Programming in C and C++ - Supervision 2

Nandor Licker

October 2019

1 Tooling

Q1 Reading a value from an unaligned address results in undefined behaviour. Provide a method using C++ template metaprogramming that reads any integral type from any location.

Q2 Build systems are also part of C++ tooling and nobody should grow old without being tortured by Makefiles first. For this assignment, create a project built using a Makefile consisting of the following:

- A text file containing a large dictionary of words.
- A python script to generate C file definining an array of strings, containing all the words.
- A header file declaring the array.
- A main C file which reports which command line arguments are in the dictionary.

Ensure incremental builds work correctly - changing any file should trigger the recompilation of all downstream dependencies. Add flags to enable any of the sanitizers.

2 Aliasing, Graphs, and Deallocation

Q1 Arena allocators are quite commonly used in practice. Declare an interface for an arena allocator in a header file and define it in a source file. The allocator should keep allocating chunks of memory of fixed size using `malloc`. Ensure you can satisfy requests whose size exceeds the chunk size as well.

Q2 Implement a stack backed by a linked list of memory chunks of fixed size, capable of allocating items of arbitrary size. Ensure you do not call `malloc`/`free` too often when you push/pop across a chunk boundary.

3 Reference Counting and Garbage Collection

Q1 C++11 introduced `std::unique_ptr` and `std::shared_ptr` to simplify memory management. To understand them, provide your own implementation of shared pointers. The class should at least provide the following methods:

```
template <typename T>
class SharedPtr {
public:
    SharedPtr(T *);
    SharedPtr(SharedPtr &&);
    SharedPtr(const SharedPtr &);
    SharedPtr &operator = (SharedPtr &&);
    SharedPtr &operator = (const SharedPtr &);
    T *get();
};
```

Consider storing the reference count on the heap somewhere. Also consider using an atomic type introduced in C++11 to represent the counter.

Q2 Why is `std::make_shared<T>()` preferred over `std::shared_ptr<T>(new T())`? Think about allocations. Implement `make_shared` for your shared pointers, removing the old constructor.

Hint: C++ offers a placement new operator which constructs an object in a given buffer instead of allocating new memory. You can also invoke destructors without `delete`:

```
T *a = new (void_pointer_to_memory) T(); // construct
a->~T();                                 // destruct
```

Isn't that neat? As a sidenote, placement new is actually the recommended way of constructing objects inside unions. Do make sure to use these features in any safety-critical context you might encounter! (Hint: Please do not, but try to sleep at night knowing that someone else used this stuff!)

```
struct A { A(); }
union B {
    int x;
    A a;
};
B b;
b.x = 5;
new (&b.a) A();
```

Q3 Comment on the challenges of interfacing between a manually-managed and garbage collected language. What information does a garbage collector need that C does not implicitly provide? How do OCaml and Lua manage this?

4 The Memory Hierarchy and Cache Optimization

Q1 *Warning: Personal Opinion.* Solemnly swear that you will always prove the necessity of applying such optimisations through thorough benchmarking before you start turning readable code into an unreadable and unportable mess.

Q2 What is the issue with the following structure?

```
struct TwoInts {
    atomic_int a;
    atomic_int b;
};
```

Q3 Comment on the performance of a linked list backed by `malloc`, compared to a linked list created using an arena allocator.

5 Debugging and Undefined Behaviour

Implement a method `bool add_signed_overflow(int a, int b, int *result)` which safely adds two integers and returns true if overflow would have occurred. Return the result, taking into account two's complement signed overflow. Write a comprehensive test suite using `googletest` and run it with UBSan. Ensure undefined behaviour does not occur. Consider producing a report using `gcov` to estimate the code and branch coverage of your tests.