



# B3 - Paradigms Seminar

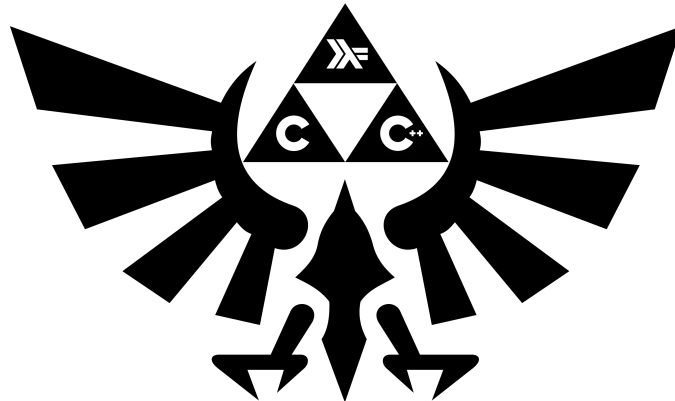
---

B-PDG-300

## Day 07 AM

---

Resistance is Futile



2.0



# Day 07 AM

language: C++



- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

All your exercises will be compiled with `g++` **and the** `-std=c++20 -Wall -Wextra -Werror` **flags**, unless specified otherwise.

All output goes to the standard output, and must be ended by a newline, unless specified otherwise.



None of your files must contain a `main` function, unless specified otherwise. We will use our own `main` functions to compile and test your code. It will include your header files.

For each exercise, the files to turn-in are path relative to the root of the directory. So you **don't** have to put everything in an `exXX` folder.



Read the examples **CAREFULLY**. They might require things that weren't mentioned in the subject...



The `*alloc`, `free`, `*printf`, `open` and `fopen` functions, as well as the `using namespace` keyword, are forbidden in C++. By the way, `friend` is forbidden too, as well as any library except the standard one.



## UNIT TESTS

---

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the “**How to write Unit Tests**” document on the intranet, available [here](#).

For them to be executed and evaluated, put a `Makefile` at the root of your directory with the `tests_run` rule as mentionned in the documentation linked above.



## EXERCISE 0 - THE FEDERATION

**Turn in:** `Federation.hpp`, `Federation.cpp`, `WarpSystem.hpp`, `WarpSystem.cpp`

The **United Planets Federation** is an alliance of people able to travel through space. They all possess the distortion speed – or warp – technology, letting them travel through subspace, and all share common values.

**Starfleet** is an organization tightly coupled to the **Federation**.

Its primary mission is to collect as much information as possible about the **Universe** (and life and everything).

The fleet also has a defensive purpose (which is why all their vessels are prepped and armed), which can turn offensive if need be.

You must create a `Federation` namespace, which contains all the elements that allow the **Federation** to exist. Within the `Federation` namespace, create a nested `Starfleet` namespace. It contains a `Ship` class, which will be used to create spaceships.

Each `Ship` must have the following attributes:

```
int _length;
int _width;
std::string _name;
short _maxWarp;
```



These properties must all be provided during the `Ship`'s construction, and cannot be later modified.

The class' constructor must have the following prototype:

```
Ship(int length, int width, std::string name, short maxWarp);
```

Upon creation, each `Ship` prints the following to the standard output:

```
The ship USS [NAME] has been finished.
It is [LENGTH] m in length and [WIDTH] m in width.
It can go to Warp [MAXWARP]!
```



You must of course replace `[NAME]`, `[LENGTH]`, `[WIDTH]` and `[MAXWARP]` with the appropriate values.

Each `Ship` requires a complex system to navigate through space, which you must have to provide. As this system is not exclusive to the **Federation's** Ships, you must create a new `WarpSystem` namespace. This namespace will house the `QuantumReactor` class, with a single attribute:

```
bool _stability;
```

which will not be provided during the object's construction, but will instead be set to `true` by default.



You must also provide an `isStable` member function which verifies the stability of the `QuantumReactor`, as well as a `setStability` member function which can modify it.

```
bool isStable();  
void setStability(bool stability);
```

`WarpSystem` will also contain a `Core` class with a single attribute:

```
QuantumReactor *_coreReactor;
```

This pointer to `QuantumReactor` must be provided when constructing the object.; a `checkReactor()` member function will provide access to the reactor, by returning a pointer to the `QuantumReactor`.

---

The `Ship` class can now have a `setupCore` member function, taking a pointer to a `Core` as a parameter and returning nothing.

This function will hold the `Core` in the `Ship` and print the following to the standard output:

```
USS [NAME]: The core is set.
```

`Ship` should also have a `checkCore` member function taking no parameters and printing the following to the standard output:

```
USS [NAME]: The core is [STABILITY] at the time.
```



[STABILITY] must be replaced by “*stable*” if `stability` is `true` and by “*unstable*” otherwise.

It must also be possible to create `Ship` objects that do not belong to the `Starfleet`.

These objects have the same functions and attributes as the other `Ships`, but the building process is slightly different.

An independent ship has a maximum speed of 1. Upon creation, it prints the following:

```
The independent ship [NAME] just finished its construction.  
It is [LENGTH] m in length and [WIDTH] m in width.
```

The other functions' output may also be different, as you will see in the example.



The following code must compile and print out what follows:

```
int main()
{
    Federation::Starfleet::Ship UssKreog(289, 132, "Kreog", 6);
    Federation::Ship Independent(150, 230, "Greok");
    WarpSystem::QuantumReactor QR;
    WarpSystem::QuantumReactor QR2;
    WarpSystem::Core core(&QR);
    WarpSystem::Core core2(&QR2);

    UssKreog.setupCore(&core);
    UssKreog.checkCore();
    Independent.setupCore(&core2);
    Independent.checkCore();

    QR.setStability(false);
    QR2.setStability(false);
    UssKreog.checkCore();
    Independent.checkCore();

    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -W -Wall -Werror -Wextra *.cpp
~/B-PDG-300> ./a.out | cat -e
The ship USS Kreog has been finished.$
It is 289 m in length and 132 m in width.$
It can go to Warp 6!$
The independent ship Greok just finished its construction.$
It is 150 m in length and 230 m in width.$
USS Kreog: The core is set.$
USS Kreog: The core is stable at the time.$
Greok: The core is set.$
Greok: The core is stable at the time.$
USS Kreog: The core is unstable at the time.$
Greok: The core is unstable at the time.$
```



## EXERCISE 1 - THE BORGS

---

Turn in: `Federation.hpp/cpp`, `WarpSystem.hpp/cpp`, `Borg.hpp/cpp`



You must reuse the `Federation` and `WarpSystem` files from the previous exercise.

The universe is a big place.

Spreading their influence from the Delta quadrant, the Borgs are a dangerous race and have incredible technology in their possession, thanks to their power of assimilation.

Create a `Borg` namespace containing a `Ship` class.

The Borg's Ships are different from the Federation's in many aspects:

- first and foremost, they have the shape of a cube. Thus, they have no width and height, but a single side length;
- they have no name either.

Their attributes must be:

```
int _side;  
short _maxWarp;
```

The Borg vessels are built from a unique model: their side is 300 meters long, and their maximum speed is Warp 9.

These values are not provided upon construction.

When a Borg `Ship` is built, it prints the following to the standard output:

```
We are the Borgs. Lower your shields and surrender yourselves unconditionally.  
Your biological characteristics and technologies will be assimilated.  
Resistance is futile.
```

A Borg vessel does not print anything when installing a `Core`.

When verifying it however, it prints, if `stability` is true:

```
Everything is in order.
```

or, if `stability` is not true:

```
Critical failure imminent.
```

---

Starfleet needs outstanding crewmen and captains to face this threat.

Create a `Captain` class inside the `Starfleet` namespace, with the following attributes:

```
std::string _name;           // provided during construction  
int _age;                    // not provided during construction
```



In addition to these attributes, add functions that let you query the captain's name and age, as well as a function that modifies their age:

```
std::string getName();  
int getAge();  
void setAge(int age);
```

Modify Starfleet's `Ship` class so that it can be led by a captain.

It must hold a pointer to a `Captain`, that can be modified using the following function:

```
void promote(Captain *captain);
```

This function must print the following to the standard output:

```
[CAPTAIN NAME]: I'm glad to be the captain of the USS [SHIP NAME].
```



Of course, replace the names by the appropriate values.

Create an `Ensign` class, with the following attribute:

```
std::string _name;
```

The only way to create an `Ensign` is the following:

```
Ensign(std::string name);
```



The following code must NOT compile:

```
Ensign Chekov;  
Ensign Chekov = (std::string)"Pavel Andreievich Chekov";
```

Upon construction, an `Ensign` prints:

```
Ensign [NAME], awaiting orders.
```





The following code will compile and display what follows:

```
int main()
{
    Federation::Starfleet::Ship UssKreog(289, 132, "Kreog", 6);
    Federation::Starfleet::Captain James("James T. Kirk");
    Federation::Starfleet::Ensign Ensign("Pavel Chekov");
    WarpSystem::QuantumReactor QR;
    WarpSystem::QuantumReactor QR2;
    WarpSystem::Core core(&QR);
    WarpSystem::Core core2(&QR2);

    UssKreog.setupCore(&core);
    UssKreog.checkCore();
    UssKreog.promote(&James);

    Borg::Ship Cube;
    Cube.setupCore(&core2);
    Cube.checkCore();

    return 0;
}
```

```
~/B-PDG-300> g++ -W -Wall -Werror -Wextra *.cpp
~/B-PDG-300> ./a.out | cat -e
The ship USS Kreog has been finished.$
It is 289 m in length and 132 m in width.$
It can go to Warp 6!$
Ensign Pavel Chekov, awaiting orders.$
USS Kreog: The core is set.$
USS Kreog: The core is stable at the time.$
James T. Kirk: I'm glad to be the captain of the USS Kreog.$
We are the Borgs. Lower your shields and surrender yourselves unconditionally.$
Your biological characteristics and technologies will be assimilated.$
Resistance is futile.$
Everything is in order.$
```



## EXERCISE 2 - GET MOVING!

Turn in: Federation.hpp/cpp, WarpSystem.hpp/cpp, Borg.hpp/cpp

At some point, your Ships will need to move.

Add the following attributes to your Ship classes:

```
Destination _location;  
Destination _home;
```

Destination is an enumeration defined in the Destination.hpp file.

\_home is set to:

```
EARTH           // for Ships of Federation::Starfleet  
VULCAN          // for Ships of Federation  
UNICOMPLEX      // for Ships of Borg
```

Upon construction, \_location = \_home.

Add the following member functions to your ships:

```
bool move(int warp, Destination d);    // set _location to d  
bool move(int warp);                  // set _location to _home  
bool move(Destination d);              // set _location to d  
bool move();                           // set _location to _home
```

These functions must return true if these 3 assertions are true:

1. warp <= \_maxWarp,
2. d != \_location,
3. QuantumReactor::\_stability == true.

They return false otherwise.



Of course, if the function does not return true, the Ship does not move.



## EXERCISE 3 - THIS IS WAR

Turn in: Federation.hpp/cpp, WarpSystem.hpp/cpp, Borg.hpp/cpp

Now that the ships can move, they need a way to attack and defend themselves.  
Provide Starfleet's Ships with these new attributes:

```
int _shield;  
int _photonTorpedo;
```

With these getters and setters:

```
int getShield();  
void setShield(int shield);  
int getTorpedo();  
void setTorpedo(int torpedo);
```

Upon construction, `_shield` is initialized to 100.

Modify `Starfleet::Ship`'s constructor to make the following calls possible:

```
Ship(int length, int width, std::string name, short maxWarp, int torpedo = 0);  
Ship();
```

They must produce the following outputs:

```
The ship USS [name] has been finished.  
It is [length] m in length and [width] m in width.  
It can go to Warp [maxWarp]!  
Weapons are set: [Torpedo] torpedoes ready.
```

Or, if no information is given:

```
The ship USS Enterprise has been finished.  
It is 289 m in length and 132 m in width.  
It can go to Warp 6!
```

Calling the constructor with no parameters will give all attributes their default value, as shown above.



If `torpedo` is set to 0 don't print the line about weapons. Meaning the calls and outputs from exercise 0 must still work and be the same.

Implement the following member functions for the `Starfleet's Ships`:

```
void fire(Borg::Ship *target);  
void fire(int torpedoes, Borg::Ship *target);
```

Every call to the `fire` function reduces by 1 or `torpedoes` the value of `_photonTorpedo`, and prints:

```
[SHIPS NAME]: Firing on target. [TORPEDO] torpedoes remaining.
```

The function then reduces by `50 * torpedoes` the target's `_shield` attribute.

If the `Ship` runs out of torpedoes, it prints:

```
[SHIP NAME]: No more torpedo to fire, [CAPTAIN NAME]!
```

Of course, the ship can't fire more torpedoes than it has in store.

If it tries to do so, it should print the following message:

```
[SHIP NAME]: No enough torpedoes to fire, [CAPTAIN NAME]!
```



Add a `getCore` member function to the `Federation::Ship` class.

It takes no parameter and returns a pointer to the `Federation::Ship`'s Core.

Add the following attributes to the Borg's vessels:

```
int _shield;           // set to 100 upon construction
int _weaponFrequency;  // provided upon construction
short _repair;         // can be provided. if not, set to 3 upon construction
```

As well as the following getters and setters:

```
int getShield();
void setShield(int shield);
int getWeaponFrequency();
void setWeaponFrequency(int frequency);
short getRepair();
void setRepair(short repair);
```

The following call to the `Borg::Ship`'s constructor must be valid:

```
Ship(int weaponFrequency = 20, short repair = 3);
```

Add the following member functions to the Borg's Ship class:

```
// reduces the `target`'s `_shield` attribute by `_weaponFrequency`
void fire(Federation::Starfleet::Ship *target);

// makes the `target`'s `QuantumReactor` unstable
void fire(Federation::Ship *target);

// reduces `_repair` by 1 (if `_repair` > 0), resets `_shield` to 100
void repair();
```

The `Borg::Ship`'s fire functions must print the following:

```
Firing on target with [WEAPONFREQUENCY]GW frequency.
```



Once again, replace `[WEAPONFREQUENCY]` with the appropriate value.

The `repair` function prints the following, if repair is possible:

```
Begin shield re-initialisation... Done. Awaiting further instructions.
```

If not, it prints:

```
Energy cells depleted, shield weakening.
```



By now, you don't need us to provide a `main` function to test your code, do you?



## EXERCISE 4 - COMMANDERS

Turn in: `Admiral.hpp/cpp`, `BorgQueen.hpp/cpp`

Now that your fleets can move around and shoot at stuff, you need some way to command them. Two classes are required to meet this requirement.

First, an `Admiral` class, belonging to the `Starfleet` namespace (don't forget this namespace is nested in another `Federation` namespace).

This class must have the following private attribute:

```
std::string _name; // provided upon construction
```

Upon construction, the `Admiral` displays:

```
Admiral [NAME] ready for action.
```

The class must hold two public method pointers:

- `movePtr`: points to the `move(Destination)` method of the `Ship` class from the `Federation::Starfleet` namespace
- `firePtr`: points to the `fire(Borg::Ship *)` method of the same class

Add the two following member functions to the `Admiral` class:

```
void fire(Federation::Starfleet::Ship *ship, Borg::Ship *target);  
bool move(Federation::Starfleet::Ship *ship, Destination dest);
```

When called, the `fire` method prints the following message:

```
On order from Admiral [NAME]:
```

This should be displayed before calling the `fire` function of the `Ship`.



You must not directly call the `move` or `fire` methods of `Ship`.

Create the `BorgQueen` class (within the `Borg` namespace), holding 3 method pointers:

- `movePtr`: points to the `move(Destination)` method of the `Borg::Ship` class
- `firePtr`: points to the `fire(Federation::Starfleet::Ship *)` method of the same class
- `destroyPtr`: points to the `fire(Federation::Ship *)` method of the same class

Add the following member functions, which will use the method pointers described above:

```
bool move(Borg::Ship *ship, Destination dest);  
void fire(Borg::Ship *ship, Federation::Starfleet::Ship *target);  
void destroy(Borg::Ship *ship, Federation::Ship *target);
```

Each method pointer will be initialized in the classes' constructors.



## EXERCISE 5 - EXAM

Turn in: Exam.hpp, Exam.cpp

Create an Exam class that makes this code compile:

```
int main()
{
    Exam e = Exam(&Exam::cheat);

    e.kobayashiMaru = &Exam::start;
    (e.*e.kobayashiMaru)(3);
    Exam::cheat = true;
    if (e.isCheating())
        (e.*e.kobayashiMaru)(4);
}
```

and output the following:

```
Terminal
~/B-PDG-300> g++ -W -Wall -Werror -Wextra Exam.cpp main.cpp
~/B-PDG-300> ./a.out | cat -e
[The exam is starting]$
3 Klingon vessels appeared out of nowhere.$
they are fully armed and shielded$
This exam is hard... you lost again.$
[The exam is starting]$
4 Klingon vessels appeared out of nowhere.$
they are fully armed and shielded$
What the... someone changed the parameters of the exam ! $
```