Structured Data

Class 37

Abstraction

- an abstraction is a model
- an abstraction defines the common characteristics of some thing
- an abstraction attempts to capture the essence of that thing

Abstraction

- an abstraction is a model
- an abstraction defines the common characteristics of some thing
- an abstraction attempts to capture the essence of that thing

- "chair" is an abstraction — it applies to all chairs
- you may be sitting in a specific chair, and up here is a different chair
- these two chairs are instances of the abstraction chair

Data Type

from the slides on 28 August:

Data Type

A data type is a set of values and a set of operations defined on those values.

- C++ has a number of built-in, or primitive data types
- bool, char, unsigned long long, etc.

Abstract Data Type

- put together the concepts of abstraction and data types and you get the notion of an abstract data type (ADT)
- an ADT is defined by the programmer
- it has one or more data fields which are primitive data types
- the programmer decides what operations may be performed on instances of the ADT

Abstract Data Type

- put together the concepts of abstraction and data types and you get the notion of an abstract data type (ADT)
- an ADT is defined by the programmer
- it has one or more data fields which are primitive data types
- the programmer decides what operations may be performed on instances of the ADT

- for example, suppose you are writing a program that needs to simulate timekeeping with a 24-hour clock
- the data fields of the Clock ADT might be
  - hours, a field that can take on values from 0 to 23
  - minutes, a field that can take on values from 0 to 59
  - seconds, a field that can take on values from 0 to 59
- and the operations might involve adding and subtracting time values with correct carries, and comparing them

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:

```
struct Movie
{
  string title;
  string director;
  unsigned year_released;
  double running_time;
};
```

- the struct tag name Movie starts with an Uppercase letter
- the data fields are variables of types that already exist
- the closing curly brace is followed by a semicolon

A Structure Variable

- a struct is a template or a blueprint for a composite variable
- this struct has four fields
- a struct is not a variable, it is a new type (actually, a simple ADT)
- since it is a type, we can declare a variable of this type, using the structure tag as the type name:

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

A Structure Variable

- a struct is a template or a blueprint for a composite variable
- this struct has four fields
- a struct is not a variable, it is a new type (actually, a simple ADT)
- since it is a type, we can declare a variable of this type, using the structure tag as the type name:

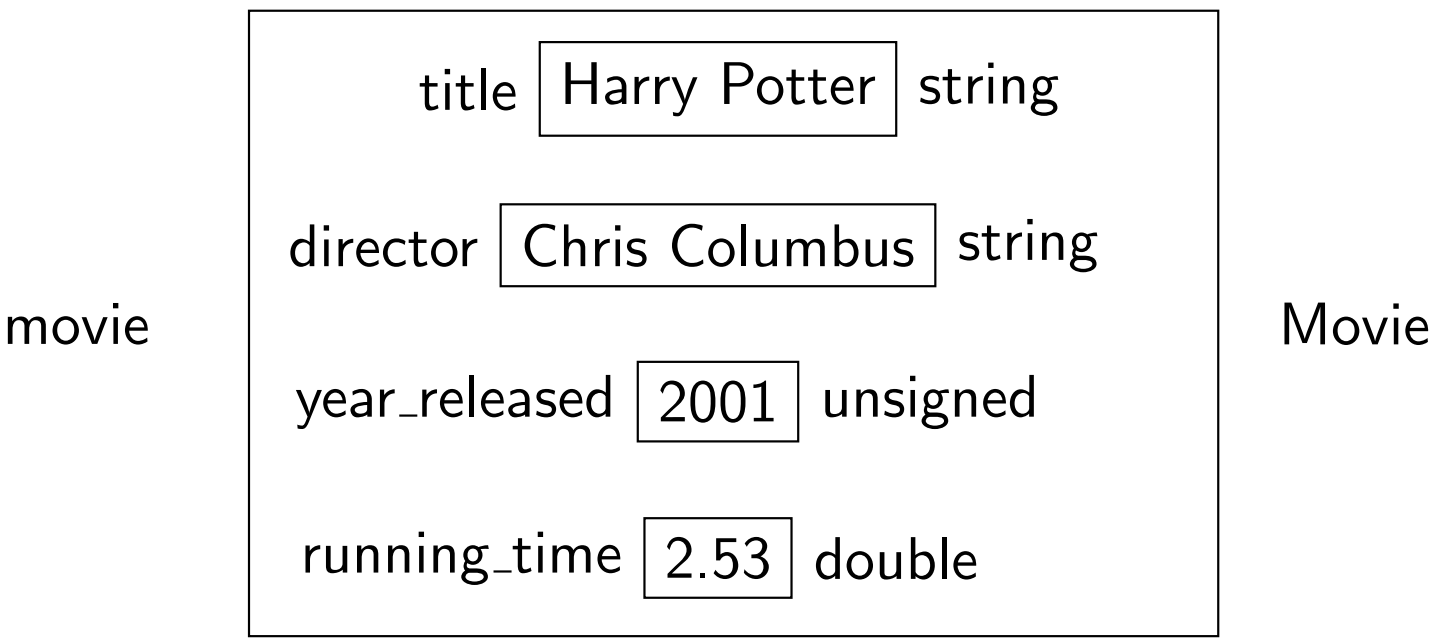
```
Movie movie {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

- movie is a variable that has four fields
- synonyms for "field" are attribute and member (Gaddis uses the latter)

Structure

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

- this declares the variable movie of data type Movie
- movie is a composite variable
- we diagram this variable schematically like this:



Accessing Structure Members

- an array is also a composite construct
- just as you cannot do things to all the elements of an array with one statement, you also cannot do things to all the members of a struct variable with one statement

Accessing Structure Members

- an array is also a composite construct
- just as you cannot do things to all the elements of an array with one statement, you also cannot do things to all the members of a struct variable with one statement

- to access individual elements of an array, we use square brackets
- to access individual elements of a vector, we use .at()

Accessing Structure Members

- an array is also a composite construct
- just as you cannot do things to all the elements of an array with one statement, you also cannot do things to all the members of a struct variable with one statement

- to access individual elements of an array, we use square brackets
- to access individual elements of a vector, we use .at()

- to access a individual structure member, we use the dot operator and dot notation

```
cout << movie.title << endl; // Harry Potter;
```

Examples

walk through the code of Program 11-1 and Program 11-2 in the text

Initializing a Struct

- 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

- 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};
```

- 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

- section 11.5 talks about arrays of structs, but we're going to talk about vectors of structs

Vectors of Structs

- section 11.5 talks about arrays of structs, but we're going to talk about 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

Vectors of Structs

- section 11.5 talks about arrays of structs, but we're going to talk about 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 you need the struct ADT definition

```
struct Movie
{
  string title;
  string director;
  unsigned year_released;
  double running_time;
};
```

Vectors of Structs

- then you use it to create vectors

```
vector<Movie> movies;

movies.push_back({"Psycho", "Hitchcock", 1960, 1.82});
movies.push_back({"Vertigo", "Hitchcock", 1958, 2.13});
movies.push_back({"Repulsion", "Polanski", 1965, 1.75});

for (auto movie : movies)
{
  cout << movie.title << ' ' << movie.year_released << endl;
}

Psycho 1960
Vertigo 1958
Repulsion 1965
```

Padding

- a note about compiling structs
- every program that uses this movie struct will have a compiler warning

```
test.cpp:10:10: warning: padding struct 'Movie' with
    4 bytes to align 'running_time' [-Wpadded]
double running_time;
```

- this is strictly an informational message that indicates the struct has extra "wasted" space between the two members `year_released` and `running_time`

Padding

- adding an extra int member suppresses the warning:

```
struct Movie
{
  string title;
  string director;
  unsigned year_released;
  int dummy;
  double running_time;
};
```

- but that would do no good and be confusing
- you can suppress the warning by adding a switch to the compiler: `-Wno-padded`
- or you can just ignore this warning

Passing Structs to Functions

- a struct variable may be passed to a function in five different ways
  1. by value
  2. by pointer
  3. by constant pointer
  4. by reference
  5. by constant reference

Passing Structs to Functions

- a struct variable may be passed to a function in five different ways
  1. by value
  2. by pointer
  3. by constant pointer
  4. by reference
  5. by constant reference

- each has a purpose

Pass by Value

- passing a struct variable by value makes a copy of the variable in the function
- typically not used at all
- rarely, used when both of these conditions hold:
  1. the struct is small, no more that a few simple members
  2. changes to the variable are not needed in the calling scope

Pass by Pointer and Constant Pointer

- rarely done in real C++ programs
- essential in C programs
- special syntax is required — see later slides
- pass by constant pointer if no changes are allowed
- pass by pointer if changes are needed in the calling scope

Pass by Reference and Constant Reference

- almost always used in real C++ programs
- pass by constant reference if no changes are allowed
- pass by reference if changes are needed in the calling scope

Location of Struct Definition

- if a struct is going to be exclusively used in one function (rare, but theoretically possible) it may be defined within that function, right after any constants
- but almost always, the struct definition will be used in multiple functions and should be defined in global scope, right after global constants, and before function prototypes