

Lecture 18 — Inlining, HLL

Patrick Lam

Good Programming Practices: Inlining

We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything! There are two ways of inlining in C++.

Implicit inlining. (defining a function inside a class definition):

```
class P {  
public:  
    int get_x() const { return x; }  
    ...  
private:  
    int x;  
};
```

Explicit inlining. Or, we can be explicit:

```
inline max(const int& x, const int& y) {  
    return x < y ? y : x;  
}
```

The Other Side of Inlining. Inlining has one big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors). Just from this your performance may go down easily.

Note also that inlining is merely a suggestion to compilers [GNU16]. They may ignore you. For example:

- taking the address of an “inline” function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

Implications of inlining. Inlining can make your life worse in two ways. First, debugging is more difficult (e.g. you can't set a breakpoint in a function that doesn't actually exist). Most compilers simply won't inline code with debugging symbols on. Some do, but typically it's more of a pain.

Second, it can be a problem for library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (Congratulations, you made a non-binary-compatible change!)

This would not be a problem for non-inlined functions—programs execute the new function dynamically at run-time.

High-Level Language Performance Tweaks

So far, we've only seen C—we haven't seen anything complex, and C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard, especially when `#define` macros and `void *` beckon.

C++11 has made major strides towards readability and efficiency—it provides light-weight abstractions. We'll look at a couple of examples.

Sorting. Our goal is simple: we'd like to sort a bunch of integers. In C, you would usually just use `qsort` from `stdlib.h`.

```
void qsort (void* base, size_t num, size_t size,
           int (*comparator) (const void*, const void*));
```

This is a fairly ugly definition (as usual, for generic C functions). How ugly is it? Let's look at a usage example.

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - (*((int*)b)));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

This looks like a nightmare, and is more likely to have bugs than what we'll see next.

C++ has a sort with a much nicer interface¹:

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp
);
```

¹...well, nicer to use, after you get over templates.

It is, in fact, easier to use:

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be a function or a functor. (Don't know what functors are? In C++, they're functions with state.) By default, `sort` uses `operator<` on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

The second question is empirical. Let's see. We generate an array of 2 million ints and sort it (10 times, taking the average).

- `qsort`: 0.49 seconds
- C++ `sort`: 0.21 seconds

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

- Custom C sort: 0.29 seconds

Now the C++ version is still faster (but it's close). But, this is quickly going to become a maintainability nightmare.

- Would you rather read a custom sort or 1 line?
- What (who) do you trust more?

Lesson

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Vectors vs Lists

Consider two problems.

1. Generate N random integers and insert them into (sorted) sequence.

Example: 3 4 2 1

- 3
- 3 4
- 2 3 4
- 1 2 3 4

2. Remove N elements one-at-a-time by going to a random position and removing the element.

Example: 2 0 1 0

- 1 2 4
- 2 4
- 2
-

For which N is it better to use a list than a vector (or array)?

Complexity analysis. As good computer scientists, let's analyze the complexity.

Vector:

- Inserting
 - $O(\log n)$ for binary search
 - $O(n)$ for insertion (on average, move half the elements)
- Removing
 - $O(1)$ for accessing
 - $O(n)$ for deletion (on average, move half the elements)

List:

- Inserting
 - $O(n)$ for linear search
 - $O(1)$ for insertion
- Removing
 - $O(n)$ for accessing
 - $O(1)$ for deletion

Therefore, based on their complexity, lists should be better.

Reality. OK, here's what happens.

```
$ ./vector_vs_list 50000
Test 1
=====
vector: insert 0.1s   remove 0.1s   total 0.2s
list:   insert 19.44s remove 5.93s   total 25.37s
Test 2
=====
vector: insert 0.11s  remove 0.11s  total 0.22s
list:   insert 19.7s  remove 5.93s  total 25.63s
Test 3
=====
vector: insert 0.11s  remove 0.1s   total 0.21s
list:   insert 19.59s remove 5.9s   total 25.49s
```

Vectors dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory. **On 64 bit machines:**
 - Vector: 4 bytes per element.
 - List: At least 20 bytes per element.
- Memory access is slow, and results arrive in blocks:
 - Lists' elements are all over memory, hence many cache misses.
 - A cache miss for a vector will bring a lot more usable data.

So, here are some tips for getting better performance.

- Don't store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.
- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient.**
- Think at a low level if you need to optimize anything.
- Readable code is good code—different hardware needs different optimizations.

References

[GNU16] GNU Compiler Collection. An inline function is as fast as a macro, 2016. Online; accessed 6-January-2016. URL: <https://gcc.gnu.org/onlinedocs/gcc/Inline.html>.