B5 - Advanced C++

B-CPP-501

R-Type Bootstrap

Write me an ECS

1.0

Entity Component System

What is an ECS ?

An ECS is an architectural design pattern, mostly used in video game development. This pattern follows the principle of composition over inheritance. Instead of defining an inheritance tree as usual in Object Oriented Programming, types are split into small yet highly reusable *components*.

A common composition pattern in OOP would be to put functions modifying these components together with the components, for example a physics components containing mass, velocity, acceleration and hit-box together with functions to move and collide entities that have this component.
In an ECS, we try to keep components as light as possible (more on that later). However, an issue arise if we try to put logic inside of components, as part of this logic could depends on multiple attributes components if they are too "small". One way to prevent this issue while keeping components as small as possible is taking the logic (functions) outside of components. These standalone functions are then free to depend on an arbitrary number of components. We call these functions *systems*.
In our physics component example, it would be split into its part (`position`, `velocity`, `mass`, `acceleration` and `hit-box`), and an array of functions would interact with all entities that have a specific sub-set of components.

We would thus have the following systems:
- **movement_system**: update all entities positions and velocities if they have the following components: `velocity`, `position` and `acceleration`.
- **gravity_system**: update entities accelerations, by taking their mass into account. The following components must be present: `acceleration` and `mass`.
- **collision_system**: handle collisions between all entities that has a hit-box. The entities must be composed from the following components: `position` and `hit-box`.

This pattern allows for a more flexible architecture, with the ability to add/remove components on the fly. You could for example handle status effect by adding a component to an entity, and handling it in the proper system (i.e. adding a `slow` component to an entity, which would be taken into account by the `movement_system` , or a `slow_system` that would update the `velocity` before new `position`s are computed).

There are two other perks with this pattern:
- the way data is organised is close to the way it could be stored in a database (each component is a row in a table)
- the way data is organised plays well with the CPU cache (more on that in a bit).

Wait, where are the entities ?

Nowhere. **Entities don't need to actually exist or contain anything.**

In an ECS, Entities are not something you should really care about. There are no Entity type that would contains its components, because there are no need for one. You can consider an entity as an index, or an ID. Some systems don't need to know which entities they are working on, as long as they get all components they need for a given entity at the same time. Other systems needs some way to modify a given

entity (add a component to it, for another system to interact with it). These systems would just have to call the object or function handling all the entities and their components, giving it the ID (or index) of the entity.

What is CPU Cache ? Is it important ?

CPU Cache is the fastest memory (excluding registers) available to your CPU. The more the CPU can rely on it, the faster it can work. The CPU Cache is organised in small chunks of memory (called cache-lines) that the CPU work with. If the piece of data the CPU wants to access is in its cache, it can work with it. If it isn't (we call it a cache-miss) and the CPU has to discard a whole cache-line and load the memory to fill it. When loading from a RAM address into its cache, the CPU always fetches more memory so it fills the cache-line. Thus putting data that will be accessed together, or sooner than other data is key to optimizing cache usage.

Is it important for the project ? Not really, so we won't try to make an optimized ECS. The point of the bootstrap is to make you write a simple ECS so you know what it looks like, and how to work with one.

By using usual OOP pattern of inheritance (or by putting every pieces of data for one type together), we fill the cache-line with fewer objects than we could if we used only what is needed for the current computation.

For example, in a video game, we usually use a main loop that have more or less these steps :
- get input
- update entities (itself composed of several steps)
- redraw

If you have to update all positions before you check for collisions, you will update every entity positions using their velocities and accelerations, before you load these again, this time using their positions and hitboxes. In an ECS, position and velocity components would be loaded only which mean more data could fit in a single cache-line (thus less load from RAM).
Using traditional OOP patterns, a full entity is loaded, which takes more space (as we also load parts of the object that we don't need), thus less entities can be loaded at once in one cache-line (thus more cache-misses happen).

Let's write an ECS

In this part, we're going to write a simple ECS. The goal is to make you understand how it's structured, not to write the best/most optimized one, nor to write the best engine by making it modular.

In this bootstrap we'll use some c++17 features. Although they can be reimplemented, it's highly recommended that you first read the documentation on classes we will use, and that you compile the bootstrap using c++17.

Our ECS will be composed of several parts :
- **Registry** : This class is the core of the ECS. It will create and manage entities, hold components containers, as well as manage systems.
- **Sparse Arrays** : these will hold one type of components. There will be a "hole" when an entity does not have the specific component. These are stored inside the registry.

In this bootstrap, we'll avoid inheritance by using a concept known as type erasure.

Think **void** *.

In c++17, a special type has been added (`std::any`), that enables type-safe type erasure, by storing the value alongside its type, and checking it when we try to cast it back.

Our components will be stored in what's called a sparse array. Instead of keeping the ID of the entity a component is part of, we will use the index in the array the component is stored in. That means we need a way to represent a non existent component. We will use `std::optional` in that effect, as it's the c++ way of achieving this.

Take some time to read the documentation of std::any and std::optional.

Step 0 - Defining entities

Entity can be defined as a `size_t`, or as a named-type, convertible to `size_t`. In this bootstrap we will implement the latter.
Write a class `entity` that stores a `size_t`, and can be converted to `size_t`. This class must be implicitly convertible to `size_t`, but should not be implicitly constructible from `size_t`.

We want the conversion to be **explicit**.

STEP 1 - SPARSE ARRAYS

1.1 - WRITING A CUSTOM CONTAINER

Our components container will be implemented as a sparse array. This kind of container works like traditional array (or vector). The main difference is that a value might no be existing at a specific index. This can be quite useful to store components that are defined for most entities, because you don't have to store the entity ID alongside the component.

Implement the following class:

```
template <typename Component> // You can also mirror the definition of std::vector,
    that takes an additional allocator.
class sparse_array {
    public:
    using value_type = ???; // optional component type
    using reference_type = value_type &;
    using const_reference_type = value_type const &;
    using container_t = std::vector<value_type>; //optionally add your allocator
        template here.
    using size_type = typename container_t::size_type;

    using iterator = typename container_t::iterator;
    using const_iterator = typename container_t::const_iterator;

    public:
    sparse_array(); // You can add more constructors.

    sparse_array(sparse_array const &); //copy constructor
    sparse_array(sparse_array &&) noexcept; //move constructor
    ~sparse_array();

    sparse_array & operator=(sparse_array const &); //copy assignment operator
    sparse_array & operator=(sparse_array &&) noexcept; //move assignment operator

    reference_type operator[](size_t idx);
    const_reference_type operator[](size_t idx) const;

    iterator begin();
    const_iterator begin() const;
    const_iterator cbegin() const;

    iterator end();
    const_iterator end() const;
    const_iterator cend() const;

    size_type size() const;

    reference_type insert_at(size_type pos, Component const &);
    reference_type insert_at(size_type pos, Component &&);

    template <class... Params>
    reference_type emplace_at(size_type pos, Params &&...); // optional

    void erase(size_type pos);

    size_type get_index(value_type const &) const;
```

```
    private:
    container_t _data;
};
```

Feel free to add functions as needed. Most functions are straightforward, you just have to call the corresponding function of the underlying `std::vector`.
Here are the functions you **have** to implement yourself:
- **insert_at**: These functions should insert the component at a specific index, erasing the old value and resizing the vector if needed.
- **emplace_at**: Same as insert_at, but the component is built in-place, or `std::move`'d into the container.
- **erase**: Erase a value at a specified index.
- **get_index**: Take a reference to an optional component, and return its index.

You won't always be able to compare two components, as entities can have identical components.

std::addressof

The `emplace_at` function build the component object in-place. To achieve this, we must destroy the already present object, then build the new one at the same place. Fortunately, `std::vector` allows access to their allocator, which can be used to do so. C++ standard recommend that developer interact with allocator though the use of a special class `std::allocator_traits`.

Read the documentation first ! std::allocator_traits

You can retrieve the type of an object at compile-time using decltype

Step 2 - Registry

In this step, you'll have to write the core class of the ECS. During the first step, the goal is to store every `sparse_array` in a container, such that we can retrieve these using their types.

Step 2.0 - Types as indices

We want to be able to store and retrieve arrays of components in the `registry`. A naive approach to do so would be to use a `std::vector` as a container, but we would then need to know which array is stored at which index. We could do so by registering our arrays in the same order every time, or by assigning an ID to each components type. However both solution would prevent us from loading new component types from dynamic libraries while ensuring there are no collisions.
Another way to handle this would be to use `std::string` in an associative container, but that wouldn't prevent the possibility of collision between two component identifiers either.
Fortunately, since C++11, a class is available that allows us to create a unique index inside an associative container, that use the actual type of an object.

Take some time to read its documentation: std::type_index

Using this type ensure that components can be loaded at runtime, while keeping an easy way to retrieve these. All we need now is to choose the best associative container, and find some way to store our components arrays.

When trying to decide which type of associative container you need, you should always think about which features you need. If you need keys to be sorted, or to search for an item, you might want to use an ordered container (`std::map`). However, if you only want to access one specific item at a time, an unordered container would be more efficient (`std::unordered_map`).

Step 2.1 - Storing multiple types in a container

In C we could pass an untyped pointer to any class using `void *`. A more object oriented style of doing so would be to use an interface or abstract class. While both ways work in C++, they are not necessarily the best ways to do so, as `void *`s are type unsafe, and `dynamic_cast`s require you to try every type, or add some identifiers. While in our case a simple interface and a static_cast would work (we use our types to retrieve the data anyway), we're going to introduce a type safe way to store unrelated data inside a container.
Since C++17, a special wrapper exists in the standard library that does just that: `std::any`. An instance of `std::any` stores a reference to an object in a type opaque way. A reference to the stored object can be retrieved using `std::any_cast`. The cast will fail and an exception will be raised in case the destination type is not compatible with the stored object.

Step 2.2 - Writing the registry.

You should now be able to write the following class :

```
class registry {
    public:
        template <class Component>
        sparse_array<Component> &register_component();

        template <class Component>
        sparse_array<Component> &get_components();

        template <class Component>
        sparse_array<Component> const &get_components() const;

    private:
    /*associative container type*/ _components_arrays;
}
```

register_components adds a new component array to our associative container.
get_components are functions to retrieve the array that stores a specific type of components.
You have to write these functions such that the index of an array is made using `std::type_index`, and the `std::any_cast` is made by the `registry` instead the system that will use the `sparse_array`s.

Step 2.3 - Managing entities

Our registry should also be responsible of managing entities. To create a new one is quite easy, as it only has to keep track of how many entities exist. A more advanced way to do so would be to keep track of dead entities so that you can reuse their IDs instead of always increasing a number. (Remember, our sparse arrays have holes in them, and while it can be more efficient than storing an array of indexes, or using associative container, the more holes there are, the less efficient they become, both in term of speed and memory footprint).

An issue arises when we think of removing an entity. Although we store `sparse_array`s in a single container, we cannot access them without knowing the type of component we need. One solution would then be to revert back to using inheritance, so we can call our `erase` function on the `sparse_array`s, but we've tried to avoid inheritance until now, so we'll try to avoid it here if possible.

We have access to the type of the component in our `register_component` function

We are going to tweak our registry and its register function to be able to create and store functions that can erase a component from every `sparse_array`s. The signature of our functions will be as follow: `void ( registry &, entity_t const &)`.
Modify your registry so it stores a container for these functions, then update `register_component` so it creates and stores said functions.

You can either use a templated function, or create a lambda function upon component registration. Use `std::function` to store these inside the container.

Add the following methods to the registry, or a class of your choice that will be stored inside the registry and manages the entities.

```
entity_t spawn_entity();
entity_t entity_from_index(std::size_t idx);
void kill_entity(entity_t const &e);

template <typename Component>
typename sparse_array<Component>::reference_type add_component(entity_t const &to,
    Component &&c);
template <typename Component, typename... Params>
typename sparse_array<Component>::reference_type emplace_component(entity_t const &to
    , Params &&...p);

template <typename Component>
void remove_component(entity_t const &from);
```

The `add_component` and `emplace_component` functions should take their parameters by lvalue reference as well as rvalue reference. You could overload `add_component`, but it won't work for `emplace_component`. When dealing with templated types however, `Type &&` (usually called a "universal reference") means that the parameter will be either an lvalue or an rvalue, depending on how the function was called. You can then call `std::forward<Type>` to conserve this property when calling another function.
You can now modify the entities so only their managing class can build them.

Ever wondered what was the use of friend ? It's a great way to call private constructor ensuring only one class can build a type.

Step 3 - Using the ECS

You should now have a basic ECS. You will use it to implement a small program that move a 2D square in an SFML window. The first step will be to define our components, and register these to the registry. Once the components are defined, you will write several systems that will do the work.

Step 3.1 - Components list

Here are the components we will use for this exercise:
- **position**: This component contains the 2D position of an entity.
- **velocity**: This component contains two variables, representing the current velocity of an entity.
- **drawable**: This component makes our entity drawable.
- **controllable**: An object with this component can be controlled by the user, using its keyboard.

Write these as `struct`s, then register them to the registry. Implementation of `controllable` and `drawable` are up to you.

Step 3.2 - Systems

We now want to add some logic to our "game". Our systems can be free functions, lambdas, or class instances that overload the `operator()` function. The signature of our systems will be as follows:

```
void example_system(registry &);
```

Let's add the following systems:
- **position_system**: this system use both `position` and `velocity` components. For all entities having both, it adds the current velocity to the position.
- **control_system**: for entities with `controllable` and `velocity` components, it set the velocity to a fixed value depending on which keys are pressed on the keyboard, or 0 if there are no inputs.
- **draw_system**: uses the `drawable` component to know *what* to draw, and the `position` to know *where* to draw.

Here is a small example, that use both `position` and `velocity`, and log these on `cerr`

```
void logging_system(registry &r) {
    auto const &positions = r.get_components<component::position>();
    auto const &velocities = r.get_components<component::velocity>();

    for (size_t i = 0; i < positions.size() && i < velocities.size(); ++i) {
        auto const &pos = positions[i];
        auto const &vel = velocities[i];

        if (pos && vel) {
            std::cerr << i << ": Position = { " << pos.value().x << ", " << pos.value
                ().y << " }, Velocity = { " << vel.value().vx << ", " << vel.value().
                vy << " }" << std::endl;
        }
    }
}
```

Step 3.3 - Wrapping it all together

Write a program that will register the components defined earlier, then create the following entities:
- An entity that is movable, using all components.
- Some static entities, that have `drawable` and `position` components, but not `velocity`.

You can now try to add new components and systems. How easy would it be to add some acceleration or deceleration (your `control_system` would interact with the acceleration component)? Or entities that move across the screen, in a loop (using a special component ?).

Step 4 - Going further

Step 4.1 - Moving systems inside the registry

While our ECS is usable right now, the systems are responsible of fetching the components, which can add a lot of boilerplate code in every functions. It would be better for the registry to call our systems directly, passing properly typed parameters.
We can modify our example system from step 3.2 thusly:

```
void logging_system(registry &r,
                    sparse_array<component::position> const &positions,
                    sparse_array<component::velocity> const &velocities) {
    for (size_t i = 0; i < positions.size() && i < velocities.size(); ++i) {
        auto const &pos = positions[i];
        auto const &vel = velocities[i];

        if (pos && vel) {
            std::cerr << i << ": Position = { " << pos.value().x << ", " << pos.value
                ().y << " }, Velocity = { " << vel.value().vx << ", " << vel.value().
                vy << " }" << std::endl;
        }
    }
}
```

We now need some way to register our function in our registry. We will use the same trick we did to “kill” an entity, wrapping the call to our systems in a lambda or a templated function.
Add the following functions in the registry:

```
    template <class... Components, typename Function>
    void add_system(Function && f); // perfect forwarding in lambda capture, anyone ?
    // or
    template <class... Components, typename Function>
    void add_system(Function const &f); // taking it by reference.

    void run_systems();
```

This function will store inside the registry a function object with the following operator() : `void operator()(registry &);` that will call our system giving it the components arrays it need. The `add_system` function take our system as its only parameter. We use a template here to be able to either write systems as lambdas, use any type with an overloaded `operator()` implemented, or a free function.

The last template parameter for `add_system` is deduced automatically, and allows for any Callable type to be used here.

Step 4.2 - Container adaptor and custom iterator

Until now, you had to check whether a component was present or not, and you either had to get the id of a component or use a standard `for` loop to iterate over multiple components. This adds some unnecessary boilerplate code in every system.
To be able to use it in a ranged base `for`, a class must expose `begin` and `end` functions, which **must** return a type that works as a legacy pointer. Here are the operations they must support:
- `operator*()` : returns a reference or a value.
- `operator->()` : returns a value such that `type->m` is equal to `(*type).m`
- `operator++()` : increments the instance, then returns an lvalue reference to it (`++val`)
- `operator++(int)` : increments the instance, and returns its previous state (`val++`)
- `operator!=(type const &, type const &)` : compares both values.

We will write our own iterator class that will at least handle these operations. As we write our own iterator, we might as well make it work with some standard algorithms, such as `std::for_each`. To do so, we must at least implement C++ `LegacyInputIterator` named requirements. Named requirements define a named category of types that can be used in a generic context, without being related by inheritance.

You should at least read about LegacyInputIterator, and might be interested in LegacyForwardIterator

For our iterator to be used in a ranged-based `for`, we need a pseudo container, as ranged-based loop are just a syntactic sugar that get transformed in a traditional loop using iterators.

```
for (auto &v : container) { /* body */ }
// gets desugarized (i.e. re-written) to
for (auto it = container.begin(), auto end = container.end(); it != end; ++it) {
    auto &v = *it;
    { /* body */ }
}
```

The container adaptor we are going to write will be built from several `sparse_array`s, and will produce iterators that can iterate upon all of them at the same time, skipping values if any component is missing. Dereferencing the iterator will give us the following type : `std::tuple<Component1 &, Component2 &, Component3 &, ...>`.

Let's first write our iterator class:

```
    template <class... Containers>
    class zipper_iterator {
        template <class Container>
        using iterator_t = ??? // type of Container::begin() return value

        template <class Container>
        using it_reference_t = typename iterator_t<Container>::reference;

        public:
        using value_type = std::tuple</* retrieve it_reference_t::value() type */
            &...>; // std::tuple of references to components
        using reference = value_type;
        using pointer = void;
        using difference_type = size_t;
        using iterator_category = /* proper iterator tag */;
        using iterator_tuple = std::tuple<iterator_t<Container>...>;

        // If we want zipper_iterator to be built by zipper only.
        friend containers::zipper<Containers...>;
        zipper_iterator(iterator_tuple const &it_tuple, size_t max);
        public:
            zipper_iterator(zipper_iterator const &z);

            zipper_iterator operator++();
            zipper_iterator &operator++(int);

            value_type operator*();
            value_type operator->();

            friend bool operator==(zipper_iterator const &lhs, zipper_iterator const
                &rhs);
            friend bool operator!=(zipper_iterator const &lhs, zipper_iterator const
                &rhs);

        private:
            // Increment every iterator at the same time. It also skips to the next
                value if one of the pointed to std::optional does not contains a
                value.
            template <size_t... Is>
            void incr_all(std::index_sequence<Is...>);

            // check if every std::optional are set.
            template <size_t... Is>
            bool all_set(std::index_sequence<Is...>);

            // return a tuple of reference to components.
            template <size_t... Is>
            value_type to_value(std::index_sequence<Is...>);
        private:
            iterator_tuple _current;
            size_t _max; // compare this value to _idx to prevent infinite loop.
            size_t _idx;

            static constexpr std::index_sequence_for<Containers...> _seq{};
        };
}
```

decltype allows you to retrieve the type of an expression at compile-time, std::declval() creates a pseudo-value of type T, at compile time

std::index_sequence_for are helpers that will allow you to work on every part of your tuple, using fold expressions

Reading std::tie will help you to write `to_value`

Next, write the zipper pseudo container:

```
template <class... Containers>
class zipper {
    public:
        using iterator = zipper_iterator<Containers...>;
        using iterator_tuple = typename iterator::iterator_tuple;

        zipper(Containers &...cs);

        iterator begin();
        iterator end();

    private:
        // helper function to know the maximum index of our iterators.
        static size_t _compute_size(Containers &... containers);
        // helper function to compute an iterator_tuple that will allow us to
            build our end iterator.
        static iterator_tuple _compute_end(Containers &... containers);
    private:
        iterator_tuple _begin;
        iterator_tuple _end;
        size_t _size;
};
```

If you don't care about the index of your entity, you can now use the zipper class in ranged-based `for`, or some standard algorithm.
You should now be able to write the `indexed_zipper` class by yourself. It works the same way as the zipper class, but its iterator has a size_t in the tuple it returns.
Using this class, we can rewrite our example system once again.

```
void logging_system(registry &r,
                    sparse_array<component::position> const &positions,
                    sparse_array<component::velocity> const &velocities) {
    for (auto &&[i, pos, vel] : indexed_zipper(positions, velocities))
        std::cerr << i << ": Position = { " << pos.value().x << ", " << pos.value().y
             << " }, Velocity = { " << vel.value().vx << ", " << vel.value().vy << "
            }" << std::endl;
}
```

4.3 - Optimizing memory footprint

While our `sparse_array` is useful, it can become memory inefficient if a component is barely used (for example, a tag component, that would only be present on the user controlled entities). In your actual r-type, you might want to have an hybrid type, that can either work as a sparse array, or as two `std::vector`s (one storing entities ids, the other storing components), depending on how dense you think the containers will be (you could also compute this density at runtime, and change the underlying storage on the fly).

You should read a bit on C++ type-safe union, and the visitor pattern (std::variant and std::visit).