War of Idioms

Useful, yet not commonly used C++ idioms

Akademy 2014

Ivan Čukić

ivan.cukic@kde.org
ivan.fomentgroup.org/blog

Disclaimer

> Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.
>
> — Philip Wadler

Disclaimer 2

“Modern C++”

Focus

- Safety
- Speed
- Beauty

Focus

- Safety
- Speed
- Beauty
- **Insanity**

MEMORY

Memory safety

> But I think that Garbage Collection is not as critical for C++ as it is for many of the other languages. We just don't generate that much garbage.
>
> Bjarne Stroustrup

Item No 1: Evil pointers

- Doing explicit `new` and `delete`

Item No 1: Evil pointers

- Doing explicit `new` and `delete`
- What does the following declaration mean?
  `type * instance();`

Item No 1: Evil pointers

- Doing explicit `new` and `delete`
- What ~~does~~ **can** the following declaration mean?
  `type * instance();`
  - (static) singleton instance?
  - should be disposed by the user?
  - creator-owned, creator disposes of it?
  - an optional value?
  - a position in an array?

Item No 1: Evil pointers

```
d(new Private());
delete d;

new QThread(this);

object->deleteLater();
```

The new No 2: The Good Guys

```
type &instance();

std::unique_ptr<type> instance();

std::weak_ptr<type> instance();

std::shared_ptr<type> instance();

std::optional<type> instance(); // *
std::expected<type, error_type> instance(); // **
```

* does not really exist yet, boost::optional does.

** also does not exist, not certain it will.

ITEM No 3: The Snowflake

`std::unique_ptr`

- Just a RAII class for `new` and `delete`
- With copying disabled
- And moving enabled

ITEM No 4: Sharing is Caring

```
static std::shared_ptr<T> instance()
{
    static std::weak_ptr<T> s_instance;

    static std::mutex s_mutex;
    std::lock_guard<std::mutex> singleton_lock(s_mutex);

    auto result = s_instance.lock();

    if (s_instance.expired()) {
        result.reset(new T());
        s_instance = result;
    }

    return std::move(result);
}
```

Virtuals

```
class Base {
public:
    // some virtual methods

    ~Base() { ::: }
};

class Derived: public Base {
public:
    ~Derived() { ::: }
}
```

ITEM No 5: D-PTR

```
class Thingie {
public:
    :::

private:
    class Private;
    Private * const d;
};

Thingie::Thingie() : d(new Private()) {}
Thingie::~Thingie() { delete d; }
```

ITEM No 5: D-PTR

```
template <typename T>
class d_ptr {
private:
    std::unique_ptr<T> d;

public:
    d_ptr();

    template <typename... Args>
    d_ptr(Args &&...);

    ~d_ptr();

    T *operator->() const;
};
```

ITEM No 5: D-PTR

```
class Thingie {
public:
    :::

private:
    class Private;
    d_ptr<Private> d;
};

Thingie::Thingie() {}
Thingie::~Thingie() {}
```

ITEM No 5: D-PTR

```
Thingie::Thingie() : d(new Private()) {}
Thingie::~Thingie() { delete d; }
// vs.
Thingie::Thingie() : {}
Thingie::~Thingie() {}

Thingie::Thingie() : d(new Private(this)) {}
Thingie::~Thingie() { delete d; }
// vs.
Thingie::Thingie() : d(this) {}
Thingie::~Thingie() {}
```

ITEM Not a Number, a Free Man

```
template <typename T>
class execaround_ptr {
    class proxy {
        proxy(T *v) :inner(v) { ::: }
        ~proxy() { ::: }

        T *operator->() { return inner; }

        T *inner;
    };

    proxy operator->() { return proxy(pointer); }

    :::
};
```

RANGES

ITEM No 7: Algorithms

Non-modifying sequence operations
all_of, any_of, none_of, for_each, count, count_if, mismatch, equal, find, find_if, find_if_not, find_end, find_first_of, adjacent_find, search, search_n

Modifying sequence operations
copy, copy_if, copy_n, copy_backward, move, move_backward, fill, fill_n, transform, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, replace, replace_if, replace_copy, replace_copy_if, swap, swap_ranges, iter_swap, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, shuffle, unique, unique_copy

Partitioning operations
is_partitioned, partition, partition_copy, stable_partition, partition_point

Sorting operations
is_sorted, is_sorted_until, sort, partial_sort, partial_sort_copy, stable_sort, nth_element

...

ITEM No 7: Algorithms

```
...

Binary search operations (on sorted ranges)
lower_bound, upper_bound, binary_search, equal_range

Set operations (on sorted ranges)
merge, inplace_merge, includes, set_difference, set_intersection,
set_symmetric_difference, set_union

Heap operations
is_heap, is_heap_until, make_heap, push_heap, pop_heap, sort_heap

Minimum/maximum operations
max, max_element, min, min_element, minmax, minmax_element,
lexicographical_compare, is_permutation, next_permutation,
prev_permutation

Numeric operations
iota, accumulate, inner_product, adjacent_difference, partial_sum
```

ITEM No 8: Ranges

SQL

```sql
SELECT "count" + "additions"
    FROM "Thingies"
    WHERE "count" < 30
        AND "additions" > 60
```

Haskell

```haskell
map (\ (count, additions) -> count + additions) (
    filter
        (\ (count, additions) ->
            count < 30 && additions > 60
        )
        thinges
    )
```

ITEM No 8: Ranges

`std::transform` - map, projection
`std::find_if` - kinda like filter

The problem is that they require iterator pairs.

ITEM No 8: Ranges

Enter `boost.range`

```
thingies | filtered(_1.count     < 30)
         | filtered(_1.additions > 60)
         | transformed(_1.count + _1.additions)
```

C++17 should have a proper support for ranges.

TYPING

ITEM No 11: Case for auto

```
std::map<std::string,
         std::pair<int, std::string>
    >::const_iterator i = ...;
```

_

ITEM No 11: Case for auto

```
std::map<std::string,
         std::pair<int, std::string>
    >::const_iterator i = ...;

auto x; // error

??? l = [] { ::: };

int l = arr.length();
```

ITEM No 11: Case for auto

```
map<string, string> items;

for (const pair<string, string>& item: items) {
    // what is wrong here?
}
```

ITEM No 12: Case against, or a case for?

```
QString row{"A"};
QString number{"1"};

auto result1 = row + number;

number = "2";

auto result2 = row + number;

qDebug() << result1;
qDebug() << result2;
```

ITEM No 13: Collecting ducks

How do we deal with collections of polymorphic objects?

- Create a super-type – interface / abstract class
- Call virtual methods everywhere we can

Can we do something that does not expose the internals in such way?

ITEM No 14: Collecting ducks

Implement polymorphic collections using duck typing?

Full frontal

When can you call a private method?

```
QVariant v((void*) nullptr);
```

Questions?

Further reading and watching:

- Modern C++ design, Andrei Alexandrescu
- C++ Seasoning, Sean Parrent
- Value Semantics and Range Algorithms, Chandler Carruth
- Systematic Error Handling in C++, Andrei Alexandrescu