Quick Sort - implementation

Can anything be improved in this code?

```
#include <iostream>
using namespace std;

void print(int *a, int n)
{
    int i=0;
    while(i<n){
        cout<<a[i]<<",";
        i++;
    }
}

void swap(int i,int j, int *a){
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}


void quicksort(int *arr, int left, int right){
    int min = (left+right)/2;
    cout<<"QS:"<<left<<","<<right<<"\n";

    int i = left;
    int j = right;
    int pivot = arr[min];

    while(left<j || i<right)
    {
        while(arr[i]<pivot)
        i++;
        while(arr[j]>pivot)
        j--;

        if(i<=j){
            swap(i,j,arr);
            i++;
            j--;
        }
        else{
            if(left<j)
                quicksort(arr, left, j);
            if(i<right)
                quicksort(arr,i,right);
            return;
        }
    }
}


int main() {
    int arr[8] = {110, 5, 10,3 ,22, 100, 1, 23};
    quicksort(arr, 0, (sizeof(arr)/sizeof(arr[0]))-1);
    print(arr, (sizeof(arr)/sizeof(arr[0])));
    return 0;
}
```

c++ algorithm sorting quick-sort

edited Jan 18 '15 at 5:51

Jamal ♦

29.2k 10 109 219

asked Jan 17 '15 at 0:15

Software Engineer

159 2 4 10

2 Please don't edit the original code based on answers. – Jamal ♦ Jan 18 '15 at 5:51

4 Answers

In general, the code you have is neat, and easy to follow. The QuickSort algorithm is relatively traditional, and the pieces I expect to see are about where I expect them to be.

Let's go through the issues that I see though... and some of them are serious...

Namespaces

Using `namespace::std` is generally a poor idea. The possibility of namespace pollution is real. In fact, you have implemented a `swap` function and there already is one `std::swap`, but we'll get to that.

Style

- You have a variable `min`, but this should be `mid`.
- sometimes you put the function parameters with the left/right indices before the array, and sometimes after. You have:

  ```
  void quicksort(int *arr, int left, int right) {
  ```

  and you also have:

  ```
  void swap(int i,int j, int *a) {
  ```

  pick one, and stick with it. I would personally recommend putting the array first, and the indices afterwards.
- Use whitespace appropriately. Operators like the `<<` operator on `cout` are operators like any others. Use the space to improve readability:

  ```
  cout<<"QS:"<<left<<","<<right<<"\n";
  ```

  should be:

  ```
  std::cout << "QS:" << left << "," << right << "\n";
  ```

Bugs

You have a few bugs in here which should be addressed:

1. There's the potential for integer overflow when calculating the mid point. This is a 'famous' bug. Your code `int min = (left+right)/2;` should be done as:

   ```
   int mid = left + (right - left) / 2;
   ```

   The above solution will not overflow.
2. You should, when partitioning the data, consider values equal to the pivot value, to be either left or right of the pivot. Your code you use a strict `<` or `>` depending on whether you are on the right or left. One of those should include `=`. Your code, as it is, will run through the actual pivot value and do some funky business with it. You end up moving the pivot around in various ways.
3. You have a potential overrun (buffer overflow) in your loop conditions. It is posible, when you get to this line in the pivoting:

   ```
   while(arr[i]<pivot)
       i++;
   ```

   for i to run off the end of the array. If all the remaining values are less than the pivot, there's nothing stopping it from going off. You still need to check against j in these loops.

Swap

C++ has a swap function, use it. To get it, from C++11 `#include<utility>` and before that `#include<algorithm>`

Algorithm

The classic quick-sort is done in 5 stages:

1. find a 'pivot value'.
2. move all values less than (or equal to) the pivot value to 'the left'.
3. move all values larger than the pivot to 'the right'.
4. quick-sort the values less than(or equal)
5. quick-sort the values larger than.

Note that many text books extract the first 3 stages in to a 'partitioning' function. The purpose of that function is to identify the pivot value, move the candidates around, and then insert the pivot value back in to the data at 'the right place'.

That last part is key, it leaves the pivot value in the exact place it is supposed to be. This means you never have to include that pivot value in the sorts again.

Let's break that logic down in to it's methods, then, with the assumption that' there's a 'pivoting' function that does the first 3 steps. That leavs a simpler quicksort that looks like:

```
void quicksort(int *arr, const int left, const int right){

    if (left >= right) {
        return;
    }

    int part = partition(arr, left, right);

    quicksort(arr, left, part - 1, sz);
    quicksort(arr, part + 1, right, sz);
}
```

Notice, in the above, that the check to make sure the inputs are valid are done on entry to the recursive function. This simplifies the last part of the function. The same code could, alternatively, be written similar to yours, as:

```
void quicksort(int *arr, const int left, const int right, const int sz){

    int part = partition(arr, left, right);
    std::cout << "QSC:" << left << "," << right << " part=" << part << "\n";
    print (arr, sz);

    if (left < part - 1) {
        quicksort(arr, left, part - 1, sz);
    }
    if (part + 1 < right) {
        quicksort(arr, part + 1, right, sz);
    }
}
```

I prefer the first.... it makes it more easy to spot the recursion terminator.

So, that's now a simpler quicksort, partition the data, sort each partition (but not the actual partitioning value which is in the right place).

How do you partition the data?

The trick here is to swap the pivot value to the front of the sequence, partition the rest of the values, and then swap the pivot value back to where it belongs:

```
int partition(int *arr, const int left, const int right) {
    const int mid = left + (right - left) / 2;
    const int pivot = arr[mid];
    // move the mid point value to the front.
    std::swap(arr[mid],arr[left]);
    int i = left + 1;
    int j = right;
    while (i <= j) {
        while(i <= j && arr[i] <= pivot) {
            i++;
        }

        while(i <= j && arr[j] > pivot) {
            j--;
        }

        if (i < j) {
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i - 1],arr[left]);
    return i - 1;
}
```

Note how the code above double-checks the buffer overflow?

Putting this all together, and leaving in some of the debug statements you have, I would have the code:

```
#include <iostream>
#include <algorithm>

void print(int *a, int n)
{
    int i = 0;
    while(i < n){
        std::cout << a[i] << ",";
        i++;
    }
    std::cout << "\n";
}

int partition(int *arr, const int left, const int right) {
    const int mid = left + (right - left) / 2;
    const int pivot = arr[mid];
    // move the mid point value to the front.
    std::swap(arr[mid],arr[left]);
    int i = left + 1;
    int j = right;
    while (i <= j) {
        while(i <= j && arr[i] <= pivot) {
            i++;
        }

        while(i <= j && arr[j] > pivot) {
            j--;
        }

        if (i < j) {
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i - 1],arr[left]);
    return i - 1;
}

void quicksort(int *arr, const int left, const int right, const int sz){

    if (left >= right) {
        return;
    }


    int part = partition(arr, left, right);
    std::cout << "QSC:" << left << "," << right << " part=" << part << "\n";
    print (arr, sz);

    quicksort(arr, left, part - 1, sz);
    quicksort(arr, part + 1, right, sz);
}

int main() {
    int arr[8] = {110, 5, 10,3 ,22, 100, 1, 23};
    int sz = sizeof(arr)/sizeof(arr[0]);
    print(arr, sz);
    quicksort(arr, 0, sz - 1, sz);
    print(arr, sz);
    return 0;
}
```

I have put this in ideone too.

answered Jan 17 '15 at 4:46

rolfl

87k 13 178 384

Technique:

Rather than indexes into an array you should write your algorithm in terms of iterators.

```
quicksort(arr, 0, (sizeof(arr)/sizeof(arr[0]))-1);
```

Much easier to write as:

```
quicksort(std::begin(arr), std::end(arr));
```

The iterator divorcees your algorithm from container it is actually working on. So it will work just as well with arrays and std::vectors and std::array etc..

Clarity

You should not need to use this magic `(sizeof(arr)/sizeof(arr[0]))-1` like this anymore. Couple of problems with it. It is very susceptible to cut and paste problems (because arrays decay into pointers very quickly the above may stop working very quickly if moved to a function (if not done correctly)).

In terms of code clarity you should write a function that does size for you:

```
template <typename T, std::size_t N>
constexpr std::size_t getSize(T const (&)[N])
{
    return N;
}
```

Now your code becomes self documenting:

```
quicksort(arr, 0, getSize(arr)-1);
```

It also shows more clearly you are using non idiomatic C++ (the -1). Ranges in C++ are expressed in terms `[beginning, end)`. ie. `end` is one past the end of the container. This is done everywhere in C++ code; breaking from this idiom is going to cause you a lot of confusion with other C++ developers.

Compilers Job

Don't do work the compiler can do for you:

```
int arr[8] = {110, 5, 10,3 ,22, 100, 1, 23};
```

The compiler is better than you at it anyway and it will prevent errors. Here you have said the number of elements is 8. As a human I can't see that at a glance I could count them to verify but as a human I am lazy and going to assume you got it correct. If you did not get it correct then we will have problems.

So let the compiler work it out.

```
int arr[] = {110, 5, 10,3 ,22, 100, 1, 23};
```

Now if the array changes size you only have to change one thing (the data). Rather than two things (data and size).

answered Jan 17 '15 at 16:47

Loki Astari
57.1k 2 64 180

I see a couple things here that can be improved, and one that looks very wrong. This is what looks very wrong:

```
while(arr[i]<pivot)
i++;
while(arr[j]>pivot)
j--;
```

It almost looks as if you didn't intend this because you don't have braces and the indentation level is the same.

This should be written like this:

```
while (arr[i]<pivot) {
    i++;
}
while (arr[j]>pivot) {
    j--;
}
```

Your code is a very neat as a general rule, but your style is a little off. C++ typically uses braces like this:

```
void SomeFunction()
{
    while (condition) {
        // some code here
    }

    if (condition) {
        // some code here
    } else {
        // else code here
    }
```

```
    // and so on

    // Many people use the Stroustrup variant too, without a cuddled else:
    if (condition) {
        // some code here
    }
    else {
        // else code here
    }
}
```

You don't always have your braces have the same style, and sometimes you don't use braces at all. You should always use braces, even if there is only one statement in the block, and you should always use the same style.

Also, it is common to put spaces around operators to improve readability: `left < j` instead of `left<j` .

Finally, it is not good practice to always include `using namespace std;` because sometimes names inside namespaces overlap, which causes compiler errors or undefined behavior; instead, you should specifically specify which namespace you are using by placing `std::` in front of the name.

answered Jan 17 '15 at 1:24

Hosch250

16k 4 53 144

I'm going to assume that you're aware of `qsort` (C implementation of quicksort) and `std::sort` (C++ sorting routine). Therefore, you are doing this as a programming exercise. There's a `reinventing-the-wheel` tag for that if you happen to do something similar in the future.

```
using namespace std;
```

Why is using namespace std bad practice? Note that the short version is that it can lead to code conflicts and make it difficult to tell where code originates. For example, someone looking at your code might think that you were using `std::swap` .

```
int i=0;
while(i<n){
    cout<<a[i]<<",";
    i++;
}
```

This might be a good time to use a `for` loop. The main reason being that the logic matches exactly.

```
// decrement n so as not to have an extra comma at the end
--n;
for ( int i = 0; i < n; ++i ) {
    std::cout << a[i] << ", ";
}

// print the last entry without a trailing comma
std::cout << a[n] << std::endl;
```

Now the loop definition is all on one line rather than being scattered across three lines. There are times when you're just as well off to use a `while` as a `for` , but this isn't one of them.

Note that we can also get rid of the trailing comma by adjusting the loop boundary.

Also adds some horizontal whitespace to make it easier to see where one token ends and another begins.

Adds an end-of-line (which also flushes output), as that made sense in this code. You can leave that off if you want the code to be a little more general. Then you should do it in your `main` function. Otherwise you end up with your cursor starting on the same line as the output, which is confusing.

```
void quicksort(int *arr, int left, int right){
```

The variable name `arr` always makes me feel like a pirate. It's not that much more descriptive than `a` .

```
cout<<"QS:"<<left<<","<<right<<"\n";
```

This is debugging code. By the time that you are getting a code review, your debugging code should be out of your function.

```
std::cout << "QS:" << left << "," << right << std::endl;
```

When you are using debugging code, consider using a `std::endl` instead of `"\n"`. This is a little slower because it triggers a flush, but when debugging, you're probably better off with slightly slower code that doesn't have anything in the output buffer. That way, if your program crashes, you can see how far it got.

```
int min = (left+right)/2;
```

I find this name confusing. The name `min` would usually be an abbreviation of minimum, but you aren't picking the minimum. You're actually taking the middle element of the range. A more typical name would be `mid`, although I would go all the way and say `middle`.

You can get a better average performance by picking a random pivot though. Picking the middle element can devolve to a complexity of $O(n^2)$ in the worst case input. The problem arises when the middle element is always either the smallest or largest element in the range.

```
    while(arr[i]<pivot)
    i++;
```

This is unclear. You should always indent the contents of loops relative to the looping command. That way we can easily see that the statement is in the loop.

```
    while ( arr[i] < pivot )
        i++;
```

Even better, go all the way and write

```
    while ( arr[i] < pivot ) {
        i++;
    }
```

This not only makes it obvious what is and is not in the loop, but it protects against accidentally adding a statement outside the loop that you intended to be inside. As a practical matter, having to debug a mistake once takes more time than adding a thousand curly brace pairs.

```
int arr[8] = {110, 5, 10,3 ,22, 100, 1, 23};
```

There may be some excuse for calling it `arr` in the `quicksort` function where you don't actually know what it represents. Here though, you should call it by something more descriptive, if only `test_data`.

```
int test_data[8] = {110, 5, 10,3 ,22, 100, 1, 23};
int test_data_count = sizeof(test_data)/sizeof(test_data[0]);
```

I'd also go ahead and determine the number of elements ahead of time. You use the same expression twice. Rather than repeating yourself, precalculate it.

```
quicksort(arr, 0, (sizeof(arr)/sizeof(arr[0]))-1);
print(arr, (sizeof(arr)/sizeof(arr[0])));
```

Now we have to change this to match our previous changes.

```
quicksort(test_data, 0, test_data_count-1);
print(test_data, test_data_count);
```

This has the side effect of making it much easier to see what you are doing. That expression is counting the number of elements in the `test_data`. We then pass one less than that as the third parameter to `quicksort` and we pass the exact amount to `print`.

answered Jan 17 '15 at 3:15

Brythan

6,466 2 12 36