|  |
| --- |
| 3 parts a (non-trivial) C++ program is typically organized into: interface, implementation, client/application. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | What items commonly go into each part (and what don't logically belong). |

|  |  |  |
| --- | --- | --- |
|  |  | User-defined data type development using C++ in *object-based* fashion (use of **class** in particular). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Information hiding*: separating the interface (*what*) from the implementation (*how*). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Header and source files and separate compilation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Key benefit: ***maintainability*** (modifiability, updatability, upgradability, *etc*.). |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Interface has lower tendency to change (with time) than implementation. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Modular programming*: building application with components that are organized as modules (interface-implementation pairs). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Header-source file pairs and separate compilation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Key benefit (of *modularity* in component development): ***reusability***. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Indigeneous* versus *exogeneous* data type. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **ourStr** Version 1 vs **ourStr** Version 2. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **IntSet** of Assignment 1 vs **IntSet** of Assignment 2. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | The "Big Four". |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Which is automatically invoked when. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Automatic (compiler-supplied) versions, their shortcomings, and impacts of the shortcomings. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | In particular: *deep copying* versus *shallow copying*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Which of the four, and under what circumstace, will the compiler not supply an automatic version when no custom version is provided. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | When and how to write custom (overriding) ones. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Operators. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Which ones "come-for-free" (and what shortcomings if any). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | How to use *operator overloading* to override compiler-supplied assignment operator and to support additional operators. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Some key C++ (especially **class**-related) features, the roles they play and how to use them (both as a client/user and a developer/supplier). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **include** directive (macro guard or inclusion guard). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Class-level* variable versus *instance-level* variable. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Data member whose declaration has static appearing as a decoration versus data member that doesn't have such decoration. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | The *invoking object* and the **this** pointer. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Accessors* and *mutators*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Non-public *helpers*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Member* functions (methods), *non-member* (free, ordinary, regular) functions, and **friend** functions. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *How-to*'s associated with each within the *interface*, *implementation* and *client*/*application* components. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Initializer*(or *initialization*)*list*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Function with *default arguments*. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Its particular usefulness in covering constructors -> "multi-role" constructor. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Condition under which a constructor with arguments will also cover the *default constructor*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **const** keyword: 3 of the usage contexts (situations) of particular interest to us and the semantic (meaning) of each. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Declaring symbolic constant. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Specifying pass-by-**const**-reference. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Specifying accessor (protecting invoking object). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Some specific items prone to misconceptions and/or oversight. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Whose data can an *accessor* not cause any side effect on. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Whose **private** data can a *member function* (method) directly access. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Under which circumstance will C++ do cloning via *copy construction* and which via *copy assignment*, and ... |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | When does "**=**" mean assignment and when does it not (but mean what instead). |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | Under which 3 contexts (of interest to a programmer) will C++ call the *copy constructor*. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | Defining a new object of a class, initializing it to an existing object (of the same class). |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | Passing an object of a class *by value* to a function. (The reason why *the argument of copy constructor cannot be passed by value*.) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | Returning an object of a class *by value* from a function. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Object lifetime: construction -> utilization -> destruction. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Who is/are the role player(s) in which of the phases. |

|  |  |  |
| --- | --- | --- |
|  |  | *Contract-based* design technique. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | What the *3 key ingredients* and the *3 key action verbs* are. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | preconditions, postconditions, class invariant |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | expect, guarantee, maintain |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | (Less formally: receive *benefits*, fulfill *obligations*, abide by *rules*.) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | How the key ingredients interrelate. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | How the key action verbs apply (relative to the key ingredients). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Who are the stakeholders (*i.e.*, contract between which parties). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | How the functions of a **class** are bound ("contractually") by invariants. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Constructors <--> invariants. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Destructors <--> invariants. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Other mutators <--> invariants. |

|  |  |  |
| --- | --- | --- |
|  |  | Good practices. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Program defensively. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Principle of *least privilege*. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Reveal/enable only what is necessary, nothing more. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *E.g.*: don't use global variables. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *E.g.*: use appropriate parameter passing mechanism (by value, reference and **const** reference) in functions. (Note: This in part motivated our discussion of C++'s intents in supporting the 3 parameter passing mechanisms.) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Use **friend** functions sparingly and only where warranted. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Where possible, trap/address precondition violations and other "surprise" situations. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Client-oriented design. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *E.g.*: specify position rather than index. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *E.g.*: don't baffle client with implementation details. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | "Don't make soup too salty". |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *E.g.*: not good to have **using** directive in header file. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | "Avoid doing the exact same thing multiple times". |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *E.g.*: not good to call a function over and over (such as in a loop) if the return value of each such call is unchanging. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Don't "throw away old TV before new TV is in hand". |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *E.g.*: when resizing a dynamic array, don't free up "old" array before the "new" array is in place. |

|  |  |  |
| --- | --- | --- |
|  |  | Computer-based problem solving concepts. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Central role of data -> storage and access techniques for data -> *data structure*s and *algorithm*s. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Classic ("tried-and-proven-to-be-commonly-useful") containers. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | At "lower-level": array, linked list, etc. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | At "higher-level": stack, queue, etc. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Abstraction*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What it is and how it helps us in solving problems. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Underpinnings of just about all aspects of CS -> underlies many concepts/paradigms/methodologies/. . . : (examples below are not in any particular order and may overlap one another) - *top-down* progressive refinement (specification/specialization) - *bottom-up* generalization - *dividing-and-conquering* - *separating concerns* - information hiding (encapsulation) - modularization (black-boxing) - parameterization - . . . |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Helps us in *managing complexities*. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Abstract data types* (ADTs). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Focusing on the interface (*what*), abstrating away (ignoring) the implementation (*how*). |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *One-to-many* (in general) relationship between interface and implementation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | C++ support: header and source files and separate compilation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Container ADTs. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | What they are and what the "3 categories of operations (besides construction and destruction) typically supported by each" are. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | (accessors, mutators, helpers) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Starter ADTs: set, multiset (bag), sequence (list) and sorted sequence (sorted list). |

|  |  |  |
| --- | --- | --- |
|  |  | Specifying/implementing/utilizing container ADTs in C++ - some aspects: |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Iterators*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What are and why. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | (enabling user to systematically step (iterate) through container without knowing internal details of container's implementation) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Enhanced scoping (name-conflict prevention) using custom *namespace*. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Symbolic representation of data type using **typedef**. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Use of **size\_t** for sizing/counting/iterating. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Pitfalls of using size\_t. |

|  |  |  |
| --- | --- | --- |
|  |  | Templating. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | What it essentially enables one to do. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What must hold true to make it possible. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | (parameterizing type) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Template *function* and template *class*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Provision *data types* must meet for successful use with templates. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Aspects of STL. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What the *3 key components* are. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *containers* (template classes), *iterators*, *algorithms*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What *left-inclusive metaphor* is. |

|  |  |  |
| --- | --- | --- |
|  |  | Introduction to algorithm analysis (include only aspects that have been covered). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Nature-of-input-dependent scenarios: *worse case*, *average case* and *best case*. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Big-O characterization and notation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Upper bound (order on resource requirement growth rate), asymptotic ("in-the-big", "settled-down"), order of magnitude ("broad-brushing"). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What it can (is intended to) capture (and what is not captured): implications and common misconceptions. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | ~~Rigorous example characterization/analysis involving O(1), O(n) and O(n~~~~2~~~~) algorithms for solving an example problem.~~ |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | Others |

|  |  |
| --- | --- |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  | Reference material. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Lecture Notes**: those prefixed 301 through 311. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Examples** posted so far (up to and including the one posted on 09/26/2018). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Class handouts - most of them were derived from notes also appearing under **Lecture Notes** and/or **Examples**. |

|  |  |  |
| --- | --- | --- |
|  |  | You may want to check out ***sample past test/exam questions*** already posted on the class homepage. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | You should ***not*** however, expect the questions to be identical in number, kind, topic coverage, *etc.* |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *You should not have to worry about questions being written on topics we have not yet covered; some such questions may appear as sample past questions because the associated topics were appropriate at that time.* |