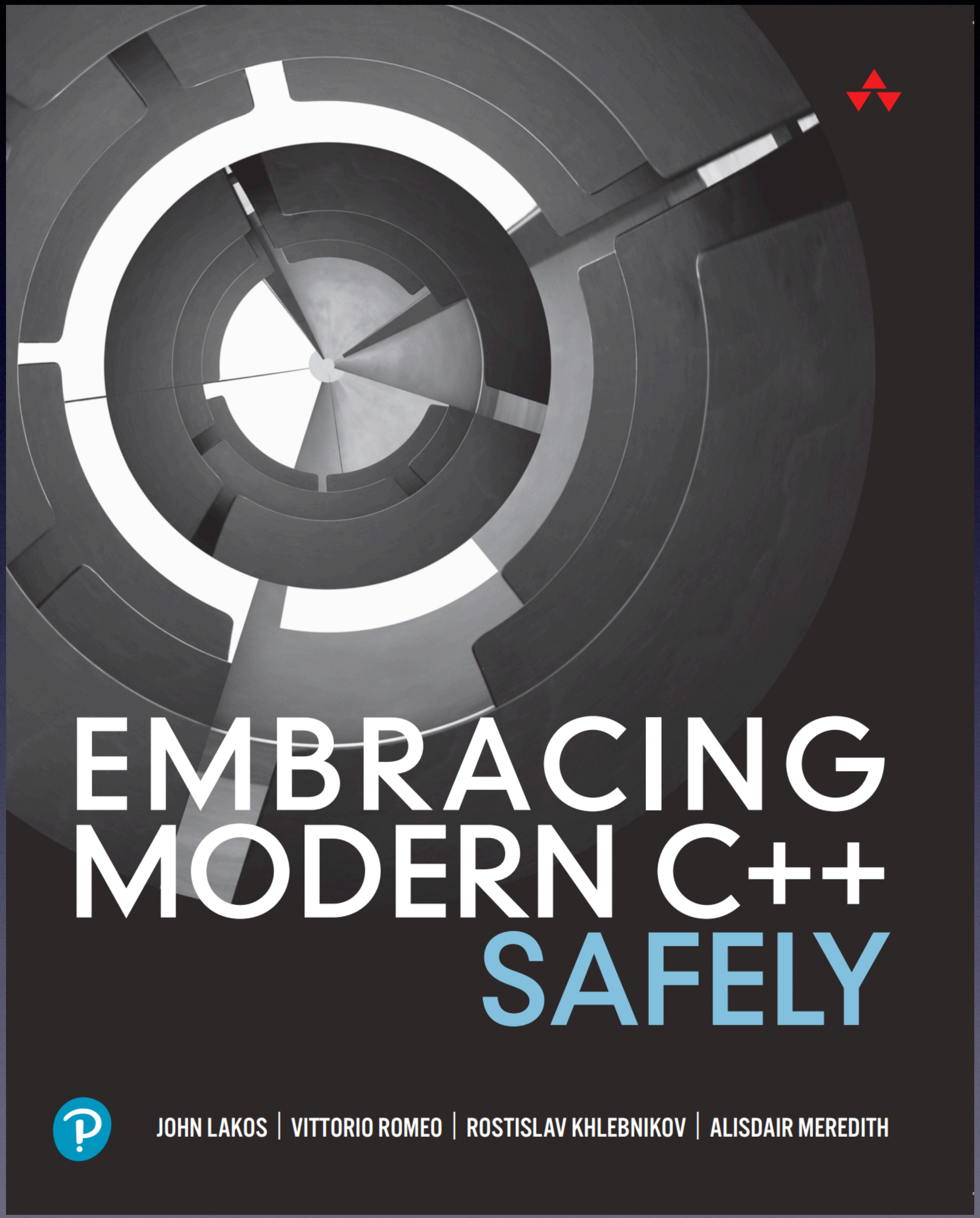
A Tour of C++ Recognised User Type Categories

NINA RANNS

Cppcon
The C++ Conference
2022
September 12th-16th

Nina Ranns

Committee member since 2013,
Cat owner,
Coffee drinker,
Chocolate lover,
Comma supervisor, and
C++ nerd



- John Lakos
- Vittorio Romeo
- Rostislav Khlebnikov
- Alisdair Meredith
- An extended team of collaborators

What Is The Book About?

- Analyze every new feature of the C++11/14 language
- Categorize risk/reward given the potential for misunderstanding (safe, conditionally safe, unsafe)
- common pitfalls and how to avoid them

Types with special provisions

- treating user defined types as built in types in certain cases
- new features require new definitions (constexpr specifier and literal types)
- “blessing” certain common programming patterns inherited from C code

C++ Object Model

C++ Object Model

- “The constructs in a C++ program create, destroy, refer to, access, and manipulate objects.”

C++ Object Model

- “The constructs in a C++ program create, destroy, refer to, access, and manipulate objects.”
- “An object occupies a region of storage in its period of construction, throughout its lifetime, and in its period of destruction.”

C++ Object Model

- “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) ”

C++ Object Model

- "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."

Types of objects

- objects of scalar types (arithmetic types, enumeration types, pointer types, pointer-to-member types, std::nullptr_t)
- objects of user defined types, i.e. class types
- arrays

Types of objects

- objects of scalar types (arithmetic types, enumeration types, pointer types, pointer-to-member types, std::nullptr_t)
- objects of user defined types, i.e. class types
- arrays

Types of objects

- objects of scalar types (arithmetic types, enumeration types, pointer types, pointer-to-member types, std::nullptr_t)
- objects of user defined types, i.e. class types
- arrays

Objects of user defined type

- normally have constructor called when lifetime begins
- normally have destructor called when lifetime ends
- normally have values copied using assignment or copy construction
- normally have unspecified layout
- normally have differing object representation and value representation

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Aggregates

- existed in C++98
- can be initialised as a collection of objects (aggregate initialisation)
- allows for designated initialisers
- definition incrementally expanded to cover more user defined types

Aggregates

- existed in C++98
- can be initialised as a collection of objects (aggregate initialisation)
- allows for designated initialisers
- definition incrementally expanded to cover more user defined types

```
struct X {
  int a,b;
  char c;
  long l;
};

X x1{1, 2, 'a', 0l};
X x2{1};
X X3{x2};
```

Aggregates

- existed in C++98
- can be initialised as a collection of objects (aggregate initialisation)
- allows for designated initialisers
- definition incrementally expanded to cover more user defined types

Aggregates

- existed in C++98
- can be initialised as a collection of objects (aggregate initialisation)
- allows for designated initialisers (since C++20)
- definition incrementally expanded to cover more user defined types

```cpp
struct X {
    int a = 1;
    int b;
    char c = 'a';
    long l = 0;
};

int main()
{
  int  a = 4;
  X x1{.c = 'a'};
  X x2{.b = a, .c = 'c', .l = 3l}; // x2.b = 4
  X x3{.a = x2.b, .c = 'a', .l = 3l}; // x2.b is uninitialised
}
```

```
struct X {
    int a = 1;
    int b;
    int d;
    char c = 'a';
    long l = 0;
};
int main()
{
  int  a = 4;
  X x1{.c = 'a'};
  X x2{.b = a, .c = 'c', .l = 3l}; // x2.b = 4
  X x3{.a = x2.b, .c = 'a', .l = 3l}; // x2.b is uninitialised
}
```

```cpp
struct X {
    int a = 1;
    int b;
    int d;
    char c = 'a';
    long l = 0;
};
int main()
{
  int  a = 4;
  X x1{.c = 'a'};
  X x2{.b = a, .c = 'c', .l = 31}; // x2.b = 4
  X x3{.a = x2.b, .c = 'a', .l = 31}; // x2.b is uninitialised
}
```

“The initializations of the elements of the aggregate are evaluated in the element order. “

Aggregates

- existed in C++98
- can be initialised as a collection of objects (aggregate initialisation)
- allows for designated initialisers
- definition incrementally expanded to cover more user defined types

Aggregates

- existed in C++98
- can be initialised as a collection of objects (aggregate initialisation)
- allows for designated initialisers
- definition incrementally expanded to cover more user defined types

Aggregates - C++03

- An aggregate is an array or a class with
  - no user-declared constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no base classes

Aggregates - C++03

- An aggregate is an array or a class with
  - no user-declared constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no base classes

Aggregates - C++03

- An aggregate is an array or a class with
  - no user-declared constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no base classes

Aggregates - C++03

- An aggregate is an array or a class with
  - no user-declared constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no base classes

Aggregates - C++11

- An aggregate is an array or a class with
  - no user-provided or explicit constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no base classes
  - no brace-or-equal-initializers for non-static data members

Aggregates - C++14

- An aggregate is an array or a class with
  - no user-provided or explicit constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no base classes

Aggregates - C++17

- An aggregate is an array or a class with
  - no user-provided, explicit, or inherited constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no virtual, private, or protected base classes

“The *elements* of an aggregate are … for a class, the direct base classes in declaration order, followed by the direct non-static data members that are not members of an anonymous union, in declaration order. “

…

```
struct Y{
  int i;
};
struct X : public Y{
  char c;
  long l;
};

X x1{{1}, 'c', 2l};
X x2{1, 'c', 2l};
```

Aggregates - C++17

- An aggregate is an array or a class with
  - no user-provided, explicit, or inherited constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no virtual, private, or protected base classes

Aggregates - C++20

- An aggregate is an array or a class with
  - no user-declared, explicit, or inherited constructors,
  - no private or protected direct non-static data members,
  - no virtual functions, and
  - no virtual, private, or protected base classes

```
struct X {
    X() = delete;
};

int main() {
    X x1; // ill-formed - default c’tor is deleted
    X x2{}; // compiles!
}
```

```cpp
struct X {
    int i{1};
    X(int) = delete;
};

int main() {
    X x1(1); // ill-formed - value c'tor is deleted
    X x2{2}; // compiles!
}
```

```cpp
struct X {
    int i{1};
    X(int) = delete;
};

int main() {
    X x1(1); // ill-formed - value c'tor is deleted
    X x2{2}; // compiles!
}
```

See P1008 for more details

Library trait - std::is_aggregate

- Introduced in C++17 to resolve the problem of emplace and construct library idioms

```
template <class U, class... Args&>
    void construct(U* p, Args&&... args);
Effects: ::new((void *)p) U(std::forward<Args>(args)...)

U(std::forward<Args>(args)...) // doesn't work if U is an aggregate
U{std::forward<Args>(args)...} // preffers std::initializer_list constructors
                               // for non aggregates
```

Library trait - std::is_aggregate

- C++20 adds aggregate parens initialisation (see P0960). Library trait obsolete ?

```
struct X{
 int i, j;
};
X x1(1);       // ok in C++20
X x2(1,2);     // ok in C++20
X x3(.i = 3); // designated initialisers not supported
```

Library trait - std::is_aggregate

- C++20 adds aggregate parens initialisation (see P0960). Library trait obsolete ?

```
struct X{
 int i, j;
};
X x1(1);       // ok in C++20
X x2(1,2);     // ok in C++20
X x3(.i = 3); // designated initialisers not supported
```

Note : "… narrowing conversions are permitted, designators are not permitted, a temporary object bound to a reference does not have its lifetime extended, and there is no brace elision. "

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Standard layout types

- Classes with “predictable” layout.
- “Standard-layout classes are useful for communicating with code written in other programming languages.”

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

```
struct X {}; // standard layout

struct Y : public X { // standard layout
 int i, j;
};

struct Z : public Y {}; // standard layout

struct Q : public X {     // not standard layout
  char c;
};
```

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

Distinct address (C++11)

- Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a base class subobject of zero size and they are of different types;

Distinct address (C++20)

- Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a subobject* of zero size and they are of different types;

Distinct address (C++20)

- Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a subobject* of zero size and they are of different types;

*in C++20, [no_unique_address] non-static members could be of zero size

Distinct address (C++20)

- Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a subobject* of zero size and they are of different types

*in C++20, [no_unique_address] non-static members could be of zero size

```
struct A {};        // standard layout
struct B : A {};    // standard layout
struct C : A {};    // standard layout

struct D : B,C {}; // not standard layout
struct E : B {     // not standard layout
    A a;
};
```

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

Standard Layout Class

- has the same access control for all non-static data members,
- has all non-static data members and bit-fields first declared either in the most derived class or in only one base class,
- has no base classes that require a distinct address,
- has no virtual functions and no virtual base classes,
- all base classes and non-static data members are of standard layout type, no reference members

Implications of Standard Layout

- layout of the object depends only on the non-static data members
- the address of a standard layout class object and its first non-static data member is the same
- the address of a standard layout class object and all of its base classes is the same
- `offsetof` is well defined with standard layout class types*

Implications of Standard Layout

- layout of the object depends only on the non-static data members
- the address of a standard layout class object and its first non-static data member is the same
- the address of a standard layout class object and all of its base classes is the same
- `offsetof` is well defined with standard layout class types*

Implications of Standard Layout

- layout of the object depends only on the non-static data members
- the address of a standard layout class object and its first non-static data member is the same
- the address of a standard layout class object and all of its base classes is the same
- `offsetof` is well defined with standard layout class types*

Implications of Standard Layout

- layout of the object depends only on the non-static data members
- the address of a standard layout class object and its first non-static data member is the same
- the address of a standard layout class object and all of its base classes is the same
- `offsetof` is well defined with standard layout class types*

* and conditionally supported with non standard layout types

Common Initial Sequence

- Two standard layout structs have a common initial sequence if some of their non-static data members and bit-fields in declaration order have layout-compatible types

Common Initial Sequence

- Two standard layout structs have a common initial sequence if some of their non-static data members and bit-fields in declaration order have the same (underlying) type

Common Initial Sequence (C++20)

- Two standard layout structs have a common initial sequence if some of their non-static data members and bit-fields in declaration order have the same (underlying) type, and either both are declared with [[no_unique_address]] or neither is.

Reading from a Union

- If an inactive member of a union has a common initial sequence with the active member, it's possible to read a member of the common initial sequence through that inactive member

- The read must happen through the union object, not through a pointer or a reference to the inactive member

Reading from a Union

```
struct A {
  int i;
  long l;
};
```

```
struct B {
  int j;
  char c;
};
```

```
union U {
  A a;
  B b;
};
```

```
U u{A{}}; // `a` is the active member;
```

```
int k = u.b.j; // ok, `b.j` is a member of the common initial sequence
int l = u.b.c; // UB, `b.c` is not a member of the common initial sequence
```

Reading from a Union

```
struct A {
  int type_id;
  long l;
};
```

```
struct B {
  int type_id;
  char c;
};
```

```
union U {
  A a;
  B b;
};
```

```
U u{A{}}; // `a` is the active member;
```

```
if (u.b.type_id == b_type) {
  /*…*/
}
```

Reading from a Union

```
struct A {
  int type_id;
  long l;
};
```

```
struct B {
  int type_id;
  char c;
};
```

```
union U {
  A a;
  B b;
};
```

```
U u{A{}}; // `a` is the active member;

B  &bref = u.b; // forms reference to inactive union member
if (b.type_id == b_type) {  // UB: reading through reference to an inactive member
  /*…*/
}
```

Library trait - std::is_standard_layout

- Introduced in C++11
- limited usability (static_assert to ensure correctness?)
- questionable quality of implementation suggest it isn’t widely used

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Trivial special member functions

Trivial default constructor

- not user-provided
- class has no virtual functions and no virtual base classes
- all the base classes and non-static data members have trivial default constructors
- class has no default member initializers for non-static data members

Trivial default constructor

- not user-provided
- class has no virtual functions and no virtual base classes
- all the base classes and non-static data members have trivial default constructors
- class has no default member initializers for non-static data members

Trivial default constructor

- not user-provided
- class has no virtual functions and no virtual base classes
- all the base classes and non-static data members have trivial default constructors
- class has no default member initializers for non-static data members

Trivial default constructor

- not user-provided
- class has no virtual functions and no virtual base classes
- all the base classes and non-static data members have trivial default constructors
- class has no default member initializers for non-static data members

Trivial copy constructor and assignment operator

- not user-provided
- class has no virtual functions and no virtual base classes
- all the base classes and non-static data members have trivial corresponding special member function

Trivial destructor

- not user-provided
- not virtual
- destructors of all base classes and non-static data members are trivial

Trivially copyable type

- at least one non-deleted* copy operation
- all copy operations are trivial
- has a trivial non-deleted destructor

Trivially copyable type

- at least one non-deleted* copy operation
- all copy operations are trivial
- has a trivial non-deleted destructor

* *C++20 definition takes into account constraints too*

Trivially copyable type

- at least one non-deleted* copy operation
- all copy operations are trivial
- has a trivial non-deleted destructor

* *C++20 definition takes into account constraints too*

Trivially copyable type

- at least one non-deleted* copy operation
- all copy operations are trivial
- has a trivial non-deleted destructor

* *C++20 definition takes into account constraints too*

Trivially copyable type

- at least one non-deleted* copy operation
- all copy operations are trivial
- has a trivial non-deleted destructor

Note that there are no requirements on access control or ambiguity of call

* *C++20 definition takes into account constraints too*

Implications of Trivially Copyable

- the value is contained in the underlying byte representation
- can be `memcpy`-ed to and from another object of same type
- can be `memcpy`-ed to and from an array of `char`, `unsigned char`, or `std::byte`
- compiler can implement the copy (copy/move construction or copy/move assignment) as `memcpy`

Implications of Trivially Copyable

- the value is contained in the underlying byte representation
- can be `memcpy`-ed to and from another object of same type
- can be `memcpy`-ed to and from an array of `char`, `unsigned char`, or `std::byte`
- compiler can implement the copy (copy/move construction or copy/move assignment) as `memcpy`

Implications of Trivially Copyable

- the value is contained in the underlying byte representation
- can be `memcpy`-ed to and from another object of same type
- can be `memcpy`-ed to and from an array of `char`, `unsigned char`, or `std::byte`
- compiler can implement the copy (copy/move construction or copy/move assignment) as `memcpy`

Library trait - std::is_trivially_copyable

- Introduced in C++11
- prerequisite for bitwise copy
- does not ensure copy operation is non-deleted - generic code should also check for availability of the relevant copy operation

```cpp
struct X
{
    X() = default;
    X(const X&) = delete;
    X(X&&) = default;
};

static_assert(std::is_trivially_copyable_v<X>);

template<class T> requires std::is_trivially_copyable_v<T>
void copy(T& dest, const T& source)
{
    std::memcpy(&dest, &source , sizeof(dest)); // ok for X ?
}
```

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Trivial Type

- trivial class type = trivially copyable + non-deleted/eligible trivial default constructor
- remnant from POD days - only used in the library to describe types which used to be required to be of POD type

Library trait - std::is_trivial

- Introduced in C++11
- limited usability - check for POD requirement ?

Library trait - std::is_trivial

- Introduced in C++11
- limited usability - check for POD requirement ?

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

What are PODs

- Defined in C++98 for C compatibility
- their physical representation mimics that of a C type (it doesn't use C++ only features which affect its physical representation)
- behave like C types in terms of construction, copying, and destruction (it doesn't make use of C++ only features that are relevant to constructions, destruction, and copying)

POD: Plain Old Data

- The original definition :
  - an aggregate (no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions)
  - no non-static data members of reference type, all non-static data members of POD type, no user-defined copy assignment operator, and no user-defined destructor.

POD in C++11

- properties relevant to object layout - standard layout types
- properties relevant to object construction, copying and destruction - trivial types
- original term no longer needed - deprecated, then removed in C++20

21 Strings library [strings]

21.1 General [strings.general]

1 This Clause describes components for manipulating sequences of any non-array POD (3.9) type. In this Clause such types are called *char-like types* , and objects of char-like types are called *char-like objects* or simply *characters*.

2 The following subclauses describe a character traits class, a string class, and null-terminated sequence utilities, as summarized in Table 61.

23 Strings library [strings]

23.1 General [strings.general]

1 This Clause describes components for manipulating sequences of any non-array trivial standard-layout (6.8.1) type. Such types are called *char-like types*, and objects of char-like types are called *char-like objects* or simply *characters*.

2 The following subclauses describe a character traits class, string classes, and null-terminated sequence utilities, as summarized in Table 74.

Library trait - std::is_pod

- Introduced in C++11
- deprecated in C++20

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Literal types

- A literal type is one for which it might be possible to create an object at compile time.
- It is not a guarantee that it is possible to create such an object, nor is it a guarantee that any object of that type will be usable at compile time.

Literal types

- A literal type is one for which it might be possible to create an object within a constant expression.
- It is not a guarantee that it is possible to create such an object, nor is it a guarantee that any object of that type will usable in a constant expression.

Constant expressions

- requirement for constant initialisation (i.e. initialisation during compile time)
- three new keywords constexpr(C++11), consteval(C++20), and constinit(C++20)

Limitations of constant expressions

- things that are not possible/correct at compile time
- things that are difficult to implement
- For more information : C++Now 2019: Daveed Vandevoorde “C++ Constants”, P0992 Andrew Sutton “Translation and evaluation A mental model for compile-time metaprogramming”

Literal types

- a scalar type
- a reference type referring to a literal type
- an array of literal type
- void (C++14)
- a literal class type

Literal class (C++11)

- it is an aggregate type or has at least one constexpr constructor or constructor template that is not a copy or move constructor
- every constructor call and full-expression in the brace-or-equal-initializers for non-static data members is a constant expression
- all of its non-static data members and base classes are of literal types.
- has a trivial destructor

Literal class (C++17)

- it is a closure type, an aggregate type or has at least one constexpr constructor or constructor template that is not a copy or move constructor
- every constructor call and full-expression in the brace-or-equal-initializers for non-static data members is a constant expression
- all of its non-static data members and base classes are of literal types.
- has a trivial destructor

Literal class (C++20)

- it is a closure type, an aggregate type or has at least one constexpr constructor or constructor template that is not a copy or move constructor

- all of its non-static data members and base classes are of literal types.

- has a constexpr destructor

Library trait - std::is_literal_type

- introduced in C++11, deprecated in C++17, removed in C++20
- the real question is - does this initialisation qualify as constant initialisation ?
- will we need the definition of literal type in the future ?

Library trait - std::is_literal_type

- introduced in C++11, deprecated in C++17, removed in C++20
- the real question is - does this initialisation qualify as constant initialisation ?
- will we need the definition of literal type in the future ?

```cpp
struct X
{
    constexpr X() = default;
    X(int){};
    constexpr ~X(){};
};

constinit X x1;    // ok
constinit X x2{1}; // error
```

Library trait - std::is_literal_type

- introduced in C++11, deprecated in C++17, removed in C++20
- the real question is - does this initialisation qualify as constant initialisation ?
- will we need the definition of literal type in the future ?

Library trait - std::is_literal_type

- introduced in C++11, deprecated in C++17, removed in C++20
- the real question is - does this initialisation qualify as constant initialisation ?
- will we need the definition of literal type in the future ?

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Structural types (C++20)

- prerequisite for non-type template parameter
- scalar type
- lvalue reference type
- literal class type with
  - all base classes and non static data members public, non mutable and of structural type

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Types with special provisions

aggregate types

standard-layout types

trivially copyable types

trivial types

POD types

literal types

structural types

implicit-lifetime types

Object lifetime

(before C++20)

- “An object is created by a definition, by a new-expression, when implicitly changing the active member of a union, or when a temporary object is created.“

Object lifetime

(before C++20)

- "An object is created by a definition, by a new-expression, when implicitly changing the active member of a union, or when a temporary object is created."

```
struct X { int a, b; };
X *make_x() {
  X *p = (X*)malloc(sizeof(struct X));
  p->a = 1;
  p->b = 2;
  return p;
}
```

Object lifetime

(before C++20)

- “An *object* is created by a definition, by a *new-expression*, when implicitly changing the active member of a union, or when a temporary object is created.“

```
struct X { int a, b; };
X *make_x() {
  X *p = (X*)malloc(sizeof(struct X));
  p->a = 1;
  p->b = 2;
  return p;
}
```

```
char *buffer = (char*)malloc(n * sizeof(char)));
char *buffer_end = buffer + sizeof(char) * n;
```

```
char *buffer = (char*)malloc(n * sizeof(char));
char *buffer_end = buffer + sizeof(char) * n;
```

- “When an expression J that has integral type is added to or subtracted from an expression P of pointer type, the result has the type of P.
— If P evaluates to a null pointer value and J evaluates to 0, the result is a null pointer value.
— Otherwise, if P points to an array element i of an array object x with n elements the expressions P + J (…) points to the array element i + j of x
— Otherwise, the behavior is undefined.

```c
char *buffer = (char*)malloc(n * sizeof(char));
char *buffer_end = buffer + sizeof(char) * n;
```

- “When an expression J that has integral type is added to or subtracted from an expression P of pointer type, the result has the type of P.
  — If P evaluates to a null pointer value and J evaluates to 0, the result is a null pointer value.
  — Otherwise, if P points to an array element i of an array object x with n elements the expressions P + J (…) points to the array element i + j of x
  — Otherwise, the behavior is undefined.

```c
char *buffer = (char*)malloc(n * sizeof(char));
char *buffer_end = buffer + sizeof(char) * n;
```

- “When an expression J that has integral type is added to or subtracted from an expression P of pointer type, the result has the type of P.
  — If P evaluates to a null pointer value and J evaluates to 0, the result is a null pointer value.
  — Otherwise, if P points to an array element i of an array object x with n elements the expressions P + J (…) points to the array element i + j of x
  — Otherwise, the behavior is undefined.

```c
struct X { int a, b; };
X *make_x() {
  X *p = (X*)malloc(sizeof(struct X));
  p->a = 1;
  p->b = 2;
  return p;
}
```

X has a trivial constructor.
X has a trivial destructor.

What if we add special lifetime rules for types which require no code for construction and no code for destruction ?

Implicit lifetime types

- scalar types
- array types
- aggregate classes and classes with a trivial destructor and at least one trivial constructor

Object lifetime

(before C++20)

- “An object is created by a definition, by a new-expression, when implicitly changing the active member of a union, or when a temporary object is created.“

Object lifetime

(after C++20)

- “An object is created by a definition, by a new-expression, by an operation that implicitly creates objects, when implicitly changing the active member of a union, or when a temporary object is created.“

Object lifetime

(after C++20)

- “An object is created by a definition, by a new-expression, by an operation that implicitly creates objects, when implicitly changing the active member of a union, or when a temporary object is created.“
- aligned_alloc, malloc, calloc, realloc, memcpy, memove, std::bit_cast, std::pmr::memory_resource.allocate()

Object lifetime

(after C++20)

- “An object is created by a definition, by a new-expression, by an operation that implicitly creates objects, when implicitly changing the active member of a union, or when a temporary object is created.“
- aligned_alloc, malloc, calloc, realloc, memcpy, memove, std::bit_cast, std::pmr::memory_resource.allocate()
- “An operation that begins the lifetime of an array of char, unsigned char, or std::byte implicitly creates objects within the region of storage occupied by the array.”

Implicit lifetime types

- Created to make C code well defined in C++
- additionally solves the problem of pointer arithmetic
- no library trait

Summary

type	when do you care	trait
aggregate type	member-wise initialisation, ability to use designated initializers	is_aggregate (C++17)
trivially copyable type	copying using memcpy, memove, and bit_cast	std::is_trivially_copyable (C++11)
trivial type	partial check for C code compatibility	std::is_trivial (C++11)
standard layout type	pointer-interconvertibility, union access through common initial sequence, offsetoff	std::is_standard_layout (C++11)
POD type	check for C code compatibility	std::is_pod (C++11, deprecated in C++20)
literal type	requirement for compile time initilisation	std::is_literal_type (C++11, deprecated in C++17, removed in C++20)
structural type	requirement for non-type template parameter	no trait
implicit-lifetime type	makes common programming patterns well defined	no trait

Thank you !