# structs in C++

- structs are a form of container, used to group multiple data items (possibly of different types) into a single logical entity

***IMPORTANT NOTE ON STRUCTS VS CLASSES:***

- in fact, the only ***technical*** difference between a C++ class and a C++ struct is that structs declare/inherit things publicly by default, whereas classes declare/inherit them privately by default
- ***by convention***, however, programmers use the two quite differently: structs being used purely as data containers, and classes being used where we wish to incorporate more advanced functionality (e.g. methods)
- in these slides we will strictly use structs in the conventional form (i.e. without associated methods)

# structs as a container

- arrays allowed us to group elements of a specific data type into a single logical entity, then refer to them by position within the array
- structs allow us to group elements of different data types into a single logical entity
- when we define a struct we are actually defining a new data type: a record or structure made up of a variety of components, or fields, each given their own name
- we can then declare variables, constants, parameters etc that are of that type, and assign values to their fields

# struct syntax

- we specify a new typename, plus types and names for each of the individual fields
- for example, information about a product in a store

```
struct productInfo {
    string name;
    float price;
};
```

- we can create variables of the new data type, and access content through variablename.fieldname, e.g.

```
productInfo x, y, z;
x.name = “Widget”;
```

# struct use, example

- we can use the var.field syntax anywhere we could use a variable of the same type, e.g. the the productInfo from the previous slide

```
productInfo p;
cout << “Enter the product name and price” << endl;
cin >> p.name >> p.price;
cout << p.name << “ $” << p.price << endl;
```

# structs as value parameters

- we can write functions to expect structs as parameters

```
void print(productInfo prod)
{
    cout << “Item: “ << prod.name;
    cout << “, $” << prod.price << endl;
}
```

- and we can call them by passing the variable name

```
print(p);  // from previous slide's example
```

# structs passed by reference

- for a function to change the content of a struct, the struct must be passed by reference

```
void fill(productInfo &prod)
{
    cin >> prod.name >> prod.price;
}
```

- we can then call the function normally

```
fill(p);
```

# Arrays of structs

- as with other data types, we often want to store a collection of them, e.g. in an array

```
const int size = 10;
productInfo products[size];
```

- we refer to a field within specific item using the arrayname, the position in the array, and the fieldname

```
for (int i=0; i<size; i++) {
    cin >> products[i].name;
    cin >> products[i].price;
}
```

# Array of structs as a parameter

- as with other kinds of arrays, we can pass them as params

```
void fillProducts(productInfo prods[], int num)
{
   for (int i=0; i<num; i++) {
      cin >> prods[i].name;
      cin >> prods[i].price;
   }
}
```

- we call by passing the array name and size

```
fillProducts(products, size);
```

# Example: searching arr of struct

- a function to search an array for a product with a specific name, returning its position (or -1 if not found)

```
int findProd(productInfo prods[], int size, string target)
{
   for (int p = 0; p < size; p++) {
       if (prods[p].name == target) {
           return p; // found a match
       }
   }
   return -1; // didn't find it anywhere in the array
}
```

# Structs for abstraction

- note that grouping the product information into a struct has allowed us to think of a product as a logical entity
- if we add, remove or alter fields from the struct, the internal details of functions like fill or print must change, but not the profile of the function and not the routines using the function

```
int main() {
    // nothing in main cares about p's internal details
    productInfo p;
    fill(p);
    print(p);
}
```

# Structs of structs

- we can create heirachies of structs, consider
- a Student struct might contain information about their marks on each of up to 10 labs
- a Course struct might then contain information about each of up to 60 students

```
const int N=10;
const int S=60;

struct Student {
   string name;
   float marks[N];
   int numLabs;
};

struct Course {
   string instructor;
   int numStudents;
   Student students[S];
};
```

# Accessing nested struct fields

- we could access specific fields directly, e.g. assigning a mark for lab j of student i in the course:

```
Course info160;  // assuming this was our course variable
cout << “Enter the mark for lab “ << j;
cout << “ for student “ << info160.students[i].name;
cin >> info160.students[i].marks[j];
// store the value inside info160:
// in the i'th position of the students array:
// in the j'th position of the marks array for the student
```

# Simpler with abstraction

- the syntax (and maintenance) is cleaner if we have functions to deal with the relevant layers, e.g.

```
void fill(Student &s); // fill one student's info
void fill(Course &c);  // whole course, uses student fill
int main()
{
    Course info160;
    fill(info160);
}
```

# Example continued

```
// fill in a single student's info
void fill(Student &s)
{
   cout << “Enter student name: ”;
   cin >> s.name;

   cout << “Enter number of labs done”;
   cin >> s.numLabs;

   for (int i=0; i<s.numLabs; i++) {
        cout << “Enter the lab mark: “;
        cin >> s.marks[i];
   }
}
```

```
// get info for whole course
void fill(Course &c)
{
    cout << “Enter instructor name: “;
    cin >> c.instructor;

    cout << “Enter the number of students: “;
    cin >> c.numStudents;

    for (int i=0; i<c.numStudents; i++) {
        fill(c.students[i]);
    }
}
```