CS161: Introduction to Computer Science I

Week 9 – Structures

12/2020

What is in CS161 today?

fit@hcmus

- ***Structures***
  - What is a structure
  - Why would we use them
  - How do we define structures
  - How do we define variables of structures
  - How do we define arrays of structures
  - How do we pass structures as arguments
- **Other compound types:**
  - Unions
  - Enumerations

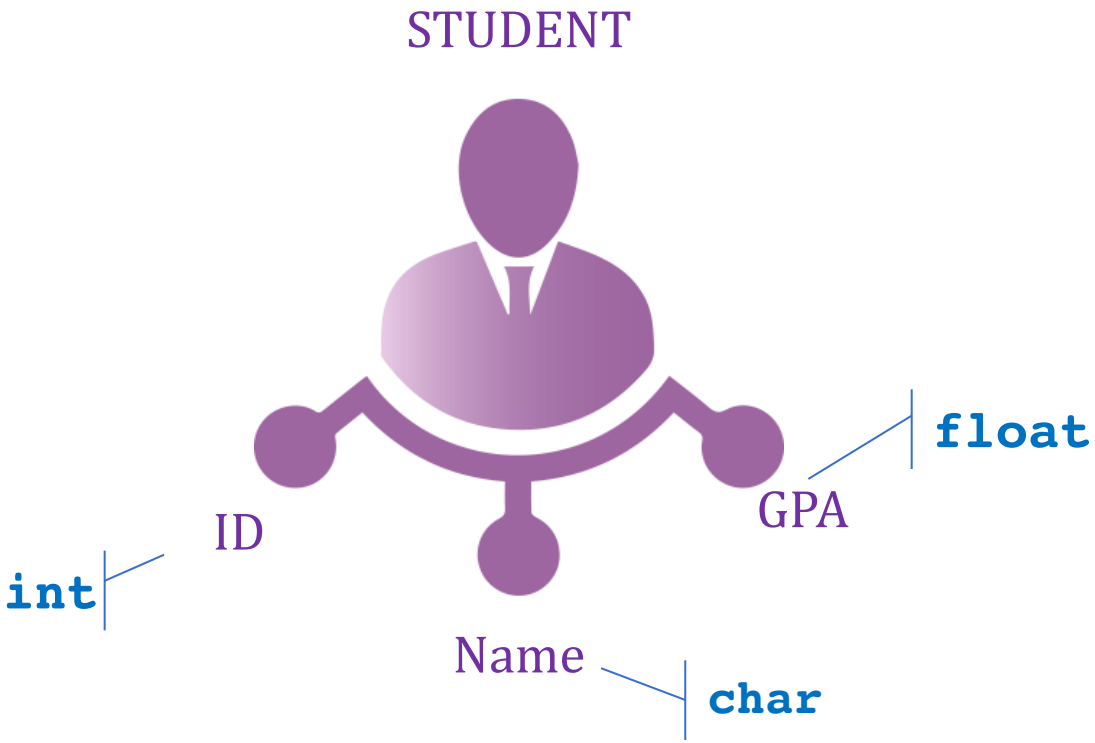
What is a Structure

- With an array, we are limited to having only a single type of data for each element...
  - think of how limiting this would be if we wanted to maintain an inventory
  - we'd need a separate array for each product's name, another for each product's price, and yet another for each barcode!

```
char name[20];
float price;
float cost;
int barcode;
```

What is a Structure

- A structure is a way for us to group different types of data together under a common name



What is a Structure

- With a structure, on the other hand, we can group each of these under a common heading
  - So, if each product can have a description, a price, a cost, and a barcode....a single structure entity can consist of an array of characters for the description, two floats for the price and cost, and an int for the barcode
  - Now, to represent the entire inventory we can have an array of these “products”

Why would we use a Structure

fit@hcmus

- Some people argue that with C++ we no longer need to use the concept of structures
- And, yes, you can do everything that we will be doing with structures, with a “class” (we will learn it in **CS202**)
- My suggestion is to use structures whenever you want to group different types of data together, to help organize your data

How do you define a Structure?

- We typically define structures "globally"
  - this means they are placed outside of the main
- We do this because structures are like a "specification" or a new "data type"
  - which means that we would want all of our functions to have access to this way to group data, and not just limit it to some function by defining it to be local

How do you define a Structure?

- Each component of a structure is called a member and is referenced by a member name (identifier).
- Structures differ from arrays in that members of a structure do not have to be of the same type. And, structure members are not referenced using an index.

How do you define members of a Structure?

- A structure might look like:

```
struct storeitem {
      char item[20];
      float cost;
      float price;
      int barcode;
}; //<-- don't forget the semicolon here
```

In this example, item, price, cost and barcode are member names. **storeitem** is the name of a new derived data type consisting of a character array, two real numbers, and an integer.

How do you define instances of a Structure?

- Once your have declared this new derived data type, you can create instances -- variables (or "objects") which are of this type (just like we are used to):

```
storeitem one_item;
```

- If this is done in a function, then **`one_item`** is a local variable...

How do you initialize a Structure variable?

- You can initialize a structure at the time that it is declared. To give a structure variable a value, follow it by an equal sign and a list of the member values enclosed in braces.
- Example:

```
struct POINT {
      int x;
      int y;
};
POINT p = {5, 7}; //POINT p = {5}; is not an error!
```

- The initializing values must be given in the order that corresponds to the order of member variables in the structure-type definition.

How do you initialize a Structure variable?

- You can initialize each member of the structure to the appropriate kind of data.
- Example:

```
struct INFLATABLE {
      char name[20];
      float volume;
      double price;
};
INFLATABLE guest = {"Gloria", 1.88, 29.99};
```

- The initializing values must be given in the order that corresponds to the order of member variables in the structure-type definition.

How do you define instances of a Structure?

- By saying:

```
storeitem one_item;
```

  - From this statement, **`one_item`** is the variable (or object)
  - We know that we can define a product which will have the members of the item name, the cost, the price, and the bar code.
  - Just think of storeitem as being a type of data which consists of an array of characters, two real numbers, and an integer.

How do you access members of a Structure?

- By saying:

```
storeitem one_item;
```

  - To access a member of a structure variable, we use a dot (the "**direct member access**" operator) after the structure variable's identifier:

```
one_item.item          //an array of chars
one_item.item[0]       //1st character...
one_item.price         //a float
one_item.barcode       //an int
```

How do you access members of a Structure?

- We can work with these members in just the same way that we work with variables of a fundamental type:
- To read in a price, we can say:

```
cin >> one_item.price;
```

- To display the description, we say:

```
cout << one_item.price;
```

What operations can be performed?

- Just like with arrays, there are very few operations that can be performed on a complete structure
- We can't read in an entire structure at one time, or write an entire structure, or use any of the arithmetic operations...
- We can use assignment, to do a "memberwise copy" copying each member from one struct variable to another

How do you define arrays of Structures?

- But, for structures to be meaningful when representing an inventory
  - we may want to use an array of structures
  - where every element represents a different product in the inventory
- For a store of 100 items, we can then define an array of 100 structures:

```
storeitem inventory[100];
```

How do you define arrays of Structures?

- Notice, when we work with arrays of any type OTHER than an array of characters,
  - we don't need to reserve one extra location
  - because the terminating nul doesn't apply to arrays of structures, (or an array of ints, or floats, ...)
  - so, we need to keep track of how many items are actually stored in this array (10, 50, 100?)

How do you define arrays of Structures?

- So, once an array of structures is defined, we

```
storeitem inventory[100];
int inv_count = 0;
//the following code gets the first product's info
cin.get(inventory[inv_count].item, 20);
cin >> inventory[inv_count].price
    >> inventory[inv_count].cost
    >> inventory[inv_count].barcode;
++inv_count; //now inventory has 1 item
```

How do you pass Structures to functions?

- To pass a structure to a function, we must decide whether we want call by reference or call by value
- By reference,
  - we can pass 1 store item:

```
return_type function(storeitem & arg);
```

  - or an array of store items: (*not really a call-by-reference parameter → Why?)*

```
return_type function(storeitem arg[]);
```

- By value, we can pass/return 1 store item:

```
storeitem function(storeitem arg);
```

How do you pass Structures to functions?

fit@hcmus

```
struct POINT{
    int x;
    int y;
};
//Get x and y from user's input
void Input(POINT & p)
{
    cout << "Please input x: ";
    cin >> p.x;
    cout << "Please input y: ";
    cin >> p.y;
}
```

Pass by Reference

How do you pass Structures to functions?

fit@hcmus

```
//Get x and y from user's input
void Input(POINT p[], int n)
{
    for(int i=0; i<n; i++)
    {
      cout << "Please input x of p[" << i <<"]: ";
      cin >> p[i].x;
      cout << "Please input y of p[" << i <<"]: ";
      cin >> p[i].y;
    }
}
```

Pass by Value/Reference?

How do you pass Structures to functions?


```
struct POINT{
    int x;
    int y;
};

//Print out p.x; p.y to the screen
void Show(POINT p)
{
    cout<< "x = " << p.x
        << "; y= " << p.y << endl;
}
```

Pass by Value

How do you pass Structures to functions?


```
//Return a new POINT which has the same y
with p but x is zero
POINT NewPoint(POINT p)
{
    POINT np = p;
    np.x = 0;
    return np;
}
```

Pass by Value

Unions

- A *union* is a data format that can hold different data types but only one type at a time.
- That is, whereas a structure can hold, say, an int ***and*** a long ***and*** a double, a union can hold an int ***or*** a long ***or*** a double.
- The syntax for union is like that for a structure

```
union one4all
{
        int int_val;
        long long_val;
        double double_val;
};//hold an int or a long or a double
```

Unions - Example

fit@hcmus

```
union one4all
{
    int int_val;
    long long_val;
    double double_val;
};// hold an int or a long or a double

one4all pail;
pail.int_val = 15;  // store an int
cout << pail.int_val;

pail.double_val = 1.38;  // store a double, int value is
lost
cout << pail.double_val;
```

Unions

- Because a union holds only one value at a time, it has to have space enough to hold its largest member. Hence, the size of the union is the size of its largest member.
- One use for a union is to save space when a data item can use two or more formats but never simultaneously.

Unions - Example

fit@hcmus

```
struct widget
{
    char brand[20];
    int type;
    union id  // format depends on widget type
    {
        long id_num; // type 1 widgets
        char id_char[20]; // other widgets
    }id_val;
};
...
widget prize;
...
if (prize.type == 1)
    cin >> prize.id_val.id_num;  // use member name to indicate mode
else
    cin >> prize.id_val.id_char;
```

Enumerations

- The C++ *enum* facility provides an alternative to const for creating symbolic constants.
- The syntax for enum resembles structure syntax.

```
enum spectrum{red, orange, yellow, green,
blue, violet, indigo, ultraviolet};
```

  - spectrum: name of a new type (called enumeration)
  - red, orange, yellow, green, blue, violet, indigo, ultraviolet are symbolic constants for the integer values 0-7 (called enumerators)

Enumerations – Setting Values

- By default, enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second enumerator, and so forth. You can override the default by explicitly assigning integer values.

```
enum bits{one = 1,two = 2,four = 4,eight = 8};
enum bigstep{first, second = 100, third};
//first = 0, second = 100, third = 101
enum {zero, null = 0, one, numero_uno = 1};
//zero = null = 0, one = numero_uno = 1
```

Enumerations – Properties

- The only valid values that you can assign to an enumeration variable without a type cast are the *enumerator values used in defining the type.*

```
spectrum band;
band = blue;  //valid, blue is an enumerator
band = spectrum(1);  //valid, spectrum(1) is orange
band = 2000;  //invalid, 2000 is not an enumerator
```

Enumerations – Properties

- Only the assignment operator is defined for

```
band = orange;  // valid
++band;  // not valid
band = orange + red;//not valid
```

- Enumerators are of integer type and can be promoted to type int, but int types are not converted automatically to the enumeration type:

```
int color = blue;  // valid, spectrum type promoted to int
band = 3;  // invalid, int not converted to spectrum
color = 3 + red;  // valid, red converted to int
```

```
struct Employee{
    string emp_id;
    string emp_name;
    string emp_sex;
};

Employee mEmp;
```

Which of the following statements about accessing the members of structure is true?