Lec_17.md

Operator Overloading

Oct. 20/2020

Note-takers Note: The fire alarm in my building is being tested today, please excuse notes if I miss anything. Sorry!

Object Assignment

main.cpp

```
int main(){
  Time x(5,2,30);
  Time y(10,30,48);

  return 0;
}
```

By default, if you try to assign `x=y`

- C++ provides a **default operator** to do this assignment
  - The `operator=` **assignment operator**

The Operator= Operator

```
Time Time::operator= (Time rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;

  return (*this);
}
```

Notes:

1. `return (*this);`
   - We need to comply with **backwards-compatibility**
     - Specifically, with the rule *Assignments are also expressions*
   - `z = (x = y)` ***must*** be defined

The Copy Constructor

Notice in the function declaration for the `operator=` function:

- `Time Time::operator= (Time rhs)`
  - We are passing in a `Time rhs` object
    - Except this is a *pass by value*, not a *pass by reference*
      - So the object is *copied*, not *given*

Whenever an object is to be created that is a copy of another object, C++ invokes the **copy constructor** of the class

- Object initialized by the **copy constructor**

Examples:

1. `Time X(Y);`
   - Y is a `Time` object
2. `Time *p = new Time(X);`
   - X is a `Time` object
     - p points to a new object which is a *copy* of X
3. `void do_something(Time x);`
   - Passing objects into functions *by value*
4. `Time X = Y;`
   - This is actually the **copy constructor**
     - We are creating `X`, and would like to initialize it with `Y`
   - This does ***not*** invoke the **assignment operator**
     - If `X` was initialized, then the **assignment operator** would be called instead of the **copy constructor**

Writing out the **copy constructor**:

```
Time::Time(Time & source){
  hour = source.hour;
  minute = source.minute;
  second = source.second;
}
```

Notes:

1. `Time::Time(Time & source)`
   - Notice we are passing `source` *by reference*
     - Do we have to *pass by reference*? Can we instead *pass by value*?
       - ***Yes***, you ***must*** pass by reference in this case
       - Otherwise, it will enter a **recursive loop**
       - Trying to create a new object `Time source` will result in a **recursive call** to the **copy constructor**
       - Which then attempts to make a new object with **copy constructor** which calls the function again
     - **Compile-Time Error** if you ***try to pass by value***
2. There is no `return`
   - It is a constructor, after all

The **copy constructor** is invoked whenever a ***new*** object is created

- The new object is a copy of an *existing object* of the same type
- If the object exists, use the **overloaded assignment operator**

Pass by Reference

Pass by reference is necessary for the **copy constructor**

- We can also use it to *avoid* copying objects

```
Time Time::operator= (Time rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;

  return (*this);
}
```

Notes:

1. `Time Time::operator= (Time rhs)`
    - `Time rhs` is *passed by value*
        - This is a copy of an object
            - Invokes the **copy constructor**
        - At the end of `operator=`, `rhs` goes out of scope
            - The **default destructor** is called too
        - All of this is costly!

```
Time Time::operator= (Time & rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;

  return (*this);
}
```

Notes:

1. `Time Time::operator= (Time & rhs)`
    - `Time rhs` is *passed by reference*
        - This is ***not*** a copy of an object, it ***is*** the object

```
Time & Time::operator= (Time & rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;

  return (*this);
}
```

Notes:

1. `Time & Time::operator= (Time & rhs)`
    - Return value is returned *by reference*
        - No copying takes place on return

Pass by reference avoids cost of copying objects, but now any function with **pass by reference** can accidentally change the value of source objects

```
Time Time::(Time & source){
  hour = source.hour;
  minute = source.minute;
  second = source.second;
  source.hour = 0;
}

Time & Time::operator= (Time & rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;
  rhs.hour = 0;

  return (*this);
}
```

Notes:

1. `source.hour = 0;`
   - This changes the value of `source.hour`
     - Outside the scope of the function
2. `rhs.hour = 0;`
   - Similarily, this changes the value of the `.hour` field on the right hand side of the `operator=` call

So how can we restore some protection?

- To avoid incorrectly rewriting/modifying `member data fields` for objects *passed by reference*

The Const Modifier

We can define the **pass by reference** parameter as a **constant**

- Locally, this means the value of the **pass by reference** parameter *cannot be changed*

```
Time Time::(const Time & source){
  hour = source.hour;
  minute = source.minute;
  second = source.second;
  source.hour = 0;
}

Time & Time::operator= (const Time & rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;
  rhs.hour = 0;

  return (*this);
}
```

Notes:

1. `Time Time::(const Time & source)`
   - Define the **pass by reference** as a **constant**
     - Cannot change the member data fields of `source`
2. `Time & Time::operator= (const Time & rhs)`
   - Same as 1. , for `rhs`
3. `source.hour = 0; , rhs.hour = 0;`
   - These are now **compile-time errors**

Applying this to other examples:

time.cpp

```
Time Time::operator+ (const Time & rhs){
  int TotalSeconds;
  Time sum;

  TotalSeconds = second + 60*minute + 3600*hour;
  TotalSeconds += rhs.second +60*rhs.minute + 3600*rhs.hour;

  sum.hour = TotalSeconds/3600;
  sum.minute = (TotalSeconds - 3600*sum.hour)/60;
  sum.second = TotalSeconds%60;

  rhs.second = 0;
  second = 0;
```

```
  return (sum);
}
```

Notes:

1. `Time Time::operator+ (const Time & rhs)`
   - Define the **pass by reference** as a **constant**
2. `rhs.second = 0;`
   - This is a **compile-time error**
3. `second = 0;`
   - This is still allowed, but...

When we say `Z = X+Y` (`X`, `Y`, `Z` all `Time` objects)

- Do we really want the value of `X` to be changed accidentally?
  - No, we don't.
  - So we want to provide modify protection to `X`
    - `X` is the object that `operator+` is called on
    - Similar to how we provided **const** protection to `rhs` earlier

time.cpp

```
Time & Time::operator+ (const Time & rhs) const{
  int TotalSeconds;
  Time sum;

  TotalSeconds = second + 60*minute + 3600*hour;
  TotalSeconds += rhs.second +60*rhs.minute + 3600*rhs.hour;

  sum.hour = TotalSeconds/3600;
  sum.minute = (TotalSeconds - 3600*sum.hour)/60;
  sum.second = TotalSeconds%60;

  rhs.second = 0;
  second = 0;

  return (sum);
}
```

Notes:

1. `Time & Time::operator+ (const Time & rhs) const { }`
   - Define the function that is being called on as a **const**
     - We cannot edit the values of `X` (or the left hand side of `X+Y`)
2. `second = 0;`
   - This is *now* also a **compile-time error**