Luxoft
A DXC Technology Company

Towards Safe C++

Type safety and safety critical domain challenges

Luxoft
A DXC Technology Company

Introduction

Safety

Safety is about eliminating unpredictability from the program, and that of course also increases level of security.

Safety and security are related. Both are about creating software that is free from constructs that lead to unpredictable behavior.

— Miroslave Zielinski, Parasoft

Introduction

Safety

"Understanding why software fails is important, but the real challenge is understanding why software works."

– Alexander Stepanov

Introduction

How to understand a C++ program

The **C++ abstract machine**: is a portable abstraction of your operating system, kernel and hardware. The abstract machine is the intermediary between your C++ program and the system that it is run on."

C++ programs describe operations that are performed on the abstract machine.
C++ implementations define the characteristics of the abstract machine and translate operations on the abstract machine into operations on the system.

- Bryce Adelstein Lelbach, Core C++ 2019 The C++ Execution Model

Introduction

How to understand a C++ program

C++ code → Abstract Machine

Binary code → Real Machine

Introduction

Observable behavior and Abstract Machine

well-formed program C++ program constructed according to the syntax rules, diagnosable semantic rules.

Implementation defined behavior the behavior of the program varies between implementations(for a well-formed program construct and correct data), and the conforming implementation must document the effects of each behavior.

- size of std::size_t

Introduction

Observable behavior and Abstract Machine

Unspecified behavior the behavior of the program varies between implementations(for a well-formed program construct and correct data), and the conforming implementation is not required to document the effects of each behavior

- order of evaluation of function parameter
- floating-point bit pattern that constitutes signaling and quiet nan.

Introduction

Observable behavior and Abstract Machine

Undefined behavior behavior for which this Standard imposes no requirements

- access to an object through a pointer of a different type
- access un-initialized variable
- access an object outside of it's lifetime
- access to non-active member of a union

Compilers are not required to diagnose undefined behavior(some simple situations are diagnosed)

Introduction

Observable behavior and Abstract Machine

```
1  #include <algorithm>
2  #include <iostream>
3  int main() {
4      //const auto p = std::minmax({1, 2}); // ok
5      //const auto& p = std::minmax({1, 2}); // ok
6      const auto p = std::minmax(1, 2); // -> dangling
7      std::cout << p.first << "\n";
8      return 0;
9  }
```

```
<source>: In function 'int main()':
<source>:6:16: warning: possibly dangling reference to a temporary [-Wdangling-reference]
    6 |     const auto p = std::minmax(1, 2); // -> dangling reference
      |                ^
<source>:6:31: note: the temporary was destroyed at the end of the full expression 'std::minmax<int>(1, 2)'
```

Introduction

Observable behavior and Abstract Machine

```
1  #include <iostream>
2
3  int main()
4  {
5      int* ptr;
6      std::cout << *ptr << "\n";
7      return 0;
8  }
```

```
<source>: In function 'int main()':
<source>:6:26: warning: 'ptr' is used uninitialized [-
Wuninitialized]
    6 |     std::cout << *ptr << "\n";
      |                          ^~~~
<source>:5:10: note: 'ptr' was declared here
    5 |     int* ptr;
      |          ^~~
```

Introduction

Type in C++

the entity that's associated with **objects**, **functions**, **references** and **expressions**, which restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits.

source: www.cppreference.com

Introduction

Type safety

Type safety means that you use the types correctly and, therefore, avoid unsafe operations i.e. program constructs that alters incorrectly the type, hence affects the entities that are associated with the type(objects, object lifetime).

Introduction

Types

Types in C++ are classified:

- **Fundamental**: int, float, bool, enums
- **Compound**: arrays, pointers, references
- **Class** types(user-defined type): class, struct and union

Introduction

Object in C++

- **Type**
- **Size**
- **Storage** duration (automatic, static, dynamic, thread-local)
- **Lifetime** (bounded by storage duration or temporary)
- **Alignment** requirement
- **Value**
- **Name/identifier**

Introduction

Type lifetime

C++’s memory model of object management is based on the object lifetime i.e. starts with constructor and ends with **destructor**. **constructors** specify the meaning of object initialization and **destructors** define the object cleanup.

For **automatic** objects, destruction is implicit executed at the end of the scope.

For **dynamic** objects placed in the (heap, dynamic memory) using new, delete is required.

Introduction

Type lifetime

- **new** and placement **new**
- **malloc**

Introduction

Object lifetime

```cpp
#include <cstring>
#include <iostream>

struct Foo{
   Foo (const int& val):x(val){ std::cout << "C'Tor\n";}
   ~Foo (){ std::cout << "D'Tor\n";}
   int x;
};

int main() {
    void* pf = std::malloc(sizeof(Foo)); // allocates memory
    Foo* f = reinterpret_cast<Foo*>(pf); // UB
    new(f) Foo{5}; // starts lifetime of object Foo
    f->~Foo(); // pseudo d'tor, std::destroy_at(f);
    free(f); // frees up allocated memory
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 C'Tor
 D'Tor
```

Use cases

SDV

Automotive Industry

Software-Defined Vehicle:

A Software-Defined Vehicle is a vehicle that’s defined, evolves and enhanced by the software, throughout its lifetime. This includes features, operations and functionality.

SDV

Automotive Industry

Software-Defined Vehicle:

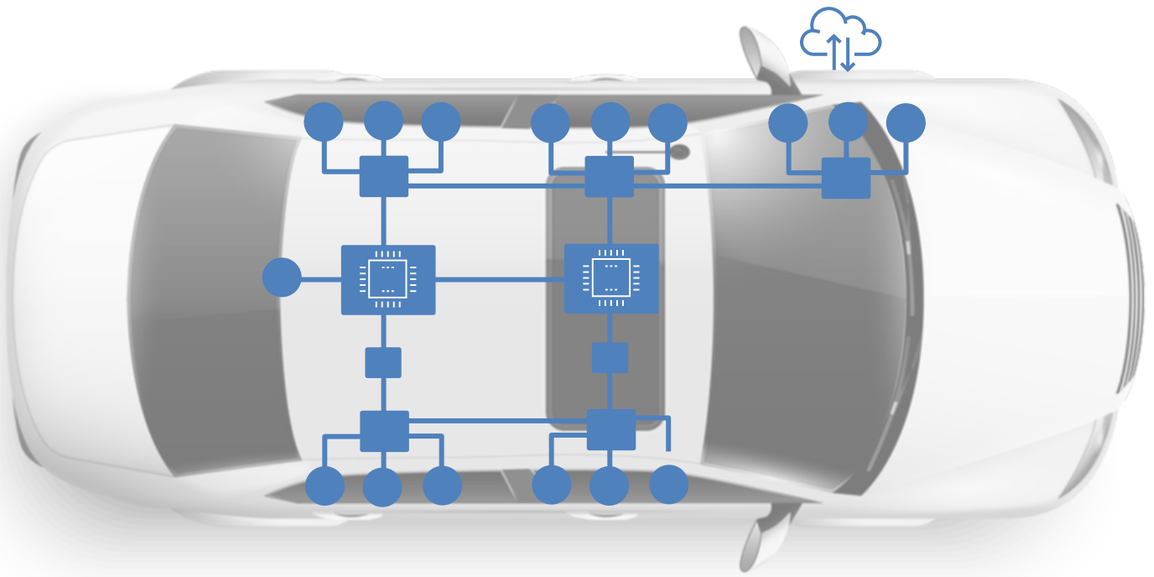
SDV car has around 70 computer(ECU)s, each one has a designated functionality such as ADAS, access, lighting, infotainment

SDV

Automotive Industry

Software-Defined Vehicle:

- Characterized by high exchange of data/messages
- These messages/events/data are exchanged over network
- Applications are performing serialization/deserialization for tons of data every single second.



Centralized E/E

Serialization Deserialization

Definition

Serialization is the process of translating a data structure or object state into a format that can be stored (e.g. files in secondary storage devices, data buffers in primary storage devices) or transmitted (e.g. data streams over computer networks) and **reconstructed later** (possibly in a different computer environment).

When the resulting series of bits is **reread** according to the serialization format, it can be used to create a **semantically identical clone** of the **original** object.

Source: https://en.wikipedia.org/wiki/Serialization

Serialization Deserialization

What about Endianess

- We need to check byte order first
- Computers(x86, ARM) use little endian byte order
- Network uses big endian byte order

Byte order swapping

How to swap byte order (classic version)

- Swap endianess
- Also know by from network to host

Byte order swapping

From Stackoverflow questions how-to-convert-big-endian-to-little-endian-in-c-without-using-library-functions

```
3  template <typename T>
4  T changeByteOrder(const T &u) // std::uint16_t x = 0b0000'0000'0000'0001u;
5  {
6    union Union
7    {
8      T t;
9      unsigned char bytes[sizeof(T)];
10   };
11
12   Union original, swapped; // no active members yet
13   original.t = u; // original.t is active
14
15   for (std::size_t i = 0; i < sizeof(T); i++)
16     swapped.bytes[i] = original.bytes[sizeof(T) - i - 1u]; // original.bytes is not active
17
18   return swapped.t; // reading non-active t member
19 } // swapped.t = 0b0000'0001'0000'0000u;
```

Byte order swapping

How to swap byte order (C)

Using union

- Accessing union’s non-active member
- Accessing object’s underlying bytes

Byte order swapping

How to swap byte order (C)

Using union

- Accessing union’s non-active member → undefined behavior
- Accessing object’s underlying bytes → undefined behavior

Byte order swapping

How to swap byte order (C)

Using union

9.5 Unions **[class.union]**

1 In a union, at most one of the non-static data members can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time. [*Note:* One special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (9.2), and if an object of this standard-layout union type

Byte order swapping

How to swap byte order (C)

Using union

Every object and reference has a *lifetime*, which is a runtime property: for any object or reference, there is a point of execution of a program when its lifetime begins, and there is a moment when it ends.

The lifetime of an object begins when:

- storage with the proper alignment and size for its type is obtained, and
- its initialization (if any) is complete (including default initialization via no constructor or trivial default constructor), except that
  - if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union, or it is made active,
  - if the object is nested in a union object, its lifetime may begin if the containing union object is assigned or constructed by a trivial special member function,
  - an array object's lifetime may also begin if it is allocated by `std::allocator::allocate`.

Source: cppreference.com

Byte order swapping

How to swap byte order (C)

Using union

- Accessing underlying bytes is not standardized yet, didn’t make it to the C++20/C++23.
- “Accessing object representations” - **CWG** C++26

Byte order swapping

Accessing object representations

Timur Doumler (papers@timur.audio)
Krystian Stasiowski (sdkrystian@gmail.com)

Document #: P1839R5
Date: 2022-06-16
Project: Programming Language C++
Audience: Core Working Group

Abstract

This paper proposes a wording fix to the C++ standard to allow read access to the object representation (i.e. the underlying bytes) of an object. This is valid in C, and is widely used and assumed to be valid in C++ as well. However, in C++ this is is undefined behaviour under the current specification.

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1839r5.pdf

Byte order swapping

1 Motivation

Consider the following program, which takes an `int` and prints the underlying bytes of its value in hex format:

```
void print_hex(int n) {
  unsigned char* a = (unsigned char*)(&n);
  for (int i = 0; i < sizeof(int); ++i)
  printf("%02x ", a[i]);
}

int main() {
  print_hex(123456);
}
```

undefined behavior

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1839r5.pdf

Byte order swapping

std::memcpy to the rescue

- It can replace places when using reinterpret_cast is considered UB
  - e.g. std::bit_cast C++20

- It can replace places that need to have access to object representations(underlying bytes)
  - by first copying the bytes from object of type T(trivially copyable) to an array of bytes of sizeof(T)

From bytes to types

Also...

- **std::memcopy** doesn't start the **lifetime** of objects(till C++20).
- using **std::memcpy** to copy bytes from object of type T1 to another object of T2, where they're trivially copyable and they share same size only, with no value corresponding between them, the behavior is undefined.

Byte order swapping

3.9 Types [basic.types]

1 [*Note:* 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (1.8), references (8.3.2), or functions (8.3.5). — *end note*]

2 For any object (other than a base-class subobject) of trivially copyable type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes (1.7) making up the object can be copied into an array of `char` or `unsigned char`.[42] If the content of the array of `char` or `unsigned char` is copied back into the object, the object shall subsequently hold its original value. [*Example:*

```
#define N sizeof(T)
char buf[N];
T obj;                          // obj initialized to its original value
std::memcpy(buf, &obj, N);      // between these two calls to std::memcpy,
                                // obj might be modified
std::memcpy(&obj, buf, N);      // at this point, each subobject of obj of scalar type
                                // holds its original value
```

C++ standard working draft paper

Byte order swapping

```
5  template <typename T> // assuming T is trivially copyable
6  T changeByteOrder(const T &val) { // std::uint16_t x = 0b0000'0000'0000'0001u;
7    struct aligned_buffer {
8      alignas(T) std::uint8_t data[sizeof(T)];
9    } original{}, swapped{};
10
11   std::memcpy(&original.data, &val, sizeof(T));
12
13   for (std::size_t k = 0u; k < sizeof(T); k++)
14     swapped.data[k] = original.data[sizeof(T) - k - 1u];
15
16   T result{};
17   std::memcpy(&result, &swapped.data, sizeof(T));
18   return result; //0b0000'0001'0000'0000u;
19 }
```

From bytes to types

std::memcpy should be used wisely !!

From bytes to types

One more thing

Class Invariants:

- The values of the members and the objects referred to by members are collectively called the state of the object (or simply, its value).
- A major concern of a class design is to get an object into a well defined state (initialization/construction), to maintain a well defined state as operations are performed, and finally to destroy the object gracefully.
- The property that makes the state of an object well-defined is called its invariant.

Bjarne Stroustrup 'The C++ Programming Language'

From bytes to types

```
3  #include <cassert>
4  enum class Color:uint8_t{RED =100u, ORANGE = 110u, GREEN = 120u};
5
6  class TrafficLight{
7   public:
8     void setLight(const Color& color){color_ = color;}
9     Color getLight(){return color_;};
10  private:
11    Color color_;
12 };
13
14 int main() {
15     TrafficLight light;
16     light.setLight(Color::RED);
17     std::memset(&light, 0U, sizeof(light));
18     return 0;
19 }
```

violates the class invariants

From bytes to types

```
4  struct RefInt{
5      RefInt(const int val):x(val){}
6      const int x;
7  };
8
9  int main() {
10     int x = 333; int y = 0;
11     RefInt ref_int_x{x};
12     RefInt ref_int_y{y};
13     std::memcpy(&ref_int_y, &ref_int_x, sizeof(RefInt));
14     return 0;
15 }
```

violates the class invariants

From bytes to types

gcc introduced

Wclass-memaccess

Warn when the destination of a call to a raw memory function such as **memset** or **memcpy** is an object of **class type**, and when writing into such an object might **bypass** the class **non-trivial** or deleted **constructor** or **copy assignment**, violate **const-correctness** or encapsulation, or corrupt virtual table pointers.
Modifying the representation of such objects may violate **invariants** maintained by member functions of the class.

https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html#index-Wclass-memaccess

From bytes to types

gcc introduced

Wclass-memaccess

```
<source>:13:16: warning: 'void* memcpy(void*, const void*, size_t)' writing to an object of
type 'struct RefInt' with no trivial copy-assignment; use copy-initialization instead [-
Wclass-memaccess]
   13 |     std::memcpy(&ref_int_y, &ref_int_x, sizeof(RefInt));
      |     ~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:4:8: note: 'struct RefInt' declared here
    4 | struct RefInt{
      |        ^~~~~~
```

From bytes to types

Finally...

- **std::memcopy**, **std::memset**, **std::memcmp** perform their operation on the **Object Represenation** of an object of type **T**.

From bytes to types

Object representation and value representation

— *end example*]

4 The *object representation* of an object of type `T` is the sequence of N `unsigned char` objects taken up by the object of type `T`, where N equals `sizeof(T)`. The *value representation* of an object is the set of bits that hold the value of type `T`. For trivially copyable types, the value representation is a set of bits in the object representation that determines a *value*, which is one discrete element of an implementation-defined set of values.[44]

From bytes to types

```
 7  struct Foo
 8  {
 9      std::uint16_t x;
10      std::uint16_t y;
11      // 4-bytes padding
12      std::size_t capacity;
13  }; //alignment: 8 byte(s) //sizeof: 16 byte(s)
14
15  int main()
16  {
17      Foo f1{0U, 0U, 0U};
18      Foo f2{0U, 0U, 0U};
19
20      const auto result = std::memcmp(&f1, &f2, sizeof(Foo));
21      assert(result == 0); // assertion fails
22      return 0;
23  }
```

- Value representation is different from object representation
- 4 bytes difference due to padding
- These padding bytes are not equal among different instances.

Assertion fails

From bytes to types

```
5  struct Foo{
6      std::uint16_t x; std::uint16_t y;// 4-bytes padding
7      std::size_t capacity;
8      bool operator ==(const Foo& other) const {
9          return (x == other.x) && (y == other.y) && (capacity == other.capacity)
10     }
11 };
12 int main() {
13     Foo f1{0U, 0U, 0U}; Foo f2{0U, 0U, 0U};
14     const auto result = (f1 == f2);
15     assert(result == true);
16     return 0;
17 }
```

comparison operator

assertion passes

From bytes to types

When use std::memcpy, std::memset, std::memcmp

- Need to distinguish between object representation and value representation
- Need to pay attention to class invariants.
- Still may result in unsafe operations

Questions

©10.6.23Luxoft, A DXC Technology Company. All rights reserved.