# Introduction to Programming – Course Notes

- **Data Type**
    - Set of values and operations on these values
    - Region of computer memory, set of operations and set of constraints
- **Programs Criteria**
    - Correctness, Speed, Size, Reliability, Power and Usability
- **Struct: Named group of data entities of different size (type).**
    - Each element of the group is called member(field).
        - Each member has a name and type!
- **Three kinds of object behavior:**
    - Command, Query and Creation
- **Access Identifiers:**
    - Public: all can read write
    - Private: only class routines can read write
    - Protected: only descendants can read write + classes of the same Java package
    - Default: only classes of the same Java package.
- **Class A conforms to B if there is at least one path in the inheritance graph between A and B**
- **Member adaptation while inheriting**
    - Change inherited member name. (Eiffel)
    - Change inherited member access specifier (Eiffel, C++, Java)
    - Change inherited member signature (Eiffel)
    - Change inherited member function body (implementation). Eiffel, Java, C++, C#
    - Change inherited routine pre and post conditions. Will be discussed at next lecture Eiffel
    - Resolve conflicting versions with multiple inheritance Eiffel, C++
- **Exception** is an unplanned event that occurs while a program is executing and disrupts the flow of its instructions.
- **Structured GoTo** : break, continue, return, raise/throw
- **Software verification:** Assure that software satisfies the expected requirements
    - Static verification (source code level)
        - Formal verification: proving that a program satisfies a formal specification
            - Deductive verification
            - Abstract interpretation
            - Automated theorem proving
            - Type systems
            - Lightweight formal methods
    - Dynamic verification (run the program)

- **Assertions in Eiffel**
    - Class objects are consistent when they have invariants
        - Invariant: set of conditions which must be true at runtime
        - Class attributes may have invariants, too
    - Class routines have
        - Require: set of pre-conditions that must be true for the routine to work. If they are violated then the problem is in the point of the call (ex. invalid arguments)
        - Ensure: set of post-conditions. If they are true then the routine did its job correctly. If they are violated then the problem is in the routine body
    - Loops have invariants
        - Set of conditions that must be true for the loop to keep running
        - Loop variants: non-negative descending function. It ensures that the loop will meet one of the invariants and terminates
        - In java: for(initialization; invariant; variant){}
    - Every assertion has a tag that will be printed at runtime if the assertion is violated
        - Routine name, source code position, call stack trace will be also printed.
- **Assertions in Java:**
    - Used to check impossible situations. For testing purposes
    - Violating them throws AssertionError
    - They don't replace error messages and shouldn't be used to check user errors or command line argument;
    - assert BooleanExpression;
    - assert BooleanExpression : String
- **How to implement same logic on different data types without rewriting the code?**
    - **C++ approach**
        - Universal Type: any type can be converted to void* while passing
        - No type checking
    - **Java approach**
        - Common base type: any type extends class Object
        - Type checking at runtime T -> Object (boxing), Object -> T (unboxing)
- **Generic programming** is a powerful mechanism of code reuse. It raises the level of abstraction and allows to do type checks at compile time keeping software quality up.
- **There are generic classes and generic functions.**
- **Type parametrized genericity can be unbounded or bounded.**
    - **Unbounded:** any variable from any type could be passed as a parameter and compiler will not complain
    - **Bounded (in java):** only particular variables of some classes could be used based on their place in the inheritance graph.
    - **Bounded (in C#):** only particular classes could be passed. They should implement a specific (one or more) interfaces or inherit from (only one) specific class.
- **Types passed could be built-in or user defined.**
    - User defined types should support all operators used in the generic function/class

- **Generics in Java**
  - Generic parameter could be a Type/enum Type
  - Syntax:

```
// Declaration
class name <T, G> {
   G obj;
   void name1 (T par) {…}
   G name2 (int par) {return obj;}
}

// Usage
name<Integer, String> obj = new name<>();
```

```
// Wildcards

// works for Integer and all its children
class name <? extends Integer> {…}

// works for String and all its super classes
(ex. Object)
class name2 <? super String> {…}
```

```
// Declaration
public static <T> void name (T par){…}
// Usage
name(5); name("5");
```

- **Generics in C++**
  - **Templates**

```
// Declaration
template < typename T > void name ( T par ) {…}
template <typename T> class name2 {…}
// Usage
name(5); name("5");
name2 obj;
```

  - C++ doesn't support bounded (constrained) genericity while java does.
  - C++ supports constant parametrization, while java doesn't.

- **Translation:** compilation vs. interpretation
- **Compilation:** take the whole program and generate machine code, optionally optimize it.
  - **Two kinds of compilation:** independent and separate
  - **Independent compilation:** program units can be compiled without information about any other program units.

  - **Separate compilation** relies on the concept of Compilation unit interface.
  - It is the process by which different portions of a computer program are compiled separately from each other.
    - Interface: set of methods names + signatures available for usage and/or inheritance
    - Routine interface is its name + signature.
  - **Separate compilation** implies static type checking for all compilation units of the system done separately from each other but taking dependencies into account
  - **There are 3 approaches to separate compilation:**
    - Programming language requires explicitly specifying import (Ada, Modula-2)
    - Concepts of package and import are part of the programming language (Java)
    - Packages and imports are not part of the programming language, but a build language (Eiffel)
- **Interpretation:** take the program instructions one by one and execute them or take the whole program and generate code for the virtual machine and then execute it.
- **JIT –Just in Time compilation:** Applied for the already generated virtual machine code to generate native code when program is started for execution.
- **AOT –Ahead of time compilation:** Compile virtual machine code into native before program started (while downloading or installing)
- **Hybrid execution (mixed) mode:** Program (executable) may contain native and virtual machine code
- **Module/class:** named group of types, routines, variables, constants and initialization. It has interface and implementation.
- **Class forms in Eiffel:**
  - Short – Flat – Short flat
- **Software reuse:** Reuse routine/function, class/module, type
- **Feature adaptation** while inheriting (Eiffel) is a kind of extended form of reuse.
- **Library**
  - A group of object files put together under a proper name.
  - A group of compiled classes/functions. Typically these classes/functions are properly tested.
  - Can be statically or dynamically linked.
- **Global data**
  - **Java**: static
  - **Eiffel:** Once functions
  - **Ada/Modula-2:** Modules
- **File** is a computer resource for recording data discretely in a computer storage device.
  - File is a class in OOP
- **File properties:** Name, Size, Type, Attributes
- **File operations**: Create, Open, ReadData, WriteData, Close, Delete, Rename, Move, Copy, Change file attributes
- **File format:** binary, text, XML, binary XML, JSON, database,

- **Persistency:** The property of data/state surviving while its creator is already destroyed (ex. Data is stored in a file)
    - **2 basic operations:** store/write object into file and read/restore object from file.
        - Object object = File.readObject();
        - File.writeObject (object);
- **Serialization** is a mechanism of converting the state of an object into a byte stream which can then be saved to a database/file/memory or transferred over a network. **Serialization implements persistence**
    - To achieve serialization in java, a class need to implement the interface Serializable
    - The byte stream is platform independent. An object serialized on one platform can be deserialized on a different platform.
- **Deserialization** is the reverse process.
- **Object-Relational Mapping (ORM):** Allows to store and retrieve Java objects in the relation database.
- **Java persistence API (JPI):** particular set of functions to support ORM
- **Name clash problem:**
    - Two classes in different packages may have the same name
        - Java way: use full name: packageName.className
        - Eiffel way: rename one for particular program
    - Several versions the same class in different packages
        - Java way: use full name: packageName.className
        - Eiffel way: Hide non-actual ones for the particular problem.
- **Serial Computing:**
    - A problem is broken into a discrete series of instructions
    - Instructions are executed sequentially one after another
    - Executed on a single processor
    - Only 1 instruction may be executed at any moment in time
- **Parallel Computing:**
    - Simultaneous use of multiple computer resources to solve a computational problem
    - A problem is broken into discrete parts that can be solved concurrently
    - Each part is further broken down to a series of instructions
    - Instructions from each part execute simultaneously on different processors
    - An overall control/coordination mechanism is employed
- **Concurrency** is when two tasks can start, run, and complete in overlapping time periods.
- **Parallelism** is when tasks literally run at the same time, (eg. on a multi-core processor).
- **Flynn's taxonomy is a classification of computer architectures**
    - **SISD:**
        - A sequential computer which exploits no parallelism in either the instruction or data streams.
    - **SIMD:**
        - A single instruction operates on multiple different data streams. Instructions can be executed sequentially, such as by pipelining, or in parallel by multiple functional units.

- **MISD:**
    - Multiple instructions operate on one data stream. This is an **uncommon** architecture which is generally used for fault tolerance.
- **MIMD:**
    - Multiple autonomous processors simultaneously executing different instructions on different data. **MIMD architectures include multi-core superscalar processors, and distributed systems**, using either one shared memory space or a distributed memory space.

- **Amdahl's Law:**
  - Execution time after improvement = Execution time unaffected + (Execution time affected / Amount of improvement)

- **Shared Memory:**
  - When you have several processors accessing the same memory and it's their only way of communication – accessing memory is simple
- **Distributed Memory:**
  - When you have several processors, each one has his own local memory, all processors are connected to an Interconnection network, which they access via I/O interface.
  - Accessing the processor's own memory is simple, while accessing a remote memory is expensive – you have to access the Interconnection network to load/store
- **Hybrid Distributed-Shared Memory:**
  - The largest and fastest computers in the world today employ both shared and distributed memory architectures.
  - The shared memory component can be a shared memory machine and/or graphics processing units
- **Parallel Programming Models:**
  - **Shared Memory (without Threads) - Simplest**
    - processes/tasks share a common address space, which they read and write to asynchronously.
    - An advantage of this model is that the notion of data "ownership" is lacking
    - An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality:
  - **Threads:**
    - This model is a shared memory programming with a single "heavy weight" process which can have multiple "light weight" with concurrent execution paths.
  - **Distributed Memory / Message Passing**
    - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.

    - **Data Parallel Model**
        - Tasks are assigned to processes and each task performs similar types of operations on different data
    - **Hybrid**
        - A hybrid model combines several other programming models.
    - **Single Program Multiple Data**
        - Multiple autonomous processors simultaneously executing the same program
    - **Multiple Program Multiple Data**
        - Multiple autonomous processors simultaneously operating at least 2 independent programs.
- **Deadlock is a state in which each member of a group is waiting for another member, including itself, to take action, such as sending a message or more commonly releasing a lock.**
- **Race conditions arise in software when an application depends on the sequence or timing of processes or threads for it to operate properly.**
- **Functional programming** treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.
    - Expressions or declarations instead of statements.
- **Pure functions:** functions that will always return the same values given the same arguments. They don't change global data nor depends on any state. (have no side-effects)
- **Functional programming languages:**
    - Common Lisp, Scheme, Clojure, Wolfram Language, Racket, Erlang, OCaml, Haskell, F#...
- **Imperative programming languages were extended to support functional style:**
    - Perl, PHP, C++11, Kotlin, Java, Eiffel, Scala.
- A programming language is said to have **first class functions** if it supports passing functions as arguments to other functions
- **Higher order functions** are functions that can either take other functions as arguments or return them as results. They enable partial application/Currying
    - **Java way –** Lambda expressions are passed as arguments and returned. They have to implement a functional interface (interface having only one method)
    - **Anonymous interface** can have state while Lambda expressions cannot
    - **Eiffel** has **'agents'** as a mechanism for functional programming.

| Imperative programming | Declarative(functional) programming |
|---|---|
| State(data) and code | No state |
| Routine – action or query | Functional as transformation of elements of one set into the other |
| Routine – procedure or function | High order functions |
| Side effects – change in global data, in class attributes | No side effects |
| Aligned with HW architecture | Aligned with mathematical approaches |
| Statements | Expressions and declarations |
| Mutable variables | Immutable variables |
| Loops | Recursion |

- **C++ Memory model:**
  - Program (Machine instruction), Read-only
  - Stack: Stores local objects.
  - Heap: dynamically allocated memory for objects.
- **Type:** Set of values, operators and relationships between this type and others.
  - Fundamental/Structured. Predefined/User-defined.
  - Modified types: constant, pointer, reference, functions, arrays.
  - (int, double, char) type modifiers: signed, unsigned, long, short.
- **Sizeof some types:**
  - Bool, char - 1 byte, Short - 2 bytes, Float, long, int (signed/unsigned) – 4 bytes.
  - Double, long double, long long – 8 bytes.
- **Translation process:** Compilation of separate codes into an object file, Link all object files.
  - **.**h files contain interface, .c or .cpp files contains implementation.
- **Object declaration syntax:** StorageClassSpecifier Type Name Initializer ;
  - **Can swap** StorageClassSpecifier and Type. (i. e. int static x =5)
  - **StorageClassSpecifiers: auto(old), register, extern, static, mutable, thread_local.**
    - global static object are available only within the TU where they were declared. They are declared once, before the program starts.
    - extern objects should be defined elsewhere and can have initial value.
- **E1[E2] = *( (E1) + (E2) ), therefore x[5] is the same as 5[x].**
- **Rules on references:**
  - No pointers to references, No arrays of references, No references to references. No “constant” references.
- **Forward declarations:** declaring methods classes without specifying implementation
- **Copy constructor** is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:
  - `ClassName (const ClassName &old_obj);`
- **Passing by reference/value:**
  - Passing by reference doesn’t invoke copy constructors.
  - Working with templates, we pass a reference instead of the value to avoid implicit compiler type conversations.
  - If we pass arrays by value, they get converted to pointers. By reference, they didn’t.
- **Class access specifiers:**
  - Public members from the base class are accessible
  - Protected members of the base class are accessible from within its derived classes only.
  - Private members of the class are accessible only within the class itself
- **Operators that can’t be overloaded: ?: | . | .* | :: | sizeof**
- **Template argument can be one of the following:**
  - A value that has an integral type or enumeration
    - Integral types: bool, char, int (with modifiers).
  - A pointer or reference to a class object
  - A pointer or reference to a function
  - A pointer or reference to a class member function or std::nullptr_t
- **How to call a template that takes no arguments?**
  - Using explicit instantiation. Call the function with the type in angle brackets
- **Template instantiation can be:**
  - Complete explicit instantiation: all types are given explicitly.
  - Incomplete explicit instantiation: some types are given explicitly, others are deduced.
  - Implicit instantiation: all types are deduced.

- **Explicit template specialization:** When we need a special function in case the parameter is of some specific type, we write template<> then the generic function replacing T with the type we need.
- **Partial template specialization: class C<???>**
  - <const T> // constant types
  - <T*> // pointer types
  - <T&> // reference types
  - <T[int-const]> // arrays
  - <type(*)(T)>//pointers to functions with parameter(s) of type T
  - <T(*)()> // pointers to functions returning type T
  - <T(*)(T)> // pointers to functions with parameter(s)of type T and returning type T
  - <(T)> // represents lists where at least one type contains T;
  - <()> // represents lists where no type contains T.
- **Making functions virtual:** Now functions in derived classes don't hide the ones from the base class, but override them.
  - The interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type) type), whereas the interpretation of a call of a non virtual member function depends only on the type of the pointer or reference denoting that object (the static type).
- The class is **abstract** if it has at least one pure virtual function (doesn't have implementation).
- **Abstract** class can contain: abstract specification of behavior, non abstract functionality, object state.

```
// Pointers to constants and constant pointers.
const int *t1; int const *t2; int *const t3;
// t1, t2 are pointers to const. t3 is a const pointer
int x = 5; int &*p = x; // Pointer to reference, Invalid.
int x = 5; int& r = x+1; // Invalid because when declaring references, right value should always be a constant expression.
int x = 5; int *&r = &x; // Reference to a pointer, Invalid for the same reason. How to fix it?
    ▪ Either declare int* p = &x and use p instead of &x
    ▪ Or declare r to be a reference to a constant pointer. int * const &r = &x;
int(*(a[10]))(int); // declares an array named 'a' of 10 elements, each element is of type pointer to a function that takes one int parameter and returns int.
typedef int(*PtrFun)(int);//Now we can use PtrFun like a class.
using PtrFun= int(*)(int); // Same
auto il = { 10, 20, 30 }; // Type is std::initialize_list
template < typename T > int spaceOf (){
    int bytes = sizeof (T);
    return bytes/4 + bytes%4>0; }
cout<< spaceOf<int>();
```

```
//template function outside the class
template< typename T > void C<T>::f() {..}
```

```
// Using C++20 concepts:
template<typename T> concpt G = requires(T x, T y){{x>y} -> bool};
template<typename T> requires G <T> T Max(T a, T b)
return a>b ? a : b;
```

```
// Lambda Expression Syntax:
  [capture clause] (parameters) -> return-type { Body }
// Return type is optional, compiler can deduce it in simple cases
        Where there is only one return statement in the body.
// Syntax used for capturing variables:
    []: can access only variables that are declared inside the scopes.
    [&]: capture all external variable by reference
    [=]: capture all external variable by value
    [=a, &b]: capture 'a' by value and 'b' by reference, equal sign can be omitted.
// The type of [](int x){return x+1;} is Function<int(int)>
```

- **Ctor-initializer:**
  - Specifies which base class constructor to invoke for the sub-object of derived class.
- **Mem-initializer:**
  - Initializer data members before calling the constructor.
- **Delegating ctor:**
  - A constructor that invokes another constructor before calling itself.
- **this pointer**
  - A constant pointer to an instance of the class.
  - Passed as the first hidden argument for any non-static member function.
  - For constant functions, it's a constant pointer to a constant instance of the class.
    - This allows overloading a function with the same name and signature if one of them is const, the difference will be in **this** first hidden argument.
    - Constant functions are safe and more generic
      - Safe because they can't modify object state (has no side effects).
      - More generic because they can be applied to both constant and non-constant class instances.
- **Stack frame:**
  - All the information needed for the function execution, stored in adjacent blocks of memory. Stack pointer points to the start of the latest frame.
    - Information to restart the execution at the end of the call.
    - Return address by code
    - Return value (if any).
    - Function arguments passed to the function in the call and local variables (if any).
- **Exception aspects:**
  - To break the "normal" flow of program control.
  - To transfer the control to some other program point.
    - All stack frames of dynamically enclosing scopes are popped until a special stack frame "the try block" is found.
  - An object that is passed together with transferring the control,
    - It contains information about what happened.
- **Program address space:**
  - Stack: local variables inside functions, grows downward
  - Heap: space requested for dynamic data; resizes dynamically, grows upward
  - Static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - Code: loaded when program starts, does not change

- **Scopes and blocks:**
  - Scope is a rule determining existence and visibility of variables.
  - Block is a compound language construct where variables (and other program entities) are declared.
- **void* malloc (int size) {}**
  - Allocates ‘size’ bytes of uninitialized storage in the heap.
  - On success, returns the pointer to the beginning of newly allocated memory. To avoid a memory leak, the returned pointer must be deallocated with free() or realloc().
  - On failure, returns a null pointer.
  - Available in <stdlib.h>
- **void free(void* ptr) {}**
  - Deallocate the previously allocated memory by malloc or similar functions.
- Bit-fields are members of a structure that have a specified limit on the number of bits the variable of that type can take, they are useful for memory management.
- Sizeof struct is not necessarily equal to $\Sigma$ of all sizes of its members
- Sizeof union is equal to the Max member size.