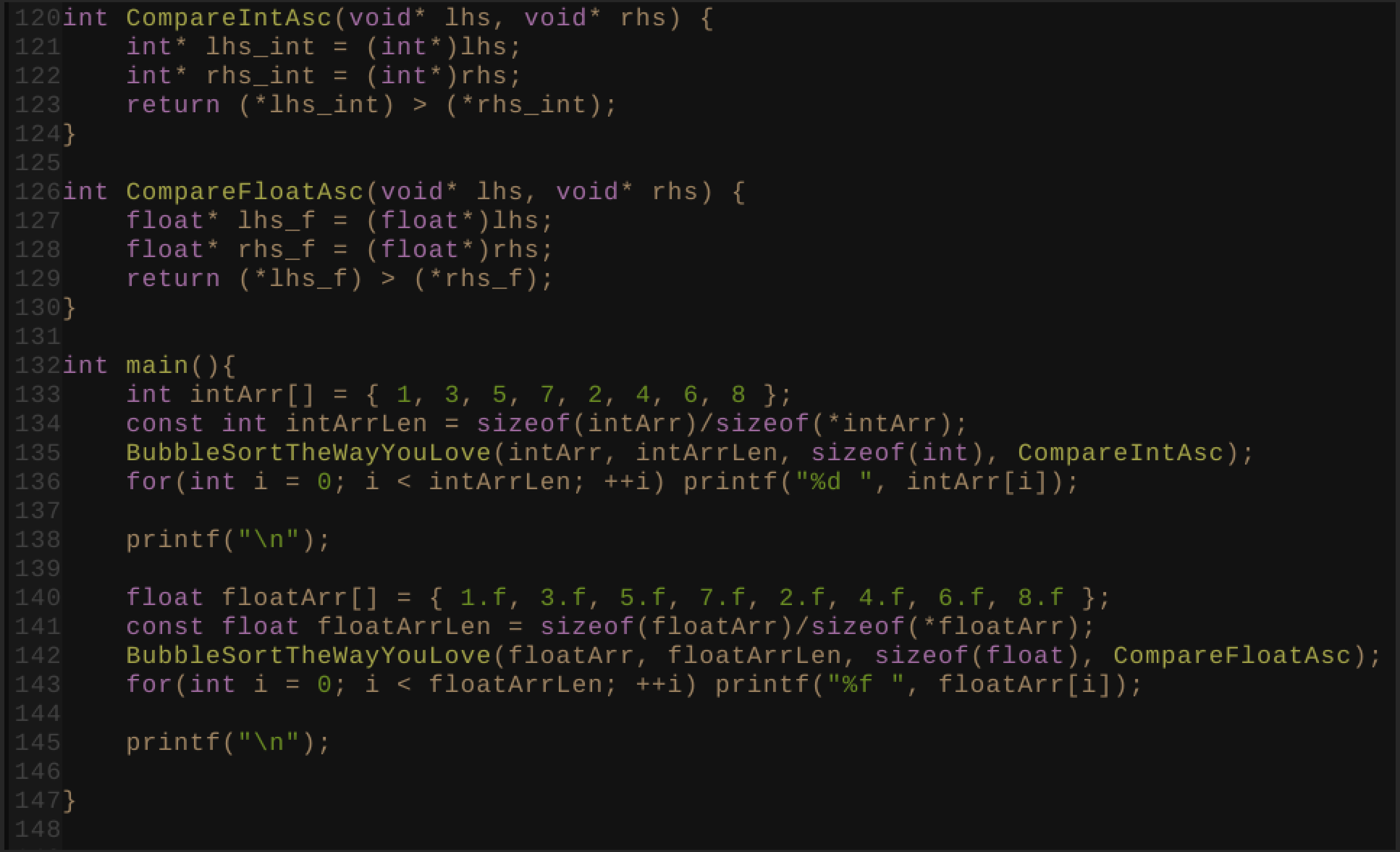
Solution

Manually swap chunks of bytes ourselves.

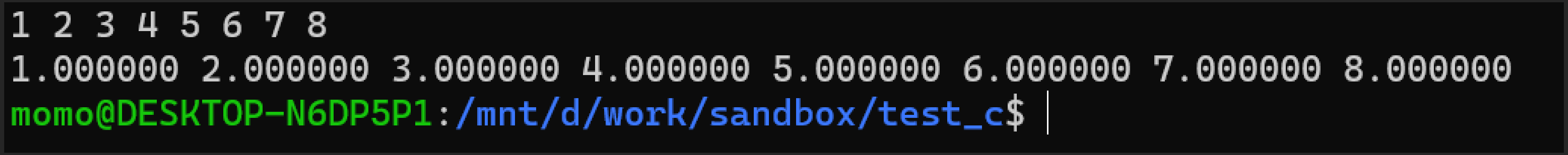


```c
void BubbleSortTheWayYouLove(void* arr,
                             int total_elements,
                             int bytes_per_element,
                             int (*comparer)(void*, void*))
{
    for(int i = 0; i < total_elements - 1; ++i) {
        for (int j = 0; j < total_elements - i - 1; ++j) {
            char* lhs = (char*)arr + (j * bytes_per_element);
            char* rhs = (char*)arr + ((j+1) * bytes_per_element);

            if (comparer((void*)lhs, (void*)rhs)) {
                // The amount of space needed
                // for temp is exactly bytes_per_element!
                char temp[bytes_per_element];

                // Copy lhs to temp
                for(int k = 0; k < bytes_per_element; ++k)
                    temp[k] = lhs[k];

                // copy rhs to lhs
                for(int k = 0; k < bytes_per_element; ++k)
                    lhs[k] = rhs[k];

                // copy temp to rhs
                for(int k = 0; k < bytes_per_element; ++k)
                    rhs[k] = temp[k];

            }
        }
    }
}

```

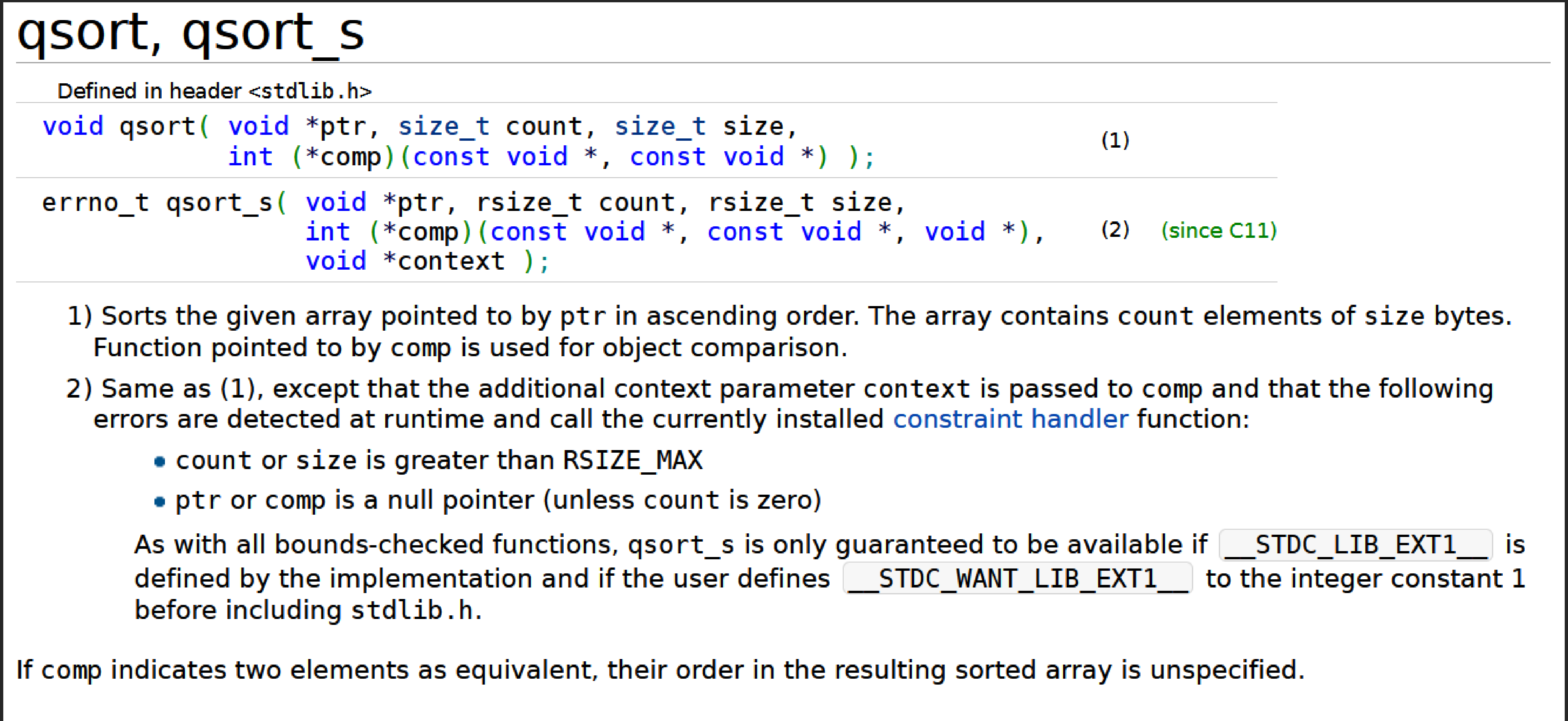
We have liftoff!



```c
int CompareIntAsc(void* lhs, void* rhs) {
    int* lhs_int = (int*)lhs;
    int* rhs_int = (int*)rhs;
    return (*lhs_int) > (*rhs_int);
}

int CompareFloatAsc(void* lhs, void* rhs) {
    float* lhs_f = (float*)lhs;
    float* rhs_f = (float*)rhs;
    return (*lhs_f) > (*rhs_f);
}

int main(){
    int intArr[] = { 1, 3, 5, 7, 2, 4, 6, 8 };
    const int intArrLen = sizeof(intArr)/sizeof(*intArr);
    BubbleSortTheWayYouLove(intArr, intArrLen, sizeof(int), CompareIntAsc);
    for(int i = 0; i < intArrLen; ++i) printf("%d ", intArr[i]);

    printf("\n");

    float floatArr[] = { 1.f, 3.f, 5.f, 7.f, 2.f, 4.f, 6.f, 8.f };
    const float floatArrLen = sizeof(floatArr)/sizeof(*floatArr);
    BubbleSortTheWayYouLove(floatArr, floatArrLen, sizeof(float), CompareFloatAsc);
    for(int i = 0; i < floatArrLen; ++i) printf("%f ", floatArr[i]);

    printf("\n");

}

```



```
1 2 3 4 5 6 7 8
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
momo@DESKTOP-N6DP5P1:/mnt/d/work/sandbox/test_c$
```



qsort, qsort_s

Defined in header <stdlib.h>

```
void qsort( void *ptr, size_t count, size_t size,
            int (*comp)(const void *, const void *) );                  (1)
```

```
errno_t qsort_s( void *ptr, rsize_t count, rsize_t size,
                 int (*comp)(const void *, const void *, void *),       (2)   (since C11)
                 void *context );
```

1) Sorts the given array pointed to by ptr in ascending order. The array contains count elements of size bytes. Function pointed to by comp is used for object comparison.

2) Same as (1), except that the additional context parameter context is passed to comp and that the following errors are detected at runtime and call the currently installed constraint handler function:

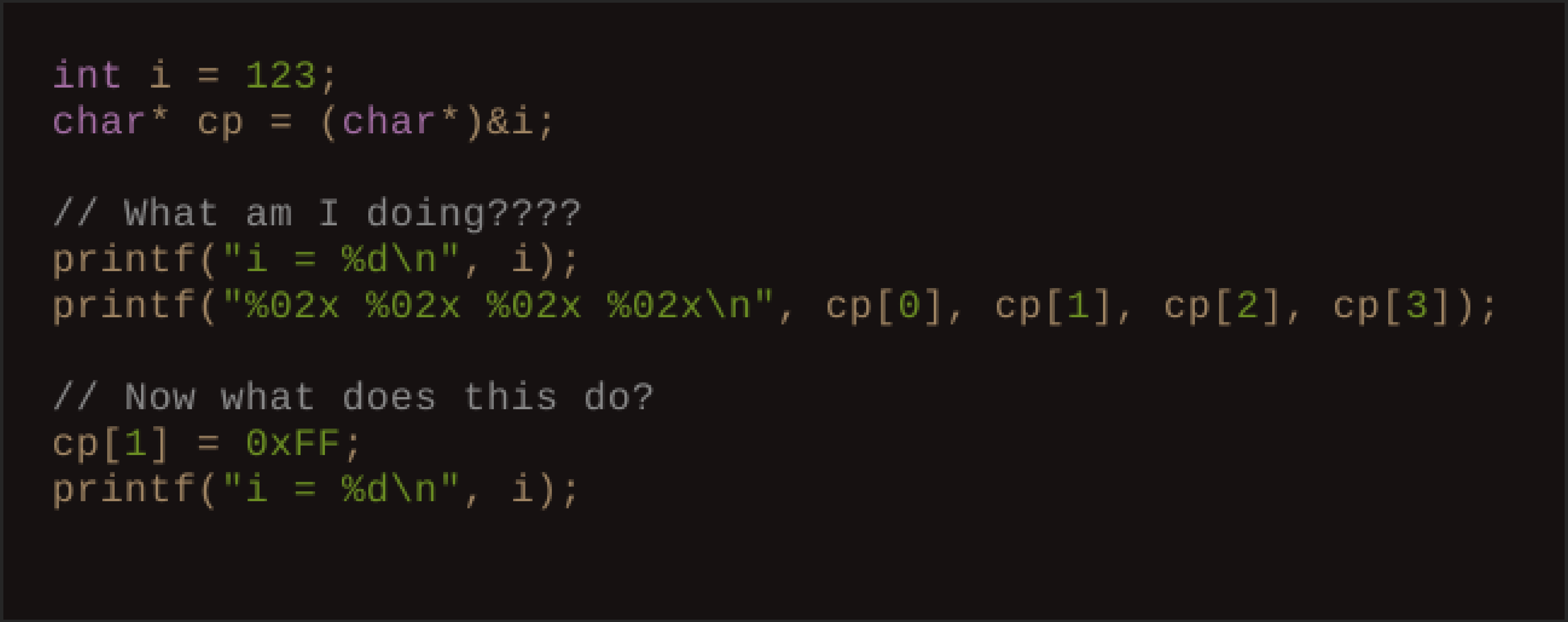
- count or size is greater than RSIZE_MAX
- ptr or comp is a null pointer (unless count is zero)

As with all bounds-checked functions, qsort_s is only guaranteed to be available if __STDC_LIB_EXT1__ is defined by the implementation and if the user defines __STDC_WANT_LIB_EXT1__ to the integer constant 1 before including stdlib.h.

If comp indicates two elements as equivalent, their order in the resulting sorted array is unspecified.

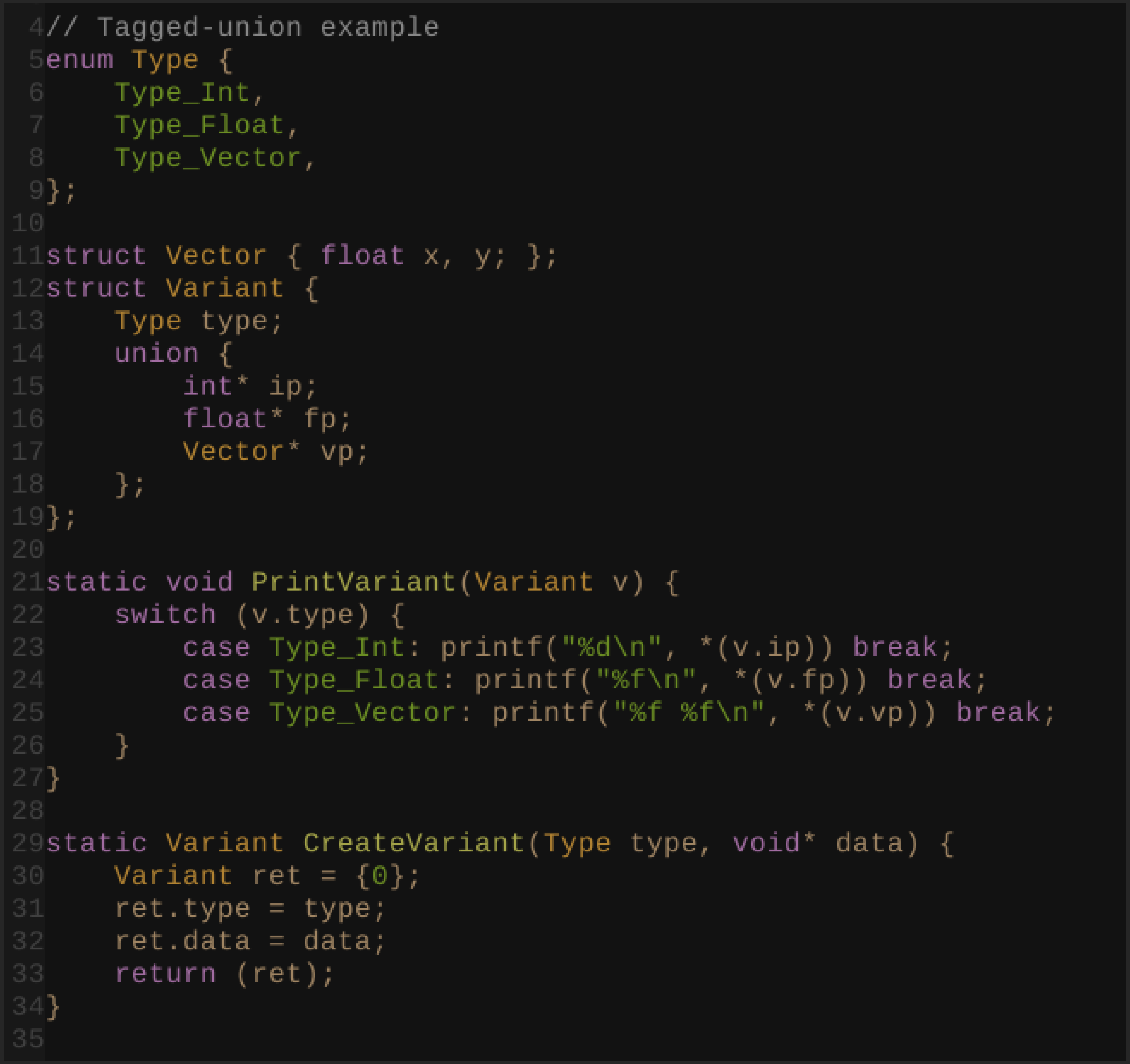
Bonus

What does this do?

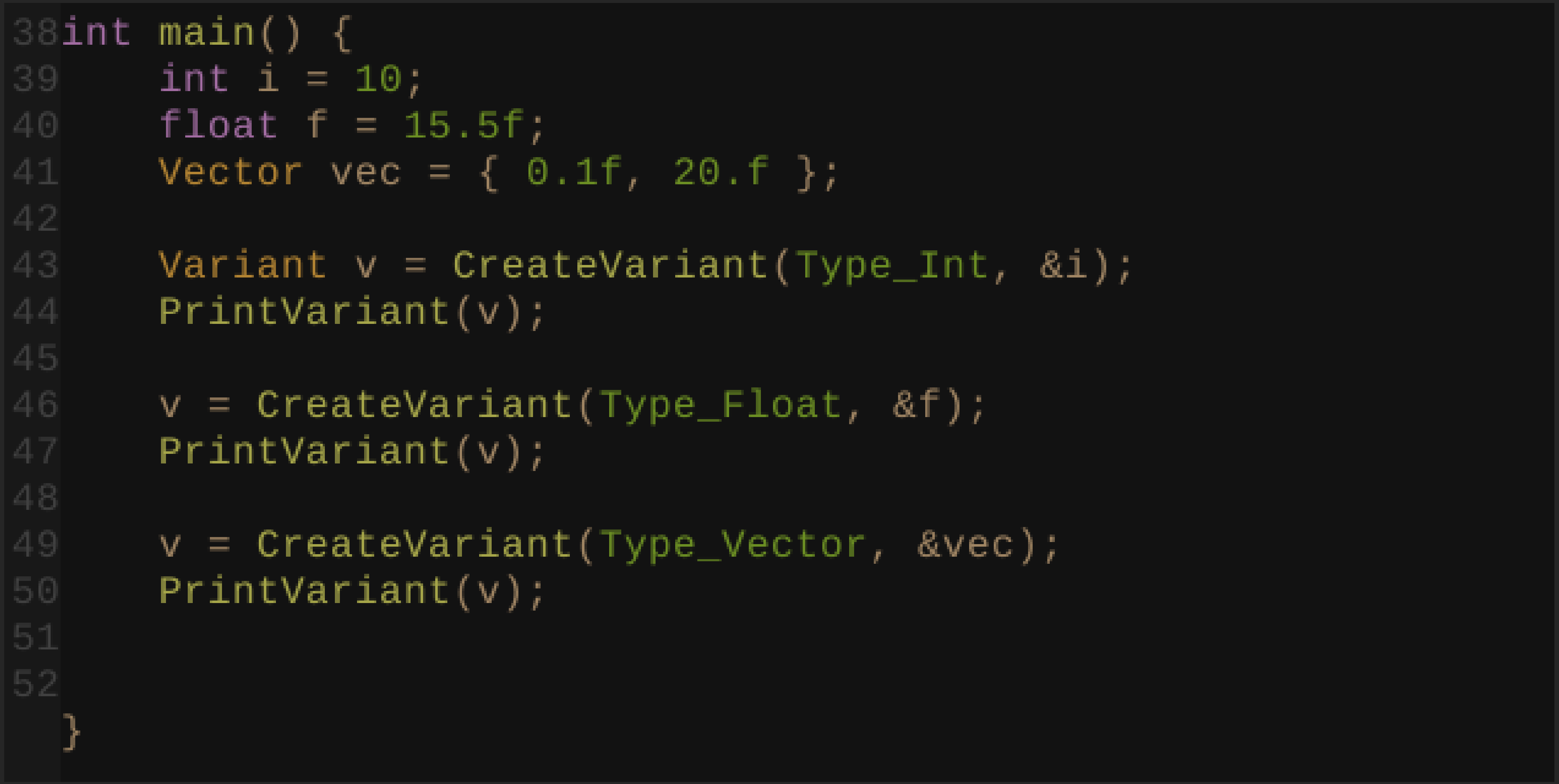


```
int i = 123;
char* cp = (char*)&i;

// What am I doing????
printf("i = %d\n", i);
printf("%02x %02x %02x %02x\n", cp[0], cp[1], cp[2], cp[3]);

// Now what does this do?
cp[1] = 0xFF;
printf("i = %d\n", i);
```

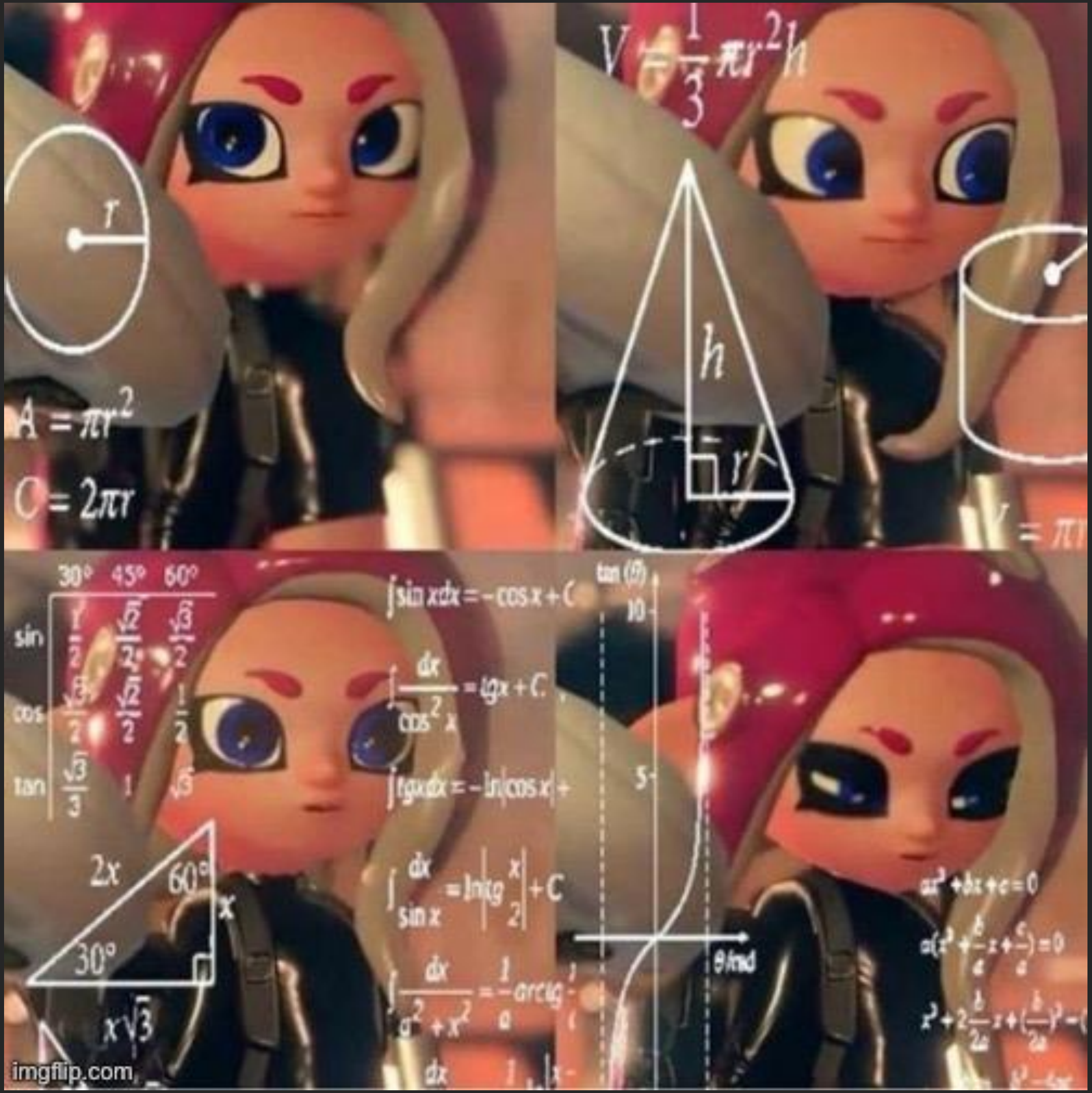
Tagged Unions



```
// Tagged-union example
enum Type {
    Type_Int,
    Type_Float,
    Type_Vector,
};

struct Vector { float x, y; };
struct Variant {
    Type type;
    union {
        int* ip;
        float* fp;
        Vector* vp;
    };
};

static void PrintVariant(Variant v) {
    switch (v.type) {
        case Type_Int: printf("%d\n", *(v.ip)) break;
        case Type_Float: printf("%f\n", *(v.fp)) break;
        case Type_Vector: printf("%f %f\n", *(v.vp)) break;
    }
}

static Variant CreateVariant(Type type, void* data) {
    Variant ret = {0};
    ret.type = type;
    ret.data = data;
    return (ret);
}

```



```
int main() {
    int i = 10;
    float f = 15.5f;
    Vector vec = { 0.1f, 20.f };

    Variant v = CreateVariant(Type_Int, &i);
    PrintVariant(v);

    v = CreateVariant(Type_Float, &f);
    PrintVariant(v);

    v = CreateVariant(Type_Vector, &vec);
    PrintVariant(v);


}
```


$V = \frac{1}{3}\pi r^2 h$
r
$A = \pi r^2$
$C = 2\pi r$
h
r
$V = \pi r$
30° 45° 60°
sin $\frac{1}{2}$ $\frac{\sqrt{2}}{2}$ $\frac{\sqrt{3}}{2}$
cos $\frac{\sqrt{3}}{2}$ $\frac{\sqrt{2}}{2}$ $\frac{1}{2}$
tan $\frac{\sqrt{3}}{3}$ 1 $\sqrt{3}$
2x
60°
x
30°
$x\sqrt{3}$
$\int \sin x dx = -\cos x + C$
$\int \frac{dx}{\cos^2 x} = tg x + C$
$\int tg x dx = -\ln|\cos x| +$
$\int \frac{dx}{\sin x} = \ln\left|tg \frac{x}{2}\right| + C$
$\int \frac{dx}{a^2 + x^2} = \frac{1}{a} arctg$
dx
tan (θ)
10
5
θ (rad)
$ax^2 + bx + c = 0$
$a(x^2 + \frac{b}{a}x + \frac{c}{a}) = 0$
$x^2 + 2\frac{b}{2a}x + (\frac{b}{2a})^2 -$
imgflip.com

Conclusions?

- Pointers are not complicated. It's what you do with them that might be complicated.
- We can pass functions around just like variables, to add customizability and scalability to another function.
- There is no magic.