Structures

CS 1044

Composite Data Types

- We can build **composite data types** by combining simple data types
- We've seen some of these already
- **Arrays** and **vectors** are composite data types made up of a number of the same type of elements

Properties of Arrays/Vectors

- Recall – Arrays and vectors are composite data types with the following properties:
  - **Homogeneous** – every element is the **same** type
  - **Ordered** – elements are referenced by **numerical** position

What if...

- What if we want a composite data type where...
  - The elements are **different** types?
  - The elements are referenced by some other property than numerical position – for example, **by name**

Example

- Let’s model an entry in a phone’s contact list
  - Person’s name
  - Address (street, city, state, ZIP)
  - Phone number (multiple?)
  - Birthday
  - What else?

Bundling Related Data

- What we're talking about is **bundling related data** together
- We don't want names, phone numbers, addresses, etc. all floating around independently – hard to manage
- Need a way to treat as a single entity that can be stored in collections, passed to functions

Structures

- C/C++ provide **structures**, composite data types that are:
  - **Heterogeneous** – elements can be different types
  - Elements are referred to **by name**
  - **Unordered** – beyond the order that elements are listed in your code, it doesn't matter
- We call the elements of a structure **fields**

Defining a Structure

```
struct contact
{
    string name;
    string street_address;
    string city;
    string state;
    int zip_code;
    vector<string> phone_numbers;
};
```

Give a meaningful name to the new data type

List all of the fields – their data types and names

Fields can also be other composite data types, like vectors

Using Structures

```
int main()
{
    contact tony;
    tony.name = "Tony Allevato";
    tony.street_address = "123 Hokie Lane";
    tony.city = "Blacksburg";
    // ...
    tony.phone_numbers.push_back("555-1234");
}
```

Declare a variable using the struct's name as the data type

Use dot-notation to access fields in the structure

If a field is a data type with functions, call them like you normally would

Using Structures in Functions

```
int test(contact c1, contact &c2)
{
    // do something interesting with c1 and c2

    return 0;
}
```

Nesting Structures

```
struct address
{
  string street_address;
  string city;
  string state;
  int zip_code;
};

struct contact
{
  string name;
  address home_address;
  address work_address;
  vector<string> phone_numbers;
};

tony.work_address.street_address = "3160G Torgersen";
```

Pull out the address as a separate type, we may want to use it elsewhere

Now we can easily put multiple addresses in a contact

Structures and Streams

- Structures cannot be automatically read/written with streams, but you can add your own support

`istream`s are like `ifstream`s, but more general (works for cin as well as files)

```
istream& operator>>(istream& stream, my_struct& ms)
{
    stream >> ms.some_field;
    getline(stream, ms.another_field);
    // and so on...
    return stream;
}
```

Use regular input operations to read data into fields

Structures and Streams

Pass by constant reference because output doesn't change the struct

```
ostream& operator<<(
                ostream& stream, const my_struct& ms)
{
    stream << ms.some_field;
    stream << ms.another_field << endl;
    return stream;
}
```

Use regular output operations to write fields to the stream

Documenting a Structure

```
/**
 * Stores contact information about one
 * person.
 */
struct contact
{
    /** The person's first and last name. */
    string name;
    /** The house number and street name. */
    string street_address;
    ...
};
```

The structure itself and all of its fields should have comments