Programming Studio #12

ECE 190

Programming Studio #12

- Topics this week:
  - Basic Data Structures
  - Enumerations
  - Dynamic Memory Allocation
- In Studio Assignment
  - Complex Number Adder

Announcements

- MP 5
  - Challenging, so start early.
  - Checkpoint 1 Due date extended by 1 week
- Exam 2
  - Regrade requests due today.
- Final Exam
  - Monday May 10th – 1:30 – 4:30
  - Conflict Exam: Friday May 7th – 1:30 – 4:30 (see website to see if you are eligible for a conflict exam). **Notify us by April 30th**

Cheating

- Reminder: We do not tolerate any form of cheating.
  - Sharing code is cheating
  - Sharing code and renaming variables is still cheating
  - Sharing code and adding/removing spaces/tabs is still cheating
- We have identified multiple cheaters on MP3 and MP4…guilty parties will be notified soon

Data Structures

- What are the basic data types in C?
  - Integer, float, double, character
- What about data in the real world?
  - Also integer, float, double, characters
  - Data is often organized into groups
    - Vectors: direction, magnitude
    - Complex numbers: real part, complex part
    - Student record: last name, first name, UIN, GPA, standing…

Structures

- Grouping of primitive data types
- Student Record Example

Student Record Structure

First Name (char[])
Last Name (char[])
netID (char[])
UIN (unsigned int)
GPA (float)

Student 1

John
Doe
doe1
123456789
3.99

Student 2

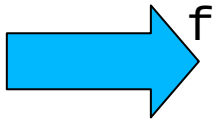
Sarah
Smith
Smith1
987654321
4.00

Structures

- We must first define the customized data type (structure)

Student Record Structure

First Name (char[])
Last Name (char[])
netID (char[])
UIN (unsigned int)
GPA (float)



```
struct student_record {
        char
first_name[12];
        char last_name[12];
        char netid[8];
        unsigned int uin;
        float gpa;
};
```

- We can then create variables whose type is the structure

```
int main(){
        struct student_record student_1;
        student_1.first_name = ``John``;
        student_1.last_name = ''Doe'';
        student_1.uin = 123456789;
                        ...
}
```

Structures

- Memory is allocated the same way as for any variable
  - Locals are allocated on the run-time stack and globals are allocated on the global data section
  - Structures occupy contiguous regions of memory blocks. Each block takes as much memory as the sum of all of the member elements
    - How much space does each student entry take up?
- Structure arrays can also be created:

```
int main(){
        struct student_record students[100];
        student[0].first_name = ``John``;
        student[0].last_name = ‘’Doe’’;
        student[0].uin = 123456789;
                        ...
}
```

Structures and Pointers

- Since structures are handled the same way as normal data types in C, pointers to structures can be created

```
int main(){
        struct student_record
students[100];
        struct student_record *st_ptr;
        st_ptr = &students[3];
```

- The pointers can be dereferenced and accessed

```
(*st_ptr).gpa = 3.25;
```

- Another way of accessing the values stored in pointers

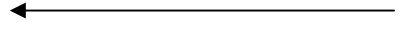
```
st_ptr->gpa = 3.25;
```

- The '->' expression is like the dereference operator '*' except it is used for dereferencing member elements of a structure

Enumerations

- What if your data types should contain certain values?
  - Storing multiple choice answers (A,B,C,D)
  - Opcodes in an assembly language
- You could just use an integer value to represent each of these things, but it is difficult to read for humans?
  - Which opcode is this? if(op == 2)
- Solution: create a list where each number is assigned a human readable representation
  - This is much easier: if(op == LD)
- Enumerated list:

```
enum opcode {BR, ADD, LD};
enum opcode op;
op = 1;
if(op == LD)
```



Is this if statement true?

Dynamic Memory Allocation

- So far we know how to allocate memory using arrays
  - Limitation: we can only allocate a constant amount of elements
  - This amount must be known at compile time (specified in the source code)
  - Waste of memory, or danger of not allocating enough space to store all data
- What if we want to allocate during run-time instead of compile time?

Dynamic Memory Allocation

- Use function "malloc(size)" to allocate memory at run time
  - Size is a parameter that tells malloc how many bytes to allocate
  - Memory is allocated in the heap
- Dynamic memory must be freed manually using "free(<pointer>)"
  - <pointer> is a pointer to the memory block that needs to be freed
  - Without freeing no other programs/variables may use this memory block (we can run out of memory)

```
int main(){
        struct student_record *students;
        int number;
        printf(``How many students?``);
        scanf(``%d``,&number);
        students = (student_record*) malloc(number * sizeof(student_record));
        /* students is now a pointer to the 0th element of the 'students'
block */
        students[0].first_name = ``John``;  /*  behaves just like an array */
        students->last_name = ''Does'';

        /* Code that initializes all student records */

        free(students);
}
```

Programming Studio Assignment

- Complex number adder
  - Create a structure for a complex number (float real_part, float complex_part);
  - Ask the user how many complex numbers will be entered
  - Dynamically allocate enough space to hold all the complex numbers
  - Ask the user to enter each number (first ask for the real part, then ask for the complex part)
  - Add all the complex numbers together:
    - The real_part of the sum is just the sum of the real parts of all the numbers
    - The complex_part of the sum is just the sum of the complex parts of all the numbers
  - Print out the answer in the format:
    "Sum = <real_part> + j <complex_part>"
  - Remember to free the memory after you're done.