Chapter 9
Structured Data: Structs and ADTs
(Data Base Programs with C++)

Mr. Dave Clausen

La Cañada High School

The need for Structs

- So far we have structured the data associated with a program in files, vectors or matrices.
- Files and “arrays” have helped us organize data that are of the same type.
- We could use “arrays” to organize data of different types using “parallel arrays” and multidimensional arrays, but there is a better way…Structs.

Data Bases and Structs

- Before we look at structs, let's remind you how data is organized in a typical data base program.
- We organize the data into records, and then organize the records into fields or categories of information that contain the same type of data in each field.
- Data may be displayed in a "table" format or a "label" or "form" format.

Form or Single Record Layout

This format displays all the fields and data elements for one record only. This is similar to a struct.

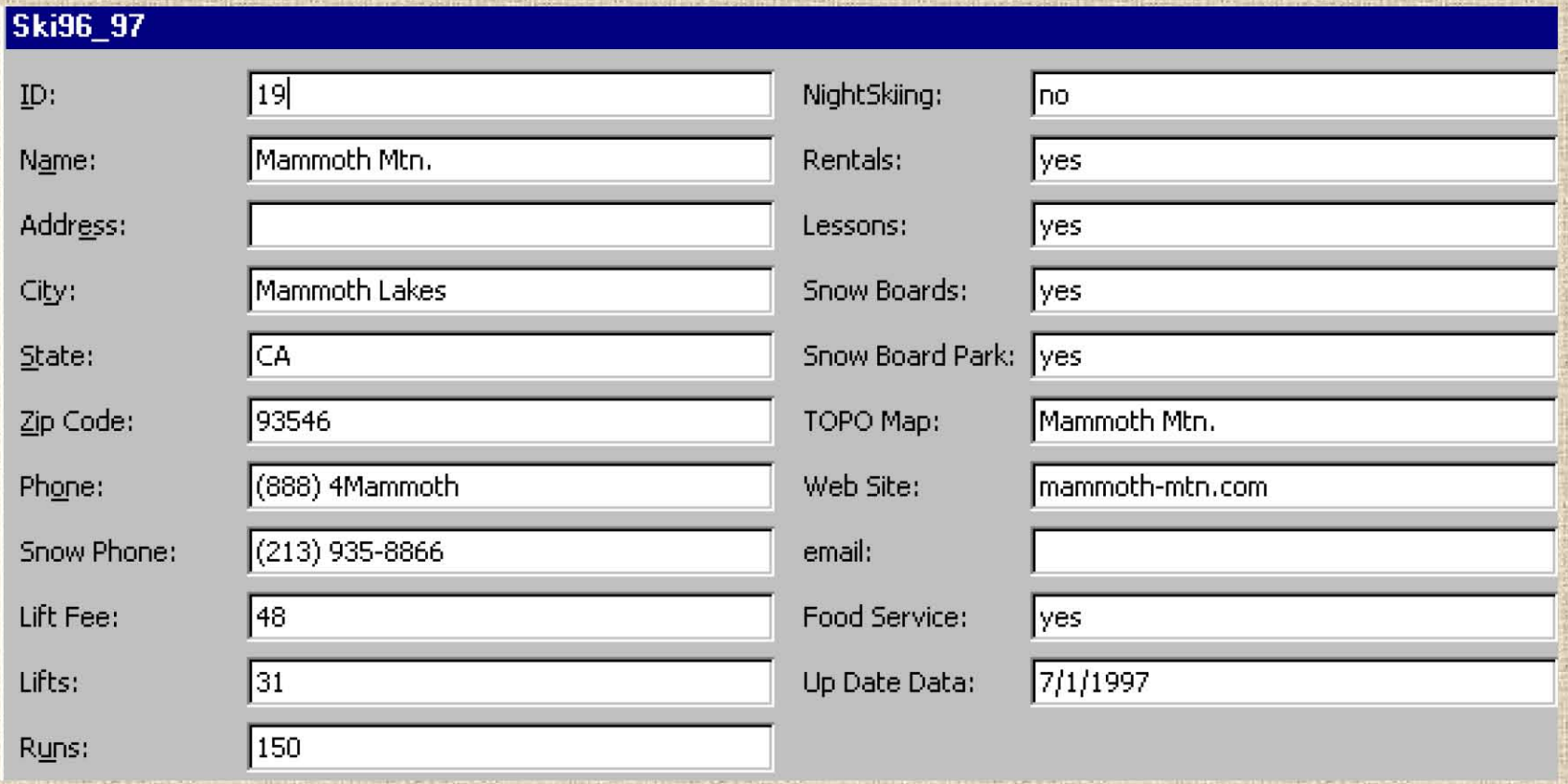
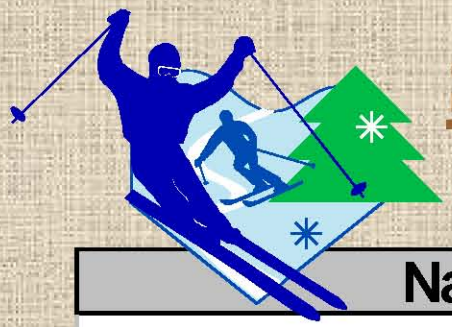


Table or Multiple Record Format

In this format:

The Rows are Records, and The Columns are the Categories or Fields. This is similar to a vector of records or structs.



Ski Resorts in California

Name	City	State	Lifts	Runs
Alpine Meadows	Tahoe City	CA	13	100
Badger Pass	Yosemite	CA	5	9
Bear Mtn.	Big Bear Lake	CA	12	36
Bear Valley	Bear Valley	CA	11	80

What is a Struct?

- A struct is a data structure that can have components or elements of different data types associated by and accessible by the same name.
- The components or fields are called members of the struct.
- When using structs it is often convenient to define a new data type, and declare variables to be of this type.

Defining a Struct Type

▣The general form for defining a struct type is:

struct <new type name> //Include the word type here

{

<data type> <member name 1>;

<data type> <member name n>;

};

//Yes, a semicolon must follow the right curly bracket.

//A structure declaration is a statement, and therefore,

//needs a semicolon to end the statement.

A Struct Example

```
struct employee_type       //We included the word type.
{
   apstring name, street, city, state, zip_code;
   int age;
   double salary;
};      //Don't forget the semicolon!
//Variable Declarations based upon the new data type.

employee_type employee1, employee2;
```

Another struct example

- Structs don't have to use different data types.
  - Windows was originally written using structs in C (classes in C++ weren't available yet), and still has some of this "legacy" code. Structs were used to keep track of the coordinates of rectangular windows…

```
struct Rect
{
    int left;       //Top left point coordinate point
    int top;
    int right;      //Bottom right point coordinate point.
    int bottom;
};
```

Remember where we define types...

- Comments
- Preprocessor Directives
- Constant Definition Section
- Type Definition Section (struct, class, typedef)
- Function Declaration Section
- Main Program Heading: int main( );
  - Declaration Section (Variables, etc…)
  - Statement Section
- Function Implementations

Structs: General Information

- When the struct variables are declared, that is when the computer allocates the memory for all the elements of the structs, not when the struct is defined.
- The members (fields) of a struct are accessed by using a member selector:
  - A selector is a period placed between the struct variable name and the name of a struct member (or field in our database terminology).

Using Struct Member Selectors

```
employee1.name = “John Doe”;
employee1.street = “102 Maple Lane”;
employee1.city = “York”;
employee1.state = “PA”;
employee1.zip_code = “12309”;
employee1.age = 21;
employee1.salary = 10000.00;
```

Visualizing the Struct as a Data Base Form

employee1

Members of the struct	Data Items
name	John Doe
street	102 Maple Lane
city	York
state	PA
zip_code	12309
age	21
salary	100000.00

Advantages of Using Structs

- If you define your struct as a data "type", and declare two variables of that same type, you can:
  - Perform aggregate assignments

    employee2 = employee1;
    - This will copy every member of one struct into the other. (We don't have to copy every member…)

    employee2.name = employee1.name;

    employee2.street = employee1.street;

    etc… (A great shortcut!)

Nested Structs & Hierarchical Records

- A component of a record can be another record. When this occurs we call them hierarchical records.
- Using struct vocabulary, a struct member can be any valid data type. When a struct is a member of another struct, we call this nested structs.
- In this case, more than one member selector may be necessary to access any data member.

An Example of Nested Structs

```
struct address_type
{
   apstring street, city, zip_code;
};
struct employee_type
{
   apstring name;
   address_type address;
   int age;
   double salary;
};  //address_type needs to be declared before employee_type
```

Let's Try Defining A Struct

```
Struct Date_Type
{




};  //Represent: Month (int or enum), Day, & Year
```

Let's Try Defining Another Struct

```
Struct Inventory_Type
{




};  //Part #, Description, Quantity, Price,  & Ship Date
```

Using Multiple Member Selectors

- If we declare variables based on these nested structs such as:

employee_type employee1, employee2;

- To access any part of the address requires 2 member selectors:

```
cout<<"Street: "<<employee1.address.street<<endl;

cout<<"City: "<<employee1.address.city<<endl;

cout<<"Zip Code: "<<employee1.address.zip_code<<endl;
```

Other Struct Advantages

- If you define your struct as a data "type" statement, you can define more complex structures, like a vector of structs.
  - apvector <employee_type> list(vector_size);
  - Each component of this vector is a struct containing the members previously defined in employee_type
    - list[0] //accesses the **entire struct** for the first employee
    - list[0].name //accesses the name member of the struct for the first employee

Using Structs with Functions

- Functions can return data of type struct. Consider the following function declarations:

```
employee_type init_employee( );
employee_type read_employee( );
employee_type print_employee( );
```

- The implementations of these functions will initialize, read, or print each member of the struct

Sample Function Implementation

```
employee_type init_employee( )
{
        employee_type employee;  //A local variable is declared

        employee.name = "John Doe";
        employee.street = "123 Anywhere Lane";
        employee.city = "SomeCity";
        employee.state = ”SomeState";
        employee.zip_code = "00000";
        employee.age = -1;
        employee.salary = -1.00;
//Use initialization values that are obviously not real data items.
        return employee;
}
```

A Vector of Structs

- **To declare a vector of structs:**
  - Ask the user how many elements they would like in the vector. Then declare the vector using:

apvector <employee_type> list(vector_size);

- **You can now declare and call functions using the vector of structs:**

  void read_list(apvector <employee_type> &list);
  - Remember, you don't have to pass the length of the vector as a parameter to each function. You can use the length member function built into the apvector class:

  list.length( )

Data Abstraction

- In Chapter 4, we said that a function is an example of abstraction, because it simplifies a complex task for its users.
- Simplification of complex **data** requires two components:
  - **A data structure** such as an vector or struct, labeled with a type name.
  - And a **set of functions** that perform operations on the data structure.
- Therefore, data abstraction is the separation of the conceptual definition of a data structure and its implementation.

Abstract Data Types (ADT's)

- An ADT is a class of objects (values), a defined set of properties of those objects (values), and a set of operations for processing the objects(values).
- The benefits of ADT's are simplicity and ease of use.
- An ADT doesn't mention how it is implemented, or how it can be used in a program.
- Many C++ ADT's are already defined in libraries.
- A library header file contains the type definitions of the data structures and the function declarations

Sample ADT Library Header Files

- The employee type ADT contains definitions for employee_type, and function declarations for init_ employee, read_ employee, and print_ employee.

  employee.h employeh.htxt

- The list type ADT contains definitions for MAX_LIST_SIZE, list_type, and function declarations for init_list, read_list, and print_list. This is based upon an array, not a vector… listvctr.h listvctrh.htxt

Let's look at the assigned program

- We can use the employee type ADT in our program, as is…
- We will have to modify the list type ADT to use vectors instead of arrays…
- We will not separate the header ".h" files from the implementation ".cpp" files in this program.
- Refer to pages 502 - 507 for this program.

P9avectr.exe