Structures
Structures

- Complex data types
- Structures
- Defined types
- Structures and functions
- Structures and pointers
- (Very) brief introduction to the STL
Representing Complex Data
Many programs require complex data to be represented

- That cannot easily be represented using base type variables and arrays
  - Base 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
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.
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

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>33</th>
<th>...</th>
<th>97</th>
<th>98</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>9101</td>
<td>7234</td>
<td>5678</td>
<td>4864</td>
<td>1789</td>
<td>2457</td>
<td>4444</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>33</th>
<th>...</th>
<th>97</th>
<th>98</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>Bob</td>
<td>Kate</td>
<td>Sue</td>
<td>Dave</td>
<td>Joe</td>
<td>Ella</td>
<td>Anna</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>33</th>
<th>...</th>
<th>97</th>
<th>98</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>last</td>
<td>Wing</td>
<td>Smith</td>
<td>Abel</td>
<td>Dodd</td>
<td>Ng</td>
<td>Moss</td>
<td>Frenc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>33</th>
<th>...</th>
<th>97</th>
<th>98</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpa</td>
<td>3.00</td>
<td>3.50</td>
<td>2.75</td>
<td>3.75</td>
<td>2.25</td>
<td>4.00</td>
<td>3.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Represents the student Dave Dodd
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

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>9101</td>
<td>7234</td>
<td>5678</td>
<td>4475</td>
<td>4864</td>
<td>3459</td>
<td>1789</td>
<td>2457</td>
<td>4444</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>Bob</td>
<td>Kate</td>
<td>Sue</td>
<td>Alan</td>
<td>Dave</td>
<td>Joy</td>
<td>Joe</td>
<td>Ella</td>
<td>Anna</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>last</td>
<td>Wing</td>
<td>Smith</td>
<td>Abel</td>
<td>Flim</td>
<td>Dodd</td>
<td>Shae</td>
<td>Ng</td>
<td>Moss</td>
<td>Frenc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpa</td>
<td>3.00</td>
<td>3.50</td>
<td>2.75</td>
<td>2.00</td>
<td>3.75</td>
<td>3.50</td>
<td>2.25</td>
<td>4.00</td>
<td>3.50</td>
</tr>
</tbody>
</table>
### Sort Error in a Parallel Array

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

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>1789</td>
<td>2457</td>
<td>3459</td>
<td>4444</td>
<td>4475</td>
<td>4864</td>
<td>5678</td>
<td>7234</td>
<td>9101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>Bob</td>
<td>Kate</td>
<td>Sue</td>
<td>Alan</td>
<td>Dave</td>
<td>Joy</td>
<td>Joe</td>
<td>Ella</td>
<td>Anna</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>last</td>
<td>Wing</td>
<td>Smith</td>
<td>Abel</td>
<td>Flim</td>
<td>Dodd</td>
<td>Shae</td>
<td>Ng</td>
<td>Moss</td>
<td>Frenc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpa</td>
<td>3.00</td>
<td>3.50</td>
<td>2.75</td>
<td>2.00</td>
<td>3.75</td>
<td>3.50</td>
<td>2.25</td>
<td>4.00</td>
<td>3.50</td>
</tr>
</tbody>
</table>
Structures
A structure declaration defines a complex type

```c
struct Student {
    int id;
    string first;
    string last;
    float gpa;
};
```
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
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
A structure can be initialized in much the same way as an array

- Using a comma separated list of values enclosed in curly brackets

```java
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

```cpp
#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);
```

declares the student structure
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;
}
void printStudent(Student st) {
    cout << st.id;
    cout << " " << st.first << " " << st.last;
    cout << " GPA: " << st.gpa;
}
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
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)
Here is the course structure:

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

And the revised student structure:

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

It is perfectly OK to nest structures and to make arrays of structures!
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

```c
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.
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
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