Many Ways to Kill an Orc (or a Hero)

PATRICE ROY

Cppcon
The C++ Conference
2024
September 15 - 20

Many ways to kill an Orc

(or a Hero)

Patrice Roy, Patrice.Roy@USherbrooke.ca, Patrice.Roy@clg.qc.ca

From the abstract…

- « ***Our game programmers and game engines involve fights between heroes and their foes*** *[…]* »

From the abstract…

- *« Our game programmers and game engines involve fights between heroes and their foes.* ***There are «classical», traditional ways to express heroes and monsters fighting each other, but C++ is a particularly expressive and versatile language, and with C++20 and C++23 there are many ways for heroes and monsters to hit at each other*** *[…] »*

From the abstract…

- *« Our game programmers and game engines involve fights between heroes and their foes. There are «classical», traditional ways to express heroes and monsters fighting each other, but C++ is a particularly expressive and versatile language, and with C++20 and C++23 there are many ways for heroes and monsters to hit at each other.* ***This is what this talk will explore*** *»*

From the abstract…

- « *Our game programmers and game engines involve fights between heroes and their foes. There are «classical», traditional ways to express heroes and monsters fighting each other, but C++ is a particularly expressive and versatile language, and with C++20 and C++23 there are many ways for heroes and monsters to hit at each other. This is what this talk will explore* »

At this point, you're probably wondering « **what's this talk *actually* about?** »

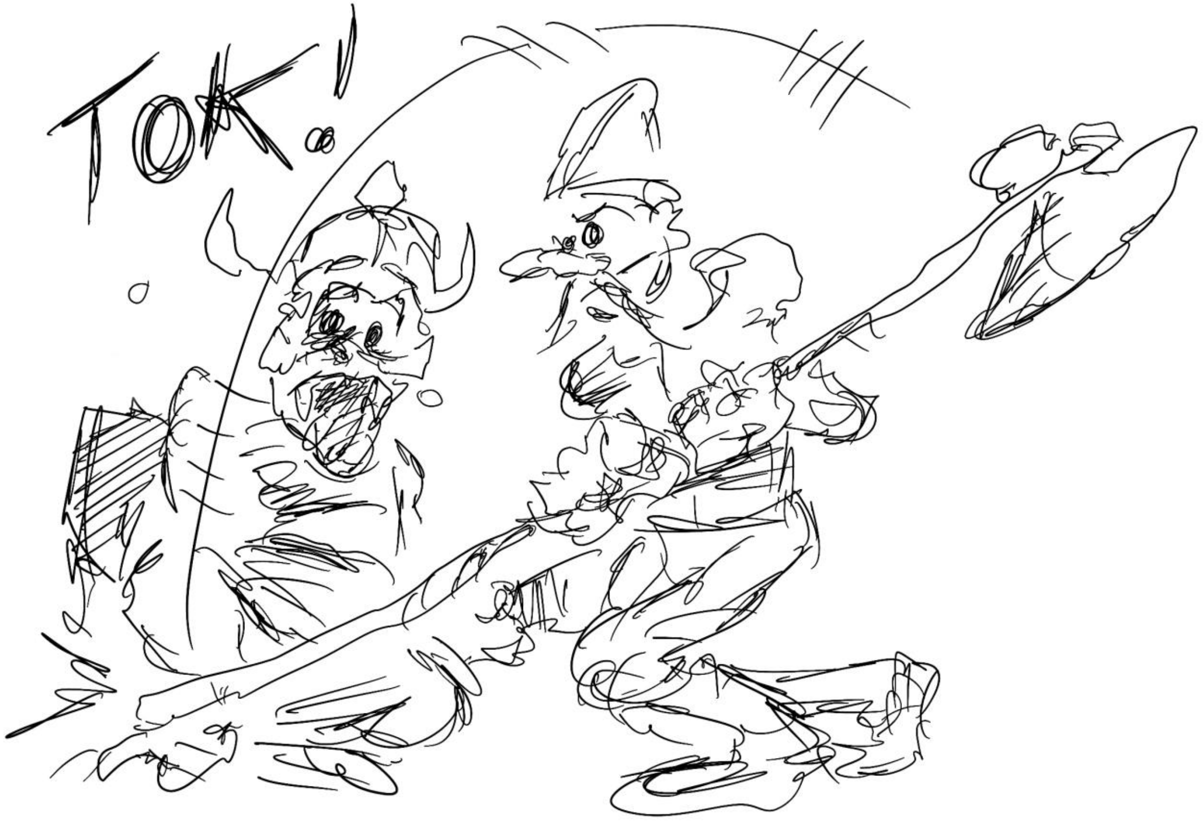
That's quite understandable

What this is about?

What this is about?

What this is about?

What this is about?



What this is about?

- Lots of hitting ensues…

What this is about?

- (censored bloodshedding…)

What this is about?

- … so no, that’s not really what this talk is about
- Well, it is *somewhat*, but indirectly

What this is about?

- **Important note**: throughout this talk, we'll be looking at different ways to do similar things
- Feel free to **raise questions**, **make comments**, **criticize**, etc. as we go along!

What this is about?

- **Important note**: throughout this talk, we'll be looking at different ways to do similar things
- Feel free to **raise questions**, **make comments**, **criticize**, etc. as we go along!

There will be **lots** of code in these slides, so pay attention mostly to what is in **boldface** (everything will be made available after the talk if you want to do a more thorough reading)

What this is about?

```
// the easy case
import std; // or #include <random>, <string>, <string_view>, <print>
class Character {
   std::string name_;
   int life_;
public:
   constexpr Character(std::string_view name, int life)
     : name_{ name }, life_{ life } {
   }
   constexpr std::string name() const { return name_; }
   constexpr bool alive() const { return life() > 0; }
   constexpr bool dead() const { return !alive(); }
   constexpr int life() const { return life_; }
   constexpr void suffer(int damage) { life_ -= damage; }
};
```

A Character can suffer
(among other things)

What this is about?

```
// the easy case
// ...
class Character { /* ... */ };
class Orc;
class Hero : Character {
   int strength_;
public:
   constexpr Hero(std::string_view name, int life, int strength)
      : Character { name, life }, strength_{ strength } {
   }
   constexpr int strength() const { return  strength_; }
   using Character::name, Character::alive, Character::dead,
         Character::life, Character::suffer;
   void hit(Orc&);
};
```

A Hero is a Character that can hit an Orc

What this is about?

```
// the easy case
// ...
class Character { /* ... */ };
class Orc;
class Hero : Character { /* ... */ };
class Orc : Character {
   int strength_;
public:
   constexpr Orc(std::string_view name, int life, int strength)
      : Character { name, life }, strength_{ strength } {
   }
   constexpr int strength() const { return  strength_; }
   using Character::name, Character::alive, Character::dead,
         Character::life, Character::suffer;
   void hit(Hero&);
};
```

An Orc is a Character that can hit a Hero

What this is about?

```
// the easy case
// ...
class Character { /* ... */ };
class Orc;
Class Hero : Character { /* ... */ };
class Orc : Character { /* ... */ };
void Hero::hit(Orc &orc) {
   orc.suffer(strength());
}
void Orc::hit(Hero &hero) {
   hero.suffer(strength());
}
```

What this is about?

```
// the easy case
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   Orc orc{ "URG", 100, dice(prng) / 10 };
   Hero hero{ "William", 100, dice(prng) / 10 };
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         hero.hit(orc);
      else
         orc.hit(hero);
   }
   std::print("{} won", orc.alive() ? orc.name() : hero.name());
}
```

Violence ensues...

What this is about?

```
// the easy case
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   Orc orc{ "URG", 100, dice(prng) / 10 };
   Hero hero{ "William", 100, dice(prng) / 10 };
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         hero.hit(orc);
      else
         orc.hit(hero);
   }
   std::print("{} won", orc.alive() ? orc.name() : hero.name());
}
```

https://wandbox.org/permlink/eYFgO1AcBCplFRa6

What this is about?

- What do you think of this implementation of the fight between good and evil?

What this is about?

- Such violence…

What this is about?

- But that’s not the point of this talk
- What we will really explore here is ways in which to implement this conflict between good and evil
- As we do this, we will discuss the design choices we make

What this is about?

- We always make design choices when we code
  - Even if we choose to just « throw brute force code » at a problem
  - It might not be the best design choice there is, but it’s still a choice
- In our « easy case » implementation, we already made some choices that might deserve discussion

What this is about?

```
class Character { /* ... name_, life_, etc. */ };
// ...
class Hero : Character {
   int strength_;
public:
   constexpr Hero(std::string_view name, int life, int strength)
      : Character { name, life }, strength_{ strength } {
   }
   using Character::name, Character::alive, Character::dead,
         Character::life, Character::suffer;
   // ...
};
```

What this is about?

```
class Character { /* ... name_, life_, etc. */ };
// ...
class Hero : Character {
   int strength_;
public:
   constexpr Hero(std::string_view name, int life, int strength)
      : Character { name, life }, strength_{ strength } {
   }
   using Character::name, Character::alive, Character::dead,
         Character::life, Character::suffer;
   // ...
};
```

Private inheritance: other classes (if non-friend) are not aware of this base-derived relationship

Deliberate exposure of base class' selected services as our own

What this is about?

- Private inheritance means client code (other than friend functions and classes) cannot fully use the base-derived relationship

```
#include <type_traits>
// ...
static_assert(std::is_base_of_v<Character, Hero>); // Ok
void f(const Character&) {}
int main() {
   // does not compile (inaccessible base class)
   f(Hero{ "", 0, 0 }); // helps avoid "slicing"
}
```

What this is about?

- In this overly simplified example, it’s reasonable to use private inheritance
  - There’s no benefit for client code to be able to use knowledge of the base-derived relationship with the implementation we wrote
- Of course, it often happens that we want public inheritance in order for that relationship to be usable by client code
  - The base class usually exposes at least one virtual member function when this happens

What this is about?

```
import std; // or #include <random>, <string>, <string_view>,
            // <print>, <format>
class Character {
   std::string name_;
   int life_;
public:
   constexpr Character(std::string_view name, int life)
      : name_{ name }, life_{ life } {
   }
   virtual std::string name() const { return name_; } // <--
   virtual ~Character() = default;                    // <--
   // alive(), dead(), life(), suffer(int)
};
```

What this is about?

```cpp
// ...
class Character { /* ... */ };
class Orc;
class Hero : public Character { // <--
   // ...
   void hit(Orc&);
};
class Orc : public Character { // <--
   // ...
   std::string name() const override { // <--
       return std::format("Me is {}", Character::name());
   }
   void hit(Hero&);
};
```

What this is about?

```
// ...
class Character { /* ... */ };
class Orc;
class Hero : public Character { /* ... */ };
class Orc : public Character { /* ... */ };
void Hero::hit(Orc &orc) { orc.suffer(strength()); }
void Orc::hit(Hero &hero) { hero.suffer(strength()); }
// we can try to benefit from that newfound knowledge
void attack(Character &from, Character &to) {
   from.hit(to); // <-- OOPS
}
void print_result(const Character &a, const Character &b) {
   std::print("{} won", a.alive() ? a.name() : b.name());
}
```

What this is about?

```
// ...
class Hero : public Character { /* ... */ };
class Orc : public Character { /* ... */ };
// ...
void attack(Character &from, Character &to) { /* ... */ }
void print_result(const Character &a, const Character &b) { /* ... */ }
int main() {
   // ...
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc); // <-- OOPS
      else
         attack(orc, hero); // <-- OOPS
   }
   print_result(orc, hero);
}
```

What this is about?

```
// ...
class Hero : public Character { /* ... */ };
class Orc : public Character { /* ... */ };
// ...
void attack(Character &from, Character &to) { /* ... */ }
void print_result(const Character &a, const Character &b) { /* ... */ }
int main() {
   // ...
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc); // <-- OOPS
      else
         attack(orc, hero); // <-- OOPS
   }
   print_result(orc, hero);
}
```

This code does not compile due to the way attack() is written: it tries to make a Character hit() a Character, but there's no such function in our design!

What this is ab

```
// ...
class Hero : public Character { /*
class Orc : public Character { /*
// ...
void attack(Character
void print_result(const Character &a
int main() {
   // ...
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc); // <-- OOPS
      else
         attack(orc, hero); // <-- OOPS
   }
   print_result(orc, hero);
}
```

```
void Hero::hit(Orc &orc) { orc.suffer(strength()); }
void Orc::hit(Hero &hero) { hero.suffer(strength()); }
void attack(Character &from, Character &to) {
   from.hit(to); // <-- OOPS
}
```

This code does not compile due to the way attack() is written: it tries to make a Character hit() a Character, but there's no such function in our design!

What this is about?

- There are, of course, many solutions to this problem

What this is about?

- There are, of course, many solutions to this problem
- One, which many would call self-evident, would be to make Character::hit(Character&) compile
  - We could make it so it always does the same thing, regardless of types (it would work with our simplified case)

What this is about?

```
// ...
class Character {
   // ... suffer() etc.
   void hit(Character &other) {
      other.suffer(strength()); // <-- OOPS
   }
};
class Hero : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
class Orc : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
```

What this is about?

```
// ...
class Character {
   // ... suffer() etc.
   void hit(Character &other) {
      other.suffer(strength()); // <-- OOPS
   }
};
class Hero : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
class Orc : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
```

Well, this would suppose any Character exposes a strength() member function, something we do not currently do

What this is about?

```
// ...
class Character {
   // ... suffer() etc.
   void hit(Character &other) {
      other.suffer(strength()); // <-- OOPS
   }
};
class Hero : public Character {
   // ...
   constexpr int strength() const { return
};
class Orc : public Character {
   // ...
   constexpr int strength() const { return
};
```

Well, this would suppose any Character exposes a strength() member function, something we do not currently do

This is starting to put pressure on the Character class, and it's unclear if we're making beneficial decisions... What if we want to make a Pacifist subclass of Character that does not fight? We could implement a strength() member function that returns zero, but that's more a patch than an actual design solution...

What this is about?

One could argue that exposing Character::hit() is a problem in the first place: are we really sure this makes sense for all types of Character?

```
// ...
class Character {
   // ... suffer() etc.
   void hit(Character &other) {
      other.suffer(strength()); // <-- OOPS
   }
};
class Hero : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
class Orc : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
```

What this is about?

```
// ...
class Character {
   // ... suffer() etc.
   void hit(Character &other) {
      other.suffer(strength()); // <-- OOPS
   }
};
class Hero : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
class Orc : public Character {
   // ...
   constexpr int strength() const { return  strength_; }
};
```

One could argue that exposing Character::hit() is a problem in the first place: are we really sure this makes sense for all types of Character?

We’re « coding blind » here. Writing code before we actually have a plan

We need a plan…

We need a plan…

- By adding features and moving members around somewhat blindly, we’re « fixing » problems but creating others
- Writing code without a plan is a perilous venture, even for such a simple problem

We need a plan…

- What do we want?
  - We want heroes to be able to hit monsters (Orcs in particular)
  - We want monsters to be able to hit heroes (otherwise it’s unfair)
  - To make things fun, let’s suppose heroes can wear armor, which potentially gives them an edge

We need a plan…

- We have **implementation** considerations
  - Let’s make it so all characters will have a name, and « life points »
  - The same rules will apply for all characters regarding the ideas of being dead, or alive
  - There can be variations in the way characters name themselves
  - Open questions: should all characters be able to suffer? To heal? etc.

We need a plan…

- We have **interface** concerns
  - Some characters are bellicose and can hit others
  - The way damage is dealt can vary with type, probably with individual objects too
  - The way damage is received can also vary with type, probably with individual objects (e.g.: when someone wears armor)

A first attempt…

A first attempt...

Character
(name_, life_, alive(),
dead(), etc.)

Some things will be common to all types of characters. We do not presume a proclivity towards violence at this stage

A first attempt…

Character
(name_, life_, alive(),
dead(), etc.)

Damageable
(suffer())

There will be types that can be hit, but are not necessarily inclined to retaliate

A first attempt…

Damageable
(suffer())

Character
(name_, life_, alive(),
dead(), etc.)

Bellicose
(hit())

There are of course types to model
those who are inclined to violence

A first attempt…

Damageable
(suffer())

Character
(name_, life_, alive(), dead(), etc.)

Bellicose
(hit())

Hero
(can wear armor)

Monster
(speaks simply)

A first attempt...

Damageable
(suffer())

Character
(name_, life_, alive(), dead(), etc.)

Bellicose
(hit())

Hero
(can wear armor)

Monster
(speaks simply)

Armor
(protection())

Orc
(smelly and proud of it)

A first attempt…

Damageable
(suffer())

Character
(name_, life_, alive(),
dead(), etc.)

Bellicose
(hit())

Hero
(can wear armor)

Monster
(speaks simply)

Does this look reasonable so far?

Armor
(protection())

Orc
(smelly and proud of it)

A first attempt…

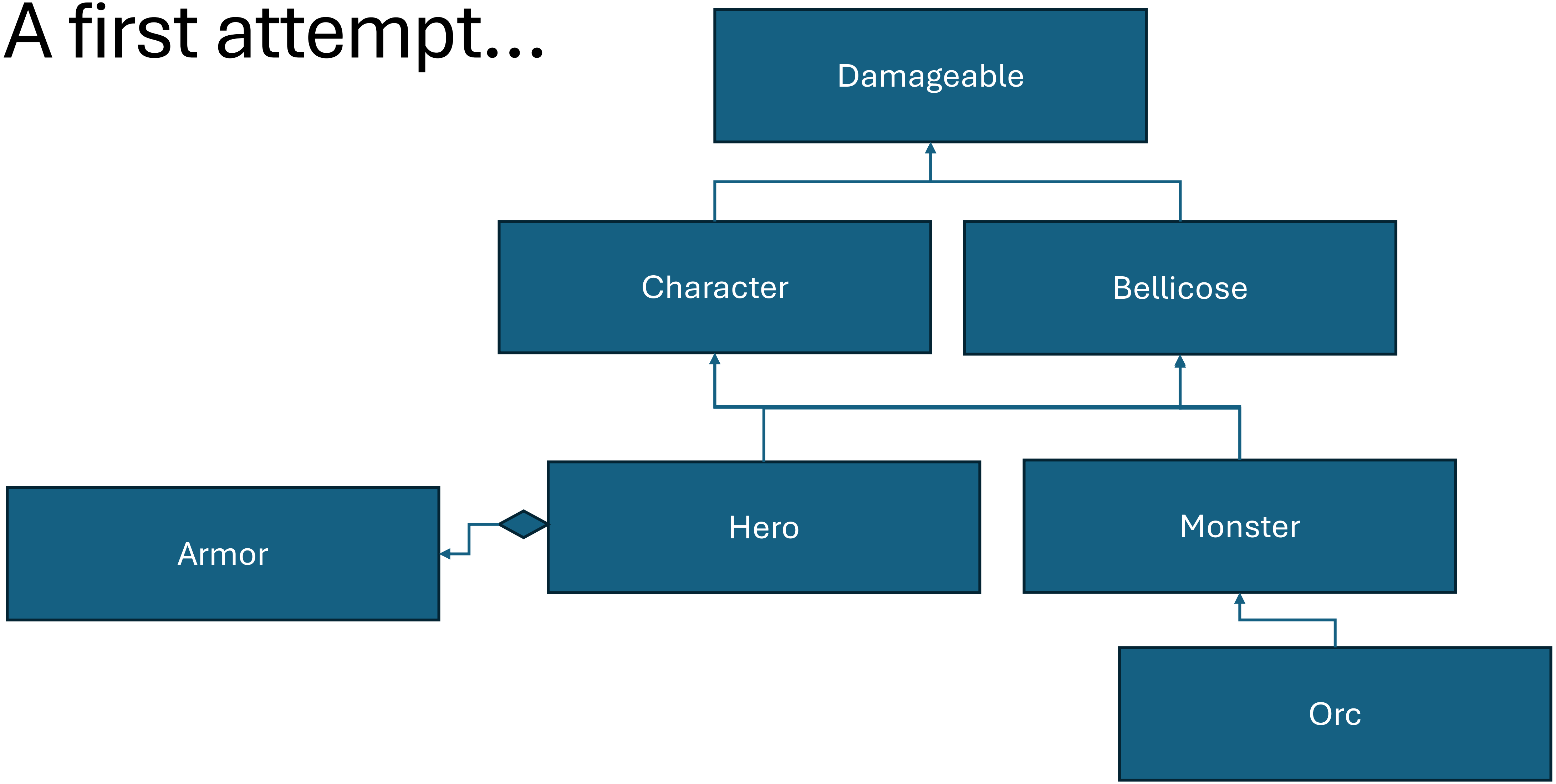
- A classical take on these ideas would be to use public inheritance and runtime polymorphism, with some composition thrown in
  - A Character *is* Damageable
  - A Bellicose entity *is* Damageable (otherwise it's unfair)
  - A Hero *is* a Bellicose Character
  - A Hero *has* an Armor
  - A Monster *is* a Bellicose Character
  - An Orc *is* a Monster

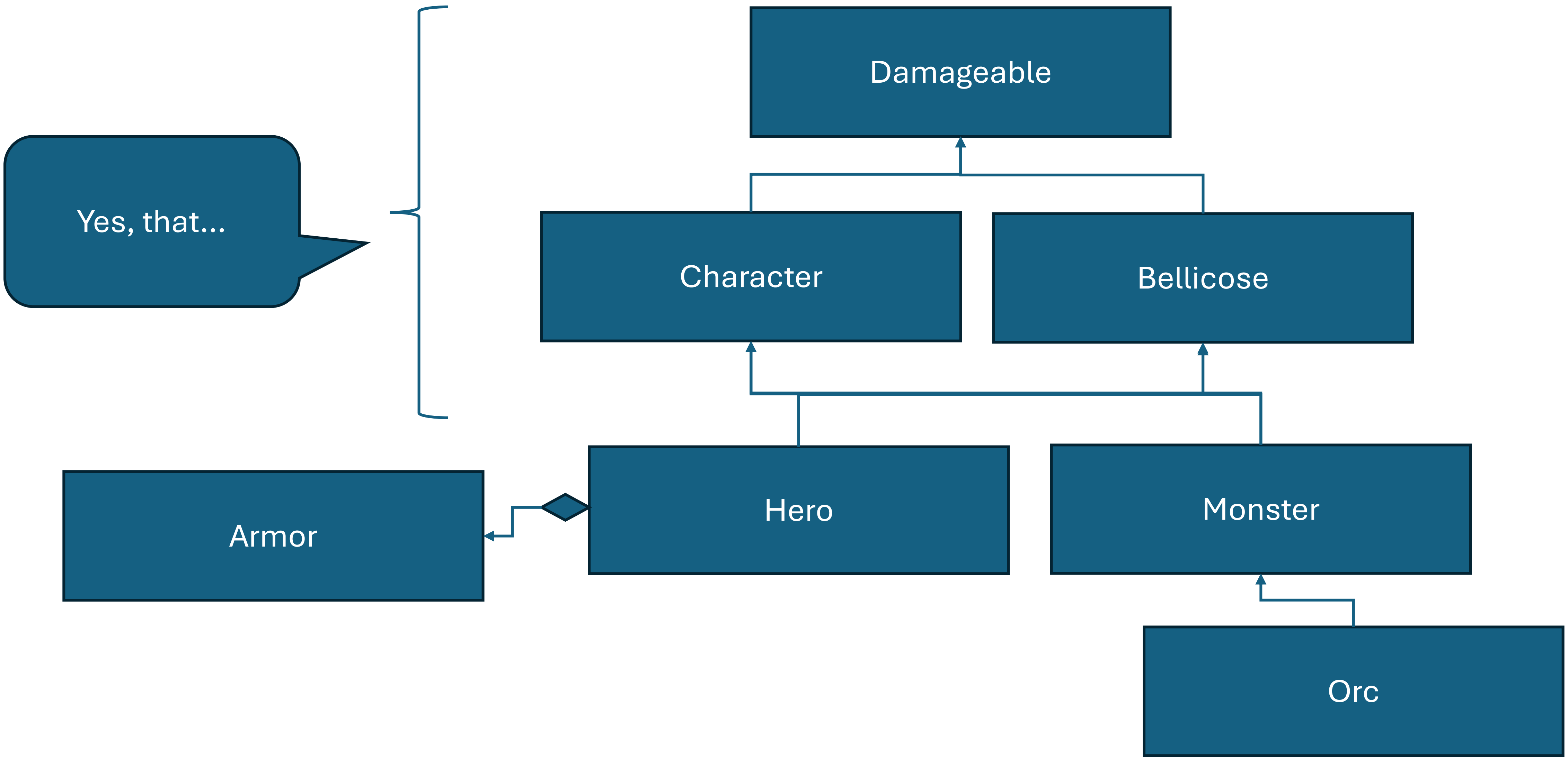
A first attempt…

- A classical take on these ideas would be to use public inheritance and runtime polymorphism, with some composition thrown in
  - A Character *is* Damageable
  - A Bellicose entity *is* Damageable (otherwise it's unfair)
  - A Hero *is* a Bellicose Character
  - A Hero *has* an Armor
  - A Monster *is* a Bellicose Character
  - An Orc *is* a Monster

Do you notice something slightly… unpleasant?

A first attempt...




Yes, that...
Damageable
Character
Bellicose
Hero
Monster
Armor
Orc

A first attempt…

```
import std;
// or #include <random>, <string>, <string_view>,
// <print>, <memory>, <utility>, <format>
struct Damageable {
   virtual void suffer(int) = 0;
   virtual ~Damageable() = default;
};
```

A first attempt…

A Character is Damageable and we plan for a diamond-shaped inheritance

```cpp
// ...
class Character : virtual public Damageable {
   std::string name_;
   int life_;
public:
   Character(std::string_view name, int life)
      : name_{ name }, life_{ life } {
   }
   virtual std::string name() const { return name_; }
   virtual ~Character() = default;
   bool alive() const { return life() > 0; }
   bool dead() const { return !alive(); }
   int life() const { return life_; }
   void suffer(int damage) override { life_ -= damage; }
   void heal(int health) { life_ += health; }
};
```

We removed constexpr from a few of our member functions, as constexpr construction and virtual base classes don't agree much (sadly)

A first attempt…

```
// ...
struct Bellicose : virtual Damageable {
    virtual void hit(Damageable&) = 0;
    virtual ~Bellicose() = default;
};
```

A first attempt…

```
// ...
struct Armor {
   virtual float protection() const { return 0; }
   virtual ~Armor() = default;
};
class Scale : public Armor {
   float protection() const override { return 0.1f; }
};
class Plate : public Armor {
   float protection() const override { return 0.2f; }
};
```

A first attempt...

This one could be abstract; it’s a matter of perspective, really

```
// ...
struct Armor {
   virtual float protection() const { return 0; }
   virtual ~Armor() = default;
};
class Scale : public Armor {
   float protection() const override { return 0.1f; }
};
class Plate : public Armor {
   float protection() const override { return 0.2f; }
};
```

A first attempt…

```
// ...
struct Armor {
   virtual float protection() const { return 0; }
   virtual ~Armor() = default;
};
class Scale : public Armor {
   float protection() const override { return 0.1f; }
};
class Plate : public Armor {
   float protection() const override { return 0.2f; }
};
```

These two are only examples, obviously

A first attempt…

```
// ...
class Hero : public Character, public Bellicose {
   int strength_;
   std::unique_ptr<Armor> armor;
   void equip(std::unique_ptr<Armor> p) { armor = std::move(p); }
   void hit(Damageable &other) override { other.suffer(strength()); }
public:
   Hero(std::string_view name, int life, int strength) : Character { name, life }, strength_{ strength } {
   }
   Hero(std::string_view name, int life, int strength, std::unique_ptr<Armor> armor) : Hero{ name, life, strength } {
      equip(std::move(armor));
   }
   int strength() const { return  strength_; }
   void suffer(int damage) override {
      Character::suffer(armor? static_cast<int>(damage * armor->protection()): damage);
   }
};
```

Alternatively, we could avoid null pointers and use a « null object » for the Armor type and remove the test altogether

A first attempt...

```
// ...
class Monster : public Character, public Bellicose {
   int strength_;
   void hit(Damageable &other) override {
      other.suffer(strength());
   }
public:
   Monster(std::string_view name, int life, int strength)
      : Character { name, life }, strength_{ strength } {
   }
   int strength() const { return  strength_; }
   std::string name() const override {
      return std::format("Me is {}", Character::name());
   }
};
```

Monsters have a... limited vocabulary

A first attempt…

```
// ...
class invalid_smell {};
class Smell {
   double val;
   static constexpr bool is_valid(double val) {
      return 0 <= val && val <= 1;
   }
   static constexpr double validate(double val) {
      return is_valid(val)? val : throw invalid_smell{};
   }
public:
   constexpr explicit Smell(double val) : val{ validate(val) } {
   }
   constexpr double value() const { return val; }
};
```

Smell is important to Orcs, so we take care to model that idea through a class

A first attempt…

```
// ...
class Orc : public Monster {
   Smell smell;
   static constexpr std::string_view qualify_smell(Smell smell) {
      using namespace std::literals;
      return smell.value() < 0.5? "not really bad"sv :
             smell.value() < 0.8? "bad"sv : "terrible"sv;
   }
public:
   Orc(std::string_view name, int life, int strength, Smell smell)
      : Monster{ name, life, strength }, smell{ smell } {
   }
   std::string name() const override {
       return std::format("{} me smell {}", Character::name(), qualify_smell(smell));
   }
};
```

A first attempt…

A Bellicose attacks a Damageable…

```
// ...
void attack(Bellicose &from, Damageable &to) { from.hit(to); }
void print_result(const Character &a, const Character &b) { std::print("{} won", a.alive() ? a.name() : b.name()); }
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   Orc orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } };
   Hero hero{ "William", 100, dice(prng) / 10 };
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc);
      else
         attack(orc, hero);
   }
   print_result(orc, hero);
}
```

A first attempt...

```
// ...
void attack(Bellicose &from, Damageable &to) { from.hit(to); }
void print_result(const Character &a, const Character &b) { std::print("{} won", a.alive() ? a.name() : b.name()); }
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   Orc orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } };
   Hero hero{ "William", 100, dice(prng) / 10 };
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc);
      else
         attack(orc, hero);
   }
   print_result(orc, hero);
}
```

https://wandbox.org/permlink/YnoHx7VSk3tJzgxq

A first attempt…

- This solution uses well-known but intrusive techniques such as public inheritance
  - It’s not necessarily bad, but it introduces coupling in our code
  - We’ve learned over time that when alternatives exist that involve less coupling, they tend to help us perform code maintenance

A first attempt…

- In particular, a class hierarchy with virtual base classes is valid C++, but is often frowned upon
  - Makes objects bigger
  - Not constexpr-friendly
  - Can complicate the construction of the shared base class (not everyone knows the rules)

A first attempt…

- In particular, a class hierarchy with virtual base classes is valid C++, but is often frowned upon
  - Makes objects bigger
  - Not constexpr-friendly
  - Can complicate the construction of the shared base class (not everyone knows the rules)
- That does not mean it does not work in practice
  - We just made it work, indeed

A first attempt…

- There are even upsides to this approach
  - Teachable (many are familiar with inheritance and virtual functions)
  - Structural commonality, which lets us write such things as:

```
// group various Bellicose entities in a single container
vector<unique_ptr<Bellicose>> attackers;
attackers.emplace_back(make_unique<Hero>(
   "William", 100, dice(prng) / 10
));,
attackers.emplace_back(make_unique<Orc>(
   "URG", 100, dice(prng) / 10, Smell{ 7.0 }
));
// can then write for(auto && p : attackers) p->hit(...);
```

A first attempt…

Complete example:
https://wandbox.org/permlink/9PU2GOxJkI89AgKH

- There are even upsides to this approach
  - Teachable (many are familiar with inheritance and virtual functions)
  - Structural commonality, which lets us write such things as:

```
// group various Bellicose entities in a single container
vector<unique_ptr<Bellicose>> attackers;
attackers.emplace_back(make_unique<Hero>(
   "William", 100, dice(prng) / 10
));,
attackers.emplace_back(make_unique<Orc>(
   "URG", 100, dice(prng) / 10, Smell{ 7.0 }
));
// can then write for(auto && p : attackers) p->hit(...);
```

Something less intrusive…

Something less intrusive…

- Our first attempt defined the Bellicose and Damageable protocols as abstractions
  - Abstract base classes, expressing what languages that do not fully support multiple inheritance would name *interfaces*
- As mentioned previously, that allows us to write code that groups either Bellicose pointers or Damageable pointers together
- … but what if we don’t need this?

Something less intrusive…

```
import std;
// or #include <random>, <string>, <string_view>,
// <print>, <memory>, <utility>, <format>
class Character { // note: no base class
   std::string name_;
   int life_;
public:
   constexpr Character(std::string_view name, int life) : name_{ name }, life_{ life } { }
   virtual constexpr std::string name() const { return name_; }
   virtual constexpr ~Character() = default;
   constexpr bool alive() const { return life() > 0; }
   constexpr bool dead() const { return !alive(); }
   constexpr int life() const { return life_; }
   virtual constexpr void suffer(int damage) { life_ -= damage; }
   constexpr void heal(int health) { life_ += health; }
};
```

Something less intrusive…

```
// ...
struct Armor {
   virtual float protection() const { return 0; }
   virtual ~Armor() = default;
};
class Scale : public Armor {
   float protection() const override { return 0.1f; }
};
class Plate : public Armor {
   float protection() const override { return 0.2f; }
};
```

Something less intrusive...

```
// ...
class Hero : public Character {
   int strength_;
   std::unique_ptr<Armor> armor;
   void equip(std::unique_ptr<Armor> p) { armor = std::move(p); }
public:
   template <class Damageable> // note: public
      void hit(Damageable &other) {
         other.suffer(strength());
      }
   Hero(std::string_view name, int life, int strength) : Character { name, life }, strength_{ strength } {  }
   Hero(std::string_view name, int life, int strength, std::unique_ptr<Armor> armor) : Hero{ name, life, strength } {
      equip(std::move(armor));
   }
   int strength() const { return  strength_; }
   void suffer(int damage) { Character::suffer(armor? static_cast<int>(damage * armor->protection()): damage); }
};
```

The non-intrusive approach here is clear from source code: an object is « Damageable » if it can suffer() an int value (we made *implicit* that an object is « Bellicose » if it can hit() something Damageable)

Something less intrusive…

```
// ...
class Monster : public Character {
   int strength_;
public:
   template <class Damageable> // note: public
      void hit(Damageable &other) {
         other.suffer(strength());
      }
   Monster(std::string_view name, int life, int strength)
      : Character { name, life }, strength_{ strength } {
   }
   int strength() const { return  strength_; }
   std::string name() const override {
       return std::format("Me is {}", Character::name());
   }
};
```

Something less intrusive…

```
// ... class Smell (no change there) ...
class Orc : public Monster {
   Smell smell;
   static constexpr std::string_view qualify_smell(Smell smell) {
      using namespace std::literals;
      return smell.value() < 0.5? "not really bad"sv :
             smell.value() < 0.8? "bad"sv : "terrible"sv;
   }
public:
   Orc(std::string_view name, int life, int strength, Smell smell)
      : Monster{ name, life, strength }, smell{ smell } {
   }
   std::string name() const override {
       return std::format("{} me smell {}", Character::name(), qualify_smell(smell));
   }
};
```

Something less intrusive...

```
// ...
template <class Bellicose, class Damageable>
   void attack(Bellicose &from, Damageable &to) { from.hit(to); }
void print_result(const Character &a, const Character &b) { std::print("{} won", a.alive() ? a.name() : b.name()); }
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   Orc orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } };
   Hero hero{ "William", 100, dice(prng) / 10 };
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc);
      else
         attack(orc, hero);
   }
   print_result(orc, hero);
}
```

We can now express our abstractions in attack() through generic code rather than through indirections to a base class

Something less intrusive...

```
// ...
template <class Bellicose, class Damageable>
   void attack(Bellicose &from, Damageable &to) { from.hit(to); }
void print_result(const Character &a, const Character &b) { std::print("{} won", a.alive() ? a.name() : b.name()); }
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   Orc orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } };
   Hero hero{ "William", 100, dice(prng) / 10 };
   while(orc.alive() && hero.alive()) {
      if(dice(prng) % 2 == 0)
         attack(hero, orc);
      else
         attack(orc, hero);
   }
   print_result(orc, hero);
}
```

https://wandbox.org/permlink/FbUrShQABaqfbLdd

Something less intrusive…

- What do you think about this version?

Something less intrusive…

- There are upsides:
  - Function calls can be direct rather than indirect
  - Objects are smaller
  - More opportunities for constexpr
    - We did not use it, but it’s there!

Something less intrusive…

- There are downsides too:
  - Forget about making a vector<Damageable*> for example as there's no Damageable type serving as base class
  - Passing inadequate types to attack() might incur obscure error messages
    - This is of course QoI
  - One could rightfully claim that our interface is insufficiently clear
    - Takes any type, even if the type names try to be expressive
    - Good function interfaces are essential to good code hygiene

Clarifying intent

Clarifying intent

- What if we made the semantics of our design explicit from the source code?

Clarifying intent

```
template <class T>
   concept Damageable = requires(T &a) {
      a.suffer(std::declval<int>());
   };
void bellicose_test(auto && d) {
   struct X { void suffer(int) {}; } x; d.hit(x);
}
template <class T> concept Bellicose = requires(T &a) {
   bellicose_test(a);
};
void attack(Bellicose auto &from, Damageable auto &to) {
   from.hit(to);
}
```

Clarifying i

```
template <class T
   concept Damage
      a.suffer(st
   };
void bellicose_te
   struct X { voi
}
template <class T>
   bellicose_test(a);
};
void attack(Bellicose auto &from, Damageable auto &to) {
   from.hit(to);
}
```

The intuitive syntax does not work in C++20:

```
template <class T>
   concept Damageable = requires(T &a) {
      a.suffer(std::declval<int>());
   };

template <class T>
   concept Bellicose = requires(T &a, Damageable auto &b) {
      a.hit(b);
   };
```

Clarifying i

```
template <class T
   concept Damage
      a.suffer(st
   };
void bellicose_te
   struct X { voi
}
template <class T>
   bellicose_test(a);
};
void attack(Bellicose auto &from, Damageable au
   from.hit(to);
}
```

The intuitive syntax does not work in C++20:

```
template <class T>
   concept Damageable = requires(T &a) {
      a.suffer(std::declval<int>());
   };

template <class T>
   concept Bellicose = requires(T &a, Damageable auto &b) {
      a.hit(b);
   };
```

«Error: placeholder type not allowed in that context» or «auto not allowed in requires expression parameter»

Clarifying i

```
template <class T>
   concept Damagea
      a.suffer(st
   };
void bellicose_te
   struct X { voi
}
template <class T>
   bellicose_test(a);
};
void attack(Bellicose auto &from, Damageable au
```

The intuitive syntax does not work in C++20:

```
template <class T>
   concept Damageable = requires(T &a) {
      a.suffer(std::declval<int>());
   };

template <class T>
   concept Bellicose = requires(T &a, Damageable auto &b) {
      a.hit(b);
   };
```

«Error: placeholder type not allowed in that context» or «auto not allowed in requires expression parameter»

I enquired about this to the Core Working Group and it seems that the intuitive syntax would have worked with C++0x concepts, but not with the less ambitious concepts that were adopted for C++20

Clarifying intent

Luckily, there are specific workarounds like bellicose_test() here. This workaround expresses what satisfying the Bellicose concept requires (!), and uses that in the implementation of the requires clause

```
template <class T>
   concept Damageable = requires(T &a) {
      a.suffer(std::declval<int>());
   };
void bellicose_test(auto && d) {
   struct X { void suffer(int) {}; } x; d.hit(x);
}
template <class T> concept Bellicose = requires(T &a) {
   bellicose_test(a);
};
void attack(Bellicose auto &from, Damageable auto &to) {
   from.hit(to);
}
```

Clarifying intent

```
template <class T>
   concept Damageable = requires(T &a) {
      a.suffer(std::declval<int>());
   };
void bellicose_test(auto && d) {
   struct X { void suffer(int) {}; } x; d.hit(x);
}
template <class T> concept Bellicose = requires(T &a) {
   bellicose_test(a);
};
void attack(Bellicose auto &from, Damageable auto &to) {
   from.hit(to);
}
```

https://wandbox.org/permlink/MXrKcTcmGzMkyrdf
for a full example

Clarifying intent

- Providing clearer semantics regarding our intents has mostly upsides
  - Clearer code, easier to understand
  - Better error messages
  - Possibility of direct function calls

Clarifying intent

- We still do not have a common base class
  - Concepts are non-intrusive
  - We cannot write a vector<Damageable auto*>

Grouping for an assault

Grouping for an assault

- What if we want to group attackers (or victims) in a container?
- The classical « pointer to base class » approach will not work without such a base class

Grouping for an assault

- … but we have this wonderful type named std::variant!

Grouping for an assault

```
// other Damageable types
struct Furniture {
   void suffer(int) { /* breaks easily */ }
};
struct Bystander : Character {
   using Character::Character;
};
template <Damageable ... Ts>
   void attack(Bellicose auto &from, std::variant<Ts...> &to) {
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

Grouping for an assault

```
// other Damageable types
struct Furniture {
   void suffer(int) { /* breaks easily */ }
};
struct Bystander : Character {
   using Character::Character;
};
template <Damageable ... Ts>
   void attack(Bellicose auto &from, std::variant<Ts...> &to) {
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

Through a variant of Damageable types, a Bellicose can directly call the appropriate hit() member function for the contained object

Grouping for an assault

```
// other Damageable types
struct Furniture {
   void suffer(int) { /* breaks easily */ }
};
struct Bystander : Character {
   using Character::Character;
};
template <Damageable ... Ts>
   void attack(Bellicose auto &from, std::variant<Ts      o) {
      std::visit([&from](Damageable auto && to) { f  om.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

> Making the actual vector of Damageable objects is relatively easy with a factory function that infers the exact types for the variant from the list of arguments. This is not a case for which I found deduction guides to work well, sadly, but maybe that's just me

Grouping for an assault

```
// other Damageable types
struct Furniture {
   void suffer(int) { /* breaks easily */ }
};
struct Bystander : Character {
   using Character::Character;
};
template <Damageable ... Ts>
   void attack(Bellicose auto &from, std::variant<Ts...>    ) {
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

Note that for variant<Ts...> to work here, it's important that each type in Ts... occurs only once. We could fix that in the code specification, it's « easy »...

Gro

```cpp
#include <type_traits>
template <class ...>
   struct type_list;
template <class TL, class T> struct contains;
template <class T, class ... Q, class U>
   struct contains <type_list<T, Q...>, U> : contains<type_list<Q...>, U> { };
template <class T, class ... Q>
   struct contains <type_list<T, Q...>, T> : std::true_type { };
template <class T>
   struct contains <type_list<>, T> : std::false_type { };
// ...
```

```cpp
// other
struct F
   void
};
struct Bystander : Char
   using Character::C
};
template <Damageable ... Ts>
   void attack(Bellicose auto &from, std::variant<Ts...> &to) {
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

Gro

```
// ...
template <class T, class TL> struct merge;
template <class T, class ...Ts>
   struct merge<T, type_list<Ts...>> { using type = type_list<T, Ts...>; };
template <class T> struct merge<T, type_list<>> { using type = type_list<T>; };
template <class T, class TL> using merge_t = typename merge<T, TL>::type;
template <class> struct remove_duplicates;
template <class T> using remove_duplicates_t = typename remove_duplicates<T>::type;
template <class T, class ... Q>
   struct remove_duplicates<type_list<T, Q...>> {
      using type = std::conditional_t<
         contains<type_list<Q...>, T>{}(),
         remove_duplicates_t<type_list<Q...>>,
         merge_t<T, remove_duplicates_t<type_list<Q...>>>
      >;
   };
// ...
```

```
// other
struct F
   void
};
struct B
   using
};
template <
   void attack(Bellicos                    riant<Ts...> &to) {
      std::visit([&fr             auto && to) { from.hit(to); }, to);
   }
template <Damageab    ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

Gro

```
// ...
template <>
   struct remove_duplicates<type_list<>> {
      using type = type_list<>;
};
// precondition: TL has no duplicatestemplate <class TL>   struct tl_to_variant;template <class ... Ts>   struct tl_to_variant<type_list<Ts...>> {
  using type = std::variant<Ts...>;
};
template <class ... Ts>
   using expunged_variant =
      typename tl_to_variant<remove_duplicates_t<type_list<Ts...>>>>::type;
```

```
// other
struct F
   void
};
struct By
   using Ch
};
template <Damageable
   void attack(Bel  ose auto &from, std::variant<Ts...> &to) {
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

Gro

```
// other
struct F
   void
};
struct By
   using Ch
};
template <Damageable
   void attack(Bel   ose auto &from, std::va
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<typename expunged_variant<Ts...>::type> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

```
// ...
template <>
   struct remove_duplicates<type_list<>> {
      using type = type_list<>;
};
// precondition: TL has no duplicatestemplate <class TL>   struct tl_to_variant;template <class ... Ts>   struct
tl_to_variant<type_list<Ts...>> {
  using type = std::variant<Ts...>;
};
template <class ... Ts>
   using expunged_variant =
      typename tl_to_variant<remove_duplicates_t<type_list<Ts...>>>::type;
```

https://wandbox.org/permlink/24PHTnyOHYSclBV0

Grouping for an assault

```
// other Damageable types
struct Furniture {
   void suffer(int) { /* breaks easily */ }
};
struct Bystander : Character {
   using Character::Character;
};
template <Damageable ... Ts>
   void attack(Bellicose auto &from, std::variant<Ts...> &to) {
      std::visit([&from](Damageable auto && to) { from.hit(to); }, to);
   }
template <Damageable ... Ts>
   std::vector<std::variant<Ts...>> make_victims(Ts &&... args) {
      return { std::forward<Ts>(args)... };
   }
```

…but let's keep things simple

Grouping for an assault

```
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   auto victims = make_victims(
      Orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } },
      Furniture{},
      Bystander{ "Fred", 20 }
   );
   Hero hero{ "William", 100, dice(prng) / 10 };
   // William swings his halberd!
   for(auto && p : victims)
       attack(hero, p);
}
```

Grouping for an assault

Through a vector of variant for our Damageable objects, we can make grouped attacks without resorting to the classical, indirect approach

```
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   auto victims = make_victims(
      Orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } },
      Furniture{},
      Bystander{ "Fred", 20 }
   );
   Hero hero{ "William", 100, dice(prng) / 10 };
   // William swings his halberd!
   for(auto && p : victims)
       attack(hero, p);
}
```

Grouping for an assault

```
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   auto victims = make_victims(
      Orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } },
      Furniture{},
      Bystander{ "Fred", 20 }
   );
   Hero hero{ "William", 100, dice(prng) / 10 };
   // William swings his halberd!
   for(auto && p : victims)
       attack(hero, p);
}
```

https://wandbox.org/permlink/Hclqv6DgPq7u1A6N

We want numbers…

We want numbers…

- There are many motivations behind design choices
  - Readability
  - Teachability
  - Cohesiveness with the rest of the codebase
  - etc.
- We’ll suppose that our concerns are size and execution speed

We want numbers…

- The sizeof and alignof operators will help us compare memory consumption
- For speed, we will use a simple yet useful little function

We want numbers…

```
// #include <vector>, <chrono>, <utility>
template <class F, class ... Args>
   auto test(F f, Args &&... args) {
      using namespace std;
      using namespace std::chrono;
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return pair{ res, post - pre };
   }
// ...
```

This is what I will use to measure execution time of some (non-void) function f() with arguments args…

We want numbers…

- Classic, pointer to base approach with virtual base classes from « A first attempt… »

We want numbers…

```
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   std::print("Classical approach (pointer to base class, virtual bases)\n");
   std::print("\tCharacter : sizeof == {}, alignof == {}\n",
              sizeof(Character), alignof(Character));
   std::print("\tHero      : sizeof == {}, alignof == {}\n",
              sizeof(Hero), alignof(Hero));
   std::print("\tMonster   : sizeof == {}, alignof == {}\n",
              sizeof(Monster), alignof(Monster));
   std::print("\tOrc       : sizeof == {}, alignof == {}\n",
              sizeof(Orc), alignof(Orc));
   // ...
```

We want nu

```
Classical approach (pointer to base class, virtual bases)
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64, alignof == 8
    Orc       : sizeof == 72, alignof == 8
```

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution         1, 100 };
   std::print("Classical approach (pointer to base class, virtual bases)\n");
   std::print("\tCharacter : sizeof == {}, alignof == {}\n",
              sizeof(Character), alignof(Character));
   std::print("\tHero      : sizeof == {}, alignof == {}\n",
              sizeof(Hero), alignof(Hero));
   std::print("\tMonster   : sizeof == {}, alignof == {}\n",
              sizeof(Monster), alignof(Monster));
   std::print("\tOrc       : sizeof == {}, alignof == {}\n",
              sizeof(Orc), alignof(Orc));
   // ...
```

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution     { 1, 100
   std::print("Classical approach
   std::print("\tCharacter : siz
            sizeof(Character),
   std::print("\tHero      : siz
            sizeof(Hero), alig
   std::print("\tMonster   : siz
            sizeof(Monster), a
   std::print("\tOrc       : siz
            sizeof(Orc), align
   // ...
```

```
Classical approach (pointer to base class, virtual bases)
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64  alignof == 8
    Orc       : sizeof ==     alignof == 8
```

```
struct Damageable {
   virtual void suffer(int) = 0;
   virtual ~Damageable() = default;
};
class Character : virtual public Damageable {
   std::string name_;
   int life_;
   // ...
};
```

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution     { 1, 100
   std::print("Classical approach
   std::print("\tCharacter : siz
            sizeof(Character),
```

Classical approach (pointer to base class, virtual bases)

```
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64  alignof == 8
    Orc       : sizeof ==     alignof == 8
```

```
struct Damageable {
   virtual void suffer(int) = 0;
   virtual ~Damageable() = default;
};
class Character : virtual public Damageable {
   std::string name_;
   int life_;
   // ...
};
```

On this compiler, sizeof(string) is 32 and alignof(string) is 8. We are paying for padding and a pointer to the vtbl for the virtual member functions and the virtual base class

We want nu

```
Classical approach (pointer to base class, virtual bases)
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64, alignof == 8
    Orc       : sizeof == 72, alignof == 8
```

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution      1, 100 };
   std::print("Classical approach (pointer to base class, virtual bases)\n");
   std::print("\tCharacter : sizeof == {}, alignof == {}\n",
              sizeof(Character), alignof(Character));
   std::print("\tHero      : sizeof == {}, alignof == {}\n",
              sizeof(Hero), alignof(Hero));
   std::print("\tMonster   : sizeof == {}, alignof == {}\n",
              sizeof(Monster), alignof(Monster));
   std::print("\tOrc       : sizeof == {}, alignof == {}\n",
              sizeof(Orc), alignof(Orc));
   // ...
```

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution       { 1, 100 };
   std::print("Classical approach (pointer to              , virtual bases)\n");
   std::print("\tCharacter : sizeof == {}                       \n",
              sizeof(Charact
   std::print("\tHero      :
              sizeof(Hero),
   std::print("\tMonster   :
              sizeof(Monster
   std::print("\tOrc       :
              sizeof(Orc), alignof(Orc));
   // ...
```

```
Classical approach (pointer to base class, virtual bases)
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64, alignof == 8
    Orc       : sizeof == 72, alignof == 8
```

```
class Hero : public Character, public Bellicose {
   int strength_;
   std::unique_ptr<Armor> armor;
   // ...
};
```

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution    e{ 1, 100 };
   std::print("Classical approach (pointer to              , virtual bases)\n");
   std::print("\tCharacter : sizeof == {}                   \n"
              sizeof(Charact
   std::print("\tHero      :
              sizeof(Hero),
   std::print("\tMonster   :
              sizeof(Monster
```

```
Classical approach (pointer to base class, virtual bases)
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64, alignof == 8
    Orc       : sizeof == 72, alignof == 8
```

```
class Hero : public Character, public Bellicose {
   int strength_;
   std::unique_ptr<Armor> armor;
   // ...
};
```

sizeof(Character) is 48. Adding sizeof(int) and sizeof(void*) adds 16 including padding, leading to 64. We get 72 due to the virtual base class overhead (the cost of this overhead depends on the implementation)

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::random_d
   std::uniform_int_distribution        { 1, 100 };
   std::print("Classical approach (pointer to              , virtual bases)\n");
   std::print("\tCharacter : sizeof == {}                 \n"
              sizeof(Charact
   std::print("\tHero      :
              sizeof(Hero),
   std::print("\tMonster   :
              sizeof(Monster
   std::print("\tOrc       :
              sizeof(Orc), alignof(Orc));
   // ...
```

```
Classical approach (pointer to base class, virtual bases)
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 72, alignof == 8
    Monster   : sizeof == 64, alignof == 8
    Orc       : sizeof == 72, alignof == 8
```

```
class Monster : public Character, public Bellicose {
   int strength_;
   // ...
};
```

sizeof(Monster) is slightly less than sizeof(Hero) since a Monster does not have an armor

We want numbers…

```
// ...
int main() {
   // ...
   constexpr int N = 1'000'000, M = 1000;
   Hero hero{ "William", 100, dice(prng) / 10 };
   std::vector<std::unique_ptr<Damageable>> victims;
   for(int i = 0; i != N; ++i)
      victims.push_back(
         std::make_unique<Orc>("URG", 100, dice(prng) / 10, Smell{ 0.7 })
      );
   // ...
}
```

We want numbers…

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, *p);
      return victims.size(); // whatever
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

We want numbers…

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, *p);
      return victims.size(); // whatever
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

We want numbers…

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, *p);
      return victims.size(); // wh
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

Attacking 1000000 victims 1000 times in **5222730us**

We want numbers...

https://wandbox.org/permlink/wccvfLba33CTBp6K

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, *p);
      return victims.size(); // whatever
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

We want numbers…

- Classic, pointer to base approach with virtual base classes from « A first attempt… »
- Variant-based approach from « Grouping for an assault »

We want numbers…

```
// ...
int main() {
   std::mt19937 prng { std::random_device{}() };
   std::uniform_int_distribution dice{ 1, 100 };
   std::print("Variant approach\n");
   std::print("\tCharacter : sizeof == {}, alignof == {}\n",
              sizeof(Character), alignof(Character));
   std::print("\tHero      : sizeof == {}, alignof == {}\n",
              sizeof(Hero), alignof(Hero));
   std::print("\tMonster   : sizeof == {}, alignof == {}\n",
              sizeof(Monster), alignof(Monster));
   std::print("\tOrc       : sizeof == {}, alignof == {}\n",
              sizeof(Orc), alignof(Orc));
   // ...
```

We want nu

```
Variant approach
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 56, alignof == 8
    Monster   : sizeof == 48, alignof == 8
    Orc       : sizeof == 56, alignof == 8
```

```
// ...
int main() {
  std::mt19937 prng { std::rand };
  std::uniform_int_distributi ice{ 1, 100 };
  std::print("Variant approach\n");
  std::print("\tCharacter : sizeof == {}, alignof == {}\n",
             sizeof(Character), alignof(Character));
  std::print("\tHero      : sizeof == {}, alignof == {}\n",
             sizeof(Hero), alignof(Hero));
  std::print("\tMonster   : sizeof == {}, alignof == {}\n",
             sizeof(Monster), alignof(Monster));
  std::print("\tOrc       : sizeof == {}, alignof == {}\n",
             sizeof(Orc), alignof(Orc));
  // ...
```

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::rand
   std::uniform_int_distributi    ice{ 1, 100
   std::print("Variant approach\n
   std::print("\tCharacter : siz
              sizeof(Character),
   std::print("\tHero      : siz
              sizeof(Hero), alig
   std::print("\tMonster   : siz
              sizeof(Monster), a
   std::print("\tOrc       : siz
              sizeof(Orc), align
   // ...
```

```
Variant approach
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 56, alignof == 8
    Monster   : sizeof == 48   lignof == 8
    Orc       : sizeof ==      alignof == 8
```

```
class Character {
   std::string name_;
   int life_;
   // ...
   virtual constexpr std::string name() const;
   virtual constexpr ~Character() = default;
   // ...
};
```

We want nu

```
Variant approach
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 56, alignof == 8
    Monster   : sizeof == 48   lignof == 8
    Orc       : sizeof ==      alignof == 8
```

```
// ...
int main() {
   std::mt19937 prng { std::rand
   std::uniform_int_distributi      ice{ 1, 100
   std::print("Variant approach\n"
   std::print("\tCharacter : siz
         sizeof(Character),
```

```
class Character {
   std::string name_;
   int life_;
   // ...
   virtual constexpr std::string name() const;
   virtual constexpr ~Character() = default;
   // ...
};
```

On this compiler, sizeof(string) is 32 and alignof(string) is 8. We are paying for padding and a pointer to the vtbl for the virtual member functions

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::rand
   std::uniform_int_distributi        ice{ 1, 100
   std::print("Variant approach\n");
   std::print("\tCharacter : sizeof                 f == {}\n"
              sizeof(Charact
   std::print("\tHero      :
              sizeof(Hero),
   std::print("\tMonster   :
              sizeof(Monster
   std::print("\tOrc       :
              sizeof(Orc), alignof(Orc));
   // ...
```

```
Variant approach
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 56, alignof == 8
    Monster   : sizeof == 48, alignof == 8
    Orc       : sizeof == 56, alignof == 8
```

```
class Hero : public Character {
   int strength_;
   std::unique_ptr<Armor> armor;
   // ...
};
```

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::rand
   std::uniform_int_distributi    ice{ 1, 100
   std::print("Variant approach\n");
   std::print("\tCharacter : sizeof =           f == {}\n"
              sizeof(Charact
   std::print("\tHero      :
              sizeof(Hero),
   std::print("\tMonster   :
              sizeof(Monster
```

```
Variant approach
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 56, alignof == 8
    Monster   : sizeof == 48, alignof == 8
    Orc       : sizeof == 56, alignof == 8
```

```
class Hero : public Character {
   int strength_;
   std::unique_ptr<Armor> armor;
   // ...
};
```

sizeof(Character) is 48. Adding sizeof(int) and sizeof(void*) adds 16 including padding, leading to 64, but this compiler is clever and removes the padding between the trailing int of Character and the leading int of Hero which keeps both types aligned to 8 but saves us 8 bytes of space per object. Cool!

We want nu

```
// ...
int main() {
   std::mt19937 prng { std::rand
   std::uniform_int_distributi    dice{ 1, 100
   std::print("Variant approach\n");
   std::print("\tCharac
              sizeof(C
   std::print("\tHero
              sizeof(H
   std::print("\tMonst
              sizeof(M
```

```
Variant approach
    Character : sizeof == 48, alignof == 8
    Hero      : sizeof == 56, alignof == 8
    Monster   : sizeof == 48, alignof == 8
    Orc       : sizeof == 56, alignof == 8
```

It only seems to do this when virtual member functions are involved. Compare:

Without: https://wandbox.org/permlink/E3Ugj2bEwFoqT8VG
With: https://wandbox.org/permlink/Ods0GoSYTdJw19ur

sizeof(Character) is 48. Adding sizeo ...) adds 16 including padding, leading to 64, but this ... piler is clever and removes the padding between the trailing int of Character and the leading int of Hero which keeps both types aligned to 8 but saves us 8 bytes of space per object. Cool!

We want numbers...

```
// ...
int main() {
   // ...
   constexpr int N = 1'000'000, M = 1000;
   Hero hero{ "William", 100, dice(prng) / 10 };
   std::vector<std::variant<Furniture, Orc, Bystander>> victims;
   for(int i = 0; i != N; ++i)
      victims.push_back(
         Orc{ "URG", 100, dice(prng) / 10, Smell{ 0.7 } }
      );
   // ...
}
```

We want numbers…

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, p);
      return victims.size(); // whatever
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

We want numbers…

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, p);
      return victims.size(); // whatever
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

We want numbers...

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, p);
      return victims.size(); // wh
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

`Attacking 1000000 victims 1000 times in 3649402us`

We want numbers...

https://wandbox.org/permlink/CYk9rGnVwQe61KHh

```
// ...
int main() {
   // ...
   auto [r0, dt0] = test([&victims, &hero] {
      for(int i = 0; i != M; ++i)
         for(auto && p : victims)
            attack(hero, p);
      return victims.size(); // whatever
   });
   using namespace std::chrono;
   std::print("Attacking {} victims {} times in {}",
              std::size(victims), M, duration_cast<microseconds>(dt0));
}
```

We want numbers…

« Traditional » approach

- `sizeof(Character): 48`
- `sizeof(Hero):      72`
- `sizeof(Monster):   64`
- `sizeof(Orc):       72`
- `10^9 attacks: 5222730us`

Variant-based approach

- `sizeof(Character): 48`
- `sizeof(Hero):      56`
- `sizeof(Monster):   48`
- `sizeof(Orc):       56`
- `10^9 attacks: 3649402us`

We want numbers…

« Traditional » approach

- `sizeof(Character): 48`
- **`sizeof(Hero): 72`**
- `sizeof(Monster): 64`
- **`sizeof(Orc): 72`**
- `10^9 atta` … `5222730us`

Variant-based approach

- `sizeof(Character): 48`
- **`sizeof(Hero): 56`**
- `sizeof(Monster): 48`
- **`sizeof(Orc): 56`**
- `10^9 attacks: 3649402us`

With the variant-based approach, each object of type Hero or of type Orc occupies 77.7% of the memory space occupied by its equivalent counterpart in the « traditional » approach

We want numbers…

« Traditional » approach

- `sizeof(Character): 48`
- `sizeof(Hero):      72`
- **`sizeof(Monster):   64`**
- `sizeof(Orc):       72`
- `10^9 atta` `5222730us`

Variant-based approach

- `sizeof(Character): 48`
- `sizeof(Hero):      56`
- **`sizeof(Monster):   48`**
- `sizeof(Orc):       56`
- `10^9 attacks: 3649402us`

With the variant-based approach, each object of type Monster occupies 75% of the memory space occupied by its equivalent counterpart in the « traditional » approach

We want numbers…

« Traditional » approach

- `sizeof(Character): 48`
- `sizeof(Hero):      72`
- `sizeof(Monster):   64`
- `sizeof(Orc):       72`
- **`10^9 attacks: 5222730us`**

Variant-based approach

- `sizeof(Character): 48`
- `sizeof(Hero):      56`
- `sizeof(Monster):   48`
- `sizeof(Orc):       56`
- **`10^9 attacks: 3649402us`**

As far as speed goes, the variant-based approach consumes 69,88% of the time consumed by the « traditional » equivalent. This is probably mostly due to better cache usage

We want numbers…

- Our choices have measurable consequences
  - Making informed decisions tends to lead to better results

We want numbers…

- Our choices have measurable consequences
- There are of course other technical factors to consider
  - Compile-times: if the set of Bellicose or Damageable types is subject to evolve a lot, the variant-based approach might require more frequent recompilation that the « traditional » one
  - Portability and maintainability: if abstracting away types is important, for example when publishing objects through a shared library, the traditional approach might also be better than a variant-based approach

Bataille royale

Bataille royale

- One hero fighting many Orcs or many heroes fighting one Orc are possibilities, but we can create much more ruckus…
- How about a « bataille royale »?

Bataille royale

- The rules are simple
  - A team made of heroes against team made of monsters
    - We’ll make sure it’s not just « anyone hits on anyone » as that would be chaos
  - The first team with no-one left alive loses…?

Bataille royale

- Let’s do this two different ways
  - One based on the traditional usage of inheritance and virtual member functions
  - Another based on concepts and variants
  - Note that these are not mutually exclusive and can be combined in innovative ways
- First, the « traditional » way

Bataille royale

```
import std;
// or #include <print>, <string>, <string_view>,
// <utility>, <random>
struct Damageable {
   virtual void suffer(int) = 0;
   virtual ~Damageable() = default;
};
```

A Damageable object can suffer()

Bataille royale

A Bellicose object is a Damageable object that can hit() any Damageable object. Note that we made hit_impl() protected here, for reasons we will soon explain

```
// ...
struct Bellicose : Damageable {
   virtual int life() const = 0;
   bool alive() const { return life() > 0; }
   bool dead() const { return !alive(); }
protected:
   virtual void hit_impl(Damageable&) = 0;
};
```

Bataille royale

```
// ...
class unknown {};
enum class Weapon { sword, axe, bow };
auto effect(Weapon w) {
   using namespace std;
   switch(w) {
      using enum Weapon;
   case sword: return pair{ "zing!", 8 };
   case axe:   return pair{ "thunk!", 7 };
   case bow:   return pair{ "dong!", 6 };
   }
   throw unknown{};
}
```

Some heroes will fight with weapons…

Bataille royale

```
// ...
enum class Spell { fireball, lightning_bolt };
constexpr int nb_spells = 2;
auto effect(Spell sp) {
   using namespace std;
   switch(sp) {
      using enum Spell;
   case fireball:       return pair{ "WOOSH!", 12 };
   case lightning_bolt: return pair{ "BZZT!", 11 };
   }
   throw unknown{};
}
```

... others will cast spells

Bataille royale

```
// ...
auto initial_life(int min, int max) {
   // bad but short and simple
   std::mt19937 prng{ std::random_device{}() };
   return std::uniform_int_distribution<> {
      min, max
   }(prng);
}
```

To make things fun, we'll randomize the initial life points of our belligerents

Bataille royale

As announced, we will constrain heroes to hitting monsters just as we will constrain monsters to hit heroes

```
// ...
struct Monster;
struct Hero : Bellicose {
   virtual std::string name() const = 0;
   void hit(Monster &); // constraining to monsters
   virtual ~Hero() = default;
};
struct Monster : Bellicose {
   virtual std::string name() const = 0;
   void hit(Hero & h) { hit_impl(h); } // constraining to heroes
   virtual ~Monster() = default;
};
inline void Hero::hit(Monster & m) { hit_impl(m); } // constraining to monsters
```

Bataille royale

A Warrior is a Hero. As such, it only hits monsters

```cpp
// ...
class Warrior : public Hero {
   std::string name_;
   Weapon weapon_;
   int life_ = initial_life(50, 100);
public:
   std::string name() const override { return name_; }
   auto weapon() const { return weapon_; }
private:
   void hit_impl(Damageable & foe) final { // we know foe is a Monster
      auto [sound, damage] = effect(weapon());
      std::print("{} : {} hits {} for {} damage\n", sound, name(), static_cast<Monster&>(foe).name(), damage);
      foe.suffer(damage);
   }
public:
   Warrior(std::string_view name, Weapon weapon) : name_{ name }, weapon_{ weapon } { }
   int life() const override { return life_; }
   void suffer(int damage) override { life_ -= damage; }
};
```

Bataille royale

A Wizard is also a Hero. As such, it only « hits » monsters (through spellcasting of course)

```
// ...
class Wizard : public Hero {
   std::string name_;
   mutable std::mt19937 prng{ std::random_device{}() };
   int life_ = initial_life(30, 60);
public:
   std::string name() const override { return name_; }
   auto spell() const { return Spell{ std::uniform_int_distribution{ 0, nb_spells - 1 }(prng) }; }
private:
   void hit_impl(Damageable & foe) final { // we know foe is a Monster
      auto [sound, damage] = effect(spell());
      std::print("{} : {} casts a spell on {} for {} damage\n", sound, name(), static_cast<Monster&>(foe).name(), damage);
      foe.suffer(damage);
   }
public:
   Wizard(std::string_view name) : name_{ name } { }
   int life() const override { return life_; }
   void suffer(int damage) override { life_ -= damage; }
};
```

Bataille royale

```
// ...
class Orc : public Monster {
   std::string name_;
   int strength_;
   int life_ = initial_life(25, 75);
public:
   std::string name() const override { return name_; }
   auto strength() const { return strength_; }
private:
   void hit_impl(Damageable & foe) final { // we know foe is a Hero
      std::print("ME {}! HIT {} FOR {} HURT!\n", name(), static_cast<Hero&>(foe).name(), strength());
      foe.suffer(strength());
   }
public:
   Orc(std::string_view name, int strength) : name_{ name }, strength_{ strength } { }
   int life() const override { return life_; }
   void suffer(int damage) override { life_ -= damage > 2 ? damage - 2 : 0 ; } // tough hide
};
```

An Orc is a Monster, one with a tough hide indeed. As such, it only hits heroes

Bataille royale

A Minotaur is also a Monster, and a raging one at that. As such, it only hits heroes

```
// ...
class Minotaur : public Monster {
   std::string name_;
   int strength_, rage_;
   int life_ = initial_life(50, 120);
public:
   std::string name() const override { return name_; }
   auto strength() const { return strength_; }
   auto rage() const { return rage_; }
private:
   void hit_impl(Damageable & foe) final { // we know foe is a Hero
      std::print("GRRRRR {}! INFLICTS {} DAMAGE ON {}!\n", name(), strength(), static_cast<Hero&>(foe).name());
      foe.suffer(strength() + rage());
   }
public:
   Minotaur(std::string_view name, int strength, int rage) : name_{ name }, strength_{ strength }, rage_{ rage } { }
   int life() const override { return life_; }
   void suffer(int damage) override { life_ -= damage; }
};
```

Bataille royale

- Conflict begins
- Avert your eyes if necessary

Bataille royale

```
// ...
// #include <memory>, <vector>, <algorithm>
template <class T>
   bool all_dead(const std::vector<T> &v) {
      namespace rg = std::ranges;
      return rg::all_of(v, [](auto && x) { return x->dead(); });
   }
template <class T, class U>
   void attack(T & a, U & b) {
      a->hit(*b);
   }
```

Bataille royale

```
// ...
template <class T, class Prng>
   T& pick_one(std::vector<T> &v, Prng & prng) {
      using namespace std;
      auto pos = partition(begin(v), end(v), [](auto && e) {
         return e->alive();
      });
      decltype(size(v)) n = distance(begin(v), pos);
      return v[std::uniform_int_distribution<decltype(v.size())>{
         0, n – 1
      }(prng)];
   }
```

Bataille royale

```cpp
// ...
int main() {
   std::mt19937 prng{ std::random_device{}() };
   std::uniform_int_distribution penny{ 0, 1 };
   std::vector<std::unique_ptr<Monster>> monsters;
   monsters.emplace_back(std::make_unique<Orc>("URG", 10));
   monsters.emplace_back(std::make_unique<Minotaur>("Timmy", 5, 3));
   std::vector<std::unique_ptr<Hero>> heroes;
   heroes.emplace_back(std::make_unique<Warrior>(
      "William", Weapon::sword
   ));
   heroes.emplace_back(std::make_unique<Wizard>("Steve"));
   // ...
}
```

Bataille royale

Such violence...

```cpp
// ...
int main() {
   // ...
   while(!all_dead(monsters) && !all_dead(heroes)) {
      if(penny(prng))
         attack(pick_one(heroes, prng), pick_one(monsters, prng));
      else
         attack(pick_one(monsters, prng), pick_one(heroes, prng));
   }
   std::print("{} won!", all_dead(heroes) ? "monsters" : "heroes");
}
```

Bataille royale

https://wandbox.org/permlink/pwjEl1p5mPtXh6Pq

```
// ...
int main() {
   // ...
   while(!all_dead(monsters) && !all_dead(heroes)) {
      if(penny(prng))
         attack(pick_one(heroes, prng), pick_one(monsters, prng));
      else
         attack(pick_one(monsters, prng), pick_one(heroes, prng));
   }
   std::print("{} won!", all_dead(heroes) ? "monsters" : "heroes");
}
```

Bataille royale

- Nothing too bad, but we did need to be careful in order to constrain our types to only hit the appropriate foes

Bataille royale

- Now, for a « concepts and variants » approach
  - Reminder: approaches are not mutually exclusive and can be combined in innovative ways

Bataille royale

```
import std;
// or #include <print>, <string>, <string_view>,
// <utility>, <random>
template <class T>
   concept Damageable = requires(T &b) {
      b.suffer(std::declval<int>());
   };
```

A Damageable object can suffer()

Bataille royale

A Bellicose object is a Damageable object that can hit() any Damageable object. Note the way both tests are expressed: one cannot constrain a concept directy in C++ today so writing

template <Damageable T> concept Bellicose

will (sadly) not work

```
// ...
template <class T>
   concept Bellicose = requires(T &b) {
      requires Damageable<T>;
      b.hit([] -> decltype(auto) {
         Damageable auto & f(); return f();
      }());
   };
```

Bataille royale

```
// ...
class unknown {};
enum class Weapon { sword, axe, bow };
auto effect(Weapon w) {
   using namespace std;
   switch(w) {
      using enum Weapon;
   case sword: return pair{ "zing!", 8 };
   case axe:   return pair{ "thunk!", 7 };
   case bow:   return pair{ "dong!", 6 };
   }
   throw unknown{};
}
```

Some heroes will fight with weapons...

Bataille royale

```
// ...
enum class Spell { fireball, lightning_bolt };
constexpr int nb_spells = 2;
auto effect(Spell sp) {
   using namespace std;
   switch(sp) {
      using enum Spell;
   case fireball:       return pair{ "WOOSH!", 12 };
   case lightning_bolt: return pair{ "BZZT!", 11 };
   }
   throw unknown{};
}
```

... others will cast spells

Bataille royale

```
// ...
auto initial_life(int min, int max) {
   // bad for short and simple
   std::mt19937 prng{ std::random_device{}() };
   return std::uniform_int_distribution<> {
      min, max
   }(prng);
}
```

To make things fun, we'll randomize the initial life points of our belligerents

Bataille royale

A Warrior is a Hero. As such, it should only hit monsters. In this example, this will come from the way client code is written, but we could have used other approaches (a type list of monsters; an empty base class common to all monster types; a tag type used for an is_monster trait; etc.)

```
// ...
class Warrior {
   std::string name_;
   Weapon weapon_;
   int life_ = initial_life(50, 100);
public:
   auto name() const { return name_; }
   auto weapon() const { return weapon_; }
   Warrior(std::string_view name, Weapon weapon) : name_{ name }, weapon_{ weapon } { }
   template <class T>
      void hit(T & foe) {
         auto [sound, damage] = effect(weapon());
         std::print("{} : {} hits {} for {} damage\n", sound, name(), foe.name(), damage);
         foe.suffer(damage);
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damage; }
};
```

Bataill

```
// ...
class Warrior {
   std::string name_
   Weapon weapon_;
   int life_ = initi
public:
   auto name() const
   auto weapon() con
   Warrior(std::stri
   template <class T
      void hit(T &
         auto [sou
         std::pri
         foe.su
      }
   auto life(
   void suff      nt d
};
```

```
// https://wandbox.org/permlink/PkL44dmsn9380fwX
struct Warrior {};
struct Wizard {};
class Orc {}; // not a Hero
#include <tuple>
using heroes = std::tuple<Warrior, Wizard>;
template <class TL, class> struct contains_type;
template <class T, class ... Q, class U>
  struct contains_type<std::tuple<T, Q...>, U>
    : contains_type<std::tuple<Q...>, U> {};
template <class T, class ... Q>
  struct contains_type<std::tuple<T, Q...>, T> : std::true_type {};
template <class T>
  struct contains_type<std::tuple<>, T> : std::false_type {};
template <class T>
  constexpr bool is_hero_v = contains_type<heroes, T>{}();
int main() {
   static_assert(is_hero_v<Warrior> &&!is_hero_v<Orc>);
}
```

Bataille royale

```
// ...
class Warrior {
   std::string name_;
   Weapon weapon_;
   int life_ = initial_life(50, 100);
public:
   auto name() const { return name_; }
   auto weapon() const { return weapon_; }
   Warrior(std::string_view name, Weapon weapon) : n
   template <class T>
      void hit(T & foe) {
         auto [sound, damage] = effect(wea
         std::print("{} : {} hits {} for {} damage\n", so
         foe.suffer(damage);
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damage; }
};
```

```
// https://wandbox.org/permlink/MFnh6kxmHNQuc70z
class Hero {};
struct Warrior : Hero {};
struct Wizard : Hero {};
class Orc {}; // not a Hero
#include <type_traits>
template <class T>
   constexpr bool is_hero_v =
     std::is_base_of_v<Hero, T>;
int main() {
   static_assert(is_hero_v<Warrior>);
   static_assert(!is_hero_v<Orc>);
}
```

Bataille royale

```cpp
// ...
class Warrior {
   std::string name_;
   Weapon weapon_;
   int life_ = initial_life(50, 100);
public:
   auto name() const { return name_; }
   auto weapon() const { return weapon_; }
   Warrior(std::string_view name, Weapon weapon) : name_{
   template <class T>
      void hit(T & foe) {
         auto [sound, damage] = effect(weapon());
         std::print("{} : {} hits {} for {} damage\n
         foe.suffer(damage);
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damag
};
```

```cpp
// https://wandbox.org/permlink/i59ETSNgfPRhLKM4
struct Warrior { using is_hero_tag = void; };
struct Wizard { using is_hero_tag = void; };
class Orc {}; // not a Hero
#include <type_traits>
template <class, class = void>
   struct is_hero : std::false_type {};
template <class T>
   struct is_hero<T, std::void_t<
      typename T::is_hero_tag
   >> : std::true_type {};
template <class T>
   constexpr bool is_hero_v = is_hero<T>{}();
int main() {
   static_assert(is_hero_v<Warrior>);
   static_assert(!is_hero_v<Orc>);
}
```

Bataille royale

```
// ...
class Warrior {
   std::string name_;
   Weapon weapon_;
   int life_ = initial_life(50, 100);
public:
   auto name() const { return name_; }
   auto weapon() const { return weapon_; }
   Warrior(std::string_view name, Weapon weapon) : name_{ name }, weapon_{ weapon } { }
   template <class T>
      void hit(T & foe) {
         auto [sound, damage] = effect(weapon());
         std::print("{} : {} hits {} for {} damage\n", sound, name(), foe.name(), damage);
         foe.suffer(damage);
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damage; }
};
```

The same reasoning goes for Wizard, Orc and Minotaur

Bataille royale

```
// ...
class Wizard {
   std::string name_;
   mutable std::mt19937 prng{ std::random_device{}() };
   int life_ = initial_life(30, 60);
public:
   auto name() const { return name_; }
   auto spell() const { return Spell{ std::uniform_int_distribution{ 0, nb_spells - 1 }(prng) }; }
   Wizard(std::string_view name) : name_{ name } { }
   template <class T>
      void hit(T & foe) {
         auto [sound, damage] = effect(spell());
         std::print("{} : {} casts a spell on {} for {} damage\n", sound, name(), foe.name(), damage);
         foe.suffer(damage);
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damage; }
;
```

Bataille royale

```
// ...
class Orc {
   std::string name_;
   int strength_;
   int life_ = initial_life(25, 75);
public:
   auto name() const { return name_; }
   auto strength() const { return strength_; }
   Orc(std::string_view name, int strength) : name_{ name }, strength_{ strength } { }
   template <class T>
      void hit(T & foe) {
         std::print("ME {}! HIT {} FOR {} HURT!\n", name(), foe.name(), strength());
         foe.suffer(strength());
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damage > 2 ? damage - 2 : 0 ; } // tough hide
};
```

Bataille royale

```
// ...
class Minotaur {
   std::string name_;
   int strength_, rage_;
   int life_ = initial_life(50, 120);
public:
   auto name() const { return name_; }
   auto strength() const { return strength_; }
   auto rage() const { return rage_; }
   Minotaur(std::string_view name, int strength, int rage) : name_{ name }, strength_{ strength }, rage_{ rage } { }
   template <class T>
      void hit(T & foe) {
         std::print("GRRRRR {}! INFLICTS {} DAMAGE ON {}!\n", name(), strength(), foe.name());
         foe.suffer(strength() + rage());
      }
   auto life() const { return life_; }
   void suffer(int damage) { life_ -= damage; }
};
```

Bataille royale

- Conflict begins
- Avert your eyes if necessary

Bataille royale

```
// ...
// #include <variant>, <vector>, <algorithm>
template <class T>
   requires requires(const T&x) { { x.life() } -> std::integral; }
   bool alive(const T &x) {
      return x.life() > 0;
   }
template <class T>
   requires requires(const T&x) { { x.life() } -> std::integral; }
   bool dead(const T &x) {
      return !alive(x);
   }
using hero = std::variant<Warrior, Wizard>;
using monster = std::variant<Orc, Minotaur>;
```

The ideas of « alive() » and « dead() » can be inferred from the life() of our types and do not need to be member functions. We could have done this with our previous approach too

Bataille royale

Expressing algorithms on variants normally means we will visit() them and directly call the desired service on each object

```
// ...
template <class T>
  bool all_dead(const std::vector<T> &v) {
    using namespace std;
    namespace rg = std::ranges;
    return rg::all_of(v, [](auto && x) {
      return visit([](auto && x) { return dead(x); }, x);
    });
  }
template <class T, class U>
   void attack(T & a, U & b) {
      std::visit([](auto && a, auto && b) { a.hit(b); }, a, b);
   }
```

Bataille royale

```
// ...
// precondition: !v.empty()
template <class T, class Prng>
   T& pick_one(std::vector<T> &v, Prng & prng) {
      using namespace std;
      auto pos = partition(begin(v), end(v), [](auto && e) {
         return visit([](auto && e) { return alive(e); }, e);
      });
      decltype(size(v)) n = distance(begin(v), pos);
      return v[std::uniform_int_distribution<decltype(v.size())>{
         0, n – 1
      }(prng)];
   }
```

Bataille royale

```
// ...
int main() {
   std::mt19937 prng{ std::random_device{}() };
   std::uniform_int_distribution penny{ 0, 1 };
   std::vector<monster> monsters{
      Orc{ "URG", 10 }, Minotaur{ "Timmy", 5, 3 }
   };
   std::vector<hero> heroes{
      Warrior{ "William", Weapon::sword }, Wizard { "Steve" }
   };
// ...
}
```

Bataille royale

Such violence...

```
// ...
int main() {
   // ...
   while(!all_dead(monsters) && !all_dead(heroes)) {
      if(penny(prng))
         attack(pick_one(heroes, prng), pick_one(monsters, prng));
      else
         attack(pick_one(monsters, prng), pick_one(heroes, prng));
   }
   std::print("{} won!", all_dead(heroes) ? "monsters" : "heroes");
}
```

Bataille royale

https://wandbox.org/permlink/gkveA2jPsKyyd6rF

```
// ...
int main() {
   // ...
   while(!all_dead(monsters) && !all_dead(heroes)) {
      if(penny(prng))
         attack(pick_one(heroes, prng), pick_one(monsters, prng));
      else
         attack(pick_one(monsters, prng), pick_one(heroes, prng));
   }
   std::print("{} won!", all_dead(heroes) ? "monsters" : "heroes");
}
```

Bataille royale

- Interestingly, we end up with the same code (syntactically) for client code with both approaches
  - At least once the containers have been constructed
  - Some of the auxiliary functions have to be written differently, of course
- This is a good thing

What to make of all this?

What to make of all this?

- You might be wondering what was the point to all of this
  - We played with various ways of doing similar things
  - We explored design issues
  - In no way was all of this exhaustive, obviously

What to make of all this?

- One of the ideas what discussing design options with you
  - It’s fun to do so!

What to make of all this?

- Another was to show some of the strengths and limitations of our language
  - Traditional approaches work, but they have some costs in terms of size, speed, coupling
  - More recent approaches work too and have different costs
  - Some features like concepts are great but still have some limits
    - We showed a few workarounds along the way

What to make of all this?

- We could have gone further
  - Imagine a program where we need to write fight(Hero,Monster) and fight(Monster,Hero) with many types of Hero and many types of Monster
  - The algorithm to pick will depend on both types
    - This could be an animation
    - For us, it will simply be a matter of printing an appropriate message on screen

What to make of all this?

- Compare the following implementations
  - Traditional object-oriented code: https://wandbox.org/permlink/do4dvYe0tQFybdbG (50 lines of code)
    - Double virtual dispatch
    - Quite a lot of coupling
    - Costly to maintain
  - Variant-based code: https://wandbox.org/permlink/OFpcXLjC57icHNKM (30 lines of code)
    - Direct function call
    - There are still maintenance costs
    - Works better with pass-by-value (pass-by-reference would impact calling code)

What to make of all this?

- We need to think before we code
- We face a diversity of problems
- We need a diversity of tools…
  - Some problems are better solved with functions
  - Some lend themselves to « traditional » object-oriented solutions
  - Some benefit from more contemporary approaches

What to make of all this?

- We’re lucky: C++ supports many paradigms and accompanies us instead of forcing us to solve problems « the right way »
  - The world is a rich yet complicated place, after all

What to make of all this?

- We can have fun solving problems, so let's do so!

:)