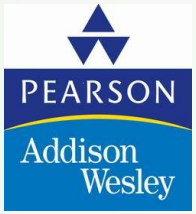
Starting Out with C++: Early Objects 5th Edition

Chapter 7
Structured Data and Classes



Structures

- A Structure is a collection of related data items, possibly of different types.
- A structure type in C++ is called struct.
- **A struct is heterogeneous** in that it can be composed of data of different types.
- In contrast, array is homogeneous since it can contain only data of the same type.

Structures

- Structures hold data that belong together.
- Examples:
  - Student record: student id, name, major, gender, start year, …
  - Bank account: account number, name, currency, balance, …
  - Address book: name, address, telephone number, …
- In database applications, structures are called records.

Structures

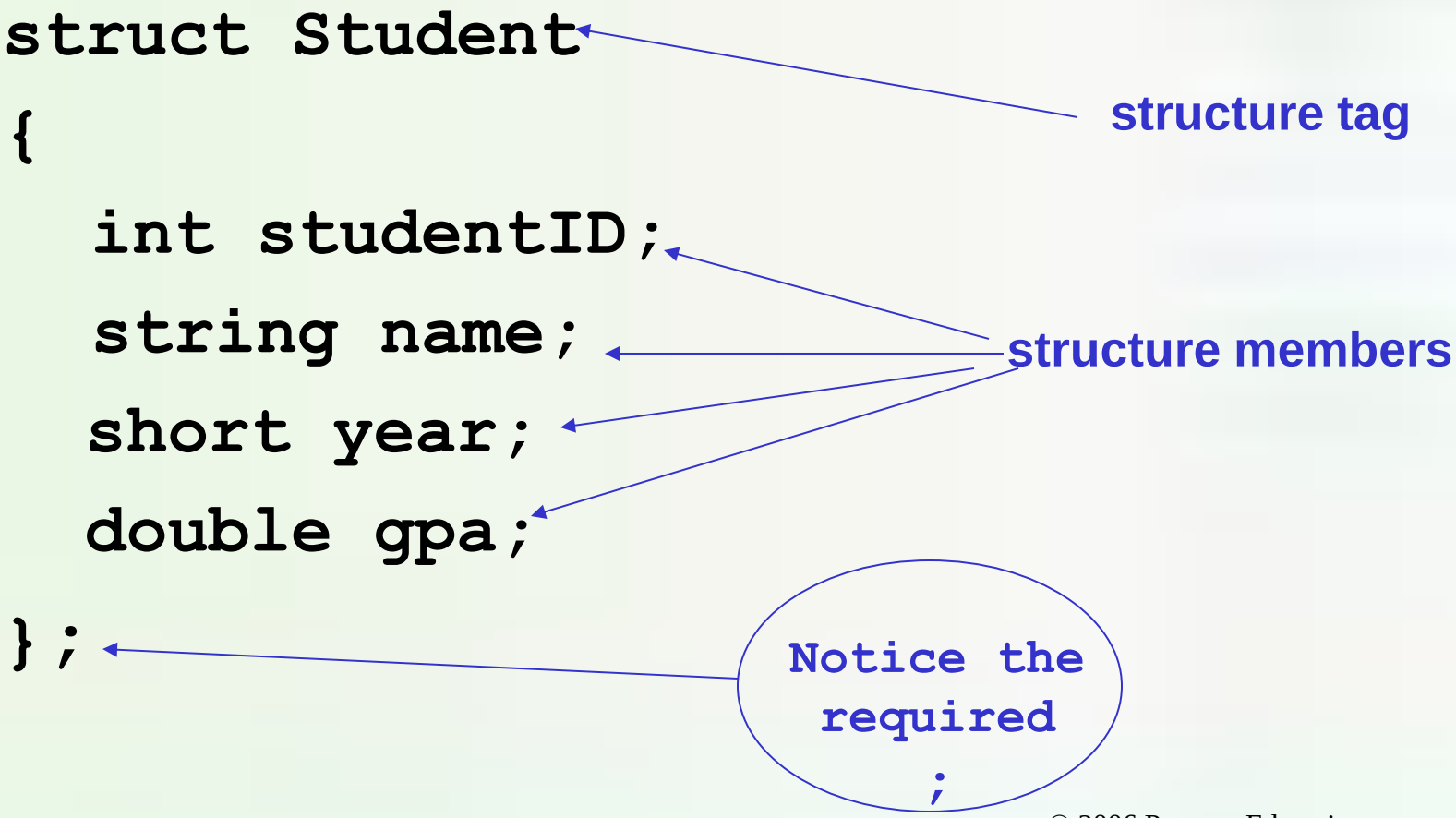
- **Individual components** of a struct type are called **members** (or fields).
- Members can be of different types (simple, array or struct).
- A struct is named as a whole while individual members are named using field identifiers.
- Complex data structures can be formed by defining arrays of structs.

7.1 Combining Data into Structures

- Structure: C++ construct that allows multiple variables to be grouped together
- Structure Declaration Format:

```
struct structure name
{
  type1 field1;
  type2 field2;
     …
  typen fieldn;
};
```

Example `struct` Declaration



`struct` Declaration Notes

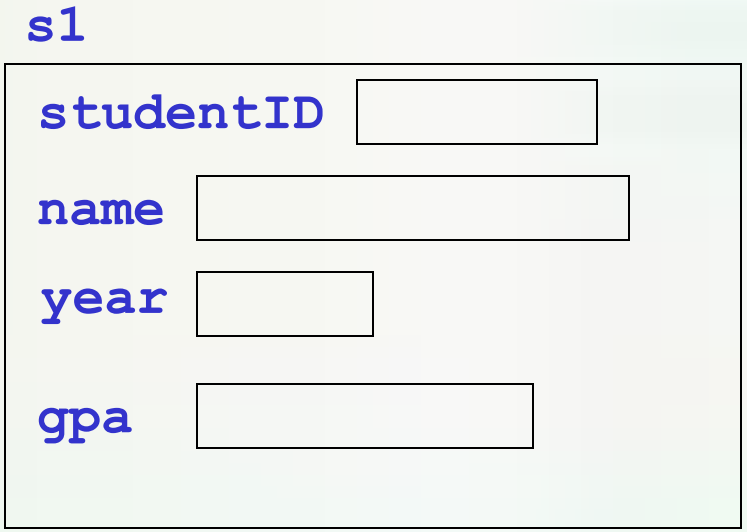
- **`struct`** names commonly begin with an uppercase letter
- Multiple fields of same type can be in a comma-separated list

```
string name,
       address;
```

Defining Structure Variables

- **`struct`** declaration does not allocate memory or create variables
- To define variables, use structure tag as type name

```
Student s1;
```



7.2 Accessing Structure Members

- Use the dot **`(.)`** operator to refer to members of **`struct`** variables

```
getline(cin, s1.name);
cin >> s1.studentID;
s1.gpa = 3.75;
```

- Member variables can be used in any manner appropriate for their data type

Displaying `struct` Members

- To display the contents of a **`struct`** variable, you must display each field separately, using the dot operator

Wrong:

```
cout << s1; // won't work!
```

Correct:

```
cout << s1.studentID << endl;
cout << s1.name << endl;
cout << s1.year << endl;
cout << s1.gpa;
```

7.3 Initializing a Structure

- Cannot initialize members in the structure declaration, because no memory has been allocated yet

```
struct Student      // Illegal
{                   // initialization
   int studentID = 1145;
   string name = "Alex";
   short year = 1;
   float gpa = 2.95;
};
```

Initializing a Structure (continued)

- Structure members are initialized at the time a structure variable is created
- Can initialize a structure variable's members with either
  - an initialization list
  - a constructor

Using an Initialization List

- An initialization list is an ordered set of values, separated by commas and contained in **{ }**, that provides initial values for a set of data members

```
{12, 6, 3}  // initialization list
            // with 3 values
```

More on Initialization Lists

- Order of list elements matters: First value initializes first data member, second value initializes second data member, etc.
- Elements of an initialization list can be constants, variables, or expressions

```
{12, W, L/W + 1} // initialization list
                 // with 3 items
```

Initialization List Example

Structure Declaration

```
struct Dimensions
{ int length,
      width,
      height;
};
```

Structure Variable

box


```
Dimensions box = {12,6,3};
```

Partial Initialization

- Can initialize just some members, but cannot skip over members

```
Dimensions box1 = {12,6};  //OK
Dimensions box2 = {12,,3}; //illegal
```

Problems with Initialization List

- Can’t omit a value for a member without omitting values for all following members
- Does not work on most modern compilers if the structure contains any string objects
  - Will, however, work with C-string members

Using a Constructor to Initialize Structure Members

- A constructor is a special function that can be a member of a structure
- It is normally written inside the **`struct`** declaration
- Its purpose is to initialize the structure's data members

Using a Constructor (continued)

- Unlike most functions, a constructor is not *called*; instead, it is automatically invoked when a structure variable is created
- The constructor name must be the same as the structure name (i.e. the struct tag)
- The constructor must have no return type

A Structure with a Constructor

```
struct Dimensions
{
   int length,
       width,
       height;

   // Constructor
   Dimensions(int L, int W, int H)
   {length = L; width = W; height = H;}
};
```

Passing Arguments to a Constructor

- Create a structure variable and follow its name with an argument list
- Example:

```
Dimensions box3(12, 6, 3);
```

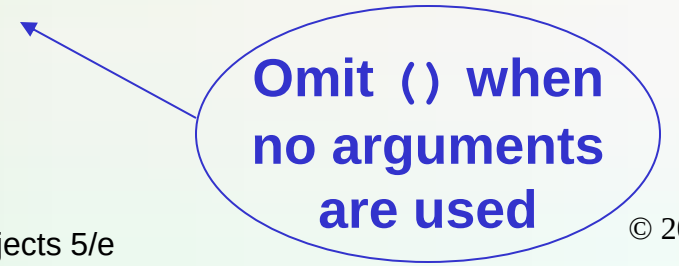
Default Arguments

- **A constructor may be written to have default arguments**

```
struct Dimensions
{
   int length,
       width,
       height;

   // Constructor
   Dimensions(int L=1, int W=1, int H=1)
   {length = L; width = W; height = H;}
};
```

Examples

```
//Create a box with all dimensions given
Dimensions box4(12, 6, 3);

//Create a box using default value 1 for
//height
Dimensions box5(12, 6);

//Create a box using all default values
Dimensions box6;
```



7.4 Nested Structures

A structure can have another structure as a member.

```
struct PersonInfo
{  string name,
           address,
           city;
};
struct Student
{  int          studentID;
   PersonInfo   pData;
   short        year;
   double       gpa;
};
```

Members of Nested Structures

- Use the dot operator multiple times to dereference fields of nested structures

```
Student s5;
s5.pData.name = "Joanne";
s5.pData.city = "Tulsa";
```

7.5 Structures as Function Arguments

- May pass members of **`struct`** variables to functions

```
computeGPA(s1.gpa);
```

- May pass entire **`struct`** variables to functions

```
showData(s5);
```

- Can use reference parameter if function needs to modify contents of structure variable

Notes on Passing Structures

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a **`const`** reference parameter

```
void showData(const Student &s)
                                  // header
```

7.6 Returning a Structure from a Function

- Function can return a **`struct`**

```
Student getStuData();  // prototype
s1 = getStuData();     // call
```

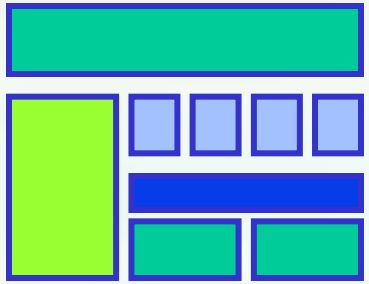
- Function must define a local structure
  - for internal use
  - to use with **`return`** statement

Returning a Structure Example

```
Student getStuData()
{ Student s;     // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;
}
```

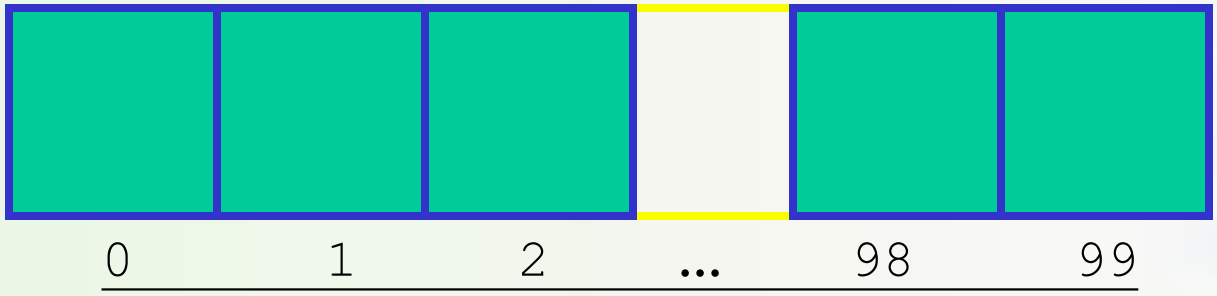
Structures

- Complex data structures can be formed by defining **arrays of structs**.

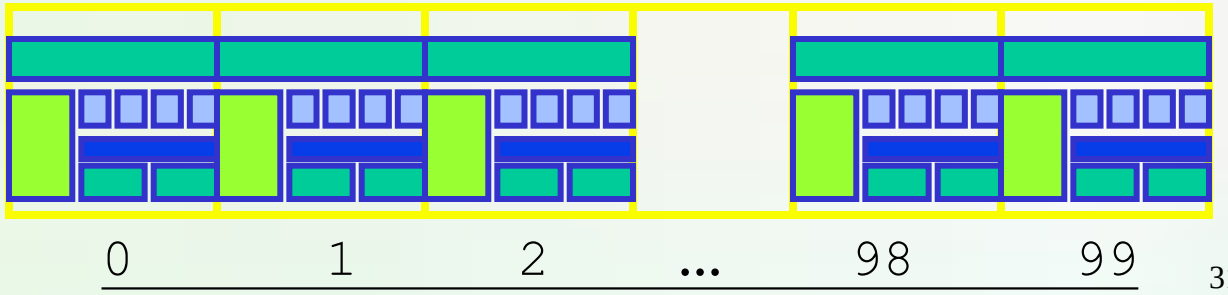


Arrays of structures

- An ordinary array: One type of data



- An array of structs: Multiple types of data in each array element.



Arrays of structures

- We often use arrays of structures.
- Example:

```
StudentRecord Class[100];
strcpy(Class[98].Name, "Chan Tai Man");
Class[98].Id = 12345;
strcpy(Class[98].Dept, "COMP");
Class[98].gender = 'M';
Class[0] = Class[98];
```