Unit 2.2

- Abstract Data Type
- Information hiding
- Encapsulation
- Type Equivalence
- Storage Management (static, stack-based, heap-based)

Mechanisms to create new data types

- Structured data
  - -Homogeneous: arrays, lists, sets
  - -Non-homogeneous: records
- Subprograms
- Type declarations to define new types and operations
- Inheritance

Structured DataType

A data structure is a data object that contains other data objects as its elements or components.

Specification

1. Number of components
   - -Fixed size - Arrays
   - -Variable size :stacks, lists. Pointer is used to link components.
2. Type of each component
   - -Homogeneous : all components are the same type
   - -Heterogeneous : components are of different types
3. Selection mechanism to identify components : index, pointer
   - Two-step process:
     - -referencing the structure
     - -selection of a particular component
4. Maximum number of components
5. Organization of the components:
   - •simple linear sequence
   - •multidimensional structures:
     - •vector of vectors (C++)

Operations on data structures

- Component selection operations

  -Sequential
  -Random

- Insertion/deletion of components

- Whole-data structure operations

- Creation/destruction of data structures

Implementation of Data Structure

Storage representation includes:

a.storage for the components

b.optional descriptor - to contain some or all of the attributes

- Sequential representation: the data structure is stored in a single contiguous block of storage, that includes both descriptor and components. Used for fixed-size structures, homogeneous structures (arrays, character strings)

- Linked representation: the data structure is stored in several noncontiguous blocks of storage, linked together through pointers. Used for variable-size structured (trees, lists)

Stacks, queues, lists can be represented in either way. Linked representation is more flexible and ensures true variable size, however it has to be software simulated.

Implementation of operations on data structures

- Component selection in sequential representation: Base address plus offset calculation. Add component size to current location to move to next component.

- Component selection in linked representation: Move from address location to address location following the chain of pointers.

Storage management

Access paths to a structured data object - to endure access to the object for its processing. Created using a name or a pointer.

Two central problems:
- Garbage : the data object is bound but access path is destroyed.
Memory cannot be unbound.
- Dangling references : the data object is destroyed, but the access path still exists.

1.Declarations and type checking for data structures

What is to be checked:

1. Existence of a selected component
2. Type of a selected component

- **Vectors and arrays**
- **A vector -** one dimensional array
- **A matrix -** two dimensional array
- **Multidimensional arrays**
- **A slice -** a substructure in an array that is also an array, e.g. a column in a matrix.
- **Implementation of array operations:**
  - **Access -** can be implemented efficiently if the length of the components of the array is known at compilation time. The address of each selected element can be computed using an arithmetic expression.
  - **Whole array operations,** e.g. copying an array - may require much memory.
- **Associative arrays**
- Instead of using an integer index, elements are selected by a key value, that is a part of the element. Usually the elements are sorted by the key and binary search is performed to find an element in the array.

Record

A record is a data structure composed of a fixed number of components of different types.
The components may be heterogeneous, and they are named with symbolic names.

Specification of attributes of a record:

-Number of components
-Data type of each component
-Selector used to name each component.

Implementation:

-Storage: single sequential block of memory where the components are stored sequentially.

-Selection: provided the type of each component is known, the location can be computed at translation time.

Efficiency of storage representation:

Other structured data objects

- Records and arrays with structured components: a record may have a component that is an array, an array may be built out of components that are records.
- Lists and sets: lists are usually considered to represent an ordered sequence of elements,
  sets - to represent unordered collection of elements.
- Executable data objects
- In most languages, programs and data objects are separate structures (Ada, C, C++).
- Other languages however do not distinguish between programs and data - e.g. PROLOG. Data structures are considered to be a special type of program statements and all are treated in the same way.

An abstract data type is:

- A set of data objects,
- A set of abstract operations on those data objects,
- Encapsulation of the whole in such a way that the user of the data object cannot manipulate data objects of the type except by the use of operation defined.

Encapsulation is a question of language design; effective encapsulation is possible only when the language prohibits access to the information hidden within the abstraction. Some languages that provide for abstract data types: Ada: packages; C++, Java, Visual Basic: classes.

Information hiding

Information hiding is the term used for the central principal in the design of programmer-defined abstract data types.

A programming language provides support for abstraction in two ways

1. By providing a virtual computer that is simpler to use and more powerful than the actual underlying hardware computer.
2. The language provides facilities that aid the programmer to construct abstractions.

When information is encapsulated in an abstraction, it means that the user of the abstraction

a. does not need to know the hidden information in order to use the abstraction,
b. is not permitted to directly use or manipulate the hidden information even if desiring to do so.

Mechanisms that support encapsulation:

Subprograms
Type definitions

- What is a data structure?
- Which are the elements of a structured data type specification?
- What types of operations are considered when specifying a structured data type?
- Which are the basic methods for storage representations generally used in implementing structured data types.
- Discuss the memory management problems when implementing structured data types.
- What is an abstract data type? How does it differ from a structured data type?

- **Encapsulation by subprograms**
  - **Subprograms as abstract operations**

Subprograms can be viewed as abstract operations on a predefined data set. A subprogram represents a mathematical function that maps each particular set of arguments into a particular set of results.

- **Specification of a subprogram** (same as that for a primitive operation):
  - the name of the subprogram
  - the signature of the subprogram - gives the number of arguments, their order, and the data type of each, as well as the number of results, their order, and the data type of each
  - the action performed by the subprogram

- Some problems in attempting to describe precisely the function computed by a subprogram:
  - Implicit arguments in the form of nonlocal variables.
  - Implicit results (side effects) returned as changes to nonlocal variables or as changes in the subprogram's arguments.
  - Using exception handlers in case the arguments are not of the required type.
  - History sensitiveness - the results may depend on previous executions

Implementation of a subprogram:

- Uses the data structures and operations provided by the language
- Defined by the subprogram body

Local data declarations
Statements defining the actions over the data.
The body is encapsulated, its components cannot be accessed separately by the user of the subprogram. The interface with the user (the calling program) is accomplished by means of
arguments and returned results.

Type checking: similar to type checking for primitive operations.
Difference: types of operands and results are explicitly stated in the program

1.Subprogram definition and invocation

- Subprogram definitions and subprogram activations

-Subprogram definition: the set of statements constituting the body of the subprogram. It is a static property of the program, and it is the only information available during translation.

-Subprogram activation: a data structure (record) created upon invoking the subprogram. It exists while the subprogram is being executed. After execution the activation record is destroyed.

- Implementation of subprogram definition and invocation

A simple (but not efficient) approach:

Each time the subprogram is invoked, a copy of its executable statements, constants and local variables is created.

A better approach:

The executable statements and constants are invariant part of the subprogram - they do not need to be copied for each execution of the subprogram. A single copy is used for all activations of the subprogram. This copy is called **code segment**. This is the **static part**. The **activation record** contains only the parameters, results and local data. This is the **dynamic part**. It has same structure, but different values for the variables.

Also see: Generic subprograms and Subprogram definitions as data objects

Type Definitions

Type definitions are used to define new data types. Note, that they do not define a complete abstract data type, because the definitions of the operations are not included.

Format: **typedef** *definiion name*

Actually we have a substitution of **name** for the **definition.**

Examples:

typedef int key_type; key_type key1, key2;

These statements will be processed at translation time and the type of *key1* and *key2* will be set to integer.

struct rational_number {**int** numerator, denominator;}; **typedef** rational_number rational; rational r1, r2;

Here *r1* and *r2* will be of type *rational_number*

Type equivalence and data object equality issues

- Name equivalence: two data types are considered equivalent only if they have the same name.

Issues

Every object must have an assigned type, there can be no anonymous types.

A singe type definition must serve all or large parts of a program.

- **Structural equivalence:** two data types are considered equivalent if they define data objects that have the same internal components.

Issues

Do components need to be exact duplicates? Can field order be different in records? Can field sizes vary?

- **Data object equality**

We can consider two objects to be equal if each member in one object is identical to the corresponding member of the other object. However there still may be a problem. Consider for example the rational numbers 1/2 and 3/6.

Are they equal according to the above definition?

In general, the compiler has no way to know how to compare data values of user-defined type. It is the task of the programmer that has defined that particular data type to define also the operations with the objects of that type.

Type definition with parameters

Parameters allow the user to prescribe the size of data types needed eg. array sizes.

Implementation: The type definition with parameters is used as a template
as any other type definition during compilation.

- Discuss about "encapsulation" and how it can be achieved with subprograms.
- Explain about "subprogram definition" and "subprogram activation record".
- Discuss the two aspects of type equivalence: name equivalence and structural equivalence
- What is data object equality?

Inheritance: derived class, abstract class, object, message, and polymorphism.

- How is inheritance defined? What can be inherited?
- Discuss briefly different types of inheritance in programming.
- Know the terms:
  - ADT
  - Derived / Super Classes
  - Methods
  - Abstract Classes and Abstraction Concept
  - Object and Messages
  - Polymorphism