CPP-08

* Smart pointers manage memory

⇒ Smart pointers apart from memory allocation behave exactly as raw pointers:

- → Can be set to nullptr
- → Use *ptr to dereference ptr.
- → Use ptr → to access methods.
- → Smart pointers are polymorphic.

⇒ Additional functions of smart pointers:

- ptr.get() ⟶ {returns the raw pointer that the smart pointer manages}
- ptr.reset(raw_ptr)
  - → freeing memory of currently managed pointer & start managing the new raw_ptr passed.

* Unique pointer

⇒ Construction of a unique pointer takes ownership of a provided raw pointer.

Syntax

```
auto p = std::unique_ptr<Type>(new Type(<Params>));
```

⇒ Unique pointer has no copy constructor.

⇒ Guarantees that memory is always owned by a single unique pointer.

* Shared pointer

⇒ Constructed just like unique pointer.
⇒ Can be copied.
⇒ Stores a usage counter and a raw pointer.
  - ↳ Increases usage counter when copied
  - ↳ Decreases usage counter when destructed
  - ↳ Frees memory when counter reaches 0.
⇒ Can be initialized from a unique pointer.

Syntax

```
auto p = std::make_shared<Type>(<Params>);
```

* When to use what?

⇒ Use smart pointer when the pointer must manage memory.
⇒ By default unique_ptr
⇒ If multiple objects must share ownership over something, use a shared_ptr to it.

* Associative Containers

* std::map

⇒ #include <map> to use std::map.
⇒ Stores items under unique keys.
⇒ Key can be any type with operator < defined.
⇒ map has keys sorted.

⇒ std::map<KeyT, ValueT> m = {
{Key, Value}, {Key, Value}, {Key, Value}};

⇒ Add item to map: m.emplace(Key, Value);

⇒ Modify or add item: m[Key] = Value;

⇒ Get (const) ref to at item: m.at(Key);

⇒ Check if Key Present: m.count(Key) > 0;

⇒ Check size: m.size();

* std::unordered_map

⇒ #include <unordered_map> to use
std::unordered_map.

⇒ Serves same purpose as std::map and have exactly same interface as std::map.

⇒ Implemented as a hash table.

↳ Key type has to be hashable
(Typically int, string as a key)

* Iterating over a maps

```
for(const auto& kv : m) {
    Const auto& Key = kv.first;
    Const auto& Value = kv.second;
    //Do important work
}
```

* Type Casting

→ Type Conversion

⇒ There are 3 ways of type casting

- → Static_cast
- → reinterpret_cast
- → dynamic_cast

* Static_cast

⇒ Syntax: `static_cast<NewType>(Variable)`

⇒ Convert type of a variable at compile time.

⇒ Rarely needed to be used explicitly.

⇒ Can happen implicitly for some types, eg. float can be cast to int.

⇒ Pointer to an object of a Derived can be upcasted to a pointer of a base class.

⇒ Enum value can be casted to int or float.

* reinterpret_cast

⇒ Syntax:

`reinterpret_cast<NewType>(Variable)`

⇒ Reinterpret the bytes of a variable as another type.

⇒ Mostly used when working binary data.

* dynamic_cast

⇒ Syntax: dynamic_cast<Base*>(derived_ptr)

⇒ Conversion happens at runtime.

⇒ GOOGLE-STYLE → Avoid using dynamic casting.

* Enumeration classes

⇒ Store an enumeration of options

⇒ Usually derived from int type.

⇒ Options are assigned consequent numbers.

⇒ Mostly used to pick path in switch.

```
enum class EnumType {OPTION_1, OPTION_2};
```

⇒ Use values as:

```
EnumType::OPTION_1
```

⇒ GOOGLE-STYLE
↳ Name enum type as other types, CamelCase

⇒ GOOGLE-STYLE
↳ Name values as constants
kSomeConstant or ALL_CAPS.

* Read & Write to binary files

⇒ We write a sequence of bytes

⇒ We must document the structure well, otherwise noone can read the file.

⇒ Writing/reading is fast

⇒ Substantially smaller than ascii-files.