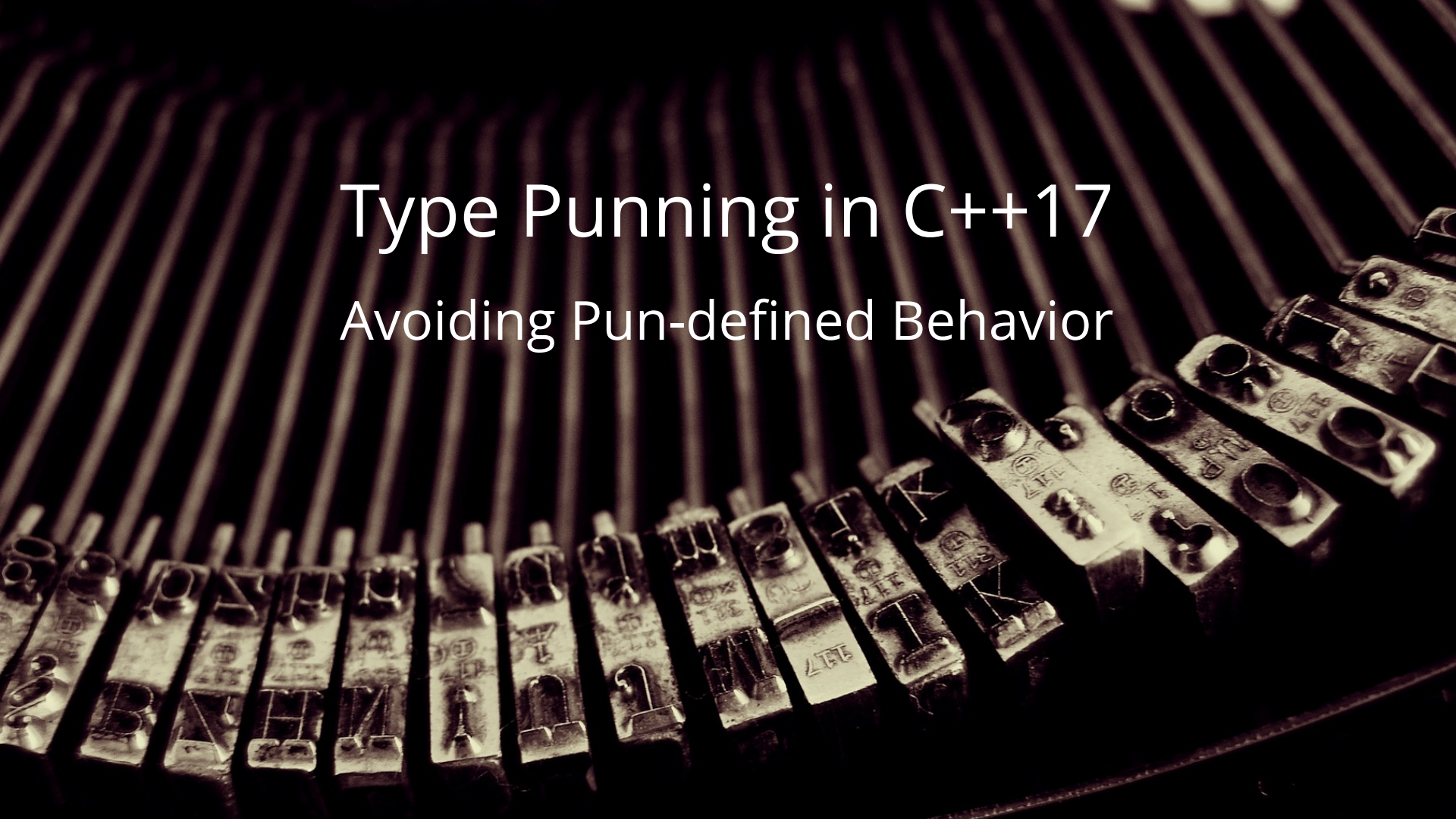


Type Punning in C++17

Avoiding Pun-defined Behavior

Type Punning in C++17

Avoiding Pun-defined Behavior

By Scott Schurr for Ripple - September 2017

Type Punning ...a common term for any programming technique that subverts or circumvents the type system of a programming language...

—Wikipedia August 2017

… In C++17

- Quotes from “The C++17 Standard” are actually from N4660, not from the final standard.

- Quotes from “The C11 Standard” are actually from N1570, not from the final standard.

- Code samples were compiled and executed with Clang and occasionally GCC.

Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary

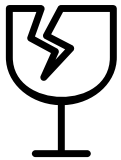

Motivations

Why Type Pun?

- Endian conversions
- Serializing between structs and unsigned char[]
- Bit manipulations in pointers
  - Testing alignment
  - Storing bits in unused portions of pointers
- Operating on internals of floating point numbers

Type Punning Dangers?

- Not portable: Relies on platform dependent representation



- Fragile: Changing compilers or compiler flags can produce different results

- Hard to maintain: Unusual techniques surprise maintainers

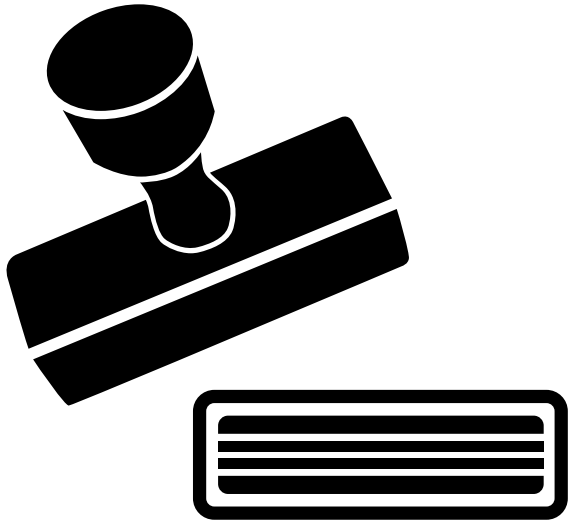
- Bug prone: Often run afoul of undefined behavior

Focus: Trivially Copyable Types

Type punning **non**-trivially copyable types is

- Seldom smart
- Usually buggy

Let’s not even go there



Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary


Definitions:
Trivially Copyable Types

Value Representation

[basic.types] § 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`.

Trivially Copyable Type

[basic.types] § 4

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.

Use `static_assert(std::is_trivially_copyable<T>::value)`

Trivially Copyable Type Rules

- Every copy and move constructor is trivial or deleted
- Every copy and move assignment operator is trivial or deleted
- At least one copy and/or move is not deleted
- Trivial non-deleted destructor
- No virtual members
- No virtual base classes
- Every subobject must be trivially copyable

Trivially Copyable Examples

```
char a;
char b[5];
struct s1 {char c[5]; int d;};
class c1 : public s1 {
    protected:
        int a;
    public:
        c1():a(7){}
};
union u1 {s1 s; c1 c;};
c1 a1[7];
```

Not Trivially Copyable

```
char& r1(a);                    // Reference

struct s2 {                     // User supplied copy ctor
    s2 (s2 const& s):ch('q') {}
    char ch;
};

class c2 {
    virtual void f() {}    // Virtual member
};
```

Definitions:
Types Of Behavior

Implementation-Defined Behavior

[defns.impl.defined]

behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

Implementation-Defined Behavior Examples

— Number of bits in a byte. See [intro.memory] § 1

— Which (if any) atomics are alway lock free. See [atomic.types.generic] § 4

Unspecified Behavior

[defns.unspecified]

behavior, for a well-formed program construct and correct data, that depends on the implementation [*Note:* The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard. — *end note*]

Unspecified Behavior Examples

— Order of evaluation of function arguments when using function call syntax. See [expr.call] § 5

— How the memory for an exception object is allocated. See [except.throw] § 4

Undefined Behavior

[defns.undefined]

Behavior for which this International Standard imposes no requirements [*Note:* Undefined behavior may be expected when

— this International Standard omits any explicit definition of behavior or…

— when a program uses an erroneous construct or erroneous data…

Undefined Behavior Examples

— Consequences of overflow of a signed integer. See [expr] § 4

— Any race condition. See [intro.races] § 20

— Accessing the non-active member of a union. Implied by [class.union] § 1

Caution: Exhibited behavior can change based on compiler flags or compiler version

Compilers Handling Undefined Behavior

[defns.undefined]

— behave during translation or program execution in a documented manner characteristic of the environment

— terminate either translation or execution

— ignore the situation completely with unpredictable results.

Optimizers specifically take that last item as an opportunity to generate faster and / or smaller code!

Compilers Will Generally Not...

- Intentionally reformat your hard drive
- Make demons fly out your nose
- Make your cat have puppies

Why Not?

- No economic incentive
- People who write compilers and optimizers want you to use their products

C++ Programs With Undefined Behavior?

- Probably most of them
- Undefined behavior is everywhere

Example

- How many multi-threaded programs have no race conditions?

Stop Worrying And Love Undefined Behavior?

No

- Do your best to keep undefined behavior out of anticipated and potential code paths

- Undefined behavior makes bugs difficult to solve

Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary

Casting Away Const

Casting Const to Non-Const

[dcl.type.cv] § 4

> Except that any class member declared `mutable` can be modified, any attempt to modify a `const` object during its lifetime results in undefined behavior.

- Casting away const is fine
- Modifying a non-mutable const is undefined behavior

Example of Modifying Const

[dcl.type.cv] § 4

```
// initialized as required
const int* ciq = new const int (3);

// cast required
int* iq = const_cast<int*>(ciq);

// undefined: modifies a const object
*iq = 4;
```

Compiler Caching of Const Values

- If the optimizer sees a value is const…
- … the optimizer is allowed to assume the value does not change

If you change a const value you are violating the optimizer’s assumption.

std::launder() [ptr.launder]

Example from P0532R0 by Nicolai Josuttis

```
struct X {
    const int n;
    const double d;
};
X* p = new X{7, 8.8};
new (p) X{42, 9.9};          // place new value into p
int b = p->n;                // undefined behavior!
int c = p->n;                // undefined behavior!
double d = p->d;             // undefined behavior!
```

std::launder() to the Rescue

Example from P0532R0 by Nicolai Josuttis

```
struct X {
    const int n;
    const double d;
};
X* p = new X{7, 8.8};
new (p) X{42, 9.9};                 // place new value into p
int b = std::launder(p)->n; // OK, b is 42
int c = p->n;                       // undefined behavior!
double d = p->d;                    // undefined behavior!
```

std::launder() Caveats

- Only operates on pointers (not values or references)
- Access questionable value…
  - Directly through std::launder or
  - Through a saved pointer returned by std::launder
- Only works with placement new

Result: std::launder is hard to use correctly

Guidance: Don’t modify non-mutable const values

Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary


Pointers

Using an Invalid Pointer is Undefined

[basic.stc.dynamic.safety] § 4

...[*Note:* The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined...

So manipulate pointers carefully

Pointer to Integer Conversions

[expr.reinterpret.cast] § 4 and § 5

A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined…

A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size … and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation defined.

Math on uintptr_t

[basic.stc.dynamic.safety] § 3.4

An integer value is an *integer representation of a safely derived pointer* only if … it is … [amongst other things] …

— the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value `P`, if that result converted by `reinterpret_cast<void*>` would compare equal to a safely-derived pointer computable from `reinterpret_cast<void*>(P)`.

std::uintptr_t Example

```
double const d[] {0.1, 0.2, 0.3, 0.4, 0.5, 0.6};
double const* p = &d[0];
auto ip = reinterpret_cast<std::uintptr_t>(p);

for (auto i = 0u; i < sizeof d; i += sizeof d[0])
{   // Not portable!  Assuming Clang and Intel
    p = reinterpret_cast<double const*>(ip + i);
    std::cout << *p << "  ";
}
std::cout << std::endl;
```

```
0.1  0.2  0.3  0.4  0.5  0.6
```

std::intptr_t and uintptr_t Caveats

- std::intptr_t and std:uintptr_t are optional types
- If no integer type is big enough to hold the bit representation of a pointer, conversion may not be supported
- Integer representation of a pointer might not be the same bit sequence as the pointer itself
- Math is allowed if you are careful. Implementation defined results.
- Consider: are you comfortable with a signed pointer?

Accessing Initialized Value Representations

[basic.lval] § 8

If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:

— the dynamic type of the object,

— ...

— a `char`, `unsigned char`, or `std::byte` type.

Indeterminate Value Definition

[dcl.init] § 12

If no initializer is specified for an object, the object is default-initialized. When storage for an object with automatic or dynamic storage duration is obtained, the object has an *indeterminate value*, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced.

Indeterminate Value Examples

```
void test_function ()
{
    char a1;
    char b1[5];
    struct s {char c1[5]; int d1;};
    class c : public s {
        protected:
            int i1;
    };
    union u {s s1; c c1;};
    u u1;
    ...
```

Accessing Indeterminate Values

[dcl.init] § 12 loose paraphrase

When accessing an indeterminate value, the result is undefined unless that access is through either

— an `unsigned char*` or

— `std::byte*`.

Accessing Any Value Representation

The representation of any trivially copyable type, either

- initialized or
- unitialized

may be accessed using either

- unsigned char* or
- std::byte*

without introducing undefined behavior.

Accessing the LSBs of a Pointer

```
char const test[] {"abcdefg"};
char const* char_ptr {&test[0]};

unsigned char const* const ptr_to_ptr {
    reinterpret_cast<unsigned char*>(&char_ptr)};

do { // Not portable!  Assuming Clang and Intel
    std::cout << int (ptr_to_ptr[0] & 0x3) << "   ";
    ++char_ptr;
} while (char_ptr < &(test[sizeof test]));
std::cout << std::endl;
```

```
0   1   2   3   0   1   2   3
```

General reinterpret_cast of Pointers

[expr.reinterpret.cast] § 7

An object pointer can be explicitly converted to an object pointer of a different type.

But, accessing through that pointer may be undefined:

- Is the punned representation valid?
- Is the punned representation properly aligned?
- Are the strict aliasing rules being followed?

Pointer Aliasing

```
int sum_twice (int* a, int* b) {
    *a += *b;
    *a += *b;
    return *a;
}

void alias_example () {
    int c[] {2, 2};  // a and  b are different
    cout << sum_twice (&c[0], &c[1]) << endl; // 6

    int d {2};       // a and  b are the same
    cout << sum_twice (&d   , &d   ) << endl; // 8
}
```

Is Aliasing a Problem?

1. Code runs fastest in registers, not memory.
2. Pointers (and references) operate on memory.
3. If a memory location *might* be aliased, the compiler must write value changes to memory before reads through the possible alias.
4. If a memory location *might* be aliased, the compiler must re-read the value from memory if the possible alias is written through.

Strict Aliasing Rules

[basic.lval] § 8

If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:

— the dynamic type of the object,

— a cv-qualified version of the dynamic type of the object,

— a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,

Strict Aliasing Rules Continued

[basic.lval] § 8

— an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),

— a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,

— a `char`, `unsigned char`, or `std::byte` type

Reason for the Strict Aliasing Rules

The Strict Aliasing Rules reduce the number of pointers and references the compiler thinks *might* be aliases.

So your program runs faster

Strict Aliasing Violation Example

```
std::uint32_t swap_halves (std::uint32_t arg)
{
    auto sp = reinterpret_cast<std::uint16_t*>(&arg);
    auto hi = sp[0];
    auto lo = sp[1];

    sp[1] = hi;
    sp[0] = lo;

    return arg;
}
```

Disabling Strict Aliasing Rules

Most compilers have a flag to turn off strict aliasing rules

- GCC: -fno-strict-aliasing
- Clang: -fno-strict-aliasing
- Visual C++: Doesn’t rely on strict aliasing rules

Consequence: your code will probably run slower!

FYI: Strict Aliasing Compiler Extensions

Your compiler generates faster code if there is no aliasing.

In C, “restrict” is a promise from the programmer that aliases to a certain pointer don’t matter. Absent from C++.

Many C++ compilers support variations on “restrict”.

Lie to the compiler at your peril!

Punning Pointers Guidance

Be wary of pointer conversions requiring reinterpret_cast to types other than:

- signed / unsigned change
- std::uintptr_t or std::intptr_t
- std::byte* or unsigned char*

Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary


Unions

Union Common Initial Sequence

[class.union] § 1

… If a standard-layout union contains several standard-layout structs that share a common initial sequence, and if a non-static data member of an object of this standard-layout union type is active and is one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of the standard-layout struct members…

Standard Layout Struct

- For definition see [class] § 7. Too long to show here.

- More constrained than trivially copyable types

... useful for communicating with code written in other programming languages ... See [class] § 9.

Use `static_assert(std::is_standard_layout<T>::value)`

Common Initial Sequence Example

```
void common_initial_sequence ()
{
    enum class type {nil, bus, car};
    struct nil {type t;};
    struct bus {type t; int max_people;};
    struct car {type t; float fuel_econ;};

    union vehicle {nil t; bus b; car c;};

    vehicle v1 {{type::nil}};
    v1.b = {type::bus, 32};
    assert (v1.t.t == type::bus); // Defined
}
```

Union Access

[class.union] § 1

In a union, a non-static data member is *active* if its name refers to an object who's lifetime has begun and not ended. At most one of the non-static data members of an object of union type 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.

C++17 Union Undefined Access

- Accessing a union's active member is defined behavior
- For a union containing standard layout structs, access to a common initial sequence can be defined behavior
- In C++17 any other non-active union member access is undefined behavior

Meanwhile in C11 (N1570)...

6.5.2.3 Structure and union members § 3 footnote 95

95) If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called “type punning”). This might be a trap representation.

C11 Non-active Union Member Access

- In C11 non-active union member access is well defined (in non-normative text).
- Any compiler that supports both C11 and C++17 is likely to exhibit C11 behavior in both cases.
- No guarantees…

Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary


Unrelated Object Types

Background: float NaN Payload

The bit-wise representation of an IEEE 754 float NaN is:

```
s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx
```

s: sign bit

x: payload

Payload not allowed to be all zeros

Type Punning Exercise

Given a payload, return a float NaN

```
float mk_nan (std::uint32_t cargo, bool pos = true)
{
    ...
}
```

Strict Aliasing Violation

```
float mk_nan (std::uint32_t cargo, bool pos = true)
{
    assert ((cargo & 0x007F'FFFF) != 0);
    assert ((cargo & 0xFF80'0000) == 0);

    cargo |= pos ? 0x7F80'0000 : 0xFF80'0000;

    // Setup for Strict Aliasing Rules violation.
    auto p_ret = reinterpret_cast<float*>(&cargo);

    // Access violates Strict Aliasing Rules.
    return *p_ret;
}
```

Using a Union

```
float mk_nan (std::uint32_t cargo, bool pos = true)
{
    assert ((cargo & 0x007F'FFFF) != 0);
    assert ((cargo & 0xFF80'0000) == 0);
    union uint_float {
        std::uint32_t i;
        float f;
    } ret {pos ? 0x7F80'0000 : 0xFF80'0000};

    ret.i |= cargo;
    // Undefined behavior accessing non-active member
    return ret.f;
}
```

Using std::memcpy

```
float mk_nan (std::uint32_t cargo, bool pos = true)
{
    assert ((cargo & 0x007F'FFFF) != 0);
    assert ((cargo & 0xFF80'0000) == 0);

    cargo |= pos ? 0x7F80'0000 : 0xFF80'0000;
    float ret;
    std::memcpy (&ret, &cargo, sizeof ret);

    return ret;
}
```

memcpy() For Type Punning?

- Most widely recommended C++ type punning technique
- Standard defines copying underlying bytes for the same trivially copyable type T. See [basic.types] § 2 and § 3
- memcpy() between unrelated types is undefined behavior
- … or not undefined since memcpy() uses unsigned char*

Access Through unsigned char*

```
float mk_nan (std::uint32_t cargo, bool pos = true)
{
    assert ((cargo & 0x007F'FFFF) != 0);
    assert ((cargo & 0xFF80'0000) == 0);
    float ret {0};
    auto const p_ret =
        reinterpret_cast<unsigned char*>(&ret);
    p_ret[0] =  cargo & 0xFF;   cargo >>= 8;
    p_ret[1] =  cargo & 0xFF;   cargo >>= 8;
    p_ret[2] = (cargo & 0xFF) | 0x80;
    p_ret[3] = pos ? 0x7F : 0xFF;
    return ret;
}
```

Type Punning Technique Costs

Technique	Assembly Instruction Count Clang Mac OS -O3
unsigned char*	9
memcpy()	*8
union	*8
strict aliasing violation	*8

* Identical code generated

Punning Unrelated Objects Guidance

1. Punning through unsigned char* should *always* work

2. memcpy() punning *almost certainly* works

3. Union punning *likely* works due to C11 compatibility

4. Strict aliasing violations are the riskiest

Punning Trivially Copyable Types

All of the identified techniques apply equally well to trivially copyable…

- Scalars
- Structs
- Classes and
- Arrays of Trivially Copyable Types

Topics

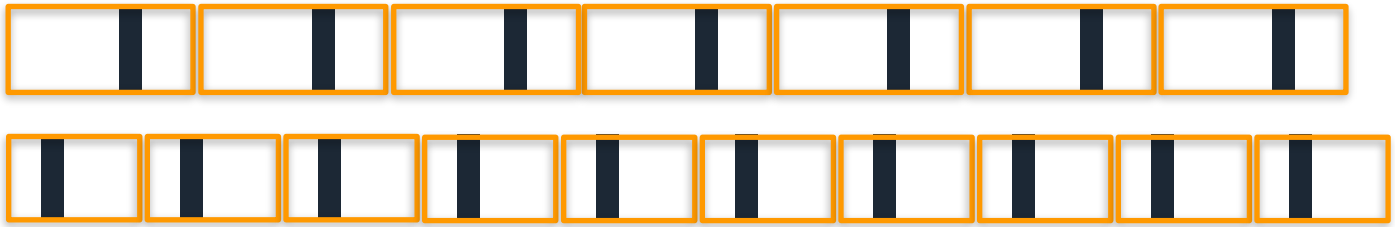
1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary


Arrays

Array Striding

From https://en.wikipedia.org.wiki/Stride_of_an_Array

Given the strides of arrays...



...can we extract pieces of the arrays?

Can We Make This Work?

```
int main ()
{
    int ints[] = {1,2,3};
    struct record { char const* text; int value; };
    record rcrd[] {{"a", 10},{"b", 11},{"c",12}};

    print_some_ints (&ints[0],
        sizeof ints / sizeof ints[0], sizeof ints[0]);
    std::cout << std::endl;

    print_some_ints (&rcrd[0].value,
        sizeof rcrd / sizeof rcrd[0], sizeof rcrd[0]);
}
```

Desired Output

```
$ ./print_some_ints
Addr: 0x7fff598737c8; value: 1
Addr: 0x7fff598737cc; value: 2
Addr: 0x7fff598737d0; value: 3

Addr: 0x7fff59873798; value: 10
Addr: 0x7fff598737a8; value: 11
Addr: 0x7fff598737b8; value: 12
$
```

print_some_ints Rev 1

```
void print_some_ints (
    int const* arr, int count, size_t stride)
{
    for (int i = 0; i < count; ++i) {
        std::cout << "Addr: " << arr
            << "; value: " << arr[0] << std::endl;
        arr = reinterpret_cast<int const*>(
            reinterpret_cast<unsigned char const*>(
                arr) + stride);
    }
}
```

Pointer Math

[expr.add] § 4

When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the expression P points to element x[i] of an array object x with n elements, the expressions P + J and J + P (where J has the value j) point to the (possibly-hypothetical) x[$i + j$] if $0 \leq i + j \leq n$; otherwise the behavior is undefined.

Rev 1 Has Undefined Behavior

- Pointer addition is defined if the pointer points to an array.
- Our unsigned char const* points into a struct, not to an array.
- Therefore addition to the unsigned char const* is undefined.

print_some_ints Rev 1

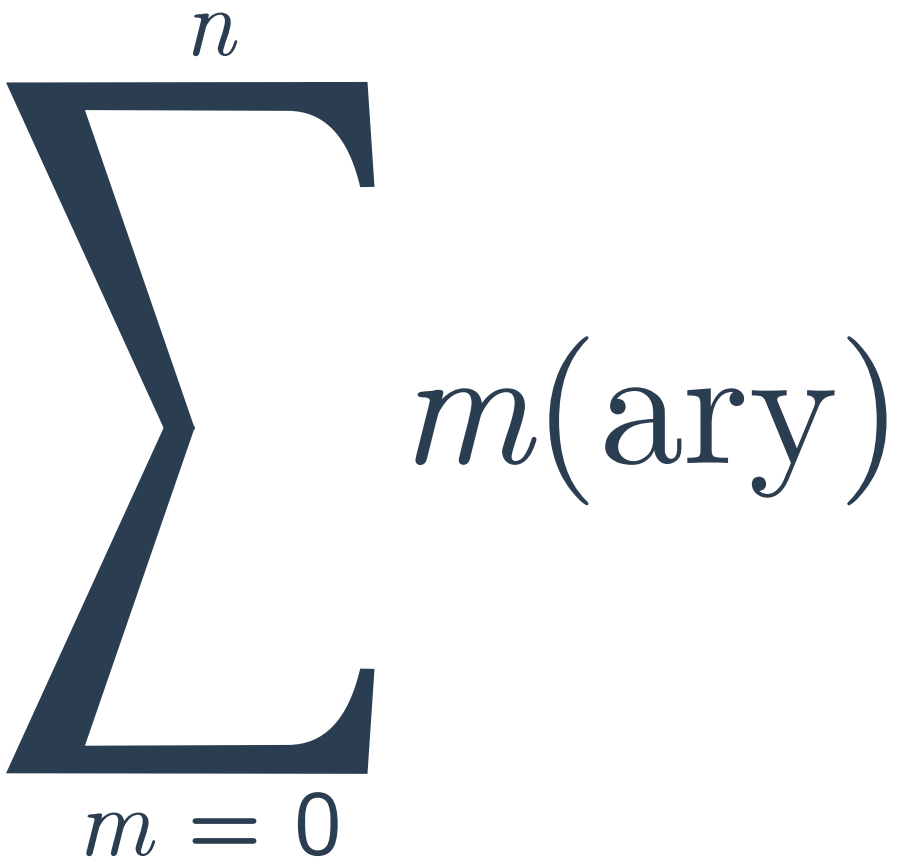
```
void print_some_ints (
    int const* arr, int count, size_t stride)
{
    for (int i = 0; i < count; ++i) {
        std::cout << "Addr: " << arr
            << "; value: " << arr[0] << std::endl;
        arr = reinterpret_cast<int const*>(
            reinterpret_cast<unsigned char const*>(
                arr) + stride);
    }
}
```

print_some_ints Rev 2

```
void print_some_ints (
    int const* arr, int count, size_t stride)
{
    for (int i = 0; i < count; ++i) {
        std::cout << "Addr: " << arr
            << "; value: " << arr[0] << std::endl;
        arr = reinterpret_cast<int const*>(
            reinterpret_cast<   std::uintptr_t   >(
                arr) + stride);
    }
}
```

Topics

1. Motivations
2. Definitions
3. Casting Away Const
4. Pointers
5. Unions
6. Unrelated Object Types
7. Arrays
8. Summary


n
m = 0
m(ary)

Punning Guidance

- Avoid modifying non-mutable const values
- Consider the effects of possible pointer aliasing
- Be aware of the strict aliasing rules
- When punning with pointers prefer…

  `std::byte*` or `unsigned char*`
- For punned pointer math, consider `std::uintptr_t`
- Avoiding undefined behavior is hard

Thanks and Acknowledgements

- Hubert Tong: Master of dark corners
- Howard Hinnant: Expertise and guidance
- Mike Miller: Nudge
- Patrick Horgan: http://dbp-consulting.com/tutorials/StrictAliasing.html
- Brian Schiller: Fonts
- ripple: Encouragement and employment


All errors are the sole property of
Scott Schurr
84·342 D.C. 1923
MARYLAND 1923


Questions?
Thanks for attending