Writing robust C++ code

Miguel Ojeda (CERN)

1st Developers@CERN Forum

2015-10-29

C++ is complex

Can you tell me what is the output of this program?

```
namespace tools {
    template <typename T> void print(T x) {
        std::cout << x << std::endl;
    }

    template <typename T> void print_repeat(T x, int n) {
        for (int i = 0; i < n; ++i)
            print(x);
    }
}

namespace something {
    struct A { };
    std::ostream & operator<<(std::ostream & os, A) { return os << 42; }
}

int main() {
    tools::print_repeat(something::A(), 4);
    return 0;
}

// ... more code ...
```

C++ is complex

Can you tell me what is the output of this program?

```cpp
namespace tools {
    template <typename T> void print(T x) {
        std::cout << x << std::endl;
    }

    template <typename T> void print_repeat(T x, int n) {
        for (int i = 0; i < n; ++i)
            print(x);
    }
}

namespace something {
    struct A { };
    std::ostream & operator<<(std::ostream & os, A) { return os << 42; }
}

int main() {
    tools::print_repeat(something::A(), 4);
    return 0;
}

namespace something {
    void print(A) {
        std::cout << "No, you cannot..." << std::endl;
    }
}
```

C++ is complex

I hereby present you C++ valid programs…

C++ is complex

I hereby present you C++ valid programs...

```
bool r = phrase_parse(first, last,
  //  Begin grammar
  (
          '(' >> double_[ref(rN) = _1]
              >> -(',' >> double_[ref(iN) = _1]) >> ')'
      |   double_[ref(rN) = _1]
  ),
  //  End grammar

  space);
```

Source: Boost Spirit [1]

C++ is complex

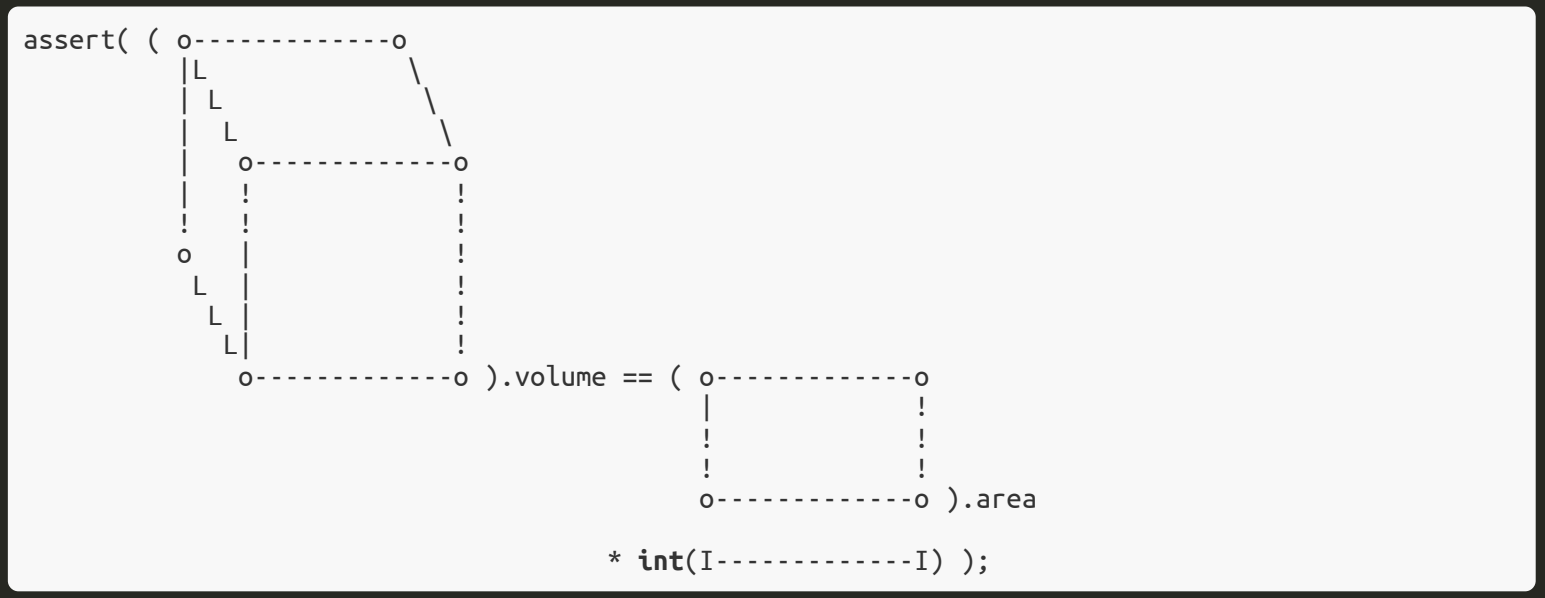
I hereby present you C++ valid programs…

```
assert( ( o-----o
          |     !
          !     !
          !     !
          !     !
          o-----o ).area == ( o---------o
                              |         !
                              !         !
                              o---------o ).area );
```

Source: eelis.net [2]

C++ is complex

I hereby present you C++ valid programs…



Source: eelis.net [2]

C++ is complex

Multiple inheritance

C++ is complex

Multiple inheritance

Exceptions

C++ is complex

Multiple inheritance

Exceptions

RAII

C++ is complex

Multiple inheritance

Exceptions

RAII

ADL

C++ is complex

Multiple inheritance

Exceptions

RAII

ADL

SFINAE

C++ is complex

Multiple inheritance

Exceptions

RAII

ADL

SFINAE

…

C++ is complex

C++ templates are Turing-complete. [3]

C++ is complex

C++ templates are Turing-complete. [3]

Therefore, compiling C++ is undecidable.

C++ is complex

C++ templates are Turing-complete. [3]

Therefore, compiling C++ is undecidable.

- *Assuming no template instantiation limits*

C++ is complex

C++ templates are Turing-complete. [3]

Therefore, compiling C++ is undecidable.

- *Assuming no template instantiation limits*

But not only that, just ***parsing*** C++ is undecidable as well. [4]

C++ is complex

See why:

```
template <...> struct TuringMachine {
    // Insert implementation of a Turing machine here
};

struct SomeType {};

template <typename T> struct S {
    static int name;
};

template<> struct S<SomeType> {
    typedef int name;
};

int x;
int main() {
    S<TuringMachine<...>::output>::name * x;
}
```

Source: Josh Haberman [4]

Writing robust C++

So how do we write robust C++?

Writing robust C++

So how do we write robust C++?

Of course, one should profit from common topics in software development.

Writing robust C++

So how do we write robust C++?

Of course, one should profit from common topics in software development.

However, for C++, several extra guidelines apply…

Writing robust C++

Your compiler is your friend:

- Enable all warnings
  - Any (global) exception should be justified
- Treat all warnings as errors
  - Do not allow commits with warnings
- Enable stack protection/canaries
  - For all (or most) functions, if performance allows
- Enable any other hardening/fortify options available
  - They are useful not only for security reasons

Writing robust C++

Diversity is good:

- Use several compilers
    - In particular, their latest version
- Compile and test for several architectures/operating systems, if possible
    - Bonus: gives you access to extra debuggers/tools
- Simulate your code, if possible
    - You can go down to simulating the memory hierarchy, if you need
- Test all debug, release and optimized builds

Writing robust C++

Consider useful libraries that add low overhead:

- Use a different allocator
  - Memory-tagging allocators (even in release builds)
  - Debugging allocators (heap checkers)
- Use checked STL library
  - e.g. -D_GLIBCXX_DEBUG

Writing robust C++

Avoid fancy C++ features unless strictly required:

- Multiple inheritance, ADL, templates, SFINAE…
- If you use them, be prepared to justify why they were needed
- Before starting a project, consider carefully exceptions and RTTI

Writing robust C++

Establish strict coding guidelines:

- Pick an existing one, if possible.
- Enforce them through the compiler/tools, if possible.

Writing robust C++

Include all the tools that you can in your test environment:

- valgrind's memcheck and helgrind/drd
- gcc's and clang's memory/thread/u.b. sanitizers
- clang-format
- Coverity
- ...

Writing robust C++

And above all, write ***defensive code***:

- Use checked/safer functions/methods, if performance allows
  - e.g. at() vs. operator[]
  - e.g. memmove() vs. memcpy()
  - e.g. strlcpy() vs. strncpy()
- Use RAII everywhere, if using exceptions

Writing robust C++

And above all, write ***defensive code***:

- Provide the basic exception guarantee
  - Minimize size of functions/methods to accomplish it
- Write down and check pre/post conditions and class invariants
  - Even in release builds, if performance allows
- …

Summary

C++ is a complex language with complex features. Therefore:

1. Aim for ***simplicity***, whenever possible
    - Require complexity to be justified and documented
2. Take advantage of compilers, ***tools*** and everything you can
    - Many are easy to setup, free and/or open source
3. Use as much ***runtime-protection/checking*** as possible
    - Most likely you can spare the overhead
4. Set up strict, justified development ***guidelines*** and rules
    - Well worth it for team projects

Questions?

References

[1] eelis.net. *Multi-Dimensional Analog Literals*. 2006.

[2] The Boost Team. *Boost Spirit Library*. 2015.

[3] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. 2003.

[4] Josh Haberman. *Parsing C++ is literally undecidable*. 2013.