Lec_20.md

Lecture 20 - When Objects have Pointers

Oct. 27/2020

Pointers are Wiggly - Prof Abdelrahman 2020

Shallow vs Deep

- **Shallow** deals with data that is stored inside an object
  - This is defined for class members which are non-dynamic
    - Remember that **dynamic** data exists outside of classes
- **Deep** deals with data stored outside object
  - **Dynamic data**

Classes with Pointers

Imagine a C++ class that contains data members which are **pointers**

- What happens?
  - Problem arise with C++'s **shallow** operations
- When pointers exist, the programmer must provide **deep** versions of these operators
  - Otherwise only the memory area for those pointers exists

Shallow Operations

The following operations are **shallow** by default:

1. Object Creation
   - Creates area to hold class members
     - No initialization for these class members
2. Object Destruction
   - Erases area that holds class members
3. Object Copying
   - Member by Member copying
4. Object Assignment
   - Member by Member assignment

This works well for **non-pointer** class members

- Like in the Time class we wrote previously
- But *not* for pointer class members

Implementations for deep operations for class Time:

Time.h

```
struct _time { int hour, minute, second; };

class Time {
```

```
  private:
    struct _time* time_ptr;
  public:
    Time ();
    Time (int h, int m, int s);
    Time (const Time & source);
    ~Time();
    int getHour();
    int getMinute();
    int getSecond();
    void setHour(int h);
    void setMinute(int m);
    void setSecond(int s);
    Time operator+ (Time rhs);
    Time operator- (Time rhs);
    void print ();
};
```

Deep Constructors

The C++ **default constructor** initializes *only* the dynamically allocated pointers

```
Time::Time() {
  time_ptr = new struct _time;
}
```

We want to write **deep constructors**

- Assign data to dynamically allocated class data members

```
Time::Time() {
  time_ptr = new struct _time;
  time_ptr->hour=0;
  time_ptr->minute=0;
  time_ptr->second=0;
}
```

Notes:

1. `time_ptr` is now **deep data**

- The value at the address pointed by `time_ptr` exists *outside* the class

```
Time::Time(int h, int m, int s) {
  time_ptr = new struct _time;
  time_ptr->hour = h;
  time_ptr->minute = m;
  time_ptr->second = s;
}
```

Notes:

1. `time_ptr` is now **deep data**

- The value at the address pointed by `time_ptr` exists *outside* the class

Deep Destructors

The C++ **default destructor** does not provide any implementation

- Invoked just before object is deleted
- **shallow** delete
  - Leaves a **dangling pointer** (memory leak)

```
Time::~Time() {

}
```

We want to avoid **memory leaks**

- Delete dynamically allocated, **deep** data

```
Time::~Time() {
  delete time_ptr;
}
```

Deep Assignment

For an assignment `X=Y;`

The C++ **default assignment operator** `operator=` directly ***updates*** the value at the address of the pointer

```
Time & Time::operator= (const Time & t) {
  time_ptr = t.time_ptr;
  return (*this);
}
```

The default version of the `operator=` operator has two large problems

1. Results in a **memory leak** * `X.time_ptr` now points to value at address of `Y.time_ptr` * No way to access the initial value of **dynamically allocated** `X.time_ptr`
2. Results in a **shared object data** * Any updates to `X.time_ptr` directly affect the value of `Y.time_ptr` * We want both objects (`X` , `Y`) to have separate data

We want a **deep assignment** that addresses both of these problems

```
Time & Time::operator= (const Time & t) {
  time_ptr->hour = t.time_ptr->hour;
  time_ptr->minute = t.time_ptr->minute;
  time_ptr->second = t.time_ptr->second;
  return (*this);
}
```

Deep Copying

The C++ **default copy** does member copying

- **Shallow copying**

```
Time::Time(const Time & src) {
  time_ptr = src.time_ptr;
}
```

Results in the same issue with the **assignment operator**

- **Shared object data**

We want a **deep copy** that addresses the issue of **shared object data**

```
Time::Time(const Time & src) {
  time_ptr = new struct _time;
  time_ptr->hour = src.time_ptr->hour;
  time_ptr->minute = src.time_ptr->minute;
  time_ptr->second = src.time_ptr->second;
}
```

Notes:

1. `Time::Time(const Time & src)`

- Must pass `src` *by reference*
  - Avoid cost of copying and recursive loop regarding copying

Const Revisited

pass by reference avoids the cost of copying objects

- removes protection of caller from callee when reassigning data values
- `const` modifier restores some of that protection

```
Time & Time::operator= (const Time & t) {
  t.time_ptr = NULL; // Compile-time error
  t.time_ptr->hour=0; // Not a compile-time error
  return (*this);
}
```

Notes:

1. `t.time_ptr = NULL;`

- Compile-time error

2. `t.time_ptr->hour=0;`

- Not a compile-time error