2023

Data-Oriented Design and Modern C++

Floris Bob van Elzelingen

C++ now

Data-Oriented Design

Modern C++?

```
class transaction {
   int quantity;
   int price;
   date purchase_date;
public:
   int get_quantity() const {return quantity;}
   void set_quantity(int q) { quantity = q;}
   int get_price() const {return price;}
   void set_price(int p) { price = p;}
   date get_purchase_date() const {return purchase_date;}
   void set_purchase_date(int pd) { purchase_date = pd;}

   virtual int income_at(date d) const;
};

class purchase : public transaction {
public:
   int income_at(date d) const override {
```

Modern C++

- Be declarative
- Be simple
- Prevent user error

Modern C++

```
struct purchase {
   int quantity;
   int price;
   date purchase_date;
};

int income_at(const purchase& p, date d) {
   return d >= p.purchase_date ? p.price*p.quantity : 0;
}

struct subscription {
   int quantity;
   int price;
   date purchase_date;
   date last_payment;
};

int income_at(const subscription& s, date d) {
   return d >= s.purchase_date ? s.price*s.quantity*(1+months_intbetween(s.purchase_date, std::min(d,
s.last_payment))): 0;
```

Data-Oriented Design?

```
struct RecordArray {
   int quantity[10];
   int price[10];
   date purchase_date[10];
};

struct SubscriptionArray {
   int quantity[10];
   int price[10];
   date purchase_date[10];
   date last_payment[10];
};

int sum_income(RecordArray* ra, int ra_size, SubscriptionArray* sa, int sa_size, date d) {
   int sum = 0;
   for (int i = 0; i < ra_size; ++i) {
      sum += ra->purchase_date[i] ? ra->price[i]*ra->quantity[i] : 0;
   }
   for (int i = 0; i < sa_size; ++i) {
      sum += sa->purchase_date[i] ? sa->price[i]*sa->quantity[i]*
```

Data-Oriented Design

- Simple transformations
- Direct access to values
- Layout data to fit to performant transformations

Data-Oriented Design

```
struct purchase {
   int quantity;
   int price;
   date purchase_date;
};

int income_at(const purchase& p, date d) {
   return d >= p.purchase_date ? p.price*p.quantity : 0;
}

struct subscription {
   int quantity;
   int price;
   date purchase_date;
   date last_payment;
};

int income_at(const subscription& s, date d) {
   return d >= s.purchase_date ? s.price*s.quantity*(1+months_intbetween(s.purchase_date, std::min(d,
s.last_payment))): 0;
```

Data-Oriented Design

```
struct RecordArray {
   int quantity[10];
   int price[10];
   date purchase_date[10];
};

struct SubscriptionArray {
   int quantity[10];
   int price[10];
   date purchase_date[10];
   date last_payment[10];
};

int sum_income(RecordArray* ra, int ra_size, SubscriptionArray* sa, int sa_size, date d) {
   int sum = 0;
   for (int i = 0; i < ra_size; ++i) {
      sum += ra->purchase_date[i] ? ra->price[i]*ra->quantity[i] : 0;
   }
   for (int i = 0; i < sa_size; ++i) {
      sum += sa->purchase_date[i] ? sa->price[i]*sa->quantity[i]*
```

C++ Memory Model

```
struct Foo {
   int a;
   int b;
   struct Bar {
     int c;
   } bar;
} foo;
```

C++ Memory Model

```
struct Foo {
   int a;
   int b;
   struct Bar {
     int c;
   } bar;
} foo;
```

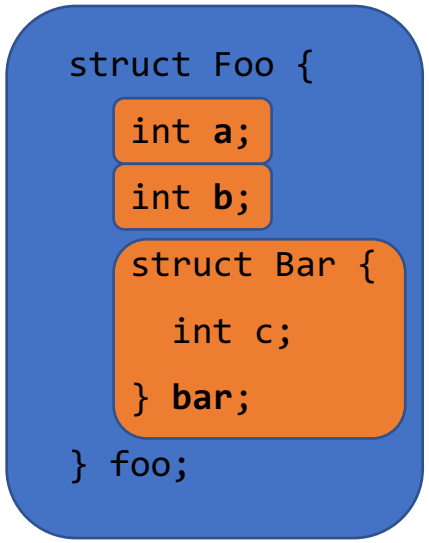
object

C++ Memory Model

```
struct Foo {
   int a;
   int b;
   struct Bar {
     int c;
   } bar;
} foo;
```

complete object

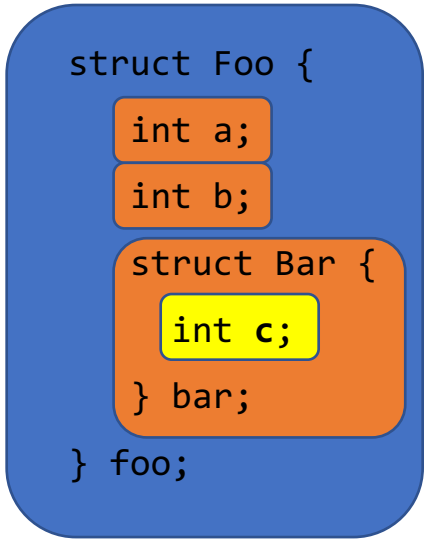
C++ Memory Model




```
member sub-object of foo    offset 0
member sub-object of foo    offset 4

member sub-object of foo    offset 8
complete object
```

C++ Memory Model




```
member sub-object of foo    offset 0
member sub-object of foo    offset 4

member sub-object of bar    offset 0
member sub-object of foo    offset 8
complete object
```

C++ Memory Model

- Logical layout is mapped to **continuous** *physical* layout
- Object identity is therefore also represented *physically* using abstractions over **memory addresses** (pointers and references)

Catching Layouts

Dealing with multiple layouts

```
struct vec3f {
  float x, y, z;
};

float f1(vec3f a) {
  return a.x + 1;
}
```

```
struct vec3f_sec {
  float z, y, x;
};

float f2(vec3f_sec a) {
  return a.x + 1;
}
```

Dealing with multiple layouts

```
struct vec3f {
  float x, y, z;
};

extern float magnitude(vec3f);

float f1(vec3f a) {
  return magnitude(a);
}
```

```
struct vec3f_sec {
  float z, y, x;
};
```

Dealing with multiple layouts

```
struct vec3f {

  float x, y, z;
};
```

```
struct vec3f_sec {
  using primary = vec3f;
  float z, y, x;
  vec3f_sec(const primary& o) : x(o.x), y(o.y), z(o.z) {}
  operator =(const vec3f& o) {…}
  operator primary() { return primary(x, y, x);}
};
```

```
extern float magnitude(vec3f);

float f1(vec3f a) {
  return magnitude(a);
}
```

```
float f2(vec3f_sec b) {
  return magnitude(b);
}
```

Dealing with multiple layouts

```
struct vec3f {
  float x, y, z;

  float magnitude() {
    return std::sqrt(x*x + y*y + z*z);
  }
};
```

```
struct vec3f_sec {
  using primary = vec3f;
  float z, y, x;
  vec3f_sec(const primary& o) : x(o.x), y(o.y), z(o.z) {}
  operator =(const vec3f& o) {…}
  operator primary() { return primary(x, y, x);}
};
```

```
float f1(vec3f a) {
  return a.magnitude();
}
```

```
float f2(vec3f_sec b) {
  return (static_cast<vec3f>(b)).magnitude();
}
```

Dealing with multiple layouts

```
struct vec3f {
  float x, y, z;

  float magnitude() {
    return std::sqrt(x*x + y*y + z*z);
  }
};
```

```
float f1(vec3f a) {
  return a.magnitude();
}
```

```
struct vec3f_sec {
  using primary = vec3f;
  float z, y, x;
  vec3f_sec(const primary& o) : x(o.x), y(o.y), z(o.z) {}
  operator =(const vec3f& o) {…}
  operator primary() { return primary(x, y, x);}

  float magnitude() {
    primary self = std::move(*this);
    auto&& result = self.magnitude();
    *this = std::move(self);
    return result;
   }
};
```

```
float f2(vec3f_sec b) {
  return b.magnitude();
}
```

Dealing with multiple layouts

```
struct vec3f {

  float x, y, z;




  vec3f& operator +=(vec3f b) {
    a.x+=b.x;
    a.y+=b.y;
    a.x+=b.z;
    return *this;
  }
};
```

```
struct vec3f_sec {
  using primary = vec3f;
  float z, y, x;
  vec3f_sec(const primary& o) : x(o.x), y(o.y), z(o.z) {}
  operator =(const vec3f& o) {…}
  operator primary() { return primary(x, y, x);}
};
```

Dealing with multiple layouts

```
struct vec3f {

  float x, y, z;




  vec3f& operator +=(vec3f b) {
    a.x+=b.x;
    a.y+=b.y;
    a.x+=b.z;
    return *this;
  }
};
```

```
struct vec3f_sec {
  using primary = vec3f;
  float z, y, x;
  vec3f_sec(const primary& o) : x(o.x), y(o.y), z(o.z) {}
  operator =(const vec3f& o) {…}
  operator primary() { return primary(x, y, x);}

  vec3f_sec& operator +=(primary b) {
    primary self = std::move(*this);
    auto&& result = (self += b;)
    *this = std::move(self);
    assert(&result == &self);
    return *this;
  }
};
```

Dealing with multiple layouts

```
struct vec3f {

  float x, y, z;




  vec3f& operator +=(vec3f b) {
    a.x+=b.x;
    a.y+=b.y;
    a.x+=b.z;
    return *this;
  }
};
```

```
struct vec3f_sec {
  using primary = vec3f;
  float z, y, x;
  vec3f_sec(const primary& o) : x(o.x), y(o.y), z(o.z) {}
  operator =(const vec3f& o) {…}
  operator primary() { return primary(x, y, x);}

  vec3f_sec& operator +=(primary b) {
    [[clang::always_inline]] primary self = std::move(*this);
    [[clang::always_inline]] auto&& result = (self += b;)
    [[clang::always_inline]] *this = std::move(self);
    assert(&result == &self);
    return *this;
  }
};
```

Dealing with multiple layouts

Requirements and limits on types

1. Member variables need to be move/copy assignable
2. Functions that leak the object's identity can not be converted

Properties of layouts

1. All layouts of the same platonic type have the same public interface and represent the same value-set
2. Any layout is therefore losslessly convertible to any other layout of the same platonic type

Hot cold split

```
struct obj {
  int pos_x;
  int pos_y;
  unsigned texture_id;
  std::string asset_id;
  std::string file_name;
};
```

```
struct obj_sec {
  /* constructors / conversion operators */
  int pos_x;
  int pos_y;
  unsigned texture_id;
  [[cold_data]] std::string asset_id;
  [[cold_data]] std::string file_name;
};
```

Hot cold split

```
struct obj {
  int pos_x;
  int pos_y;
  unsigned texture_id;
  std::string asset_id;
  std::string file_name;
};
```

```
int f(obj& o) {
  return o.asset_id == "" ? o.pos_x : o.pos_y;
}
```

```
struct obj_sec {
  /* constructors / conversion operators */
  int pos_x;
  int pos_y;
  unsigned texture_id;
  struct COLD {
    std::string asset_id;
    std::string file_name;
  };
  std::unique_ptr<COLD> cold_data;
};
```

```
int f(obj_sec& o) {
  return o.cold_data->asset_id == "" ? o.pos_x : o.pos_y;
}
```

Hot cold split

```
struct obj {
  int pos_x;
  int pos_y;
  unsigned texture_id;
  std::string asset_id;
  std::string file_name;
};
```

```
int f(obj& o) {
  return o.asset_id == "" ? o.pos_x : o.pos_y;
}
```

```
struct obj_sec {
  /* constructors / conversion operators */
  int pos_x;
  int pos_y;
  unsigned texture_id;
  struct COLD {
    std::string asset_id;
    std::string file_name;
  };
  std::unique_ptr<COLD> cold_data;
  property(std::string, cold_data->asset_id) asset_id;
  property(std::string, cold_data->file_name) file_name;
};
```

```
int f(obj_sec& o) {
  return o.asset_id == "" ? o.pos_x : o.pos_y;
}
```

Hot cold split

- Using properties we can ensure trivial conversion from fractured layouts to other layouts of the same type
- Properties are proxies for the objects they represent

Structure of Arrays

```
struct vec {
   int x, y, z;

   vec& operator +=(const vec& rhs) {
      x += rhs.x;
      y += rhs.y;
      z += rhs.z;

      return *this;
   }

   int magnitude() { return std::sqrt(x*x + y*y + z*z); }
};

std:array<vec, 10> primairy;
```

```
array_soa_vec<Size> {
   std::array<int, Size> data_x;
   std::array<int, Size> data_y;
   std::array<int, Size> data_z;
}

array_soa_vec<10> secondary;
```

Structure of Arrays

```
struct vec {
   int x, y, z;

   vec& operator +=(const vec& rhs) {
      x += rhs.x;
      y += rhs.y;
      z += rhs.z;

      return *this;
   }

   int magnitude() { return std::sqrt(x*x + y*y + z*z); }
};

std:array<vec, 10> primairy;
```

```
array_soa_vec<Size> {
   std::array<int, Size> data_x;
   std::array<int, Size> data_y;
   std::array<int, Size> data_z;
}

array_soa_vec<10> secondary;
```

Structure of Arrays

```
struct vec {
   int x, y, z;

   vec& operator +(const vec& rhs) {
      x += rhs.x;
      y += rhs.y;
      z += rhs.z;

       return *this;
   }

   int magnitude() { return std::sqrt(x*x + y*y + z*z); }
};

std:array<vec, 10> primairy;
```

```
array_soa<vec, Size> {
   std::array<int, Size> data_x;
   std::array<int, Size> data_y;
   std::array<int, Size> data_z;

   struct proxy {
      array_soa* vec;
      int index;
      property(int, vec->data_x[index]) x;
      property(int, vec->data_y[index]) y;
      property(int, vec->data_z[index]) z;
      operator vec() { return {x, y, z}; }
   }

   proxy operator[](int index) { return {this, index}; }
}

array_soa_vec<10> secondary;
```

Structure of Arrays

```
std:array<vec, 10> primairy;

int foo() {
   return std::accumulate(primary.cbegin(), primary.cend(),
vec{}).magnitude();
}
```

```
array_soa_vec<10> secondary;

int foo_sec() {
   return std::accumulate(secondary.cbegin(),
secondary.cend(), vec{}).magnitude();
}
```

Structure of Arrays

```
std:array<vec, 10> primairy;

int foo() {
   for (auto& vec : ar2)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   auto* ptr = ar2.data();
   for (int i =0; I < ar2.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

```
array_soa_vec<10> secondary;

int foo_sec() {
   for (auto vec : ar1)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   auto ptr = ar1.data();
   for (int i =0; I < ar1.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

Structure of Arrays

```
std:array<vec, 10> primairy;

int foo() {
   for (auto_ref vec : ar2)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   Ptr auto ptr = ar2.data();
   for (int i =0; I < ar2.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

```
array_soa_vec<10> secondary;

int foo_sec() {
   for (auto_ref vec : ar1)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   Ptr auto ptr = ar1.data();
   for (int i =0; I < ar1.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

Structure of Arrays

```
std:array<vec, 10> primairy;

int foo() {
   for (vec& vec : ar2)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   vec* ptr = ar2.data();
   for (int i =0; I < ar2.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

```
array_soa_vec<10> secondary;

int foo_sec() {
   for (auto_ref vec : ar1)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   Ptr auto ptr = ar1.data();
   for (int i =0; I < ar1.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

Structure of Arrays

```
std:array<vec, 10> primairy;

int foo() {
   for (ref(vec) vec : ar2)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   PtrTo<vec> auto ptr = ar2.data();
   for (int i =0; I < ar2.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

```
array_soa_vec<10> secondary;

int foo_sec() {
   for (ref(vec) vec : ar1)
      if (vec.y > 10) vec.z += vec.x;
}

int bar() {
   PtrTo<vec> auto ptr = ar1.data();
   for (int i =0; I < ar1.size(); ++i, ++ptr)
      if (vec->y > 10) vec->z += vec->x;
}
```

Overview

Mono-layout	Multi-layout
T& t	ml_ref(T) t
T* t	ml::PtrTo<T> auto t
auto& t	ml::auto_ref t
auto* t	ml::Ptr auto t
std::unique_ptr<T> t	ml::unique_ptr<T> auto t
std::shared_ptr<T> t	ml::shared_ptr<T> auto t

auto_ref

```
template <typename T>
struct auto_ref;

template <typename T>
concept Proxy = is_proxy<T>::value;

template <typename T>
concept Value = !is_proxy<T>::value;

template <Value T> auto_ref(T& t) -> auto_ref<T>;
template <Proxy T> auto_ref(T t) -> auto_ref<T>;
```

ref(T)

```
#define ref(T) ref_ ## T

template <typename T, typename Blueprint>
concept ProxyFor = is_proxy<T>::value && std::is_same<Blueprint, typename proxy_info<T>::blueprint>::value;

template <typename T, typename Blueprint>
concept ValueFor = !is_proxy<T>::value && (std::is_same<Blueprint, T>::value || std::is_same<Blueprint, typename
proxy_info<T>::blueprint>::value);

template <typename T>
struct ref_vec;

template <ValueFor<vec> T> ref_vec(T& t) -> ref_vec<T>;
template <ProxyFor<vec> T> ref_vec(T t)  -> ref_vec<T>;
```

Ptr auto

```
template <typename T>
concept Ptr = std::is_pointer<T>::value || is_pointer_to_proxy<T>::value;
```

PtrTo<T> auto

```
template <typename T, typename Blueprint>
concept PtrTo = std::is_same<T, Blueprint*>::value || (is_pointer_to_proxy<T>::value && std::is_same<Blueprint, typename proxy_info<T>::blueprint>::value);
```

Rising to the challenge

Optional

```
struct obj {
  int value;
  int height;
  std::optional<additional_data_1> data_1;
  std::optional<additional_data_2> data_2;
};

std::array<obj, 10> array;
```

```
struct obj_core {
   int value;
   int height;
};

struct {
  std::array<obj_core, 10> data_obj;
  std::map<size_t, additional_data_1> data_1;
  std::map<size_t, additional_data_2> data_2;
} array;
```

Optional

```
std::array<obj, 10> array;

void a() {
  for (auto& o : array) {
    if (o.data_1) {
      process1(o.value, o.height * o.data_1->factor);
    } else {
      process1(o.value, 1);
    }

    if (o.data_2) {
      process2(o.height*o.data_2->size, o.data_2->factor);
    }
  }
}
```

```
} array;

void b() {
  for (auto [index, data] : array.data_1) {
    auto& o = array.data_obj[index];
    process1(o.value, o.height * data.factor);
  }

  for (auto index : std::views::iota(0, array.size())
                  | std::views::filter([](size_t index){
                      return !array.data_1.contains(index);
                  })) {
    auto& o = array.data_obj[index];
    process1(o.value, 1);
  }

  for (auto [index, data] : array.data_2) {
    auto& o = array.data_obj[index];
    process2(o.height*data.size, data.factor);
  }
}
```

Optional

```
std::array<obj, 10> array;

void a() {
  for (auto& o : array) {
    if (o.data_1) {
      process1(o.value, o.height * o.data_1->factor);
    } else {
      process1(o.value, 1);
    }

    if (o.data_2) {
      process2(o.height*o.data_2->size, o.data_2->factor);
    }
  }
}
```

```
} array;

void b() {
  for (auto& o : array.filter_deref<"data_1">()) {
    process1(o.value, o.height * o.data_1.factor);
  }

  for (auto& o : array.filter_deref<"!data_1">()) {
    process1(o.value, 1);
  }

  for (auto& o : array.filter_deref<"data_2">()) {
    process2(o.height*o.data_2.size, o.data_2.factor);
  }
}
```

Variadic

```
struct obj {
  int value;
  int height;
  std::variant<additional_data_1, additional_data_2> data;
};

std::array<obj, 10> array;
```

```
struct obj_core {
  int value;
  int height;
};

struct {
  std::array<obj_core, 10> data_obj;
  std::map<size_t, additional_data_1> data_1;
  std::map<size_t, additional_data_2> data_2;
} array;
```

Variant

```
std::array<obj, 10> array;

void a() {
  for (auto& o : one::array) {
    if (auto* data_1 =
std::get_if<additional_data_1>(o.data); data_1 != nullptr) {
            process1(o.value, o.height * data_1->factor);
    } else {
      auto& data_2 = std::get<additional_data_2>(o.data)
      process2(o.height*data_2.size, o.data_2.factor);
    }
  }
}
```

```
} array;

void b() {
  for (auto [index, data] : array.data_1) {
    auto& o = array.data_obj[index];
    process1(o.value, o.height * data.factor);
  }

  for (auto [index, data] : array.data_2) {
    auto& o = array.data_obj[index];
    process2(o.height*data.size, data.factor);
  }
}
```

Variant

```
std::array<obj, 10> array;

void a() {
  for (auto& o : one::array) {
    if (auto* data_1 =
std::get_if<additional_data_1>(o.data); data_1 != nullptr) {
            process1(o.value, o.height * data_1->factor);
    } else {
      auto& data_2 = std::get<additional_data_2>(o.data)
      process2(o.height*data_2.size, o.data_2.factor);
    }
  }
}
```

```
} array;

void b() {
  for (auto& o : array.filter_deref<"data is additional_data_1">()) {
    process1(o.value, o.height * o.data.factor);
  }

  for (auto& o : array.filter_deref<"data is additional_data_2">()) {
    process2(o.height*o.data.size, o.data.factor);
  }
}
```

Variant

```
std::array<obj, 10> array;

void a() {
  for (auto& o : one::array) {
    if (auto* data_1 =
std::get_if<additional_data_1>(o.data); data_1 != nullptr) {
            process1(o.value, o.height * data_1->factor);
    } else {
      auto& data_2 = std::get<additional_data_2>(o.data)
      process2(o.height*data_2.size, o.data_2.factor);
    }
  }
}
```

```
fn c(ar : &[Foo; 10]) {
  for foo in ar.iter() {
    match foo.data {
      AdditionalData::One{factor, ..}
        => process1(foo.value, foo.height as f32 * factor),
      AdditionalData::Two{size, factor, ..}
        => process2(foo.height*size, factor)
    }
  }
}
```

Rust enums

```
enum AdditionalData {
   One { name : String, id : String, factor : f32},
   Two { size : i32, factor : f32}
}

struct Foo {
   value : i32,
   height : i32,
   data : AdditionalData
}

fn c(ar : &[Foo; 10]) {
  for foo in ar.iter() {
    match foo.data {
      AdditionalData::One{factor, ..}
        => process1(foo.value, foo.height as f32 * factor),
      AdditionalData::Two{size, factor, ..}
        => process2(foo.height*size, factor)
    }
  }
}
```

Rust enums

P1371R2: *Pattern Matching*

```
to_variadic_enum(struct AdditionalData {
   struct One {std::string name; std::string id; float factor};
   struct Two { int size; float factor};
};);

struct Foo {
   int value;
   int height;
   AdditionalData data;
}

void c(const std::array<Foo, 10>& ar) {
  std::ranges::for_each(ar, [](auto_ref foo) {
    inspect (foo.data) {
      <AdditionalData::One> one: process1(foo.value, foo.height * one.factor);
      <AdditionalData::Two> two: process2(foo.height*two.size, two.factor)
    }
  });
}
```

Ad Hoc

```
var Core = {
  value : 0,
  height : 0
}

var AdditionalDataTwo = {
  size : 0,
  factor : 0
};

function f(core,additionalDataTwo) {
  var combined = {...core, ...additionalDataTwo};
  process2(combined.height * combined.size, combined.factor)
}
```

```
struct Core  {
  int value;
  int height;
};

struct AdditionalDataTwo {
  int size;
  int factor;
};
```

Ad Hoc

```
var Core = {
  value : 0,
  height : 0
}

var AdditionalDataTwo = {
  size : 0,
  factor : 0
};

function f(core,additionalDataTwo) {
  var combined = {...core, ...additionalDataTwo};
  process2(combined.height * combined.size, combined.factor)
}
```

```
struct Core  {
  int value;
  int height;
};

struct AdditionalDataTwo {
  int size;
  int factor;
};

void f(Core core, AdditionalDataTwo additionalDataTwo) {
  auto_ref combined = restructure_ref(core, additionalDataTwo);
  process2(combined.height * combined.size, combined.factor)
}
```

Ad Hoc

```
var Core = {
  value : 0,
  height : 0
}

var AdditionalDataTwo = {
  size : 0,
  factor : 0
};

function f2(arrCore,arrAdditionalDataTwo) {
  arrCore.map((value,index) => {
    return {...value,...arrAdditionalDataTwo[index]}
  })
  .forEach(combined => process2(combined.height *
     combined.size, combined.factor));
}
```

```
struct Core  {
  int value;
  int height;
};

struct AdditionalDataTwo {
  int size;
  int factor;
};
using DataTwo = AdditionalDataTwo
void f(std::array<Core,10> core, std::array<DataTwo, 10> DataTwo) {
  auto combined_v = std::views::zip_transform(restructure_ref<Core,
DataTwo>, core, DataTwo);
  std::ranges::for_each(combined_v, [](const auto_ref combined) {
process2(combined.height * combined.size, combined.factor);}
}
```

Ad Hoc

```
var Core = {
  value : 0,
  height : 0
}

var AdditionalDataTwo = {
  size : 0,
  factor : 0
};

function f2(arrCore,arrAdditionalDataTwo) {
  arrCore.map((value,index) => {
    return {...value,...arrAdditionalDataTwo[index]}
  })
  .forEach(combined => process2(combined.height *
     combined.size, combined.factor));
}
```

```
struct Core  {
  int value;
  int height;
};

struct AdditionalDataTwo {
  int size;
  int factor;
};

void f(const Core& core, const AdditionalDataTwo& additionalDataTwo) {
  auto combined_v = zip_restructure_ref(core, additionalDataTwo);
  std::ranges::for_each(combined_v, [](const auto_ref combined) {
process2(combined.height * combined.size, combined.factor);}
}
```

The Gap

We need properties

```
struct Asset {
  struct COLD {
    std::string asset_id;
    std::string file_name;
  };
  std::unique_ptr<COLD> cold_data;
  property(std::string, cold_data->asset_id) asset_id;
  property(std::string, cold_data->file_name) file_name;

  int pos_x;
  int pos_y;
  unsigned texture_id;
};
```

We need properties

```
struct Asset {
  CPP_PROP_data({
    struct COLD {
      std::string asset_id;
      std::string file_name;
    };
    std::unique_ptr<COLD> cold;
  })

  CPP_PROP_property(std::string, cold->asset_id) asset_id;
  CPP_PROP_property(std::string, cold->file_name) file_name;

  int pos_x;
  int pos_y;
  unsigned texture_id;

  ...
};
```

We need properties

```
struct Asset {
  CPP_PROP_data({
    struct COLD {
      std::string asset_id;
      std::string file_name;
    };
    std::unique_ptr<COLD> cold;

    CPP_PROP_data_t(std::unique_ptr<COLD>&& c) : cold(std::move(c)) {}
    CPP_PROP_data_t(const CPP_PROP_data_t& other) : cold(std::make_unique<COLD>(*other.cold)) {}
    CPP_PROP_data_t(CPP_PROP_data_t&& other) : cold(std::move(other.cold)) {}
  })

  CPP_PROP_property(std::string, cold->asset_id) asset_id;
  CPP_PROP_property(std::string, cold->file_name) file_name;

  int pos_x;
  int pos_y;
  unsigned texture_id;

  ...
};
```

We need properties

```
#define CPP_PROP_data(DATA) struct CPP_PROP_data_t DATA CPP_PROP_data_m;
```

```
CPP_PROP_data({
  struct COLD {
    std::string asset_id;
    std::string file_name;
  };
  std::unique_ptr<COLD> cold;

  CPP_PROP_data_t(std::unique_ptr<COLD>&& c)
    : cold(std::move(c)) {}
  CPP_PROP_data_t(const CPP_PROP_data_t& other)
    : cold(std::make_unique<COLD>(*other.cold)) {}
  CPP_PROP_data_t(CPP_PROP_data_t&& other)
    : cold(std::move(other.cold)) {}
})
```

```
struct CPP_PROP_data_t {
  struct COLD {
    std::string asset_id;
    std::string model_file_name;
  };
  std::unique_ptr<COLD> cold;

  CPP_PROP_data_t(std::unique_ptr<COLD>&& c)
    : cold(std::move(c)) {}
  CPP_PROP_data_t(const CPP_PROP_data_t& other)
    : cold(std::make_unique<COLD>(*other.cold)) {}
  CPP_PROP_data_t(CPP_PROP_data_t&& other)
    : cold(std::move(other.cold)) {}
} CPP_PROP_data_m;
```

We need properties

```
struct Asset {
  CPP_PROP_data({
    struct COLD {
      std::string asset_id;
      std::string file_name;
    };
    std::unique_ptr<COLD> cold;

    CPP_PROP_data_t(std::unique_ptr<COLD>&& c) : cold(std::move(c)) {}
    CPP_PROP_data_t(const CPP_PROP_data_t& other) : cold(std::make_unique<COLD>(*other.cold)) {}
    CPP_PROP_data_t(CPP_PROP_data_t&& other) : cold(std::move(other.cold)) {}
  })

  CPP_PROP_property(std::string, cold->asset_id) asset_id;
  CPP_PROP_property(std::string, cold->file_name) file_name;

  int pos_x;
  int pos_y;
  unsigned texture_id;

  ...
};
```

We need properties

```
#define CPP_PROP_property(TYPE_T, ACCESSOR) \
CPP_PROP_NO_UNIQUE_ADRESS cpp_prop::property<TYPE_T, CPP_PROP_data_t, \
decltype([](CPP_PROP_data_t*CPP_PROP_prop_data)->TYPE_T&{return (CPP_PROP_prop_data->ACCESSOR;}) >
```

```
CPP_PROP_property(
  std::string,
  cold->asset_id
) asset_id;
```

```
[[no_unique_address]]
cpp_prop::property<
  std::string,
  CPP_PROP_data_t,
  decltype([](CPP_PROP_data_t*CPP_PROP_data)->std::string&
    {return CPP_PROP_data->cold->asset_id;})
> asset_id;
```

We need properties

```
template <typename T, typename Base, typename Getter>
struct property {
  T& get() {
    return (Getter{}(reinterpret_cast<Base*>(this)));
  }
  const T& get() const {
    return (Getter{}(reinterpret_cast<Base*>(this)));
  }


  T& operator= (const T& rhs) { return get() = rhs; }
  T& operator= (T&& rhs) { return get() = std::forward<T>(rhs); }
  operator T&() { return get(); }
  operator const T&() const { return get(); }
  T* operator&() { return &get(); }
  const T* operator&() const { return &get(); }
};
```

```
T = std::string
Base = CPP_PROP_data_t
Getter = [](CPP_PROP_data_t* CPP_PROP_data) ->std::string& {
  return CPP_PROP_data->cold->asset_id;
}
```

We need properties

- UB in library implementation
- Unexpected ADL

We need properties

```
struct Asset {
   struct COLD {
      std::string asset_id;
      std::string file_name;
   };
   std::unique_ptr<COLD> cold_data;
   property(std::string, cold_data->asset_id) asset_id;
   property(std::string, cold_data->file_name) file_name;

   int pos_x;
   int pos_y;
   unsigned texture_id;
};
```

We need properties

N4477: *operator dot R2*

```
struct Asset {
   struct COLD {
      std::string asset_id;
      std::string file_name;
   };
   struct HOT {
      int pos_x;
      int pos_y;
      unsigned texture_id;
   };

   COLD& operator.() { return *cold_data;}
   HOT& operator.() { return hot_data; }
private:
   std::unique_ptr<COLD> cold_data;
   HOT hot_data;
};
```

We need properties

P0352R1: *Smart References through delegation*

```
struct COLD {
   std::string asset_id;
   std::string file_name;
};

struct HOT {
   int pos_x;
   int pos_y;
   unsigned texture_id;
};

struct Asset : public using COLD, public using HOT {
   operator COLD& () { return *cold_data;}
   operator HOT& () { return hot_data; }
private:
   std::unique_ptr<COLD> cold_data;
   HOT hot_data;
};
```

We need properties

P1950R2: *indirect_value: A Free-Store-Allocated Value Type For C++*

```
struct COLD {
   std::string asset_id;
   std::string file_name;
};

struct Asset {
   indirect_value<COLD> cold_data;
   int pos_x;
   int pos_y;
   unsigned texture_id;

};

void use_asset(Asset& asset) {
   asset.pos_x = 1;
   asset.texture_id = 0;
   asset.cold_data->asset_id = 3;
}
```

We need reflection

- Build smart containers
- Generate other layouts (and their interfaces)
- Build decent proxy references

We need reflection

- Inspect all public, protected, and private class members
- Reflect into any scope even those unrelated to the callsite

What’s next

What’s next?

- We cannot cover the problem space

- Standardize tools **not** solutions (properties, reflection)
- Standardize vocabulary types (ptr, ref) and the ability to link different layouts

Questions?

https://godbolt.org/z/KohYo4sTP

Floris Bob van Elzelingen

C++Now, Aspen

May 11th 2023

@FlorisBob

florisbobvanelzelingen@gmail.com