ABSTRACT DATA TYPES (ADT)

RECALL

DATA TYPES

Recall: A type is a name construct that specify a set of values and a set of allowable operations that can be carried out on them. A data type is a classification identifying one of various types of data, such as real-valued, integer (or whole numbers) or Boolean (true or false values), dates.

- Determines the possible values for that type
- The operations that can be done on values of that type
- The meaning of the data.
- The way values of that type can be stored.

TYPE CLASSES

i) **Primitive types**

Primitive data types are predefined types of data, which are supported by the programming language (basic or built in types). For example numeric types, character and string and Boolean types are all primitive data types. Programmers can use these data types when creating variables in their programs.

ii) **Composite types**

Types derived from more than one primitive type. This can be done in a number of ways. The ways they are combined are called data structures. Composing a primitive type into a compound type generally results in a new type, common composite types are:

- ***Arrays*** : stores a number of elements of the same type in a specific order accessed using an integer index. May be fixed-length or expandable.
- ***Record*** (***tuple*** or **struct**) Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members.
- ***Union*** : A union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float and an integer; whereas, in a union, there is only one value at a time.
- **Tagged unions** (also called a variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
- ***Set*** :abstract data structure that can store certain values, without any particular order, and no repeated values. Values themselves are not retrieved from sets, rather one test a value for membership to obtain a boolean "in" or "not in".
- **Objects** : contains a number of data fields, like a record, and also a number of program code fragments for accessing or modifying them. Data structures not containing code, like those above, are called plain old data structure.

iii) **Abstract types**

Types that do not specify an implementation. For instance, a stack (which is an abstract type) can be implemented as an array (a contiguous block of memory containing multiple values), or as a linked list (a set of non-contiguous memory blocks linked by pointers).

Abstract types can be handled by code that does not know or "care" what underlying types are contained in them. Programming that is agnostic about concrete data types is called generic programming. Arrays and records can also contain underlying types, but are considered concrete because they specify how their contents or elements are laid out in memory.

Examples include: smart pointer is the abstract counterpart to a pointer, Hash or dictionary or map, queues, stacks, trees, graphs.

MODULARITY

A design technique of breaking down a program into smaller, more manageable subtasks/subsections called modules: in structural programming this is achieved through functions and procedures or blocks.

A module therefore is a contiguous section of code that can be separated from the overall cod and performs a specific task given specific resources and produces specific results.

Modularity is a technique that keeps the complexity of a large program manageable by systematically controlling the interaction of its components. You can focus on one task at a time in a modular program without other distractions.

A module is any named program unit that can be implemented as an independent entity.

A well designed module has a single purpose, and presents a narrow interface to other modules.

Modularity is mainly achieved via abstraction.

Modules include

- A single, stand-alone function
- A method of a class
- A class
- Several functions or classes working closely together

Other blocks of code

Advantages of modularization

- Makes code easy to write & read.
- Modularized code is easy to debug.
- Isolates errors and eliminates redundancies.

ABSTRACTION

A mode of thought by which we concentrate on the general ideas rather than on specific manifestations of the idea: In programming, abstraction is the distinction made between what a piece of code does and how it’s is implemented e.g. in C++ consider the distinction between a .h file (what the program does) and a .c file (how the program does it i.e. it’s implementation)

TYPES OF ABSTRACTION

There are three major types of abstraction:

i) ***Procedural / Functional Abstractions***

Concentrating on what the function does and what it requires to perform the task rather than the specific steps (the how) it undertakes to complete the task. This is achieved by: **function prototyping**, **function calls** and **parameter listings** (REFF: APP 1.).

For example, suppose that a program needs to operate on a sorted array of names the program may, for instance, need to search the array for a given name or display the names in alphabetical order. The program thus needs a function S that sorts an array of names. Although the rest of the program knows that function S will sort an array, it should not care how S accomplishes its task.

ii) ***Data Abstraction***

Asks that you think in terms of **what** you can do to a collection of data independently of **how** you do it. Data abstraction is a technique that allows you to develop each data structure in relative isolation from the rest of the solution. The other modules of the solution will "know" what operations they can perform on the data, but they should not depend on how the data is stored or how the operations are performed. Again, the terms of the contract are what and not how. Thus, data abstraction is a natural extension of functional abstraction.

iii) ***Control Abstraction (will be discussed under topic "Recursion")***

ADT – ABSTRACT DATA TYPES (A CASE OF DATA AND PROCEDURAL ABSTRACTION)

A collection of data together with a set of operations on that data are called an **abstract data type,** or **ADT.**

Terms

Information Hiding**:** Refers to the shielding of data or information from outside entities/ unauthorized access, modification or manipulation.
Encapsulation**:** Refers to wrapping up together the data and the operations for accessing, manipulating them into a single entity.

The description of an ADT's operations must be rigorous enough to specify completely their effect on the data, yet it must not specify how to store the data nor how to carry out the operations. For example, the ADT operations should not specify whether to store the data in consecutive memory locations or in disjoint memory locations. You choose a particular **data structure** when you **implement** an ADT.

Recall that a data structure is a construct that you can define within a programming language to store a collection of data. For example, arrays and structures, which are built into C++, are data structures. However, you can invent other data structures.

ADTs versus Data Structures

- An abstract data type is a collection of data and a set of operations on that data; user defined constructs.
- A data structure is a construct within a programming language that stores a collection of data.