CSCI 135
Structures

Structs

Recall: an array is a homogeneous fixed-size collection of data with efficient random access.

A **structure** is a *heterogenous* collection of data with *access by field name* – *i.e.*, a struct groups together data of different types, so that they can be manipulated as a whole.
Each element of a structure is called a *field* or *member*.

BUT since a structure is accessed by field name, its format needs to be explicitly defined (unlike for arrays) before declaring variables of that type.

Examples:

- Year – Month – Day
- Name – ID – Salary

Definition of Structure Types

struct <typeName> {<fields>} <variables>;

where typeName is an optional identifier naming the new struct type (called a **tag**), and variables is an optional list of identifiers.

```
struct Date                 Date is like a type name
{
   int year;                year is name of a field
   int month;
   int day;
};
struct Employee             uppercase, in C++ (not C) style
{
   string name;             some prefer field names to lineup
   int    id;
   double salary;
} empl1,empl2;              also declaring empl1,empl2 vars
Date mydate;                mydate is a variable
```

⚠ Date **defines** what the type looks like – it does not **declare** any variables or allocate any memory for any such variables (mydate is later declared and allocated)

Accessing Structure Fields

Dot operator: selects fields in a struct

```
struct Date                          Struct defs often in global scope
{
   int year;
   int month;                        Or enum from jan..dec
   int day;
};                                   Don't forget the semicolon
int main() {
   Date mydate={2001,1,2};           Initializer (jan 02, 2001)
   struct Date someDate;             Alternate declaration syntax
   ...
   mydate.year = 2015;
   mydate.month = 1;
   mydate.day = 1;
}
```

? What if you didn't have the semicolon after the struct definition

Accessing Structure Fields

Dot operator: selects fields in a struct

```
struct Date                          Struct defs often in global scope
{
   int year;
   int month;                        Or enum from jan..dec
   int day;
};                                   Don't forget the semicolon
int main() {
   Date mydate={2001,1,2};           Initializer (jan 02, 2001)
   struct Date someDate;             Alternate declaration syntax
   ...
   mydate.year = 2015;
   mydate.month = 1;
   mydate.day = 1;
}
```

? What if you didn't have the semicolon after the struct definition

! Next program token is the name of a variable of type Date

Example - Compute Age

```
// Assume Date's definition is in scope
int age(Date today, Date birthdate) {
   myage=today.year-birthdate.year;
   if ((today.month < birthdate.month) ||
        ((today.month == birthdate.month) &&
         (today.day < birthdate.day)))
     myage --;
   return (myage);
}
```

- Entire date treated as single variable!

Some Structure Features

- ☺ Can assign entire struct with one statement (unlike arrays)
- ☺ Passed/returned like any other simple type by value or by reference
- ☹ Can not compare entire structs (*i.e.*, no ==)

```
Date tomorrow(Date today) { function named tomorrow
  int lastday(int month);
  Date tempdate = today;
  if (today.day == lastday(today.month))
    tempdate.day = 1
  else tempdate.day = tempdate.day+1;
  if (tempdate.day == 1) {
    tempdate.month = (tempdate.month + 1);
    if (tempdate.month == 13) tempdate.month=1;   };
  if ((tempdate.month == 1) && (tempdate.day == 1))
    tempdate.year = tempdate.year+1;
  return tempdate;                    not a primitive return val!
};
```

Some Structure Features

- ☺ Can assign entire struct with one statement (unlike arrays)
- ☺ Passed/returned like any other simple type by value or by reference
- ☹ Can not compare entire structs (*i.e.*, no ==)

```
Date tomorrow(Date today) { function named tomorrow
  int lastday(int month);
  Date tempdate = today;
  if (today.day == lastday(today.month))
    tempdate.day = 1
  else tempdate.day = tempdate.day+1;
  if (tempdate.day == 1) {
    tempdate.month = (tempdate.month + 1);
    if (tempdate.month == 13) tempdate.month=1;   };
  if ((tempdate.month == 1) && (tempdate.day == 1))
    tempdate.year = tempdate.year+1;
  return tempdate;                    not a primitive return val!
};
```


Leap years! What needs changing?

Some Structure Features

- ☺ Can assign entire struct with one statement (unlike arrays)
- ☺ Passed/returned like any other simple type by value or by reference
- ☹ Can not compare entire structs (*i.e.*, no ==)

```
Date tomorrow(Date today) { function named tomorrow
  int lastday(int month);
  Date tempdate = today;
  if (today.day == lastday(today.month))
    tempdate.day = 1
  else tempdate.day = tempdate.day+1;
  if (tempdate.day == 1) {
    tempdate.month = (tempdate.month + 1);
    if (tempdate.month == 13) tempdate.month=1;   };
  if ((tempdate.month == 1) && (tempdate.day == 1))
    tempdate.year = tempdate.year+1;
  return tempdate;                    not a primitive return val!
};
```

🐞 Leap years! What needs changing?

⚠ Add year argument to lastday (and look up leap year rules!)

Dot Operator Revisited

Recall: We used the dot operator earlier with strings. What is the difference when used with structs?

```
typedef string MsgType;
MsgType msg = "hi mom!";
cout << msg.length();
```

i.e., execute member function `length` on a variable/object:

- Assuming that member function (`length`) is defined on that type (`MsgType` $\equiv$ `string`)
- Careful, function is called on the variable/object, not the type

```
struct Date {...};
Date someday;
cout << someday.month;
```

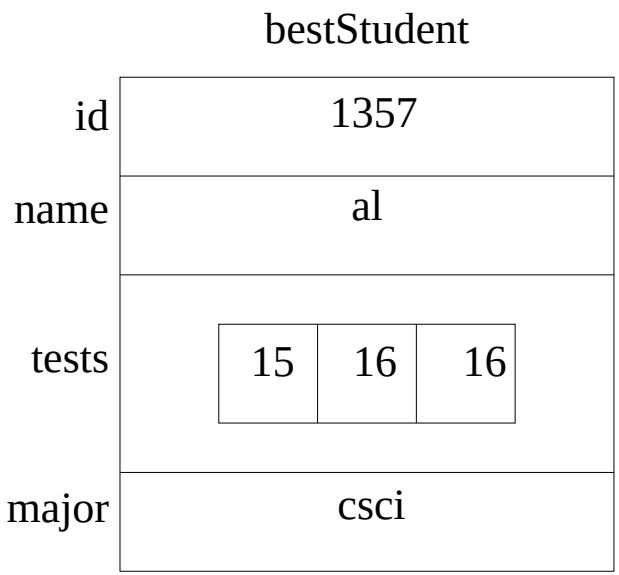
i.e., access field/member `month` of structure:

- Assuming that the field/member (`month`) is defined for that structure definition (Date)
- Careful, field is associated with the variable, not the type.

Structure Fields/Members - 1

Fields/Members can be any type (including ints, arrays, other structs, ...)!

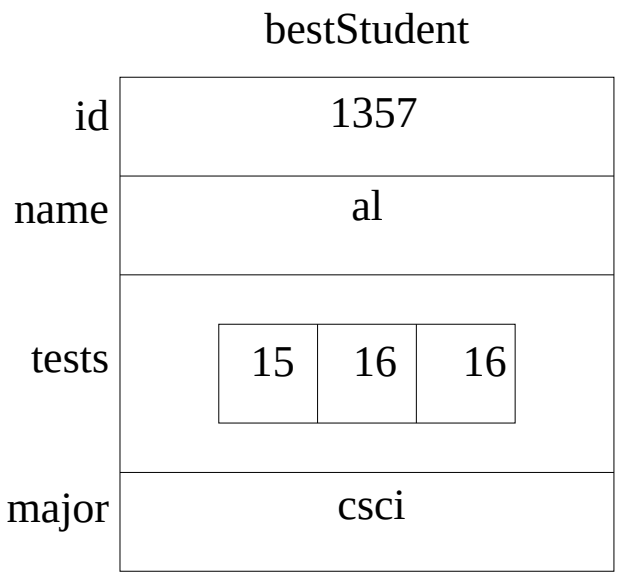
Example: Add 1 to bestStudent's Test 0 grade



Structure Fields/Members - 1

Fields/Members can be any type (including ints, arrays, other structs, ...)!

Example: Add 1 to bestStudent's Test 0 grade




```
typedef enum {csci,math,engl} Major;
struct Student {
  int id;
  string name;
  int tests[3]; Field is itself a composite!
  Major major;  Type Major, variable major
} bestStudent =
    {1357,"al",{12,16,16},csci};
bestStudent.tests[0] = 14;
(bestStudent.tests[0])++;
```

Structure Fields/Members - 2

Same as before

```
typedef enum {csci,math,english} Major;
struct Student {
  int id;
  string name;
  int tests[3];
  Major major;
};
```

Add 1 point to csci majors' test 1 grades:

```
const int MAX_CLASS_SIZE = 256;
Student students[MAX_CLASS_SIZE];
for (int i = 0; i<MAX_CLASS_SIZE; i++)
  if (students[i].major == csci)
    (students[i].tests[1])++;
```

Structure Fields/Members - 3

Same as before

```
typedef enum {csci,math,english} Major;
struct Student {
  int     id;
  string  name;
  int     tests[3];
  Major   major;
};
const int MAX_CLASS_SIZE = 256;
```

```
struct Section {
   Major    department;
   string   instructorName;
   int      numStuds;
   Student  students[MAX_CLASS_SIZE];
} cs135eve;
int sum = 0; double test0avg;
for (int i=0; i<cs135eve.numStuds; i++)
  sum += cs135eve.students[i].tests[0];
test0avg = sum / cs135eve.numStuds;
```

Structure Fields/Members - 3

Same as before

```
typedef enum {csci,math,english} Major;
struct Student {
  int     id;
  string  name;
  int     tests[3];
  Major   major;
};
const int MAX_CLASS_SIZE = 256;
```

```
struct Section {
   Major    department;
   string   instructorName;
   int      numStuds;
   Student  students[MAX_CLASS_SIZE];
} cs135eve;
int sum = 0; double test0avg;
for (int i=0; i<cs135eve.numStuds; i++)
   sum += cs135eve.students[i].tests[0];
test0avg = sum / cs135eve.numStuds;
```

⚠ better to encapsulate MAX_CLASS_SIZE inside Section.

Example: Estimate π

Approach: generate random points in a 1×1 square and check how many of them fall in NE quadrant of radius-1 circle to estimate $\frac{\pi}{4}$

```
struct Point {
  double x,y;    Alternate syntax for fields
};
Point randomPoint() {
  // Precond: PRG has been seeded
  Point temp;
  temp.x = (double) rand() / (double) INT_MAX;
  temp.y = (double) rand() / (double) INT_MAX;
  return temp;
};
int main() {
  int numPoints = 10000; int circPts = 0;
  Point randomPoint();     Point pt;
  srand(time(0));          // seed PRG
  for (int i = 0; i<numPoints; i++) {
    pt = randomPoint();
    if (pt.x*pt.x + pt.y*pt.y < 1) circPts++;
  };
  cout << (double) circPts / (double) numPoints * 4. << endl;
}
```

- This is called a Monte-Carlo method

Other Struct Related Issues

- Structure padding (when interfacing with embedded systems)
- Unions (to reduce memory usage)
- Memory allocation (we will talk about later)

Some Software Engineering Styles/Guidelines

- Write pre/post conditions for every function and/or block of code. (possibly written as asserts)
- Modularize your code into functions well. Make sure you have the right input (arguments) and output (return value) from every function.
- Think about loop invariants for complicated loops, and the code writes itself.
- Design your data structures first, and the code writes itself. And pay attention to encapsulation (you will thank yourself later when debugging/maintaining!)

⚠ These guidelines are sometimes self-contradictory!

Some Exercises

1. Using the above data structure definition, compare the test0 average of all csci, math, and english students. Print the averages, ordered by major.
2. Generalize the above to many (say, 64) majors. *i.e.*, need to use a sorting algorithm on averages, instead of brute-forcing all 3!=6 orderings with 3 majors.
3. Modify the Section definition above to allow arbitrarily large class sizes. Write a function `getTestAvg(Section sec, int testNum)`.
4. Design a data structure (using tag Transcript) to represent a student transcript, using the data types we have covered so far.
5. Write a function to print the name of every student with gpa>3.5. To complicate things, assume that the gpa must be computed for each student (*i.e.*, it isn't precomputed and stored as a field in the Transcript structure).