More Structures

Class 38

Copying Struct Variables

- unlike arrays, it is perfectly legal to copy one struct to another in a single statement

```
Movie movie1 {"Psycho", "Hitchcock", 1960, 1.82};
Movie movie2;

movie2 = movie1;
```

- now movie2 is an exact duplicate of movie1
- copied member by member

Comparing Struct Variables

- like arrays, you cannot compare struct variables with relops
- what would this even mean? `movie1 < movie2`

Comparing Struct Variables

- like arrays, you cannot compare struct variables with relops
- what would this even mean? `movie1 < movie2`

- do you mean alphabetic less-than by title?
- numeric less-than by release date?

Comparing Struct Variables

- like arrays, you cannot compare struct variables with relops
- what would this even mean? `movie1 < movie2`

- do you mean alphabetic less-than by title?
- numeric less-than by release date?

- rather, you need to write a function to compare two struct variables

Comparing Struct Variables

```
bool less_than_by_date(const Movie& movie1,
                       const Movie& movie2)
{
  return movie1.year_released < movie2.year_released;
}
```

Comparing Struct Variables

```
bool less_than_by_date(const Movie& movie1,
                       const Movie& movie2)
{
  return movie1.year_released < movie2.year_released;
}
```

write a function for alphabetic less-than by title, with ties broken by date

Comparing Struct Variables

```
bool less_than_by_date(const Movie& movie1,
                       const Movie& movie2)
{
  return movie1.year_released < movie2.year_released;
}
```

write a function for alphabetic less-than by title, with ties broken by date

```
bool less_than_alphabetic(const Movie& movie1,
                          const Movie& movie2)
{
  if (movie1.title == movie2.title)
  {
    return movie1.year_released < movie2.year_released;
  }
  return movie1.title < movie2.title;
}
```

Printing Struct Variables

- you cannot do this: `cout << movie1`

Printing Struct Variables

- you cannot do this: `cout << movie1`
- you could write a function `print_movie`, but it is much better conceptually to write this function:

```
string to_string(const Movie& movie)
{
  string result = movie.title + "; " + movie.director +
    " (" + to_string(movie.year_released) + ") ";

  unsigned hours = static_cast<unsigned>(movie.running_time);
  unsigned minutes =
    static_cast<unsigned>((movie.running_time - hours) * 60.0);

  result += to_string(hours) + " hr " + to_string(minutes) + " min";
  return result;
}
```

```
Psycho; Hitchcock (1960) 1 hr 49 min
```

Nested Structures

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

```
struct Time
{
  unsigned hour;
  unsigned minute;
};

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

Nested Structures

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

```
Movie movie1 {"Psycho", "Hitchcock", 1960, {1, 49}};

Movie movie2;
movie2.title = "Vertigo";
movie2.running_time.hour = 2;
movie2.running_time.minute = 8;
```

Nested Structures

- the to_string function now becomes:

```
string to_string(const Movie& movie)
{
  string result = movie.title + "; " +
    movie.director + " (" +
    to_string(movie.year_released) + ") " +
    to_string(movie.running_time.hour) + " hr " +
    to_string(movie.running_time.minute) + " min";
  return result;
}
```

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;
```

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 want to use the dot operator after dereferencing the pointer:
  `cout << *mptr.title;`

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 want to use the dot operator after dereferencing the pointer:
  `cout << *mptr.title;`
- but this doesn't work, because the precedence of dot is higher than the precedence of dereference
- the statement means: go to the variable mptr, select its title field (which doesn't exist), and then dereference that (which makes no sense) to find the thing to print (which doesn't work at all)

Pointers to Structure Variables

- instead we have to do this:

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

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 and old-fashioned
- 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

Dynamically Allocating Structures

- with the ability to have pointers to structure variables, we can dynamically allocate them
- this is essential in C, rarely done in C++ until CS310

```
Movie* mptr = new Movie;
mptr->title = "Billy Jack";
mptr->director = "Tom Laughlin";
mptr->year_released = 1971;
mptr->running_time.hour = 1;
mptr->running_time.minute = 54;

cout << to_string(*mptr) << endl;
delete mptr;
```

```
Billy Jack; Tom Laughlin (1971) 1 hr 54 min
```

Overloading

- a topic from 6.14 that we skipped at the time
- program 6-27, on page 360, defines two functions with the same name

```
int square(int number);
double square(double number);
```

Overloading

- a topic from 6.14 that we skipped at the time
- program 6-27, on page 360, defines two functions with the same name

  ```
  int square(int number);
  double square(double number);
  ```

- the name of the function, square, is overloaded

Overloading

- a topic from 6.14 that we skipped at the time
- program 6-27, on page 360, defines two functions with the same name

```
int square(int number);
double square(double number);
```

- the name of the function, square, is overloaded

- both functions have the same purpose
- they operate on arguments of different types, and return different types

Signatures

- in C++, every function has a signature
- the signature consists of
    1. the function's name
    2. the data types of the function's parameters, in order

Signatures

- in C++, every function has a signature
- the signature consists of
  1. the function's name
  2. the data types of the function's parameters, in order

- this is the information that is contained in the function prototype

Signatures

- in C++, every function has a signature
- the signature consists of
  1. the function's name
  2. the data types of the function's parameters, in order

- this is the information that is contained in the function prototype

- a function name can be overloaded if the types in the parameter list in the function signatures differ
- different number or arrangement of types

Examples

- all the following are legal examples of overloading

```
void foo(int i, double d); // different order of types
void foo(double d, int i);

void bar(int i, int j);    // different number of parameter
void bar(int i, int j, int k);

void baz(int x);    // different types
void baz(double x);
```

- however, the following is not legal

```
void foo(int x);  // only the return types differ
int foo(int x);   // not ok
```

Ambiguous

this is ok:

```
void foo(int i, double d);
void foo(double d, int i);
... in main
foo(5, 10.0); // ok, calls first one
```

Ambiguous

this is ok:

```
void foo(int i, double d);
void foo(double d, int i);
... in main
foo(5, 10.0); // ok, calls first one
```

and this is ok:

```
void foo(int i, double d);
... in main
foo(5, 10); // ok, promotes 10 to 10.0
```

Ambiguous

this is ok:

```
void foo(int i, double d);
void foo(double d, int i);
... in main
foo(5, 10.0); // ok, calls first one
```

and this is ok:

```
void foo(int i, double d);
... in main
foo(5, 10); // ok, promotes 10 to 10.0
```

but this won't compile due to ambiguity:

```
void foo(int i, double d);
void foo(double d, int i);
... in main
foo(5, 10); // doesn't know which one to call
```

Not Doing

- we will not do section 11.11 Enums at this time