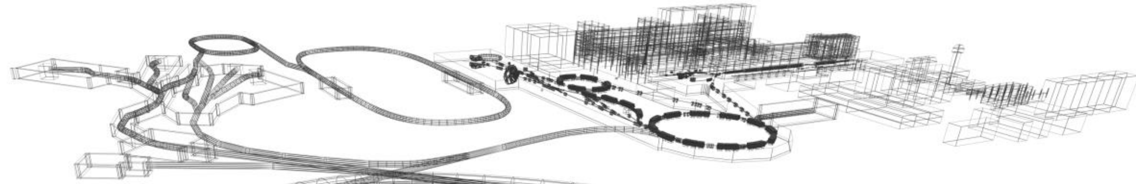
Read the Hardware

Matteo Alfonsi (GSI – System Planning SFRS)



Read a program that **Reads the Hardware** and get how, some time ago, you (or somebody else) did it

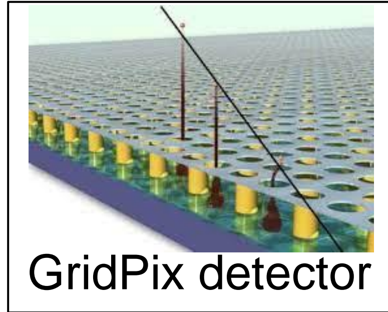
FAIR

Read a program that
Reads the Hardware
and get how, some time ago,
ou (or somebody else) did it

Read a program that
Reads the Hardware
and get how, some time ago,
you (or somebody else) did it

despite

1) Type punning

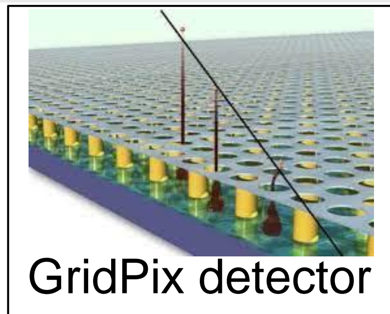

GridPix detector

```
struct TimepixHit {
  uint16_t t;
  uint8_t x;
  uint8_t y;
} hit;
```

1) Type punning

FAIR GSI

```
union {
  uint32_t raw_bits;
  struct {
    uint16_t t;
    uint8_t x;
    uint8_t y;
  } timepix_hit;
} data;

data.raw_bits = TimepixReadWord();

std::cout << data.timepix_hit.x << "\t"
<< data.timepix_hit.y << "\t"
<< data.timepix_hit.t << std::endl;
```



GridPix detector

```
struct TimepixHit {
  uint16_t t;
  uint8_t x;
  uint8_t y;
} hit;

uint32_t* raw_bits =
reinterpret_cast<uint32_t*>(timepix_hit);

*raw_bits = TimepixReadWord();

std::cout << hit.x << "\t" << hit.y
<< "\t" << hit.t << std::endl;
```

2) Reading binary files

```
struct LecroyBinaryHeader {
  char template_name[16];
  uint16_t comm_type;
  uint16_t comm_order;//18_19
  uint32_t size_descriptor_block;
  uint32_t size_usertext_block;
  uint32_t padding28_31;
  uint32_t size_trigtime_array;
[…]
  uint16_t wave_source; //328_329
  uint16_t undesired_padding330_331;
  uint32_t undesired_padding332_335;
} header;

static_assert
( sizeof(LecroyBinaryHeader) == 336 )
```

- Data structures from devices are always troublesome because of possible different padding from machine to machine

2) Reading binary files

```
struct LecroyBinaryHeader {
  char template_name[16];
  uint16_t comm_type;
  uint16_t comm_order;//18_19
  uint32_t size_descriptor_block;
  uint32_t size_usertext_block;
  uint32_t padding28_31;
  uint32_t size_trigtime_array;
[…]
  uint16_t wave_source; //328_329
  uint16_t undesired_padding330_331;
  uint32_t undesired_padding332_335;
} header;

static_assert
( sizeof(LecroyBinaryHeader) == 336 )
```

```
fstream_wf.read(
  reinterpret_cast<char*>(&header),
  sizeof(header) );

[…]
std::vector<SampleType> samples;
[…]
samples.resize( header.NbSamples() );
char* address =
  reinterpret_cast<char*>(samples.data());
fstream_wf.read( address,
  samples.size()*sizeof(SampleType) );
```

3) Bit fields, signed/unsigned

FAIR GSI

```
template <typename T, unsigned int nLSbits, unsigned int... moreFields>
constexpr auto SplitBitfield(T bitfield) noexcept {
  static_assert( std::is_integral<T>::value ); static_assert( std::is_unsigned<T>::value );
  static_assert( nLSbits <= (sizeof(T)<<3) ); static_assert( nLSbits > 0U );

  if constexpr( sizeof...(moreFields) == 0 ) {
    return std::make_tuple( T{bitfield & mask} );
  } else {
    constexpr unsigned int mask = (1U << nLSbits) -1 ;
    return std::tuple_cat( std::make_tuple( T{bitfield & mask} ),
                           SplitBitfield<T, moreFields...>(bitfield >> nLSbits) );
  }
}
```

- Compliers have some implementation freedom with bit shift on signed integers.

3) Bit fields, signed/unsigned

```
template <typename T, unsigned int nLSbits, unsigned int... moreFields>
constexpr auto SplitBitfield(T bitfield) noexcept {
  static_assert( std::is_integral<T>::value ); static_assert( std::is_unsigned<T>::value );
  static_assert( nLSbits <= (sizeof(T)<<3) ); static_assert( nLSbits > 0U );

  if constexpr( sizeof...(moreFields) == 0 ) {
    return std::make_tuple( T{bitfield & mask} );
  } else {
    constexpr unsigned int mask = (1U << nLSbits) -1 ;
    return std::tuple_cat( std::make_tuple( T{bitfield & mask} ),
                           SplitBitfield<T, moreFields...>(bitfield >> nLSbits) );
  }
}
```

```
auto DecodeEvent(UInt_t* datawords_ptr, UInt_t* datawords_end) {
[…]
auto [ev_counter, trailer_tag, vme_geo] = SplitBitfield<UInt_t, 24, 3, 5>( *datawords_ptr );
[…]
```

3) Bit fields, signed/unsigned

FAIR GSI

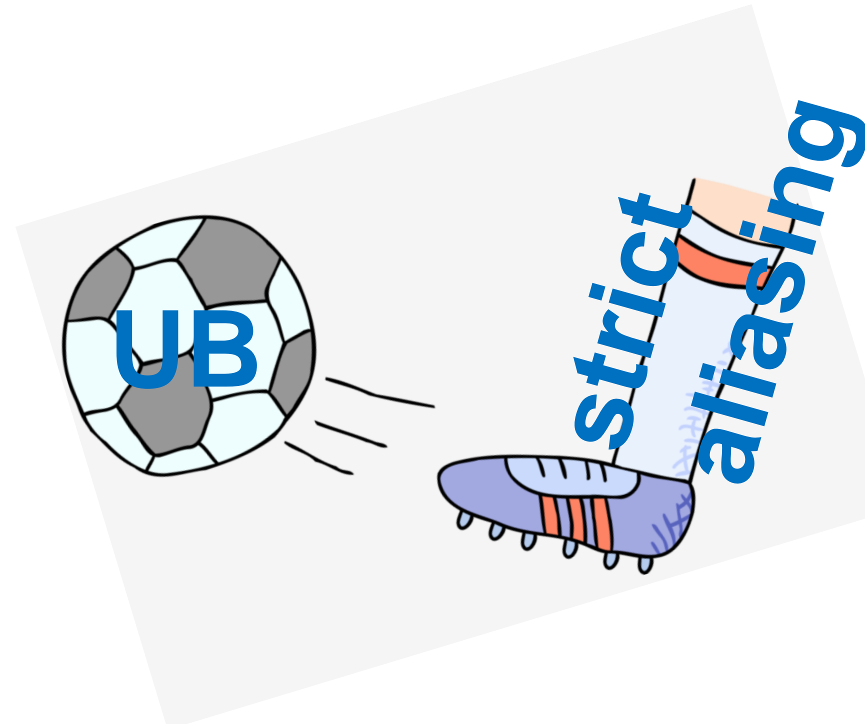
```
template <typename T, unsigned int nLSbits, unsigned int... moreFields>
constexpr auto SplitBitfield(T bitfield) noexcept {
  static_assert( std::is_integral<T>::value ); static_assert( std::is_unsigned<T>::value );
  static_assert( nLSbits <= (sizeof(T)<<3) ); static_assert( nLSbits > 0U );

  if constexpr( sizeof...(moreFields) == 0 ) {
    return std::make_tuple( T{bitfield & mask} );
  } else {
    constexpr unsigned int mask = (1U << nLSbits) -1 ;
    return std::tuple_cat( std::make_tuple( T{bitfield & mask} ),
                           SplitBitfield<T, moreFields...>(bitfield >> nLSbits) );
  }
}

auto ptr_data = reinterpret_cast<UInt_t*>( mbssubevent->GetDataField() ); //Int_t* from MBS

auto DecodeEvent(UInt_t* datawords_ptr, UInt_t* datawords_end) {
[…]
auto [ev_counter, trailer_tag, vme_geo] = SplitBitfield<UInt_t, 24, 3, 5>( *datawords_ptr );
[…]
```

Are all of them proper C++?

Type punning in (1)

Binary files readout in (2)

Signed/unsigned in (3)



Type punning (1) is UB

```
union {
  uint32_t raw_bits;
  struct {
    uint16_t t;
    uint8_t x;
    uint8_t y;
  } timepix_hit;
} data;

data.raw_bits = TimepixReadWord();

std::cout << data.timepix_hit.x << "\t"
<< data.timepix_hit.y << "\t"
<< data.timepix_hit.t << std::endl;
```

- Access to not active member of a union is UB in C++
- However this is still legal in C
- There are exceptions worth to be learned (see references later)

Type punning (1) is UB

- UB both in C and C++
- Aliasing allowed only between "similar types", check references to learn the rules - enjoy ;-)
- There are differences between the rules in C and the rules in C++

```
struct TimepixHit {
  uint16_t t;
  uint8_t x;
  uint8_t y;
} hit;

uint32_t* raw_bits =
reinterpret_cast<uint32_t*>(timepix_hit);

*raw_bits = TimepixReadWord();

std::cout << hit.x << "\t" << hit.y
<< "\t" << hit.t << std::endl;
```

Cast to char* in (2) allowed

```
struct LecroyBinaryHeader {
```

- Aliasing to char, unsigned char or std::byte always allowed, but…
  …enjoy in the detailed rules what you can and you cannot do with this pointer to bytes.

- The other way around is UB

```
static_assert
( sizeof(LecroyBinaryHeader) == 336 )
```

```
fstream_wf.read(
  reinterpret_cast<char*>(&header),
  sizeof(header) );

[…]
std::vector<SampleType> samples;
[…]
samples.resize( header.NbSamples() );
char* address =
  reinterpret_cast<char*>(samples.data());
fstream_wf.read( address,
  samples.size()*sizeof(SampleType) );
```

Signed to unsigned in (3) OK

```
template <typename T, unsigned int nLSbits, unsigned int... moreFields>
constexpr auto SplitBitfield(T bitfield) noexcept {
  static_assert( std::is_integral<T>::value ); static_assert( std::is_unsigned<T>::value );
```

- Aliasing to the corresponding unsigned (cv-qualified or not) type is allowed, in C as well as C++

- From read sources, also the other way around should be OK.

```
  }
}
```

```
auto ptr_data = reinterpret_cast<UInt_t*>( mbssubevent->GetDataField() ); //Int_t* from MBS

auto DecodeEvent(UInt_t* datawords_ptr, UInt_t* datawords_end) {
[…]
auto [ev_counter, trailer_tag, vme_geo] = SplitBitfield<UInt_t, 24, 3, 5>( *datawords_ptr );
[…]
```

Sources

Systematic discussions:

1. **Type punning in modern C++ - Timur Doumler - CppCon 2019**
   https://www.youtube.com/watch?v=_qzMpk-22cc
2. **What is the Strict Aliasing Rule and Why do we care? – shafik on github**
   https://gist.github.com/shafik/848ae25ee209f698763cffee272a58f8

Specific discussions:

- Just google and you will get a jungle of specific threads on StackOverflow, Reddit…

Why do these rules exists?

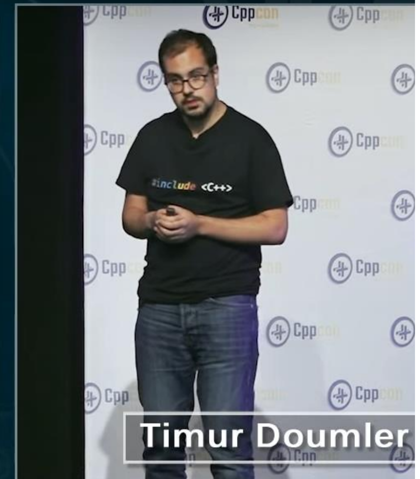
- Refer to the CppCon Talk [1]:

Type punning in modern C++ - Timur Doumler - CppCon 2019

...but why?

- aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

36



Type punning
in modern C++

So what should I do?

- … `std::memcpy()` !!!

```
float f = 43.f;
uint32_t d;
static_assert( sizeof(float) == sizeof(uint32_t) );
//and types should be std::is_trivially_copyable
std::memcpy( &d , &f, sizeof(d) );
```

- This is really weird, because *you are copying* data !!!
- However, all read sources reassure that, at any typical optimization level, the optimizer immediately catches the pattern and optimize away the copy

So what should I do if C++20 ?

- … `std::bit_cast<>()` !!!

```
//included in bit_cast: static_assert( sizeof(float) == sizeof(uint32_t) );
// and check that types are std::is_trivially_copyable
uint32_t d = std::bit_cast< uint32_t >( 43.f );
```

- You do not feel anymore that you are copying data – though, without optimizations, you are probably just calling `std::memcpy` (confusing info here)

- Bonus: `std::bit_cast<>()` is `constexpr`

More to address & to come with C++23

- As of CppCon Talk [1], many real-life cases are not yet properly covered, in particular due to the object lifetime approach.
- For C++23, **`std::start_lifetime_as<>()`** proposed to extend the options that starts object lifetime with the case of assigning it to a bunch of memory.

```
void process(Stream* stream)
{
    std::unique_ptr<char[]> buffer = stream->read();

    if (buffer[0] == WIDGET)
        processWidget(std::start_lifetime_as<Widget>(buffer.get()));  // new in P0593
    else
        // ...
}
```

- From now on compiler assumes this type without calling any constructor, and this should work for any **`std::is_trivially_constructible`** object

Summary

- Dealing with hardware implies dealing with bit patterns and data representation in memory; worth to invest some effort in readability of your code
- More important: strict aliasing changes the rules with which you was used with type punning… How many are aware of them ??
- Many "approaches that have always worked" – and still works and very likely will work for more years – are, de facto, Undefined Behaviour
- How many are aware of this ?? How many physicists ??
- Modern C++ uses `std::memcpy()`, `std::bit_cast<>()` in C++20, and more tools are coming for proper type punning.
- Still, I personally found the situation quite confusing, and quite complicate for physicists that are asking just for simple tools to perform their data analysis and/or read out their measurement devices…

Summary (2)

r/cpp Search Reddit

johannes1971 · 2 yr. ago

Just an observation, but I find it amazing and shameful that C++ makes it so hard to correctly read data from a binary file that a forum of C++ experts cannot figure out how to do it and end up giving lots of conflicting advice. This is not an uncommon task, and it should be trivially simple for a low-level language like C++.

That it is so hard to get right, while there are so many ways to write it that are officially "wrong" but do actually work, is, in my opinion, a complete failure of language design.

39 Reply Share Report Save Follow

- I do not share the final conclusion, but I share the frustration…