**C++ To C**

**Part 1: Encapsulation**

* These 2 days are **game-changer** in the way you will understand C++.
* Historical point: Back in the days, there wasn’t a C++ compiler, but a translator to C (called **cfront**).
* We will write the same product as this translator would provide.

**Name mangling**

* What is it?
* Ex1:
  + Create **main.c** file, declaring **int func(int a, int b, const char\* c)**;
  + Call **func** from **main()**, with **func(1, 2, “hello”);**
  + Compile it: **gcc main.c**
  + What is the error you get?
  + Now compile it with **g++ main.c**
  + What is the difference in the error you get?
* Why do we need it? (**C++ allows multiple declarations with the same name)**
* Who needs it? **(the linker! Different compilation units, same symbol?)**
* Mangling is hard **(Math\_power vs. Math::power, etc.)**
* NO STANDARD FOR NAME MANGLING!

**Function and Operator Overloading**

* The compiler mangles an overloaded function/operator with a unique name, so the linker can distinguish one from another.
* Costs us another function call, while built-in operator might be equivalent to as little as single machine instruction.

**References**

* What are they? **(an alias to another object/variable)**
* The compiler can decide on how it is implementing it.
* Mostly: In the same block (local reference) – the compiler will replace it with the original. When passed to functions, implemented as const pointer.
* Ex2: How can we tell its size? **(sizeof will give us the size of the variable it refers to… We can use a struct)**

**Inline Functions**

* Calling a function has an overhead. What is it?
* **Jumping** to another memory address, **allocating memory** for local and temporary variables, returning the **result**, **remembering where we stopped and where should we get back to**.
* Inline: Embeds the result of the function without actually creating and calling it. It is a **suggestion!**
* What is the syntax?

**Classes and Data Member**

* What is Encapsulation? **(binding the data and the functions controlling it, so we keep both of them safe from external misuse).**
* What is Data Abstraction? **(providing only the necessary details, hiding the implementation)**
* What’s the difference between a class and a struct? Size? Padding?
* SHOW AN EXAMPLE.
* How are the access qualifiers implemented (public, protected, private)?
* THE DIFFERENCE IS ONLY IN THE COMPILATION LEVEL!

**Classes and Member Functions**

* Are member functions part of the class/object? How is it implemented? **(like any other function – an address in the memory, text segment)**
* Difference between a class and an object?
* this

**Static Variables**

* static data members are not a part of the object and they act like C global variables (other files can approach them, only under the class namespace). They are stored once and initialized in the \*.cpp.
* SHOW EXAMPLE – STATIC MEMBERS
* const static variables are basically literals, no memory space (unlike in C). Can be initialized in the class in the header.
* static member functions are not a part of a class. Do they have **this**? **(NO!)** But, like static data members, they are restricted to the scope and the access of the class.

**Global and Static Objects**

* Static and global variables are initialized in compilation time in C and are written before main() takes place. In C++, it means that code can be executed before main!

**Constructors**

* What do constructors do?
* **They DO NOT allocate memory! They initialize the object (“new” is a different story – it allocates and then calling the CTOR)**.
* **MIL** is done by the order of definition, not the order of initialization we wrote. First, all the base classes are being initialized, then our object, and then the actual body of the CTOR is done.
* **Has a this!**

**Destructors**

* Again – do not free any memory! Destructing the object using **this**.

**Automatically Provided Functions**

* If are not defined explicitly, the compiler will plant them (probably as inline so not real function will exist in symbol table etc.):
  + Default CTOR (if not CTOR at all was defined)
  + Copy CTOR (if the programmer did not write one). Called explicitly (**SomeClass ob2(ob1), SomeClass obj3 =** ob2) or implicitly (in a function call **void foo(SomeClass sc)**)
  + DTOR (if not instance of the class is created - not provided, also for CTORs)
  + Assignment operator

**FIRST EXERCISE**

**Part 2: Inheritance**

* Two simple classes, base and derived. What is the memory layout of the derived class? **(it contains the base class’s data member and the addition of its own ones)**
* How would you implement such a thing in C? **(using structs!)**

**Member functions**

* A Base’s member function expects a Base\* as the provided **this**. But if we call it from the Derived class, how does it happen? **(simple C casting! Pointers are addresses. The compiler will think that he is handling a Base class in terms of memory layout, but that doesn’t matter to us)**

**CTOR/DTOR**

* As we saw before, CTOR is calling the base’s CTOR, then initializes the object’s member, then executing the CTOR body.
* DTOR works exactly the opposite.
* The Base’s CTOR can be called explicitly (MIL) and if not, the default one will be called. If the Base does not have a default CTOR – compilation error.

**Auto-generated Copy CTOR or Operator=**

* **IF WE DEFINE OUR OWN COPY CTOR**, AND FORGET TO CALL THE BASE ONE EXCPLICITLY**, THE COMPILER WILL NOT CALL IT FOR US**, BUT CALL THE DEFAULT CTOR. WE WILL HAVE AN UNINITIALIZED BASE!
* **SAME THING WITH ASSIGNMENT OPERATOR – WITHOUT OUR EXPLICIT CALL, THE OLD VALUES WILL BE ASSIGNED.**

**Multiple Inheritance**

* The Dimond problem (without virtual inheritance, calling a Base’s method will turn to be ambiguate unless class namespace is used or the derived implements this function. Also, pointers to the Bases will be different but comparing them using **==** will be true (pointing the same object!)
* Virtual inheritance.

**Empty classes**

* In C empty struct = 0
* In C++ empty class = 1 (every entity must have a unique address in the memory).
* **Empty Class Optimization** is when a class inherits from an empty class, the size 1 (+ a possible padding) **will not take place** because it is not necessary.

**Default Parameters**

* How is it implemented?
* The compiler plants the value in the call to the function.

**SECOND EXERCISE / HOMEWORK**