Section B: Structured data objects, Subprograms and Programmer Defined Data Type

Structured data objects & data types:

A data object that is constructed as an aggregate of other data objects, called components, is termed a structured data object or data structure. A component may be elementary or it may be another data structure(e.g., a component of an array me be a number or it may be a record, character string, or another array).

Specification & Implementation of Structured data types:

The major attributes for specifying data structures include the following :

Number of components. A data structure may be of fixed size if the number if components is invariant during its lifetime or of variable size. Variable-sized data structure types usually define operations that insert and delete components from structures.

Arrays and records are common examples of fixed-size data structure types; stacks, lists, sets, tables and files are examples of variable-size types.

Type of each components. A data structure is homogeneous if all its components are of the same type. It is heterogeneous if its components are of different types.

Ex. Arrays, sets and files are usually homogeneous, whereas records and lists are usually heterogeneous.

A data structure type need a selection mechanism for identifying individual components of the data structure.

Implementation of Operations on Data Structures

Sequential representation. Random selection of a component often involves a base-address-plus-offset calculation using an accessing formula. The relative location of the entire block is the base address. The accessing formula, given the name or subscript of the desired component (e.g., the integer subscripts of an array component), specifies how to compute the offset of the component.

Linked representation. Random selection of a component from a linked structure involves following a chain pointers from the first block of storage in the structure to the desired component. Selection of a sequence of components proceeds by selecting the first component and then following the link pointer from the current component to the next component for each subsequent selection.

Declaration and Type Checking for Data Structures

The basic concepts and concerns surrounding declarations and type checking for data structures are similar to those discussed for elementary data objects. However , structures are ordinarily more complex because there are more attributes to specify or component selection operations must be taken into account. These are two main problems:

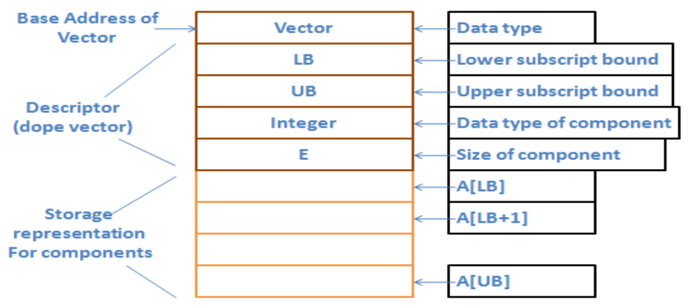
1.) Existence of a selected component. The arguments to a selection operation may be of the right types, but the component designated may not exist in the data structures.

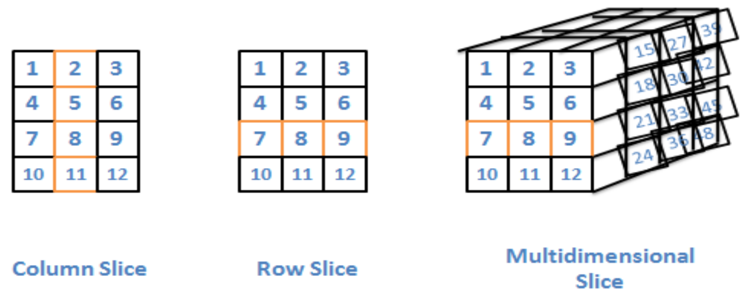
2.) Type of a selected component. A selection sequence may define a complex path through a data structure to the desired component.

Vector and Arrays

A vector is a data structure composed of a fixed number of components of the same type organized as a simple linear sequence.

A component of a vector is selected by giving its subscript, an integer (or enumeration value) indicating the position of the components in the sequence. A vector is also termed a one-dimensional array or linear array. A two-dimensional array, or matrix , has its components organized into a rectangular grid of rows and columns. Both a row subscript and a column subscript are needed to select a component of a matrix. Multidimensional arrays of three or more dimensions are defined in a similar manner.





Records

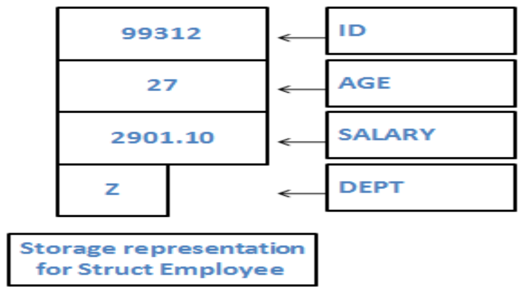
A data structure composed of a fixed number of components of different types is usually termed a record.

Int>>>>>>>>>>

Int>>>>>>>>>>

Float>>>>>>>>>

Char>>>>>>>>>



Implementation of vectors.
The homogeneity of components and fixed size of vector make storage and accessing of individual components straightforward.

Packed and unpacked storage representation. A packed storage representation is one in which components of a vector are packed into storage sequentially without regard for placing a component at the beginning of an addressable word or byte of storage.

Storage Implementation of records.

lvalue (R.I) = alpha + Ki when alpha is the base address and the Ki is the ith component.

```
struct EmployeeType{

int ID



}
```

SETS

A set is a data object containing an unordered collection of distinct values. In contrast, a list is an ordered collection of values, some of which may be repeated.

In Programming Languages, the term set is sometimes applied to a data structure representing an ordered set. An ordered set is actually a list with duplicate values removed; it requires no special consideration. The unordered set, however, admits two specialized storage representations that merit attention.

Bit-string representation of sets. The bit string storage representation is appropriate where the size of the underlying universe of values (the values that may appear in set data objects) is known to be small.

Hash-coded representation of sets. A common alternative representation for a set is based on the technique of hash coding or scatter storage. This method may be used when the underlying universe of possible values is large (e.g., when the set contains numbers or character strings).

Unions

- Unions are declared, created, and used exactly the same as struts, ***EXCEPT*** for one key difference:
    - Structs allocate enough space to store all of the fields in the struct. The first one is stored at the beginning of the struct, the second is stored after that, and so on.
    - Unions only allocate enough space to store the largest field listed, and all fields are stored at the same space - The beginnion of the union.
- This means that all fields in a union share the same space, which can be used for any listed field but not more than one of them.
- In order to know which union field is actually stored, unions are often nested inside of structs, with an enumerated type indicating what is actually stored there. For example:

```
 typedef struct Flight {
   enum { PASSENGER, CARGO } type;
   union {
      int npassengers;
      double tonnages;  // Units are not necessarily tons.
   } cargo;
} Flight;

Flight flights[ 1000 ];

flights[ 42 ].type = PASSENGER;
flights[ 42 ].cargo.npassengers = 150;

flights[ 20 ].type = CARGO;
flights[ 20 ].cargo.tonnages = 356.78;
```

- The example above does not actually save any space, because the 4 bytes saved by using a union instead of a struct for the cargo is lost by the int needed for the enumerated type. However a lot of space could potentially be saved if the items in the union were larger, such as nested structs or large arrays.
- Unions are sometimes also used to break up larger data items into smaller pieces, such as this code to extract four 8-bit bytes from a 32-bit int:

```
int nRead;

union {
   unsigned int n;
   unsigned char c[ 4 ];
} data;

 // ( Code to read in nRead, from the user or a file, has been omitted in this example )

 data.n = nRead;
```

```
for( int i = 0; i < 4; i++ )
   printf( "Byte number %d of %ud is %ud\n", i, nRead, data.c[ i ] );
```

Pointers

A *pointer type* is a type in which the range of values consists of memory addresses and a special value, nil (or null)

Uses:

1. Addressing flexibility
2. Dynamic storage management

Design Issues:

- What is the scope and lifetime of pointer variables?
- What is the lifetime of heap-dynamic variables?
- Are pointers restricted to pointing at a particular type?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should a language support pointer types, reference types, or both?

Problems with pointers

1. Dangling pointers (dangerous)
    - A pointer points to a heap-dynamic variable that has been deallocated
    - Creating one:
        - Allocate a heap-dynamic variable and set a pointer to point at it
        - Set a second pointer to the value of the first pointer
        - Deallocate the heap-dynamic variable, using the first pointer
2. Lost Heap-Dynamic Variables (wasteful)
    - A heap-dynamic variable that is no longer referenced by any program pointer

    - Creating one:

        a. Pointer p1 is set to point to a newly created heap-dynamic variable

        b. p1 is later set to point to another newly created heap-dynamic variable

    - The process of losing heap-dynamic variables is called *memory leakage*

Evolution of Data Types

FORTRAN I (1956) - INTEGER, REAL, arrays

Ada (1983) - User can create a unique type for every category of variables in the problem space and have the system enforce the types

Def: A *descriptor* is the collection of the attributes of a variable

Design Issues for all data types:

1. What is the syntax of references to variables?
2. What operations are defined and how are they specified?

Primitive Data Types

These types are supported directly in the hardware of the machine and not defined in terms of other types:

– **Integer:** Short Int, Integer, Long Int (etc.)
– **Floating Point:** Real, Double Precision

Stored in 3 parts, sign bit, exponent and mantissa (see Fig 5.1 page 199)

– **Decimal:** BCD (1 digit per 1/2 byte)

Used in business languages with a set decimal for dollars and cents

– **Boolean:** (TRUE/FALSE, 1/0, T/NIL)
– **Character:** Using EBCDIC, ASCII, UNICODE, etc.

Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types; sometimes more
- Usually exactly like the hardware, but not always; some languages allow accuracy specs in code e.g. (Ada)

type SPEED is digits 7 range 0.0..1000.0;

type VOLTAGE is delta 0.1 range -12.0..24.0;

- IEEE Floating Point Standard 754
  - Single precision: 32 bit representation with 1 bit sign, 8 bit exponent, 23 bit mantissa
  - Double precision: 64 bit representation with 1 bit sign, 11 bit exponent, 52 bit mantissa

Decimal and Boolean

Decimal

– For business applications (money)
– Store a fixed number of decimal digits (coded)
– *Advantage:* accuracy
– *Disadvantages:* limited range, wastes memory

Boolean

– Could be implemented as bits, but often as bytes
– *Advantage:* readability

Character Strings

- Characters are another primitive data type which map easily into integers.

- We’ve evolved through several basic encodings for characters:
  - 50s – 70s: EBCDIC (Extended Binary Coded Decimal Interchange Code) -- Used five bits to represent characters
  - 60s – 00s: ASCII (American Standard Code for Information Interchange) -- Uses seven bits to represent 128 possible “characters”
  - 90s – 00s - : Unicode -- Uses 16 bits to represent ~64K different characters. Needed as computers become less Eurocentric to represent the full range of non-roman alphabets and pictographs.

Character String Types

Values are sequences of characters

Design issues:

- Is it a primitive type or just a special kind of array?
- Is the length of objects static or dynamic?

Typical String Operations:

- Assignment
- Comparison (=, >, etc.)
- Catenation
- Substring reference
- Pattern matching

Subprograms

The ability to define subprograms is fundamental to all programming languages. Even Charles Babbage's Analytical Engine, built in the 1840's, had the ability to reuse collections of instruction cards [Sebesta 2002]. You probably already have an intuitive idea of what a subprogram provides. Subprograms associate a name with a sequence of processing instructions in a way that the statements can be reused at different points in the program by calling the subprogram name. For example:

```
float circumference(float radius) {
   return 2 * radius * 3.14159;
}
```

The subprogram above computes the circumference of a circle given its radius. The subprogram is a function because it returns a value.

Reuse is the most obvious benefit of defining a subprogram but subprograms also provide a form of process abstraction. They encapsulate programming language statements and hide implementation. The code inside a subroutine is safe from outside influences. Code outside of a subprogram can't modify local variables in a subprogram or transfer control to statements inside a subroutine except through the entry point. Subprograms also increase the level of abstraction in a program. For example, the code on the right below is more abstract than the code on the left:

Not very abstract	Process abstraction through subprograms

float c,r;
...
c = 2 * r * 3.1415;
	float c,r;
	...
	c = circumference(r);
	...
	function float circumference(**float** r){
	    return 2 * r * 3.1415;;
	}

The subprogram circumference() is an abstraction of the mathematical formula for calculating the circumference of a circle. It allows the programmer to calculate the circumference of a circle without worrying about the details of the computation. The code on the right is easier to understand and maintain because it is at a higher level of abstraction. Programs that are easier to understand also tend to be more reliable.

Abstract Data Types

[What are ADT's?]

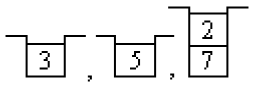
Abstract data types are user-defined data types which have many of the same characteristics of primitive data types.

1. An abstract data type defines a new type.
2. The type defines allowable values and operations on those values.
3. The implementation of data values and their operations are hidden. The only way to manipulate values of an abstract data type is through the operations defined for the type.

For example, the following Java class definition defines a stack container as an ADT:

```
// An instance of a stack can hold up to 10
// integer elements.
public class Stack {
    private static final int MAX_ELEMENTS = 10;
    private int data[];
    private int top;

    public Stack() {
        data = new int[MAX_ELEMENTS];
        top = 0;
    }

    public void push(int i) {
        data[top++] = i;
    }

    public int pop() {
        return (data[--top]);
    }

    public boolean empty() {
        return (top==0);
    }
}
```

The following table shows how abstract data type stack compares to the primitive data type short:

	Data Types	
	Primitive Data Type	**Abstract Data Type**
Type:	short	Stack
Values:	-32,768 to 32,767	 , etc
Operations:	+, -, *, /, ++, --, &, |, ==, !=, etc	push(), pop()
Implementation:	(Two's complement binary numbers, but you shouldn't depend on this. The implementation is hidden.)	(Array, but you shouldn't depend on this. The implementation is hidden.)

The table above shows there is a close parallel between primitive data types and abstract data types with one exception--the values of an abstract data type can be much more complex. In the example above a stack can hold zero to 10 elements and each element is an integer value.

Modern programming languages such as C++, Java and C# have direct support for defining abstract data types. The class construct in all three languages can be used to define a new type with a public interface and private implementation. Older languages such as Ada and Modula II which don't support the class construct can still be used to implement ADT's but require a little extra effort by clients of the ADT. Consider the following stack ADT expressed as an Ada package:

```
package Stack_Package is
   MAX_STACK_SIZE : constant := 100;
   type Stack is private;
   procedure push(stack_instance : in out Stack; value : in integer);
   procedure pop(stack_instance : int out Stack; value : out integer);
   function empty(stack_instance : Stack) return boolean;
   private
      type Stackdata is array(1 .. MAX_STACK_SIZE) of integer;
      type Stack is
         record
            data : Stackdata;
            tos : integer := 0;
         end record;
end Stack_Package;
```

The example above isn't a complete program. It includes only the specification package for the stack abstract data type. The implementation for the methods declared above would be specified in a separate body package. The specification package above defines a new type Stack. Clients can declare variables of type Stack but can't access the implementation of a Stack. The methods push(), pop() and empty() operate on instances of type Stack.

Modules lack one important feature which classes have. Types can be defined in a module but a module doesn't define a type. You can't create an instance of a module the way you can create an instance of a class. An abstract data type can be implemented as a module but if you need multiple instances of the abstract data type there must be some way of specifying which object operations should be applied to.

[Benefits of ADT's?]

The benefits provided by abstract data types are similar to the benefits provides by primitive data types. Hiding implementation reduces coupling between client code and implementation. This makes it possible to change implementation without affecting client code. Also, defining transformations on data values as operators on an abstract data type raises the level of abstraction. A client code that uses the method "pop()" to remove a value from a stack is more abstract than client code that references an array element. Coding at a higher-level of abstraction usually results in code that is easier to understand and maintain. Code that is easier to understand is usually also more reliable.

The real value of using ADT's is that they can define abstractions that are closer to the problem domain. If you could eavesdrop on a group of end-users it's doubtful you would hear them talking about ints and floats or even stacks and queues. More likely you would hear words from the problem domain like report, transaction and history file. If you create abstract data types that represent elements from the problem domain you can write your code in the language of the problem domain. The code will be easier to understand because the problem domain will serve as a model for how the code works.

Abstraction – in a nutshell is making visible what you want the external world to see and keeping the other details behind the wraps. Abstraction, as a process, denotes the extracting of the essential details about an item, or a group of items, while ignoring the inessential details. It hides the complexity. It is a technique by which we decide what information to hide and what to expose.

Information hiding – is making inaccessible certain details which would not affect other parts of the system. Just hiding it so that it is not exposed.

Encapsulation – is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object. If encapsulation was "the same thing as information hiding," then one might make the argument that "everything that was encapsulated was also hidden." This is obviously not true. For example, even though information may be encapsulated within record structures and arrays, this information is usually not hidden and is available for use. Encapsulation is just getting a set of operations and data together which belong together and putting them in a capsule.

Abstraction, information hiding, and encapsulation are different but related. Abstraction is a technique that helps us identify which specific information should be visible, and which information should be hidden. Encapsulation is the technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible. Encapsulation can be thought of as the implementation of the abstract class.