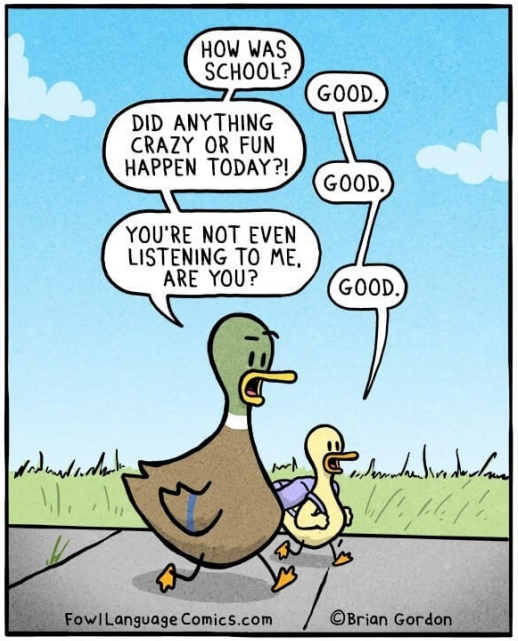
Foundation of Computer Science with C++

Dr Kafi Rahman, PhD
Email: kafi@truman.edu
Text: 605-270-7645
Truman State University

Learning from a Quiz

One question at a time



Quiz of the Day (1 of 5)

- In this course, at minimum, we are going to use which of the following compiler?
  - C++11 with clang (llvm)
  - C++11 with gcc
  - C++14 with code::blocks

Quiz of the Day (2 of 5)

- Which of the following is not an example of computer Hardware?
  - computer monitor
  - computer printer
  - keyboard
  - computer chair

Quiz of the Day (3 of 5)

- Which of the following is not a Characteristics of a Truman Student?
  - Ask questions and passionately seek knowledge
  - Welcome and value new and diverse perspectives
  - Take generously money, gold, and financial resources and spend them in computer games.
  - Live emotionally and physically healthy lives

Quiz of the Day (4 of 5)

- Which of the following is something you should avoid as a Truman student?
  - discuss the mechanics of editing, compiling, and running a program
  - discuss the details of any assignment, in person or electronically with other students
  - discuss the mechanics of using the codeBlocks IDE
  - use code from the instructor or the course book without reference

Quiz of the Day (5 of 5)

- Which of the following is something you should do as a Truman student?
  - look at any portion of another student's code (to solve assignment) or writeup (during the exam)
  - show any portion of your code or writeup to another student
  - discuss the general strategy for completing an assignment
  - copy assignment solutions from any sources whenever possible including from the instructor and the course book

Do you have any questions?

Foundation over CS180

- I will be assuming the following:
  - Familiar to work with all primitive C++ data types including string, and vectors
  - Capable of defining functions and divide a program in subunits
  - Has proper understanding of structure types in C++
  - Knows necessary syntax to pass variables by reference including a vector of structures
- Thats a lot of assumptions!
  - Let us talk about an interesting topic that we know.

Data Abstraction

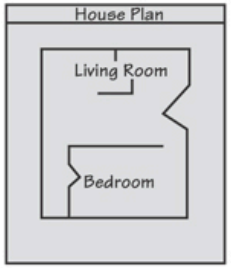
- an abstraction is a model
- an abstraction defines the common characteristics of some thing by using
  - a set of attributes, and
  - a set of behaviors

class
<CAR>

Data Abstraction (cont)

- Data Abstraction provides a blueprint and many things can be built from that blueprint

Blueprint that describes a house.



Instances of the house described by the blueprint.

Data Abstraction (cont)

- Common use for data abstraction is to group together a set of information that describes a person or a thing.
  - a thing like a cat

Abstract Data Type

- ADT is defined by the programmer
  - it has one or more data fields which may be primitive data types

Abstract Data Type (cont)

- Suppose we are writing a program that calculates time
  - In our program, we need to store time
  - Hence, we can adopt a Time ADT

- Now, The data fields of the Time ADT might be
  - a field name hour (values from 1 to 12)
  - a field named minute (values from 0 to 59)
  - a field named second (values from 0 to 59)

C++ Structures

- the primary C++ mechanism for building ADTs is the struct
- imagine you wish to build a system for maintaining information about movies
- you might define a Movie structure like this:

```
1  struct Movie
2  {
3    string title;
4    string director;
5    unsigned year_released;
6    double running_time;
7  };
```

C++ Structures (cont)

- the data fields are attributes of the structure
- the closing curly brace is followed by a semicolon

```
1  struct Movie
2  {
3    string title;
4    string director;
5    unsigned year_released;
6    double running_time;
7  };
```

A Structure Variable

- a struct is a template or a blueprint for a ADT (composite) variable
- this struct has four fields
- a struct can be used as a type
- hence, we can declare a variable of this type, using the structure name as the type name:

```
Movie movie {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

Initializing a Struct Variable

- a struct variable can be initialized when it is declared by filling all the fields in order (note no assignment operator)

```
Movie movie {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

Initializing a Struct Variable (cont)

- or by assigning them one-by-one after declaration:

```
Movie movie;
movie.director = "Chris Columbus";
movie.year_released = 2001;
movie.title = "Harry Potter";
movie.running_time = 2.53;
```

Vectors of Structs

- it is perfectly legal to have a single struct variable
- but the real power of structs comes with collections of structs i.e., arrays or vectors of structs
- a vector of structs is created just like any vector
- first, we need the struct ADT definition

```
1  struct Movie
2  {
3    string title;
4    string director;
5    unsigned year_released;
6    double running_time;
7  };
```

Printing Struct Variables

- we cannot output a struct variable like this: cout << movie_x;
- we could write a function print movie, but it is much better conceptually to write this function:

```
14  int main()
15  {
16    string title = "Harry Potter";
17    string director_name = "Director";
18    unsigned release_date = 2001;
19    float running_time = 1.20;
20
21    unsigned hours = static_cast<unsigned> (running_time);
22    unsigned minutes = static_cast<unsigned> ((running_time-hours) * 60.0);
23
24    string output = title + "; " + director_name;
25    output += " (" + to_string(release_date) +") ";
26    output += to_string(hours) + " hr " + to_string(minutes) + " min";
27
28    cout<<output;
29  }
30
31  Display: Harry Potter; Director (2001) 1 hr 12 min
```

Printing Struct Variables

- we cannot output a struct variable like this: cout << movie_x;
- we could write a function print movie, but it is much better conceptually to write this function:

```
1  string to_string(const Movie& m)
2  {
3    string result = m.title + "; " + m.director +
4      " (" + to_string(m.year_released) + ") ";
5
6    unsigned hours = static_cast<unsigned>(m.running_time);
7    unsigned minutes = static_cast<unsigned>((m.running_time - hours) * 60.0);
8
9    result += to_string(hours) + " hr " + to_string(minutes) + " min";
10   return result;
11 }
```

Printing Struct Variables

- we cannot output a struct variable like this: cout << movie_x;
- we could write a function print movie, but it is much better conceptually to write this function:

- Movie mov {"Psycho", "Hitchcock", 1960, 1.82};
- cout<< to_string(mov);

```
1  string to_string(const Movie& m)
2  {
3    string result = m.title + "; " + m.director +
4      " (" + to_string(m.year_released) + ") ";
5
6    unsigned hours = static_cast<unsigned>(m.running_time);
7    unsigned minutes = static_cast<unsigned>((m.running_time - hours) * 60.0);
8
9    result += to_string(hours) + " hr " + to_string(minutes) + " min";
10   return result;
11 }
12
13 Output: Psycho; Hitchcock (1960) 1 hr 49 min
```

Nested Structures

- a member can be of any data type
- including a programmer-defined struct ADT

```
1  // Defining a Time Structure
2  struct Time
3  { unsigned hour;
4    unsigned minute;
5  };
6
7  // Movie structure uses Time ADT
8  struct Movie
9  { string title;
10   string director;
11   unsigned year_released;
12   Time running_time; //ADT
13 }
```

Nested Structures

- nested members can be initialized via nested initializer lists
- nested members are accessed via nested dots

```
1  Movie movie_x {"Psycho", "Hitchcock", 1960, {1, 49}};
2
3  Movie movie_y;
4  movie_y.title = "Vertigo";
5  movie_y.running_time.hour = 2;
6  movie_y.running_time.minute = 8;
```

Nested Structures

- the to string function now becomes:

```
1  // display the struct as string
2  string to_string(const Movie & m)
3  { string result = m.title + "; " + m.director +
4        " (" + to_string(m.year_released) + ") " +
5        to_string(m.running_time.hour) + " hr " +
6        to_string(m.running_time.minute) + " min";
7
8    return result;
9  }
```

Pointers to Structure Variables

- a pointer variable can point to a structure location in memory

```
Movie movie {"Psycho", "Hitchcock", 1960, {1, 49}};
Movie * mptr = &movie;
```

- immediately there is a problem, however to access a member, we use the dot operator after dereferencing the pointer:
  - but this doesn't work, because the precedence of dot is higher than the precedence of dereference

```
cout << *mptr.title;
```

Pointers to Structure Variables

- instead we have to do this:

```
Movie movie {"Psycho", "Hitchcock", 1960, {1, 49}};
Movie* mptr = &movie;
cout << (*mptr).title;
```

- this syntax is required in C, and works in C++, but is considered awkward

- instead C++ uses the dereference-then-select operator `->`

```
cout << mptr->title;
```

- this operator means: first dereference mptr, then go to the title field of the thing mptr is pointing to, and print that