C++ Teasers

Frank B. Brokken
University of Groningen

December 2013 - Octobr 2023

1 About the teasers

If you're interested in C++, then this document may contains some interesting (and maybe teasing) questions.

Every now and then I may update the questions.

Most of these should be easy to answer, especially if you've attended my C++ courses. Can't attend them? I may have a tailored course for you. Just contact me if you're interested.

2 Questions

- When to define enums? Should they be strongly typed? When should they be? When is defining a stronly typed enum not really necessary?
- Why use ++idx rather than idx++ in, e.g.,

```
for (size_t idx = 0; idx != end; ++idx)
    ...
```

- Why prefer != over < in for-stmnt conditions?
- How can you be sure that you're not falling in the off-by-one trap when using for-stmnts? Both in the incrementing and decrementing case?
- What's your philosophy for using a for-statement? And a while-statement? (or don't you have one?)
- When do you prefer an if-stmnt over a switch-stmnt? And v.v.?
- Do you ever prefer neither? If so, what's your alternative?
- What is your philosophy for defining classes?

- How do you implement (rather than design) your class members? What is your software design philosophy here?
- `new Type[n]` calls `Type's` default constructor. What do you do if you need to define `n Type` objects using a non-default constructor?
- The intent is to use placement new to enlarge arrays of `Object` objects. Assume initially you do `ptr = new Object[1]`. Since placement new is used to create a larger array, we use `operator delete ptr` to return the previously allocated array just before assigning `ptr` to the enlarged array.

  Why does the program (usually) crash with a run-time error complaining that `ptr` contains is an invalid pointer?
- Do you have any good arguments for using placement new?
- Why won't the compiler allow you to pass a `Type **` argument to a function defining a `Type const **` parameter?
- Why should you put '`const`' *behind* the things that are constant, and not before?
- How can you provide context to signal handlers?
- What is your approach to using shared memory? Can you do so without violating principles of designing reusable software?
- How do you distinguish lvalue and rvalue uses of the index operator?
- How to use the `copy` generic algorithm and `std::istream_iterators` to fill a `std::vector<std::string>` with the *lines*, rather than blank-delimited words from an `std::istream`?
- Why is it a bad idea to use 'virtual' for members of derived classes if their base classes also specified 'virtual'? What should you do instead?
- Did you ever have to implement a swap-member? How do you implement a swap member if your class features reference members?
- When do you consider adding members having rvalue reference parameters to your classes?
- Why is inheritance used?
- Is inheritance ever useful without polymorphism?
- What is a polymorphic base class? Why would you use one?
- What are VTables and where are they? Can you organize your software in such a way that you have one answer that's always correct?
- Why do objects of polymorphic classes occupy more memory than objects of non-polymorphic classes?
- What is static polymorphism? Do you have an example?
- What's so interesting about rvalue reference parameters in function templates?

- How do you design a `const_iterator` which is derived from a `std::iterator` created as a `std::input_iterator_tag`, but which also allows you to use a matching `std::reverse_iterator`?
- Why don't classes derived from `Base` have to be polymorphic when storing newly allocated objects of such classes in a `std::shared_ptr<Base>` or `std::unique_ptr<Base>` object?
- Could you design and implement a class template expecting a `typename Base`, accepting a pointer to any class that is derived from `Base` (either at construction-time or using a `reset` member, just like `std::shared_ptr<Base>` or `std::unique_ptr<Base>`), properly deleting the `Derived` class object when the object of your class goes out of scope? (Your class may not use or rely on polymorphism either).
- Why isn't partial specialization available for function templates?
- Assume you have a function template

```
template <typename Tp>
void fun(Tp tp)
{
    std::cout << "hi\n";
}
```

  You can use this in a program:

```
int main(int argc, char **argv)
{
    fun(argc);
    fun(argv[0]);
}
```

  What modification do you propose to ensure that you cannot create this program anymore because `fun` cannot be instantiated for plain `int` types? (you may not use template meta-programming techniques.)