Structures

Lecture 5

Road Map

- Most of the rest of the class is about *object-oriented programming* (OOP)

- A core idea behind OOP is ***encapsulation***: often it is useful to group data together

OOP = Classes

- OOP in C++ is built around the *class*
  - Encapsulates member variables and functions
- To build up to classes, we will start with *structures*, which are historically* simpler
  - Encapsulates variables
- Caveat: in C++, structures and classes are *nearly* identical – however, their typical usage is quite different (more on this later)

Structures

- A **`struct`** is a way of defining a data type composed of **member variables** (sometimes *fields*) that belong together

- Member variables can be primitives (e.g. **`int`**, **`double`**, **`char`**), arrays, or structures/ classes (e.g. **`string`**, **`vector`**)

Structure Basics

1. Structure definition
   - Creates a new variable type!

2. Declare a variable of the type

3. Use member variables

Structure Definition

```
struct <struct name>
{
   <type> <member variable name>;
   <type> <member variable name>;
   …
}; // very important to remember
   // the semi-colon after the
   // close brace
```

Example

```
struct MyDate
{
   string month;
   int day;
   int year;
};
```

Declaring Variables

- Structures are typically defined outside any function (i.e. global scope)

- Once the structure has been defined, you can create variable(s) of that type

```
<struct name> <variable name>;
```

Example

```
#include <iostream>
#include <string>
using namespace std;

struct MyDate
{
	string month;
	int day;
	int year;
};

int main()
{
	MyDate md1, md2;

	cout << "Hello World!" << endl;

	return 0;
}
```

Using Member Variables

- Member variables are accessed with the dot operator

```
<struct variable>.<member variable>
```

- Member variables can be read/changed

Example

```
#include <iostream>
#include <string>
using namespace std;

struct MyDate
{
        string month;
        int day;
        int year;
};

int main()
{
        MyDate md1, md2;

        md1.month = "February";
        md1.day = 1;
        md1.year = 2015;
        md2.month = "February";
        md2.day = 2;
        md2.year = 2015;

        cout << "Today: " << md1.month << " "
                         << md1.day << ", "
                         << md1.year << endl;

        cout << "Tomorrow: " << md2.month << " "
                             << md2.day << ", "
                             << md2.year << endl;

        return 0;
}
```

Exercise

Define a **`struct NumPair`** that has two double fields (**`num1`**, **`num2`**). Create a variable **`x`**, of type **`NumPair`**; get 2 numbers from the keyboard; and initialize the members of **`x`**. Finally, print out the average of the members of **`x`**.

Answer

```
#include <iostream>
using namespace std;

struct NumPair
{
    double num1;
    double num2;
};

int main()
{
    NumPair x;
    cin >> x.num1 >> x.num2;
    cout << ( ( x.num1 + x.num2 ) / 2 ) << endl;

    return 0;
}
```

Hierarchical Structures

Structures can contain member variables that are themselves structures

```
struct WITClass
{
	string dept;
	int num;
};

struct WITStudent
{
	string wnumber;
	WITClass favorite;
	WITClass plan[4];
};

…

WITStudent bob;
bob.wnumber = "11001100";
bob.favorite.dept = "COMP";
bob.favorite.num = 201;
bob.plan[0].dept = "COMP";
bob.plan[0].num = 355;

…
```

Exercise

Define a **`Person`** structure with **`first_name`** and **`last_name`** fields (both of type **`string`**), as well as **`dob`** (of type **`MyDate`**). Then in your **`main`** function, make a variable of type **`Person`**, read in values for these fields from the keyboard, and output a personalized welcome message to the user.

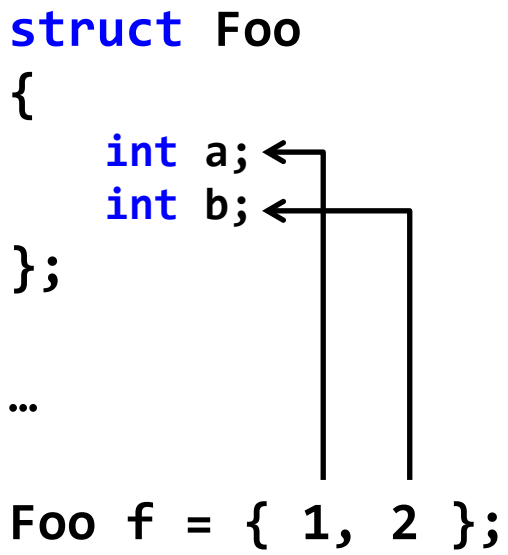
```
struct MyDate
{
    string month;
    int day;
    int year;
};
```

Answer

```
#include <iostream>
#include <string>
using namespace std;

struct MyDate
{
        string month;
        int day;
        int year;
};

struct Person
{
        string first_name;
        string last_name;
        MyDate dob;
};

int main()
{
        Person p;
        cin >> p.first_name >> p.last_name
            >> p.dob.month >> p.dob.day
            >> p.dob.year;

        cout << "Hello " << p.first_name << " "
             << p.last_name << " - " << p.dob.month << " "
             << p.dob.day << " is coming up soon!"
             << endl;

        return 0;
}
```

Initializing Structures

You can declare and initialize a structure variable as follows…



Note: the order of the *initializer list* **must** match the order of the member variables in the structure definition

Example

```
#include <iostream>
#include <string>
using namespace std;

struct MyDate
{
	string month;
	int day;
	int year;
};

struct Person
{
	string first_name;
	string last_name;
	MyDate dob;
};

int main()
{
	Person p = { "bob", "doe", { "february", 14, 1990 } };

	cout << "Hello " << p.first_name << " "
	     << p.last_name << " - " << p.dob.month << " "
	     << p.dob.day << " is coming up soon!"
	     << endl;

	return 0;
}
```

Pointers to Structures

Structures have memory addresses like any other variable

```
MyDate *d_p = &bday; // get the address
```

To access the member variables via a structure pointer, you can dereference the pointer and use the dot operator

```
(*d_p).month = "January"; // change the month
```

OR you can use an equivalent "arrow" shortcut

```
d_p->month = "January"; // change the month
```

Example

```
#include <iostream>
using namespace std;

struct MyDate
{
	string month;
	int day;
	int year;
};

int main()
{
	MyDate bday = { "December", 25, 1982 };
	cout << bday.month << " " << bday.day
	     << ", " << bday.year << endl;

	MyDate *d_p = &bday;
	cout << (*d_p).month << " " << (*d_p).day
	       << ", " << (*d_p).year << endl;

	d_p->month = "January";
	(*d_p).year = 1983;
	cout << d_p->month << " " << d_p->day
	     << ", " << d_p->year << endl;

	cout << bday.month << " " << bday.day
	     << ", " << bday.year << endl;

	return 0;
}
```

```
December 25, 1982
```

```
December 25, 1982
```

```
January 25, 1983
```

```
January 25, 1983
```

Exercise

Consider the following structure definition and variable initialization:

```
struct StudentInfo
{
    char name[10];
    double gpa;
};

StudentInfo s[] = {
    {"Alice",3.8}, {"Bob",3.6},{"Cathy",3.9},{"Dylan",3.8}
};
```

Write a program to output the average GPA, as well as the names of the students with the lowest and highest GPAs.

Answer

```
#include <iostream>
using namespace std;

struct StudentInfo
{
        char name[10];
        double gpa;
};

int main()
{
        StudentInfo s[] = {
                {"Alice",3.8},{"Bob",3.6},{"Cathy",3.9},{"Dylan",3.8}
        };
        StudentInfo *min_st = &s[0], *max_st = &s[0];

        int num_students = 4;
        double sum_gpa = s[0].gpa;

        for ( int i=1; i<num_students; i++ )
        {
                sum_gpa += s[i].gpa;

                if ( s[i].gpa < min_st->gpa )
                        min_st = &s[i];

                if ( s[i].gpa > max_st->gpa )
                        max_st = &s[i];
        }

        cout << "Avg: " << ( sum_gpa / num_students ) << endl;
        cout << "Min: " << min_st->name << " (" << min_st->gpa << ")" << endl;
        cout << "Max: " << max_st->name << " (" << max_st->gpa << ")" << endl;

        return 0;
}
```

Reusing Member Names

Two or more structures may use the same member names

```
struct NumPair
{
   double num1;
   double num2;
};

struct NumTriple
{
   double num1;
   double num2;
   double num3;
};
```

Structure Assignment

You can use the = operator to copy the values of all member variables from one structure [of a type] to another

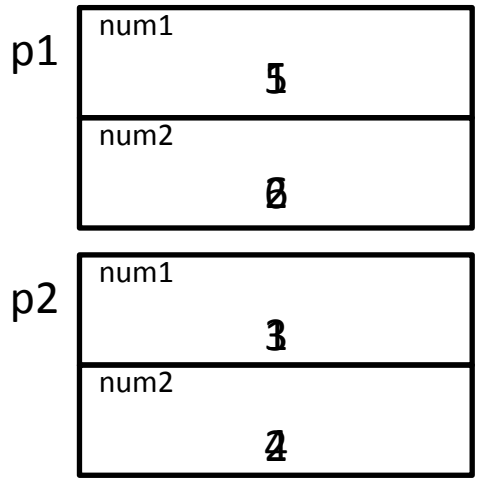
```
NumPair p1 = { 1, 2 }, p2;
p2 = p1;
```

This is equivalent to assigning all the individual member variables separately

```
p2.num1 = p1.num1;
p2.num2 = p1.num2;
```

Example

```
#include <iostream>
using namespace std;

struct NumPair
{
	double num1;
	double num2;
};

int main()
{
	NumPair p1 = { 1, 2 }, p2 = { 3, 4 };

	cout << p1.num1 << " " << p1.num2 << endl
		<< p2.num1 << " " << p2.num2 << endl;

	p2 = p1;
	p1.num1 = 5;
	p1.num2 = 6;

	cout << p1.num1 << " " << p1.num2 << endl
		<< p2.num1 << " " << p2.num2 << endl;

	return 0;
}
```




```
1 2
3 4


5 6
1 2
```

Structures as Arguments

- A function can have parameter(s) of a structure type that are call-by-value and/or call-by-reference

```
void outputPair(NumPair p);
void outputPair(NumPair& p);
```

- Call-by-value will copy all the member variables, whereas call-by-reference will essentially user a pointer
- Thus, call-by-value can become very expensive for large structures (i.e. those with many/big members)
- BUT call-by-reference allows the function to *change* the contents of the structure!

The `const` Modifier (take 2)

- So far we have used **`const`** to make constant variables

```
const double pi = 3.14159;
```

- The **`const`** modifier can also be used with function parameters (typically used with call-by-reference) to promise that the function *will NOT* change the parameter

Example

```
#include <iostream>
using namespace std;

const double PI = 3.14159;
const double E = 2.71828;

struct NumPair
{
        double num1;
        double num2;
};

void outputPair(const NumPair& p)
{
        cout << p.num1 << " " << p.num2 << endl;
}

void changePair1(NumPair p)
{
        p.num1 = 1;
}

void changePair2(NumPair& p)
{
        p.num1 = 2;
}

void changePair3(const NumPair& p)
{
        p.num1 = 3;
}
```

```
int main()
{
        NumPair pair = { PI, E };
        outputPair( pair );

        changePair1( pair );
        outputPair( pair );

        changePair2( pair );
        outputPair( pair );

        changePair3( pair );
        outputPair( pair );

        return 0;
}
```

Compile ERROR

```
In function 'void changePair3(const NumPair&)':
error: assignment of member 'NumPair::num1' in read-only object
  p.num1 = 3;
```

Example

```
#include <iostream>
using namespace std;

const double PI = 3.14159;
const double E = 2.71828;

struct NumPair
{
        double num1;
        double num2;
};

void outputPair(const NumPair& p)
{
        cout << p.num1 << " " << p.num2 << endl;
}

void changePair1(NumPair p)
{
        p.num1 = 1;
}

void changePair2(NumPair& p)
{
        p.num1 = 2;
}

// void changePair3(const NumPair& p)
// {
//      p.num1 = 3;
// }
```

```
int main()
{
        NumPair pair = { PI, E };
        outputPair( pair );

        changePair1( pair );
        outputPair( pair );

        changePair2( pair );
        outputPair( pair );

        // changePair3( pair );
        // outputPair( pair );

        return 0;
}
```

```
3.14159 2.71828
3.14159 2.71828
2 2.71828
```

Structures as Return Types

- Functions can *return* a structure type

```
NumPair makePair();
```

- Typical process
  - The function declares a structure variable
  - The function sets the members
  - The function returns the structure

Example

```
#include <iostream>
using namespace std;

struct NumPair
{
        double num1;
        double num2;
};

void outputPair(const NumPair& p)
{
        cout << p.num1 << " " << p.num2 << endl;
}

NumPair multPair(const NumPair& p, double factor)
{
        NumPair product = p;
        product.num1 *= factor;
        product.num2 *= factor;
        return product;
}

int main()
{
        NumPair pair1 = { 1, 2 };
        outputPair( pair1 );

        NumPair pair2 = multPair( pair1, 2.0 );
        outputPair( pair1 );
        cout << "x 2.0 = " << endl;
        outputPair( pair2 );

        return 0;
}
```

```
1 2
```

```
1 2
x 2.0
2 4
```

Exercise

Given the following definition of **`MyDate`**, write the function **`makeDate`** that takes as parameters **`month`** (**`string`**; by reference, but should not change), **`day`** (**`int`**), and **`year`** (**`int`**) and returns a **`MyDate`** structure.

```
struct MyDate
{
    string month;
    int day;
    int year;
};
```

Answer

```
MyDate makeDate(const string& month,
                int day, int year)
{
  MyDate d = { month, day, year };
  return d;
}
```

Wrap Up

- A structure (`struct`) is a collection of related variables (*members*)
  - Members can be different types, including primitives, arrays, and/or other structures/classes
- The members are accessed using the dot (`.`) operator: **`<struct variable>.<member>`**
- Structures can be used as function parameters and return types
  - Use **`const <struct type>&`** for efficiency and change protection