Structures

Structures

- Complex data types
- Structures
- Defined types
- Structures and functions
- Structures and pointers
- (Very) brief introduction to the STL

Representing Complex Data

Representing Data

- Many programs require complex data to be represented
  - That cannot easily be represented using base type variables and arrays
    - B*ase types* are the defined C++ types like *int*, *char*, *float*, etc.
- C++ allows *structures* to be defined and used in a program
  - A structure is a complex type that collects a number of variables, arrays or even other structures

Representing Students

- Let's assume that we want to represent student data for a small university
- For each student we want to maintain
  - First name – a string
  - Last name – a string
  - Student number – an integer
  - GPA – a float

this is a simplified version of what would be necessary, *some* of the complicating issues will be discussed later

Students without Structures

- If we wanted to maintain a list of a hundred students we could use four separate arrays
  - One for each of the four attributes
    - First name, last name, student number, GPA
  - Referred to as *parallel* arrays
- Whenever we want to retrieve student data we would use the same index in all arrays

Parallel Arrays

index	0	1	2	…	33	…	97	98	99
id	9101	7234	5678		4864		1789	2457	4444

index	0	1	2	…	33	…	97	98	99
first	Bob	Kate	Sue		Dave		Joe	Ella	Anna

index	0	1	2	…	33	…	97	98	99
last	Wing	Smith	Abel		Dodd		Ng	Moss	Frenc

index	0	1	2	…	33	…	97	98	99
gpa	3.00	3.50	2.75		3.75		2.25	4.00	3.50

Represents the student Dave Dodd

Parallel Array Drawbacks

- Maintaining parallel arrays can be fiddly
  - We must make consistent changes to all arrays
    - If we delete an element from one, elements at the same index must be deleted from the others
    - If we sort one array then all the other arrays must be sorted in the same way
  - Passing student data to functions is tedious
    - A function to print student data needs four parameters
- Or we can use structures
  - Or classes

Sort Error in a Parallel Array

Parallel arrays with sample data – assume we want to sort by ID

index	0	1	2	3	4	5	6	7	8
id	9101	7234	5678	4475	4864	3459	1789	2457	4444

index	0	1	2	3	4	5	6	7	8
first	Bob	Kate	Sue	Alan	Dave	Joy	Joe	Ella	Anna

index	0	1	2	3	4	5	6	7	8
last	Wing	Smith	Abel	Flim	Dodd	Shae	Ng	Moss	Frenc

index	0	1	2	3	4	5	6	7	8
gpa	3.00	3.50	2.75	2.00	3.75	3.50	2.25	4.00	3.50

Sort Error in a Parallel Array

Sort ID array

The data is now corrupt – students no longer have the right IDs

index	0	1	2	3	4	5	6	7	8
id	1789	2457	3459	4444	4475	4864	5678	7234	9101

index	0	1	2	3	4	5	6	7	8
first	Bob	Kate	Sue	Alan	Dave	Joy	Joe	Ella	Anna

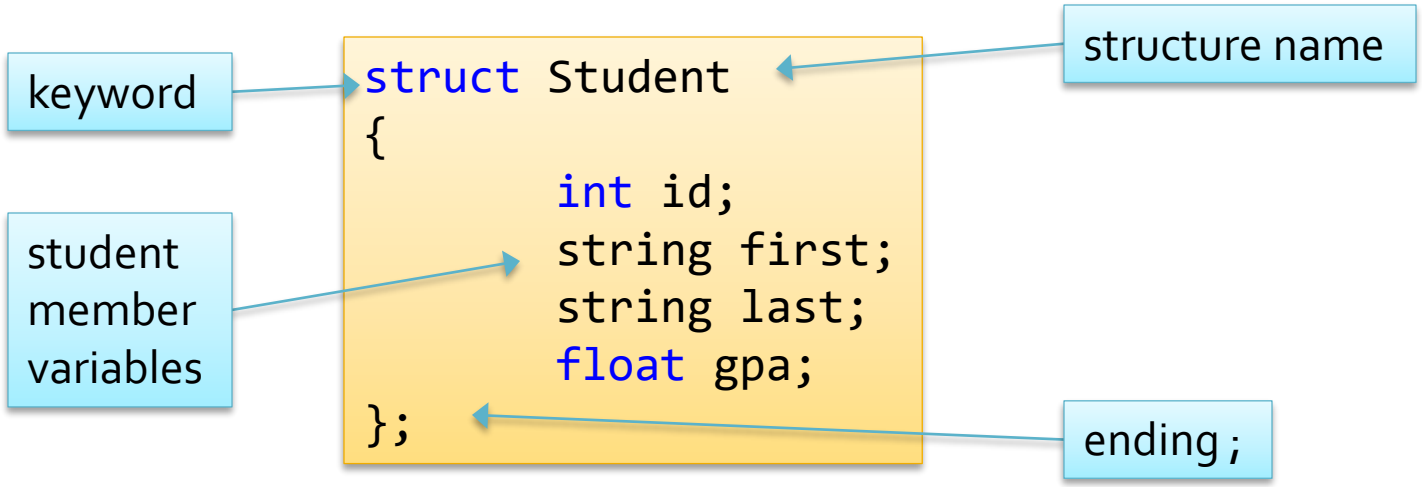
index	0	1	2	3	4	5	6	7	8
last	Wing	Smith	Abel	Flim	Dodd	Shae	Ng	Moss	Frenc

index	0	1	2	3	4	5	6	7	8
gpa	3.00	3.50	2.75	2.00	3.75	3.50	2.25	4.00	3.50

Structures

Structure Declarations

- A structure declaration defines a complex type



Defining a Structure Variable

- A structure declaration describes what data types are associated with a *struct*
  - It does not allocate space for any variables
    - It is like a blueprint for a type
- Space is allocated when a variable is declared
  - `Student st1;`
  - Instructs the compiler to reserve space for an int, two strings and a float
    - These four components of *st1* are allocated memory on the stack in sequence

What Goes Where

- It is common for *struct* definitions to appear outside any function – including main
  - So that they are available anywhere in the file
  - If a *struct* is defined inside a function it is only available in that function
- Variables of a *struct* type are declared wherever they are needed
  - Just like any other variable

Initializing a Structure

- A structure can be initialized in much the same way as an array
  - Using a comma separated list of values enclosed in curly brackets

```
Student st1 = { 123, "bob", "bobson", 2.5 };
```

Accessing Structure Members

- Variables of a structure have to be accessed individually using the *member operator* (.)
  - To access a structure variable use the structure name and the variable name separated by a dot
    - `s1.id = 12345;`
  - They can then be used like any variable of the same type
    - And can be accessed, assigned new values, passed to functions and so on

Enter Student Data

- We will look at an example that enters and prints student data
  - The student structure is declared
  - A student variable is defined
  - The user is requested to enter values for the structure attributes
  - The student data is printed

Student Data – Declarations

declares the student structure

```
#include "<iostream>"
#include "<string>"
using namespace std;

// Student structure
struct Student
{
        int id;
        string first;
        string last;
        float gpa;
};

// Forward Declarations
void printStudent(Student st);
```

the student structure

Student Data – main

```
int main()
{
        Student s1;
        cout << "Enter ID: ";
        cin >> s1.id;
        cout << "Enter first name: ";
        cin >> s1.first;
        cout << "Enter last name: ";
        cin >> s1.last;
        cout << "Enter GPA: ";
        cin >> s1.gpa;
        cout << endl;

        printStudent(s1);
        return 0;
}
```

the one and only student

use dot notation to access student attributes (member variables)

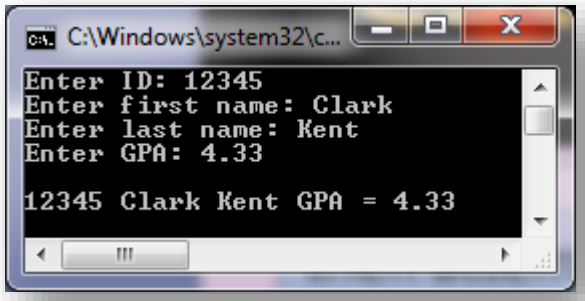
call the print function

Student Data – Print Function

and finally the print function definition

```
void printStudent(Student st)
{
        cout << st.id;
        cout << " " << st.first << " " << st.last;
        cout << " GPA: " << st.gpa;
}
```

note: passed by value



Structures and Functions

- Structures can be used as function parameters and arguments
  - In the same way as any other variable
  - Parameter passing is pass by value
    - Structure variables are not pointers
    - Unlike array variables
- When a structure is passed to a function the parameter is a copy of the original
  - Even if the original structure contained arrays!

Arrays of Structures

- It is possible to create arrays of structures
  - They are declared like any other kind of array
    - e.g. `Student class[100];`
- Individual elements are also accessed like any other array
  - *struct* attributes are accessed with dot notation
  - Let's say we want to find the first name of the student with index 15
    - `class[15].first` not class.first[15] …

Assigning Structures

- Unlike arrays, one structure can be assigned to another structure of the same type
  - Again, even if the structure contains an array
- A note on memory allocation
  - A structure might include an array in dynamic memory
    - It's array variable is really a pointer to that array
    - Pointer size is constant, even though the size of arrays in two different structures might vary
  - Two structures that contain different sized arrays are still the same size in bytes

More Complex Types

- Let's make the example more complex
  - It isn't realistic to just record GPA
  - GPA is calculated from the grades that a student receives for courses
- We will create a simple course structure
  - Department (like CMPT)
  - Number (like 130)
  - Grade (A, B, C, D and F)

Course and Student Structure

Here is the course structure

```
// Course structure
struct Course
{
        string department;
        int number;
        char grade;
};
```

it is perfectly OK to nest structures and to make arrays of structures

And the revised student structure

```
struct Student
{
        int id;
        string first;
        string last;
        Course* grades;
        int coursesTaken;
        int maxCourses
} ;
```

an array of courses!

struct Methods

- There is an issue with the Student structure
  - What happens with the array of courses when we create a new Student?
    - The array has not yet been created by calling new
    - So the array variable should be set to *NULL* or *nullptr*
- It is possible to define methods for C++ structures
  - Functions that belong to the structure
  - We will write a *constructor* for Student

Course and Student Structure

```
struct Student
{
        int id;
        string first;
        string last;
        Course* grades;
        int coursesTaken;
        int maxCourses;

        Student() {
                id = 0;
                first = last = "";
                grades = NULL;
                coursesTaken = 0;
                maxCourses = 0;
        }
};
```

The revised student structure

This is the definition of the constructor for a Student – it sets the initial values of the attributes

Note that there is no return type

Constructors

- A constructor is a special kind of function
  - That is used to initialize the member variables of a struct or a class
  - It is called automatically whenever a new Student variable is created
    - `Student s1;`
- Setting the array to *NULL* allows us to write a function to insert values into a student
  - To recognize that the array has not been created
  - And create the array using new

Notes

- The Student struct still has some issues
  - We need two variables to deal with the array size
    - maxCourses records the actual size of the array
    - Which could be increased if necessary
    - coursesTaken records the number of courses that the student has actually taken
- We would need to write an insertion function to ensure that these values are set correctly
  - And that the array is created
  - Using a *vector* instead of an array would simplify some of these issues