Expanding Our View Of Structures

Expanding Our View Of structs, Reading

A **struct** is a container for attributes that are associated with objects of that type. In our previous example, name, student ID, and GPA are attributes of a Student.

It's very useful to be able to create such objects, but did you notice that the C++ library's objects have even more useful features -- things like **fin.open** to open a file by name, or **if (a < b)** to compare two string objects alphabetically? These are "actions" and "behaviors" that can also be associated with objects.

This is not an object-oriented programming course so we're not going to get too deep into the C++ syntax for adding actions and behaviors to our objects. We'll cover just what we need to support our study of data structures. Let's start by adding a behavior to our Student objects -- the ability to output themselves, nicely formatted, to the computer console. Our previous example code already has an output function, so we'll change that from being a separate stand-alone void function into a behavior associated with the object.

|  |  |  |
| --- | --- | --- |
| // previous coding struct Student {   string name;   int studentID;   float gpa; };  void **output**(**const** Student&); |  | // new coding struct Student {   string name;   int studentID;   float gpa;   void **output**( ) **const**; }; |

What just happened is a **really big deal**, so let's study the details. First, the *function name* -- it really does not matter what we name the function (as long as it's a valid C++ identifier) but by using the same name "output", we just show that it is the same thing. So it's **bolded**.

But did you notice the other bolded code? The **const**? It seems tiny, but it's going to turn out to be *hugely* important! The parameter also went away, and that's associated with what happened to the const as we'll soon see.

The Host Object

The easiest way to make sense of what just happened is to look at how these functions are called from main.

|  |  |  |
| --- | --- | --- |
| // calling a void function   Student joe;   ...   output(**joe**); |  | // activating an object's behavior   Student joe;   ...   **joe.**output( ); |

It's all about the placement of the object's name, joe. In the standard function call (above and to the left), the object is a *parameter* shared with the function by read-only reference. Using the "behavior", the "dot operator" that's used to access any attribute (like **joe.name**) is also used to access the behavior. So **joe.output**(**);** (note the parentheses) executes that behavior (that is, calls the function).

joe is no longer a parameter in the function call. It needs another description. It's called the "**host object**".

The Trailing const

This is where the const comes in -- in the void function call, the parameter was designated as read-only inside the function. Any attempt to modify the value of any attribute inside the function would result in a compiler error. But with the parameter going away, that const and its meaning also go away. To get it back and apply it to the host object, the C++ syntax is the "trailing const". It's placed after the parentheses and before the semicolon:

  void output( ) **const**; // the *trailing const*, designates the host object as read-only

The Function Name

The new function name is actually *not* "output". Its "full name" is **"Student::output"**. That's how you can tell the difference between the function as a stand-alone and one that's defined in the context of a struct.

Data Members and Member Functions

The struct definition is starting to get a bit crowded now. Up to now we had just "attributes", but now there are functions too. And as we add functions, there will be lots of different kinds of functions -- ones that cannot modify the host objects, ones that *can*, ones that initialize an object, ones that define symbols (like less-than and square brackets), and ones that manage dynamically-allocated memory. We're going to need some names to keep these straight!

To start, we'll separate the attributes from the functions, and refer to these as "**data members**" and "**member functions**". Data members include what we've been calling "attributes" plus some additional values we'll need for various overhead purposes. They are distinguished from member functions by one very simple syntax -- member functions have parentheses, and data members do not.

struct Student  
{  
  string name; // a data member  
  int studentID; // a data member  
  float gpa; // a data member  
  void output**( )** const; // a member function  
};

The **this** Keyword

Member functions need to be defined just like any other function. They are either **written below main** or in a separate file of a multi-file project. There are **two issues** we have to address when converting what we know about writing stand-alone functions to writing these new member functions -- the **function name** and the **"lost" parameter** (now the "host object"):

|  |  |  |
| --- | --- | --- |
| // a stand-alone function void output(const Student& s) {   cout.width(32);   cout << s.name;   cout.width(10);   cout << s.studentID;   cout.setf(ios::fixed);   cout.precision(2);   cout.width(10);   cout << s.gpa << endl; } |  | // a member function, v.1 void **Student::**output( ) **const** {   cout.width(32);   cout << (\***this).**name;   cout.width(10);   cout << (\***this).**studentID;   cout.setf(ios::fixed);   cout.precision(2);   cout.width(10);   cout << **(\*this).**gpa << endl; } |
| // a member function, v.2 void **Student::**output( ) **const** {   cout.width(32);   cout << **this->**name;   cout.width(10);   cout << **this->**studentID;   cout.setf(ios::fixed);   cout.precision(2);   cout.width(10);   cout << **this->**gpa << endl; } |  | // a member function, v.3 void **Student::**output( ) **const** {   cout.width(32);   cout << name;   cout.width(10);   cout << studentID;   cout.setf(ios::fixed);   cout.precision(2);   cout.width(10);   cout << gpa << endl; } |

The above shows three different ways to deal with the "lost" parameter. It was easy in the stand-alone function -- the parameter has a name. But without a parameter, how do we refer to the "host object" that replaces it? There are three equally valid ways to refer to the host.

In C++ the reference to the host object is with a pointer named "**this**". If there's a trailing const for the function, "this" is a read-only pointer. Otherwise not. As a pointer it can be "dereferenced" to access the object -- hence the **(\*this)** in v.1 above. The "infix operator" (or "arrow) simplifies the three operators it takes to use dereferencing (star, parentheses, and dot), so v.2 has **this->** instead. But C++ makes this even easier by letting us leave off any direct reference to the host object, as in v.3. If a variable name is not defined locally in a member function, either as a parameter or in a declaration statement, C++ assumes the variable to be a data member and associates it with the host object. We'll use v.3 mostly.

Just as important as the host object -- note the function name, **Student::output**, and the **trailing const**. If a function is declared with a trailing const in its prototype within a struct, then its definition needs one too!

So applying what we've just studied, here is our new version of the sample from the previous reading:

// identifying comments code block  
// Programmer: *First Last*  
// Programmer's ID: *1234567*  
  
// C++ libraries code block  
#include <iostream>  
#include <string>  
using namespace std;  
  
// C libraries code block  
#include <cstdlib> // for atoi and atof  
  
struct Student  
{  
  string name; // a data member  
  int studentID; // a data member  
  float gpa; // a data member  
  void output( ) const; // a member function  
  void input( ); // a member function  
};  
  
int main( )  
{  
  // identification output code block  
  cout << "Programmer: *First Last*\n";  
  cout << "Programmer's ID: *1234567*\n";  
  cout << "File: " << \_\_FILE\_\_ << endl;  
  
  // create an array of student objects  
  const int CAPACITY = 2;  
  Student x[CAPACITY]; // an array of objects  
  
  // enter data for each student  
  for (int i = 0; i < CAPACITY; i++)  
    x[i].input( );  
  
  // output each student's data  
  for (int i = 0; i < CAPACITY; i++)  
    x[i].output( );  
}  
  
void Student::input( ) // a value-returning function that returns an object  
{  
  cout << "Enter a student's name ";  
  getline(cin, name);  
  
  char buf[100];  
  cout << "Enter " << name <<"'s student ID: ";  
  cin >> buf; studentID = atoi(buf);  
  cin.ignore(1000, 10);  
  
  cout << "Enter " << name << "'s GPA: ";  
  cin >> buf; gpa = atof(buf);  
  cin.ignore(1000, 10);  
}  
  
void Student::output( ) const  
{  
  cout.width(32);  
  cout << name;  
  cout.width(10);  
  cout << studentID;  
  cout.setf(ios::fixed);  
  cout.precision(2);  
  cout.width(10);  
  cout << gpa << endl;  
}

Note that the "Student::input" function is *not* value-returning anymore. Instead it acts upon its host object -- that is, it "mutates" its host -- or at least it has the *ability* to do so. It does *not* have the trailing const that "Student::output" has, because that would restrict its ability to modify (or mutate) the host object.

Study the above code listing and make sure you understand the following features:

1. The "name", "studentID", and "gpa" references in the function definitions that are not declared there, are data members of the host object.
2. The "Student::output" function may not modify the values of any of the data members belonging to its host object -- if it tried, it would result in a compiler error. But "Student::input" can and does.
3. "Student::input" does not create and return a Student object. It doesn't have to because the array declaration in main creates two uninitialized objects, which each become host objects in separate calls in the for-loop.

Make sure you fully understand exactly what's going on in the sample code above before proceeding. Contact your professor and/or use the Q&A section of this module for help, because everything that follows builds on this.