Lec_16.md

Lecture 16 - Pointers, Scopes, Arrays and Overloading

Oct. 15/2020

Note-takers Note: Good luck on the 241 midterm!

Announcements:

Quiz 3 Runs on Monday (October 19)

- 30 minutes in duration
  - Any time during the day
- You **cannot** go back to quiz if you have left
- Please scroll down to make sure you don't miss questions!

Material Covered:

- Everything from **start of term** until **end of today's lecture**

Arrays of Objects

How to declare an array a of 10 `DayOfYear` objects?

- Easy, use `DayOfYear a[10];`
  - Don't even need **dynamic allocation**

But what constructor gets called?

- Actually, the **default constructor** is called
  - So if you haven't defined the **default constructor**, you ***can not*** define an array this way
- **Default constructor** is called `n` times
  - In the above case, `n` ='10'

What about with **dynamic allocation**?

- `DayOfYear* a = new DayOfYear[n];`
  - The **default constructor** is called, `n` times

What about deleting **dynamically allocated** arrays of objects?

- `delete [] a;`
  - The **default destructor** is called, `n` times

So how can we specify the **constructor** that should run when the **objects** in the **array** are instantiated?

main.cpp

```
int main(){
  DayOfYear* a[10];
  for(int i = 0;i < 10;++i){
    a[i] = new DayOfYear(1,2);
  }
```

```
  //delete [] a;
}
```

Notes:

1. `DayOfYear* a[10];`
   - What constructor is called?
     - None, the above only creates an **array of pointers**
       - No objects have been **instantiated**
2. `a[i] = new DayOfYear(1,2);`
   - Uses the `DayOfYear(int,int)` constructor to **instantiate** the `DayOfYear` objects
3. `//delete [] a;`
   - ***Cannot*** call `delete` on `a`
     - `a` has ***not*** been dynamically allocated
       - It is a *stack* variable, not a *heap* variable

main.cpp

```
int main(){
  DayOfYear* a[10];
  for(int i = 0;i < 10;++i){
    a[i] = new DayOfYear(1,2);
  }
  for(int i = 0;i < 10;++i){
    delete a[i];
  }
}
```

Notes:

1. `delete a[i];`
   - Can call `delete` on `a[i]` to delete individual `DayOfYear` objects

Overloading

Start with an example: the **Time Class**

- Want to be able to add `time` objects
  - e.g. Add one time object representing noon, and one time object representing 4pm

We can **overload** default operators to define custom behaviour for them.

- e.g. we can define what the `+` operator does for *objects*

What exactly is `x+y` ?

- What does the `+` represent?
  - It is **Syntactic Sugar** in C++
  - ***Exactly the same*** as `x.operator+(y)`
    - Where `x` and `y` are compatible **objects**
    - the `operator+` function is a **class member** of `x`

The `.operator` operator

- Is ***always*** called on the object *on the left*
  - `x+y` or `x.operator+(y)` is called ***on x***

time.h

```
class Time{
  private:
    int hour,minute,second;
  public:
    Time();
    Time(int h,int m,int s);
    ~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();
};
```

Notes:

1. `Time operator+ (Time rhs);`
   - This **overloads** the `+` operator with a specific behaviour
2. `Time operator- (Time rhs);`
   - This **overloads** the `-` operator with a specific behaviour

What exactly is that behaviour??

- We define it in the **function definition**

time.cpp

```
Time Time::operator+ (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;

  return (sum);
}
```

Notes:

1. `TotalSeconds = second + 60*minute + 3600*hour;`
   - Set `TotalSeconds` to be the *value* of the current object
     - In this case, the *current object* is the one the operator is being *called on*
     - Or, the **left hand side**
2. `TotalSeconds += rhs.second +60*rhs.minute + 3600*rhs.hour;`
   - Adds **right hand side** object values to `TotalSeconds`
     - `TotalSeconds` currently represents **left hand side**
3. `sum.hour = TotalSeconds/3600;`
   - Remember that the **operator+** operator is defined in the **scope** of the class

        - Can access **data members** of the sum object without using **accessor** functions
4. `return sum;`
    - Return the sum object (of type `Time`)
        - So `Time z = x+y;` is valid
            - As long as `x` and `y` are `Time` objects
    - ***Must*** use this `return`
        - The **assignment operator *must be evaluatable***

What does it mean to be **evaluatable**?

- First of, I just define the term **evaluatable** to mean "able to be evaluated"
- Essentially, `(x+y)` must **evaluate** to something.
    - In other words, `z = (x+y)` must be defined
        - This means that the `operator+` ***must have a return type***

time.cpp

```
Time Time::operator- (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;

  return (sum);
}
```

Notes:

1. `TotalSeconds -= rhs.second +60*rhs.minute + 3600*rhs.hour;`
    - Does the same thing as `operator+` but subtracts instead.

Default Operators

One specific operator is given by default

- The `operator=` operator, or the **assignment operator**

time.cpp

```
Time Time::operator= (Time rhs){
  hour = rhs.hour;
  minute = rhs.minute;
  second = rhs.second;
  return (*this);
}
```

Notes:

1. `operator=`
    - Sets the object on the left to the object on the right
        - Think about `int x = 5;`
            - Sets variable `x` equal to `5`
2. `return (*this);`

- What is `(*this)` ??
  - Evidently, its an **object** of type `Time`
    - Just look at function return type
- Why do we return `(*this)` ?
  - Because `Z=X=Y` is valid
    - The expression `X=Y` assigns `X` the value of `Y`
      - But also has a evaluatable value
      - `(X=Y)` will *evaluate* to `X`
    - `Z=(X=Y)` is identical to `Z=X`

Overloading Restrictions

We can now **overload *most*** operators. Which ones can we not **overload**?

- `.` : **field access operator**
- `::` : **scope resolution operator**
- `? :` : **ternary/conditional operator**
- `sizeof` : **object size operator**

And many more. Check out the TB or online C++ reference for the full list. :)

Object Copying

What if we wanted an object to be created that is a *copy* of another object?

- C++ invokes the **Copy Constructor** of the class
- A new object is initialized by the **Copy Constructor**

But remember when you pass variables into functions via parameters, they are **passed by value**

- What is passing by value for **objects**?
  - Actually, it is **Object Copying**
  - Invocations of the **copy constructor** as new objects are created

Additional to the `operator=` **assignment operator**, the **copy constructor** is ***also given to you by default by c++***

Examples of **Object Copying**:

```
Time X(Y);
Time *p = new Time(X);

Time X = Y;
```

Notes:

1. `Time X = Y;`
   - This calls the **copy constructor**
     - Creating a *new* object `X`

Wait, how is this different from `operator=` (the **assignment operator**)

```
Time X(1,1,1);
Time Y(0,0,0);
X = Y;
```

The above invokes the **assignment operator**

```
Time Y(0,0,0);
Time X = Y;
```

The above invokes the **copy constructor**

time.cpp

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

Notes:

1. This is the **copy constructor**
2. `Time::Time(Time & source){ }`
    - The source object ***must*** be passed by reference.
        - This is a **requirement** for the **copy constructor**