CS107, Lecture 9
C Generics – Function Pointers

Reading: K&R 5.11
Generics So Far

• `void *` is a variable type that represents a generic pointer “to something”.
• We cannot perform pointer arithmetic with or dereference (without casting first) a `void *`.
• We can use `memcpy` or `memmove` to copy data from one memory location to another.
• To do pointer arithmetic with a `void *`, we must first cast it to a `char *`.
• `void *` and generics are powerful but dangerous because of the lack of type checking, so we must be extra careful when working with generic memory.
We can use `void *` to represent a pointer to any data, and `memcpy/memmove` to copy arbitrary bytes.
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}

We can cast to a char * in order to perform manual byte arithmetic with void * pointers.
memset is a function that sets a specified amount of bytes at one address to a certain value.

```c
void *memset(void *s, int c, size_t n);
```

It fills $n$ bytes starting at memory location $s$ with the byte $c$. (It also returns $s$).

```c
int counts[5];
memset(counts, 0, 3);  // zero out first 3 bytes at counts
memset(counts + 3, 0xff, 4)  // set 3rd entry’s bytes to 1s
```
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

| 4 | 2 | 12 | -5 | 56 | 14 |

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

![Array of unsorted integers]

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the **bubble sort** algorithm.

```
2  4  12  -5  56  14
```

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the **bubble sort** algorithm.

![](image)

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the **bubble sort algorithm**.

| 2 | 4 | -5 | 12 | 56 | 14 |

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
Let’s write a function to sort a list of integers. We’ll use the *bubble sort* algorithm.

Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
Let’s write a function to sort a list of integers. We’ll use the **bubble sort algorithm**.

Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

| 2 | -5 | 4 | 12 | 14 | 56 |
Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-5</td>
<td>4</td>
<td>12</td>
<td>14</td>
<td>56</td>
</tr>
</tbody>
</table>

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

In general, bubble sort requires up to $n - 1$ passes to sort an array of length $n$, though it may end sooner if a pass doesn’t swap anything.
Bubble Sort

• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

-5  2  4  12  14  56

• Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Only two more passes are needed to arrive at the above. The first exchanges the 2 and the -5, and the second leaves everything as is.
void bubble_sort_int(int *arr, size_t n) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                int tmp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = tmp;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void bubble_sort_int(int *arr, size_t n, bool ascending) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if ((ascending && arr[i - 1] > arr[i]) ||
                (!ascending && arr[i] > arr[i - 1])) {
                swapped = true;
                int tmp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = tmp;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void bubble_sort_int(int *arr, size_t n, bool ascending) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if ((ascending && arr[i - 1] > arr[i]) || (!ascending && arr[i] > arr[i - 1])) {
                swapped = true;
                int tmp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = tmp;
            }
        }
        if (!swapped) {
            return;  
        }
    }
}

We can add parameters, but they only help so much. Or even different comparisons (strcmp, etc.)?
void bubble_sort_int(int *arr, size_t n) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (should_swap(arr[i - 1], arr[i])) {
                swapped = true;
                int tmp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = tmp;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void bubble_sort_int(int *arr, int n) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                swap_int(&arr[i - 1], &arr[i]);
            }
        }
        if (!swapped) {
            return;
        }
    }
}

How can we make this function generic, to sort an array of any type?
Integer Bubble Sort

```c
void bubble_sort_int(int *arr, int n) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                swap_int(&arr[i - 1], &arr[i]);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Let’s start by making the parameters and swap generic.
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                swap(&arr[i - 1], &arr[i], elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
Key Idea: Locating i-th Elem

A common generics idiom is getting a pointer to the i-th element of a generic array.

Remember from last lecture, how to locate the last element:

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

How can we generalize this to get the location of the i-th element?
A common generics idiom is getting a pointer to the i-th element of a generic array.

Remember from last lecture, how to locate the last element:

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

How can we generalize this to get the location of the i-th element?

```c
void *ithElem = (char *)arr + i * elem_bytes;
```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
A Generics Conundrum

• We’ve hit a snag – there is no way to generically compare elements. They could be any type and have complex ways to compare them.

• How can we write code to compare *any two elements of the same type*?

• That’s not something that bubble sort can ever know how to do. **BUT** – our caller should know how to do this, because they’re supplying the data….let’s ask them!
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void bubble_sort(void *arr, int n, int elem_size_bytes, 
    function compare_fn) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prevElem, p_currElem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
    }
    if (!swapped) {
        return;
    }
}

So, how can we receive this comparison function? The function will be a parameter. And it is job will be to tell us how the two elements compare.
void bubble_sort(void *arr, int n, int elem_size_bytes,
                  bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}

So the expected return will be a bool, and the function’s expected input will be two void * arguments.
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

```c
bool (*compare_fn)(void *a, void *b)
```
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

```c
bool (*compare_fn)(void *a, void *b)
```

Return type (bool)
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function pointer name (compare_fn)
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

```c
bool (*compare_fn)(void *a, void *b)
```

Function parameters
(two void *s)
Here’s the general variable type syntax:

```c
[return type] (*[name])([[parameters]])
```
void bubble_sort(void *arr, int n, int elem_size_bytes, 
               bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) { return; }
    }
}
bubble_sort is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.

```c
bool integer_compare(void *ptr1, void *ptr2) {
    ...
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort(nums, nums_count, sizeof(nums[0]), integer_compare);
    ...
}
```
Function Pointers

bool string_compare(void *ptr1, void *ptr2) {
    ...
}

int main(int argc, char *argv[]) {
    char *classes[] = {"CS106A", "CS106B", "CS107", "CS110"};
    int arr_count = sizeof(classes) / sizeof(classes[0]);
    bubble_sort(classes, arr_count, sizeof(classes[0]), string_compare);
    ...
}

bubble_sort is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.
void bubble_sort(void *arr, int n, int elem_size_bytes, bool (*compare_fn)(void *a, void *b))

• Bubble Sort is written as a generic library function to be imported into potentially many programs to be used with many types. It must have a single function signature but work with any type of data.

• Its comparison function type is part of its function signature – the comparison function signature must use one set of types but accept any data of any size. How do we do this?
  • The function will instead accept pointers to the data via void * parameters
  • This means that the functions must be written to handle parameters which are pointers to the data to be compared
Function Pointers

This means that functions with generic parameters must always take *pointers to the data they care about*.

We can use the following pattern:
1) Cast the void *argument(s) and set typed pointers equal to them.
2) Dereference the typed pointer(s) to access the values.
3) Perform the necessary operation.

(steps 1 and 2 can often be combined into a single step)
bool integer_compare(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *
    int *num1ptr = (int *)ptr1;
    int *num2ptr = (int *)ptr2;

    // 2) dereference typed points to access values
    int num1 = *num1ptr;
    int num2 = *num2ptr;

    // 3) perform operation
    return num1 > num2;
}

This function is created by the caller specifically to compare integers, knowing their addresses are necessarily disguised as void *so that bubble_sort can work for any array type.
Function Pointers

```c
bool integer_compare(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *
    int *num1ptr = (int *)ptr1;
    int *num2ptr = (int *)ptr2;

    // 2) dereference typed points to access values
    int num1 = *num1ptr;
    int num2 = *num2ptr;

    // 3) perform operation
    return num1 > num2;
}
```

However, the type of the comparison function that e.g. `bubble_sort` accepts must be generic, since we are writing one `bubble_sort` function to work with any data type.
void bubble_sort(void *arr, int n, int elem_size_bytes, 
    bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) { 
            return;
        }
    }
}
bool integer_compare(void *ptr1, void *ptr2) {
    return *(int *)ptr1 > *(int *)ptr2;
}
Comparison Functions

• Function pointers are used often in cases like this to compare two values of the same type. These are called **comparison functions**.

• The standard comparison function in many C functions provides even more information. It should return:
  • < 0 if first value should come before second value
  • > 0 if first value should come after second value
  • 0 if first value and second value are equivalent

• This is the same return value format as `strcmp`!

```c
int (*compare_fn)(void *a, void *b)
```
int integer_compare(void *ptr1, void *ptr2) {
    return *(int *)ptr1 - *(int *)ptr2;
}

Comparison Functions
void bubble_sort(void *arr, int n, int elem_size_bytes, int (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem) > 0) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
Comparison Functions

• **Exercise:** how can we write a comparison function for bubble sort to sort strings in alphabetical order?

• The common prototype provides even more information. It should return:
  • < 0 if first value should come before second value
  • > 0 if first value should come after second value
  • 0 if first value and second value are equivalent

```
int (*compare_fn)(void *a, void *b)
```
int string_compare(void *ptr1, void *ptr2)
{
    // cast arguments and dereference
    char *str1 = *(char **)ptr1;
    char *str2 = *(char **)ptr2;
    // perform operation
    return strcmp(str1, str2);
}
Function Pointer Pitfalls

• If a function takes a function pointer as a parameter, it will accept it if it fits the specified signature.

• *This is dangerous!* E.g. what happens if you pass in a string comparison function when sorting an integer array?
Function Pointers

• Function pointers can be used in a variety of ways. For instance, you could have:
  • A function to compare two elements of a given type
  • A function to print out an element of a given type
  • A function to free memory associated with a given type
  • And more...
The meaning of “callback” functions

Library writer
- Writes generic algorithmic functions
- Relies on user-provided nelems, sizeof(elem), function pointer

User/caller
- Knows the data
- Might not know the algorithm (hence the use of library function)
- Writes the callback function to pass into library function

```c
void print_array(void *arr, size_t nelems,
                 int elem_size,
                 void(*print_fn)(void *)) {
    ...
}
```

```c
void print_string(void *ptr) {
    ...
}
```

```c
int main(int argc, char *argv[]) {
    ...
    print_array(str_array, n_elems,
                sizeof(str_array[0]), print_string);
    ...
}
```

The library uses a user-written (and user-provided!) callback function to perform complex operations on generic data.
Common Utility Callback Functions

• Comparison function – compares two elements of a given type.

\[
\text{int (*cmp_fn)(void *addr1, void *addr2)}
\]

• Printing function – prints out an element of a given type

\[
\text{void (*print_fn)(void *addr)}
\]

• There are many more! You can specify any functions you would like passed in when writing your own generic functions.
Function Pointers As Variables

In addition to parameters, you can make normal variables that are functions.

```c
int do_something (char *str) {
    printf("%s\n", str);
    return strlen(str);
}

int main(int argc, char *argv[]) {
    // Do something with variables
    int (*func_var)(char *) = do_something;
    char *str = "testing";
    int retval = func_var(str);
    printf("%d\n", retval);
    return 0;
}
```
Generic C Standard Library Functions

- **qsort** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.

- **bsearch** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

- **lfind** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

- **lsearch** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can’t find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
Generic C Standard Library Functions

- **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.
Recap

• We use `void *` pointers and memory operations like `memcpy` and `memmove` to make data operations generic.

• We use `function pointers` to make logic/functionality operations generic.

• Function pointers also allow us to pass logic around in our programs.

• Functions handling generic data must use `pointers to the data they care about`, since any parameters must have `one type` and `one size`. 
• Generic function: **Iterate through a generic array.**

```c
for (int i = 0; i < nelems; i++) {
    void *curr_p = (char *)base + i * elem_size_bytes;
    ...
}
```

• User setup: **Compute the number of elements in a local array.**

```c
int *int_array[] = ...; // declared locally
size_t nelems = sizeof(int_array) / sizeof(int_array[0]);
```
Recall print_array:
```c
void print_array(void *arr, size_t nelems, int elem_size_bytes,
                 void(*print_fn)(void *)) {
    for (int i = 0; i < nelems; i++) {
        void *elem_ptr = (char *)arr + i * elem_size_bytes;
        printf("%d: ", i + 1);
        print_fn(elem_ptr);
        printf("\n");
    }
}
```

What would happen here?

1. Fill in the blank so that print_array can print an array of strings.
   
   A. (char *) ptr   
   B. *(char **) ptr   
   C. *(char *) ptr   
   D. **(char ***) ptr

2. Why would A or D not “work”?

```c
1 void print_string(void *ptr) {
2    char *str = ???
3    printf("%s", str);
4 }
```
What would happen here?

```c
void print_array(void *arr, size_t nelems, int elem_size,
                 void(*print_fn)(void *)) {
    ...
    void *elem_ptr = (char *)arr + i * elem_size_bytes;
    print_fn(elem_ptr);
    ...
}
```

```c
void print_string(void *ptr) {
    char *str = ???;
    printf("%s", str);
}
```

```c
int main(int argc, char *argv[]) {
    char *str_array[] = {"aardvark", "beaver", "capybara"};
    size_t n_elems = sizeof(str_array) / sizeof(str_array[0]);
    print_array(str_array, n_elems, sizeof(str_array[0]), print_string);
    ...
}
```

What is the actual type of the parameter passed into `print_string`?

As a caller: Remember what the true types of parameters are.
Practice: Count Matches

• Let’s write a generic function `count_matches` that can count the number of a certain type of element in a generic array.

• It should take in as parameters information about the generic array, and a function parameter that can take in a pointer to a single array element and tell us if it’s a match.

```c
int count_matches(void *base, int nelems,
                   int elem_size_bytes,
                   bool (*match_fn)(void *));
```
int count_matches(void *base, int nelems, int elem_size_bytes, bool (*match_fn)(void *)) {
    int match_count = 0;
    for (int i = 0; i < nelems; i++) {
        void *curr_p = (char *)base + i * elem_size_bytes;
        if (match_fn(curr_p)) {
            match_count++;
        }
    }
    return match_count;
}