C++ for Blockchain Applications

Godmar Back

C++ History (1)

- 1979 C with Classes
  - classes, member functions, derived classes, separate compilation, public and private access control, friends, type checking of function arguments, default arguments, inline functions, overloaded assignment operator, constructors, destructors
- 1985 CFront 1.0
  - virtual functions, function and operator overloading, references, new and delete operators, const, scope resolution, complex, string, iostream
- 1989 CFront 2.0
  - multiple inheritance, pointers to members, protected access, type-safe linkage, abstract classes, io manipulators
- 1990/91 ARM published (CFront 3.0)
  - namespaces, exception handling, nested classes, templates

C++ History (2)

- 1992 STL implemented in C++
- 1998 C++98
  - RTTI (dynamic_cast, typeid), covariant return types, cast operators, mutable, bool, declarations in conditions, template instantiations, member templates, export
- 1999 Boost founded
- 2003 C++03
  - value initialization
- 2007 TR1
  - Includes many Boost libraries (smart pointers, type traits, tuples, etc.)

C++ History (3)

- 2011 C++11
  - Much of TR1, threading, regex, etc.
  - Auto, decltype, final, override, rvalue refs, move constructors/assignment, constexpr, list initialization, delegating constructors, user-defined literals, lambda expressions, list initialization, generalized PODs, variadic templates, attributes, static assertions, trailing return types
- 2014 C++14
  - variable templates, polymorphic lambdas, lambda captures expressions, new/delete elision, relaxed restrictions on constexpr functions, binary literals, digit separators, return type deduction for functions, aggregate initialization for classes with brace-or-equal initializers....
- 2015
  - Several TS: Filesystem, Parallelism, Transactional Memory, Concurrency, Concepts

C++ History (4)

- 2017 C++17
  - fold-expressions, class template argument deduction, auto non-type template parameters, compile-time if constexpr, inline variables, structured bindings, initializers for if and switch, u8-char, simplified nested namespaces, guaranteed copy elision, lambda capture of *this, constexpr lambda, [[fallthrough]], [[nodiscard]], and [[maybe_unused]]
- 2017
  - TS: Ranges, Coroutines, Networking, Parallism v2, modules
- 2020 C++20
  - Coroutines, Modules, Ranges and Concepts TS
  - feature test, 3-way comparison operator <=> and operator==() = default, designated initializers, init-statements and initializers in range-for, [[no_unique_address]], [[likely]], [[unlikely]], pack-expansions in lambda captures, consteval, constinit, abbreviated function templates

Excerpt from “A Brief, Incomplete, and Mostly Wrong History of Programming Languages”

1983 - Bjarne Stroustrup bolts everything he's ever heard of onto C to create C++. The resulting language *is so complex that programs must be sent to the future to be compiled* by the Skynet artificial intelligence. Build times suffer. Skynet's motives for performing the service remain unclear but spokespeople from the future say "there is nothing to be concerned about, baby," in an Austrian accented monotones.

James Iry, 2009

http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

Stroustrup, 2006

We need relatively complex language to deal with absolutely complex problems. I note that English is arguably the largest and most complex language in the world (measured in number of words and idioms), but also one of the most successful.

C++ has drifted towards becoming an “expert friendly” language. In a gathering (in person or on the Web) of experts, it is hard to build a consensus (or even interest) for something that “just” helps novices. The general opinion (in such a gathering) is typically that the best we can do for novices is to help them become experts. But it takes time to become an expert and most people need to be reasonably productive during the time it takes. More interesting, many C++ novices have no wish or need to become experts in C++. If you are a physicist needing to do a few calculations a week, an expert in some business processes involving software, or a student learning to program, you want to learn only as many language facilities as you need to get your job done. You don’t want to become a language expert—you want to be (or become) an expert in your own field and know just enough of some programming language to get your work done. When supported by suitable libraries, C++ can be used like that—it is widely used like that. However, there are traps, pitfalls, and educational approaches that make such “occasional use” of C++ unnecessarily difficult.

C++ as a “multi-paradigm” language

- ``programming using more than one programming style, each to its best effect." - Stroustrup
- For example,
  - Procedural programming style (“a better C”)
  - Data abstraction (encapsulation)
  - Object-oriented programming (runtime polymorphism)
  - Generic programming (compile time polymorphism)
  - Functional style (recursion and higher-order functions)
  - Declarative style (metaprogramming)

What is “Modern C++”?

- A subset of these paradigms, characterized by
- Absence of raw pointers and explicit memory manipulation via new/delete
  - instead use of references to encapsulated objects and exploitation of compiler optimizations such as copy elision
  - smart pointers otherwise
- Use of type inference (auto) wherever possible
- Extensive “script”-like use of high-level STL classes
  - Pairs, tuples, etc.

The familiar parts of C++ (if you know C)

- C-style syntax (semicolon as statement terminator, use of { } braces, similar keyword set (if/for/while etc.)
- Similar single-pass compilation process
  - Separate compilation units
- Statically typed language
  - Every variable and expression’s type must be known to the compiler at compile-time (even if variable is declared with `auto`)
- Primitive integer/char types
- Composite types (e.g. struct + class)
- Programmer control over stack-allocated vs heap-allocated objects

The familiar parts of C++ (if you know Java)

- Stricter static type checking
- Similar support for encapsulation via private/protected etc. modifiers
  - Albeit a bit more complex
- Similar mechanisms for runtime polymorphism (e.g. inheritance, virtual functions/overriding)
  - Similar constructor syntax
- Similar function overloading mechanism
- Basic exception handling (try/catch)
- Similar set of core functionality in standard library (STL)
  - Although paradigms and implementation differ significantly

(Of course, these are all things Java inherited from C++, not the other way around.)

Lack of memory safety

- C++, like C, is not a type- or memory safe language.
- Programmers can side-step the language's type system and write unsafe code that directly accesses the underlying machine's memory
- The specification allows for undefined behavior of many syntactic constructs
- Serious interaction with compiler optimizations: optimizing compilers use the assumption that the programmer wrote only well-defined code

Object Initialization

- In Java, all objects are heap-allocated and subject to garbage collection.
- In C++, objects can be heap-allocated or automatically allocated (as local variables or temporaries)
  - Such objects can then be passed to and returned from functions
  - There exist numerous ways to initialize either kind
- This ability has ripple effects
- Object creation, destruction, copying and move may have user-provided side effects/is under user control

```
Point p0;
Point p1();
Point p2{};
Point p3 = { 1, 2 };
Point p4 = { 1 };
Point p5 {1, 2};
Point p6 = Point{1, 2};
Point();
Point{1, 2};
new Point;
new Point();
new Point{};
new Point{2, 3};
new Point{1};
```

```
struct Point {
    int x, y;
};
```

```
Point p0;  // default initialization (uninitialized)
Point p1(); // func declaration: extern Point p1(void);
Point p2{}; // zero initialization
Point p3 = { 1, 2 }; // aggregate + copy initialization
Point p4 = { 1 }; // aggregate + copy initialization
Point p5 {1, 2};  // aggregate + direct initialization
Point p6 = Point{1, 2}; // copy initialization
Point();  // zero-initialized temporary
Point{1, 2};  // aggregate-initialized temporary
new Point;  // uninitialized heap object
new Point();  // zero-initialized heap object (C++03)
new Point{};  // zero-initialized heap object (C++11)
new Point{2, 3}; // aggregate-initialized
new Point{1};    // heap objects
```

```
struct Point {
    int x, y;
};
```

Constructor Example

- Code works also with

```
struct Point {
    int x, y;
    Point() { }
    Point(int x, int y = 0) : x{x}, y{y}
    { }
};
```

Constructor Example

- Or, using encapsulation

```
class Point {
    int x, y;
public:
    Point() { }
    Point(int x, int y = 0) : x{x}, y{y}
    { }
};
```

Meme by Timur Doumler:
https://mobile.twitter.com/timur_audio/status/1004017362381795329

Initialisation in C++17

	Default init `;`	**Copy init** `= value;`	**Direct init** `(args);`	**Value init** `();`	**Empty braces** `{};  = {};`	**Direct list init** `{args};`	**Copy list init** `= {args};`
Built-in types	Uninitialised. Variables w/ static storage duration: Zero-initialised	Initialised with value (via conversion sequence)	1 arg: Init with arg >1 arg: Doesn't compile	Zero-initialised	Zero-initialised	1 arg: Init with arg >1 arg: Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile
auto	Doesn't compile	Initialised with value	Initialised with value	Doesn't compile	Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile	Object of type std::initializer_list
Aggregates	Uninitialised. Variables w/ static storage duration: Zero-initialised***	Doesn't compile	Doesn't compile (but will in C++20)	Zero-initialised***	Aggregate init**	1 arg: implicit copy/move ctor if possible. Otherwise aggregate init**	1 arg: implicit copy/move ctor if possible. Otherwise aggregate init**
Types with std::initializer_list ctor	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor if there is one, otherwise std::initializer_list ctor	std::initializer_list ctor if possible, otherwise matching ctor	std::initializer_list ctor if possible, otherwise matching ctor****
Other types with no user-provided* default ctor	Members are default-initialised	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Zero-initialised***	Zero-initialised***	Matching ctor	Matching ctor****
Other types	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor	Matching ctor	Matching ctor****

*not user-provided = not user-declared, or user-declared as `=default` *inside* the class definition
**Aggregate init copy-inits all elements with given initialiser, or value-inits them if no initialiser given
***Zero initialisation zero-initialises all elements *and initialises all padding to zero bits*
****Copy-list-initialisation considers explicit ctors, too, but doesn't compile if such a ctor is selected

References and Call-by-Reference

- Can define a reference variable that is an alias of another variable (without using pointers)

```
int a = 4;
int & b = a;  // now b and a can be used interchangeably
```

- In C/Java all parameters are “call-by-value”; C++ supports “call-by-reference”

```
void func(A a)          // copies A before calling
void func(A &a)         // does not copy and can change
void func(const A &a)   // does not copy and does not change
void func(A *a)          // passes a copy of a pointer
```

- Unintended copies can be expensive!

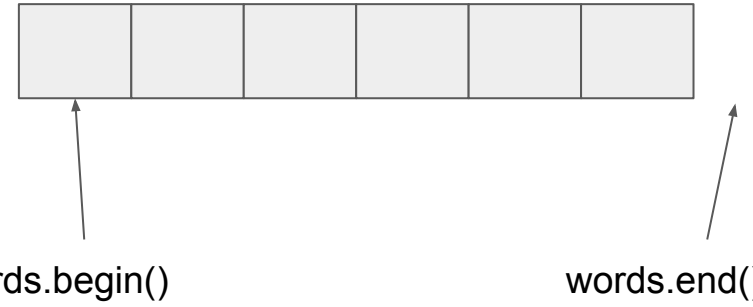
Operator Overloading

- Operators (e.g., +, <<, [], etc.) are like infix representations of function calls
- Think of `cout << “hello”` as `operator<<(cout, “hello”)`
- The meaning of these operators can be defined, or “overloaded” by the programmer
- Used in the standard library, e.g.
  - iostream (cout, cin)
  - subscript operator [] for containers
  - operator ++, * for iterators

Generic Programming via Templates

- Generic Programming is an abstraction technique
  - E.g. write code that works for all types to which an algorithm might be applied
  - While allowing for specializations for certain types
- In C++, generic programming is directly supported by the compiler, resulting in zero overhead when templates are instantiated
- C++ templates are far richer than Java's
  - E.g. support scalar types, constexpr, variadic templates
- Use of the standard library requires an understanding of how templates are instantiated

STL Example 1




```
#include <vector>
#include <iostream>

int
main()
{
    std::vector<std::string> words;
    std::string s;
    while (std::cin >> s)
        words.push_back(s);

    for (auto it = words.begin(); it != words.end(); it++) {
        auto & word = *it;
        std::cout << word << std::endl;
    }
}
```

words.begin() is an iterator, which is a generalized pointer representing a location in a data structure.
A pair of iterators is called a range.

STL Example 1 (with range-based for loop)

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

int
main()
{
    std::vector<std::string> words;
    std::string s;
    while (std::cin >> s)
        words.push_back(s);

    sort(words.begin(), words.end());
    for (auto & word : words)
        std::cout << word << std::endl;
}
```

Type Coercions

- How does `while (std::cin >> s) { … }` work?
- The compiler looks for opportunities to convert a type A into an (expected) type B. Here, std::istream provides a user-defined conversion operator ‘bool’ to convert to a boolean
- Conversely, constructors are used to create objects
  - E.g. string s; s = “a”;
- The compiler’s tendency to do so can be restricted via ‘explicit’ keyword to avoid unexpected coercions

STL Example 2

Iterators to map are (key, value) pairs, allowing in-place manipulation.

```
#include <map>
#include <iostream>

int
main()
{
    std::map<std::string, int> wordcount;
    std::string s;
    while (std::cin >> s)
        wordcount[s]++;

    for (auto it = wordcount.begin(); it != wordcount.end(); it++) {
        std::cout << it->first << " " << it->second << std::endl;
    }
}
```

STL Example 2

```
#include <map>
#include <iostream>

int
main()
{
    std::map<std::string, int> wordcount;
    std::string s;
    while (std::cin >> s)
        wordcount[s]++;

    for (auto & [word, count] : wordcount)
        std::cout << word << " " << count << std::endl;
}
```

C++17 supports “structured binding declarations”

STL Example 2 with find()

```
#include <map>
#include <iostream>

int
main(int ac, char *av[])
{
    std::map<std::string, int> wordcount;
    std::string s;
    while (std::cin >> s)
        wordcount[s]++;

    auto it = wordcount.find(av[1]);
    if (it != wordcount.end()) {
        std::cout << it->first << " " << it->second << std::endl;
    } else {
        std::cout << av[1] << " not found" << std::endl;
    }
}
```

Use of .find() returns an iterator which eithers points at the “end” or the (key, value) pair

namespaces

- C++ supports multiple namespaces in addition to the global (::) namespace
  - E.g. std, eosio
- uses ADL (Argument Dependent Lookup)

```
#include <algorithm>
#include <utility>

int exchange(int &a, int &b)
{
    std::swap(a, b);
}

int exchange(std::pair<int, int> &a, std::pair<int, int> &b)
{
    swap(a, b); // ADL finds std::swap since a is of type std::pair
}
```

Namespaces, using, and typedef

- `using` keyword brings namespaces or types from a different namespace in scope

```
using namespace std;         // frowned upon, all of std:: is now in scope
using std::vector;  // can say 'vector' instead of 'std::vector'

using namespace eosio;    // brings all of eosio into scope
using eosio::contract; // can say 'contract' instead of 'eosio::contract'

// Type aliases
typedef eosio::contract Contract; // ditto (old-style)
using Contract = eosio::contract; // 'Contract' is an alias for eosio::contract
using pii = std::pair<int, int>;  // pair of int+int
using bet_table = eosio::multi_index<"bets"_n, bets>;
```

`using` to access base class elements

- `using` keyword can also be used to bring protected base class fields or methods into scope

```
#include <queue>

template <typename T>
struct visible_queue : std::priority_queue<T> {
    // expose base class's 'c' element publicly.
    using std::priority_queue<T>::c;
    // expose base class constructor
    using std::priority_queue<T>::priority_queue;
};

int main() {
    visible_queue<int> vq{std::less<int>{}};
}
```

User-defined literals

```
#include <string>
#include <iostream>
#include <boost/io/ios_state.hpp>

struct name
{
    uint64_t code = 0;

    name(const std::string &s) {
        for (auto c : s) {
            code <<= 4;
            auto ch = tolower(c);
            if (ch >= 'a')
                code |= (ch - 'a' + 10);
            else
                code |= (ch - '0');
        }
    }
};
```

```
name operator"" _n (const char *s,
                    size_t len)
{
    return {{s, len}};
}

std::ostream &
operator <<(std::ostream &os,
            const name &n)
{
    boost::io::ios_flags_saver ifs(os);
    return os << n.code << ","
              << std::hex << n.code;
}

int
main()
{
    std::cout << "abcd"_n << std::endl;
}
```

Techniques shown on previous slide

- User-defined literals, e.g. _n
  - Can write “abcd”_n and compile will construct a “name” object
  - Also possible to write 35_km etc.
- RIAA Pattern in output operator
  - Ios flags are restored upon return
- Type coercion for return values and constructor arguments
  - `return {{s, len}};`
  - Needs a name, so equivalent to `name{{s, len}}`
  - Only constructor takes a `string, so string{const char *, int}` is considered.
- Inline initialization for instance fields

Lambda expressions

- Consider this example of Inversion of Control
- A function is passed to another function to be invoked
  - Function needs access to variables that are in scope (here: VAL)

- How can this be implemented?

```
#include <iostream>

struct object { int v; };

template <typename Function>
static void
apply(object &obj, Function fun)
{
    fun(obj);
}

int
main()
{
    object obj;
    int VAL = 42;
    apply(obj, [VAL](object & o) {
        o.v = VAL;
    });
    std::cout << obj.v << std::endl;
}
```

Lambda expressions

- Implemented as functors with overload call operator
- Compiler ensures that copies or references to captured variables are accessible
- Example
  - [x, y] - capture copies of x, y
  - [&x, y] - capture reference to x, copy of y
  - [=] - capture copies of everything
  - [&] - capture references to everything
- Programmer is responsible for safety!

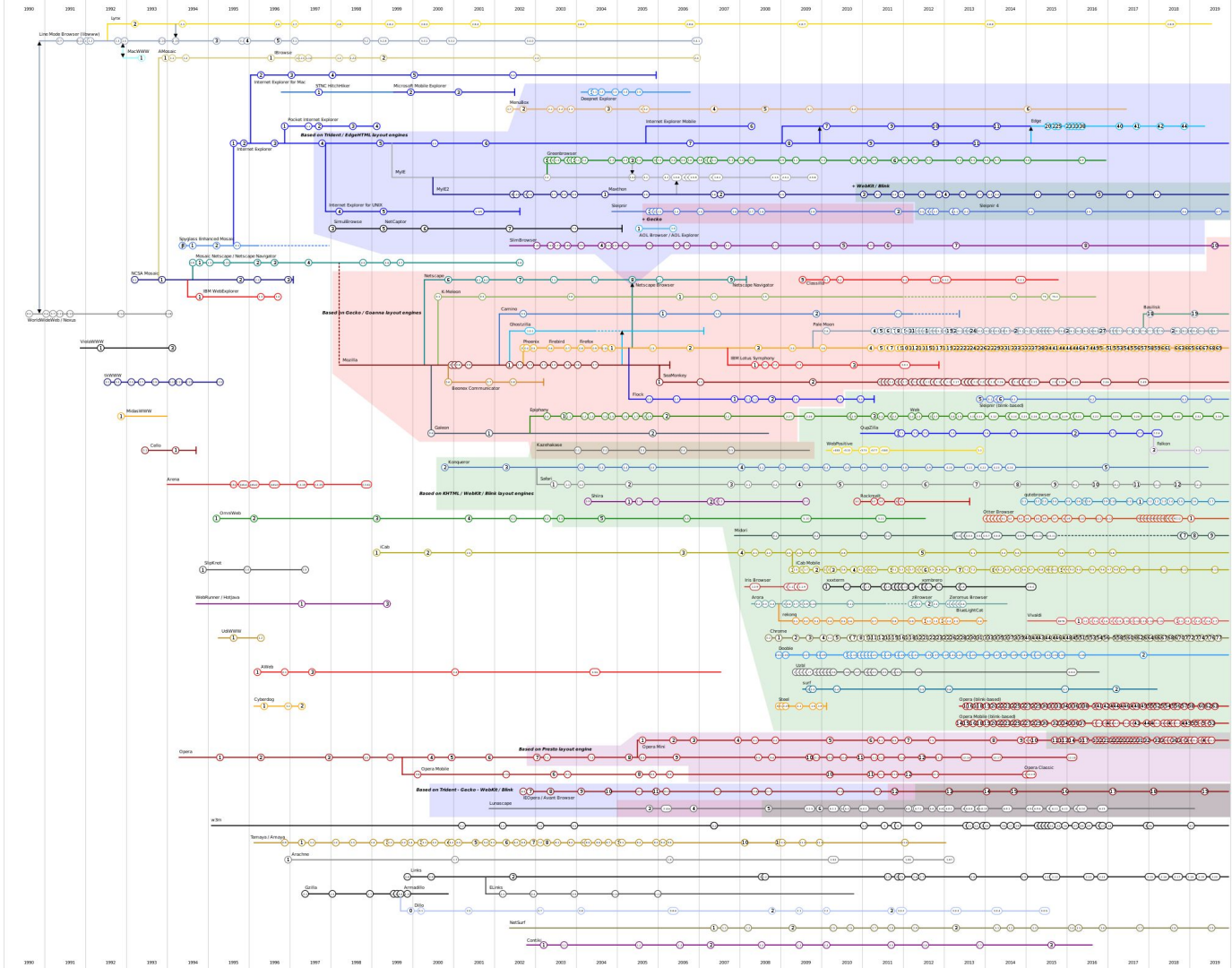
```
int
main()
{
    object obj;
    int VAL = 42;

// compiler generates this
    struct functor {
        int _VAL;
        functor(int _VAL) : _VAL{_VAL} {}

        void operator()(object &o) {
            o.v = _VAL;
        }
    } F(VAL);
// ....
    apply(obj, F);
    std::cout << obj.v << std::endl;
}
```

C++ and WASM

- During the 2nd browser war, high-performing JavaScript engines were created that provided reliable mobile code platforms



asm.js

- 2014 Alon Zakai developed asm.js, a low-level subset of JavaScript that
  - Can be compiled to efficient code by JavaScript engines
  - Can function as a target for C++ compilers (Emscripten)
- Performant enough to run 3D game engines in the browser
- These ideas led to a standardization effort to add direct support for a virtual environment to browser

```
size_t strlen(char *ptr) {
  char *curr = ptr;
  while (*curr != 0) {
    curr++;
  }
  return (curr - ptr);
}
```

```
function strlen(ptr) {
  ptr = ptr|0;
  var curr = 0;
  curr = ptr;
  while ((MEM8[curr>>0]|0) != 0) {
    curr = (curr + 1)|0;
  }
  return (curr - ptr)|0;
}
```

Source: https://en.wikipedia.org/wiki/Asm.js

WebAssembly (WASM)

- WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.
- Compilation target for clang
- Direct support added in major browsers
- Standalone virtual machines exist, e.g. binaryen

```
int32_t
add_2_integers(int32_t a, int32_t b)
{
    return a + b;
}
```

```
(func $1 (; 2 ;) (type $3)
  (param $0 i32)
  (param $1 i32) (result i32)
  (i32.add
   (local.get $0)
   (local.get $1)
  )
 )
```

Deterministic Execution in WASM

- Implementing a replicated state machine such as in a network of blockchain nodes requires safe and deterministic execution
- WebAssembly aims to provide this
  - Even if source language (C++) allows for undefined behavior (!)
- Safety through the use of a sandbox with a linear memory model
- Determinism by specification with well-defined exceptions:
  - Certain floating point operations (e.g. NaN propagation as per IEEE-754)
  - Resource exhaustion
  - Host functions
- When used in a blockchain environment, these remaining sources of nondeterminism must be addressed
  - E.g. use of software floating point with implementation-defined behavior