Overloading and Templates I

DM2233 Advanced Data Structures & Algorithms

Module Schedule

Week	Lecture	Remarks
1	Overloading and Templates I	
2	Overloading and Templates II	Labour Day (Fri)
3	Overloading and Templates III	
4	Overloading and Templates IV	
5	Exception Handling I	
6	Exception Handling II	
7	Preprocessing / Assignment 1	Vesak Day (Mon)
Week 8 and 9: Mid-Sem Break		
10	Sorting and Searching I	
11	Sorting and Searching II	
12	Sorting and Searching III	
13	Binary Tree I	Hari Raya Puasa (Fri)
14	Lab Test	
15	Binary Tree II	
16	Binary Tree III	SG50 Day (Fri)
17	Standard Template Library / Assignment 2	National Day (Mon)

Objective

- "this" pointer
- Friend function
- Operator Overloading
- Binary Operator Overloading

"this" pointer

- Every object of every class maintains a pointer to itself
  - The name of the pointer is "this"
  - "this" is not accessible outside of the object
  - "this" need not be defined

"this" pointer

- Consider the following

```
class timeType {
  private:
    int hr;
    int min;
    int sec;

  public:
    timeType (int hr, int min, int sec) {
      hr = hr;
      min = min;
      sec = sec;
    }
}
```

"this" pointer

- Solution

```
class timeType {
  private:
    int hr;
    int min;
    int sec;

  public:
    timeType (int hr, int min, int sec) {
      this->hr = hr;
      this->min = min;
      this->sec = sec;
    }
}
```

"this" pointer

- Another example

```
class timeType {
  private:
    int hr, min, sec;

  public:
    timeType (int hr, int min, int sec) {
      this->hr = hr;
      this->min = min;
      this->sec = sec;
    }

    timeType setMeeting (void) {
      hr = 9;
      min = sec = 0;
      return * this;
    }

    void print (void) {
      cout << hr << " " << min << " "
           << sec << endl;
    }
}
```

```
void main (void) {
  timeType t1 (17, 30, 0);
  timeType t2 (0, 0, 0);

  t1.print ();
  t2 = t1.setMeeting ();

  t2.print ();
  t1.print ();
}
```

output

```
17 30 0
9 0 0
9 0 0
```

"this" pointer

- Pro
  - Makes it easier for you to refer to the class which you are working on.
- Con
  - Nil

Friend Function

- A friend function is a non-member function
  - However, it has access to all members of the class
- Function prototype must exist within class definition
- Function prototype is preceded with the keyword friend
  - A good way out of private variables restrictions; use with caution

Friend Function

- Typical use of class members.

```
class timeType {
  private:
    int hr, min, sec;

  public:
    int getHr (void) {return hr;}
    int getMin (void) {return min;}
    int getSec (void) {return sec;}
}
```

```
void prtTime (timeType t) {
  cout << t.getHr() << "hr "
       << t.getMin() << "min "
       << t.getSec() << "sec"
}
```

```
void main (void) {
  timeType tt (17, 30, 0);

  prtTime (tt);
}
```

Friend Function

```
class timeType {
  private:
    int hr, min, sec;

  public:
    int getHr (void) {return hr;}
    int getMin (void) {return min;}
    int getSec (void) {return sec;}
}
```

```
void prtTime (timeType t) {
  cout << t.getHr() << "hr "
       << t.getMin() << "min "
       << t.getSec() << "sec"
}
```

```
void main (void) {
  timeType tt (17, 30, 0);

  prtTime (tt);
}
```

```
class timeType {
  private:
    int hr, min, sec;

  public:
    friend void prtTime (timeType);
}
```

```
void prtTime (timeType t) {
  cout << t.hr << "hr "
       << t.min << "min "
       << t.sec << "sec"
}
```

```
void main (void) {
  timeType tt (17, 30, 0);

  prtTime (tt);
}
```

Friend Function

- Pro
  - Allow non-member function to access the class’s private and protected variables and methods
- Con
  - Bypass the “protection” which C++ provides to private and protected variables.
    - VERY dangerous in the hands of noob programmers!

Operator Overloading

```
class timeType {
  private:
    int hr;
    int min;
    int sec;

  public:
    timeType (int hr, int min, int sec) {
      this->hr = hr;
      this->min = min;
      this->sec = sec;
    }
}
```

- How nice if we can do the following

```
timeType t1 (3, 45, 0);
timeType t2 (2, 5, 20);
timeType t3 (0, 0, 0);

t3 = t1 + t2;
cout << t1;
t1 ++;
if (t1 == t2) ...
```

Operator Overloading

- C++ allows most operators to be extended
- Relational, arithmetic, insertion and extraction operators can now be applied to objects of user defined classes
- Most operators can be overloaded
  - +, -, *, /, %, +=, -=, *=, /=, %=, <, >, <=, >=, ==, !=, ++, --, =
- These cannot be overloaded
  - ., .*, ::, ?:, sizeof

Operator Overloading

- Cannot create *new* operator
- Cannot change operator precedence
- Cannot change operator associativity
- Cannot change the number of parameters an operator takes
- Overloaded operators cannot have default parameters

Operator Overloading

- Syntax:
  - `returnType operator<op> (arguments)`
- `operator` is a reserved word
- `operator` is value-returning
- `<op>` is the operator to overload

Binary Operator Overloading

- To overload the + operator
- We have a rect type with width and height
- We want rect1 + rect2 to return a rect that adds the width and height of rect1 and rect2

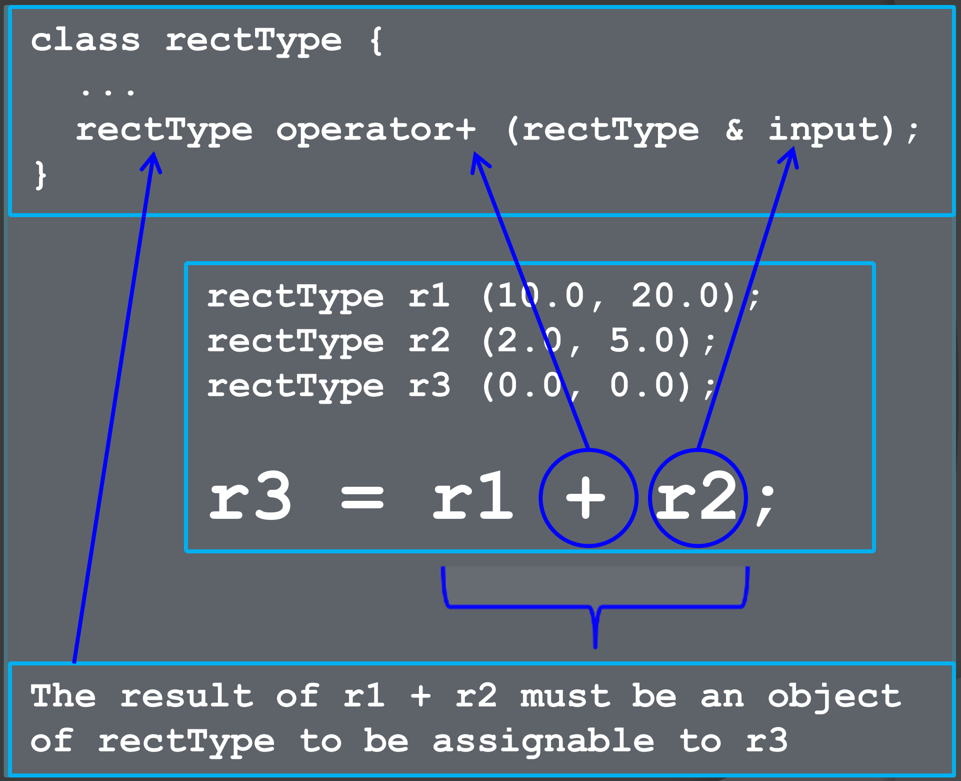
```
class rectType {
  private:
    double width;
    double height;

  public:
    recType (double w, double h) {
      width = w;
      height = h;
    }

    recType operator+ (rectType & input);
}
```

Binary Operator Overloading

- What about r1?
  - r1 is the object that + is acting upon
  - r1 + … as compared to r1.print()



```
class rectType {
  ...
  rectType operator+ (rectType & input);
}
```

```
rectType r1 (10.0, 20.0);
rectType r2 (2.0, 5.0);
rectType r3 (0.0, 0.0);

r3 = r1 + r2;
```

```
The result of r1 + r2 must be an object
of rectType to be assignable to r3
```

Binary Operator Overloading

- operator+ fleshed out

```
rectType rectType::operator+ (rectType & input) {
  rectType rtemp (0.0, 0.0);

  rtemp.width = width + input.width;
  rtemp.height = height + input.height;

  return rtemp;
}
```

```
rectType r1 (10.0, 20.0);
rectType r2 (2.0, 5.0);
rectType r3 (0.0, 0.0);

r3 = r1 + r2;
// r3.width = 12.0
// r3.height = 25.0
```

Binary Operator Overloading

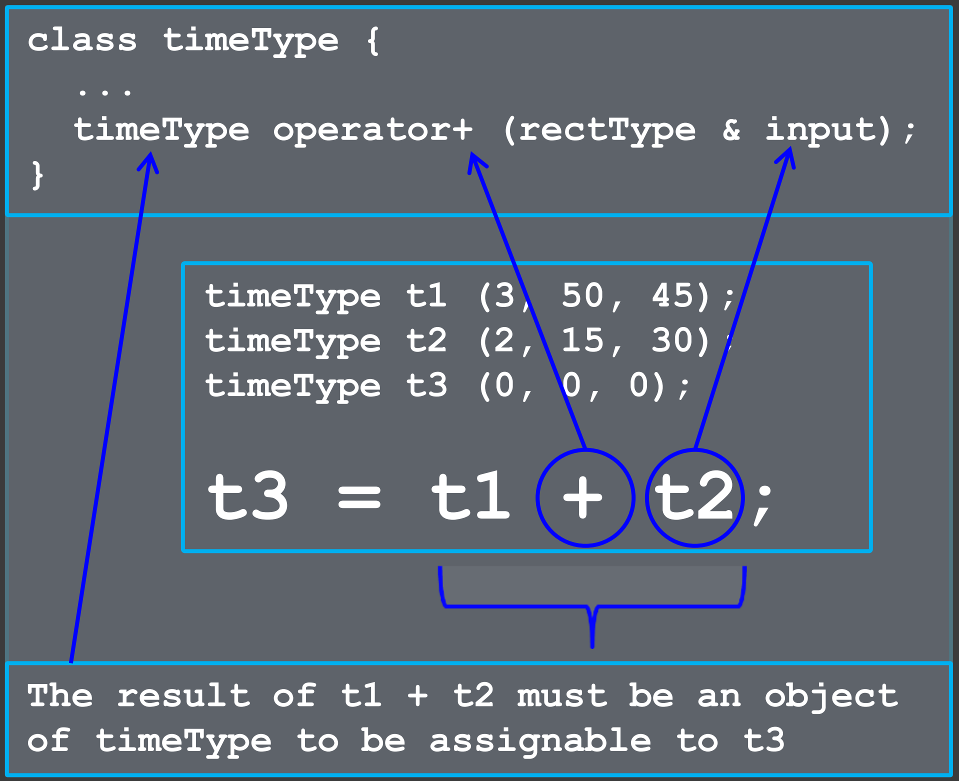
- Another example

```
class timeType {
  private:
    int hr;
    int min;
    int sec;

  public:
    timeType (int hr, int min, int sec) {
      this->hr = hr;
      this->min = min;
      this->sec = sec;
    }

    timeType operator+ (timeType & input);
}
```

Binary Operator Overloading



```
class timeType {
   ...
   timeType operator+ (rectType & input);
}
```

```
timeType t1 (3, 50, 45);
timeType t2 (2, 15, 30);
timeType t3 (0, 0, 0);

t3 = t1 + t2;
```

The result of t1 + t2 must be an object of timeType to be assignable to t3

Binary Operator Overloading

- operator+ fleshed out

```
timeType timeType::operator+ (timeType & input) {
  timeType ttemp (0, 0, 0);
                                                  ttemp
  ttemp.sec = sec + input.sec;      ---------     0  0 75
  if (ttemp.sec >= 60) {
    ttemp.min += ttemp.sec / 60;    ---------     0  1 75
    ttemp.sec %= 60;                ---------     0  1 15
  }

  ttemp.min += min + input.min;     ---------     0 66 15
  if (ttemp.min >= 60) {
    ttemp.hr += ttemp.min / 60;     ---------     1 66 15
    ttemp.min %= 60;                ---------     1  6 15
  }

  ttemp.hr += hr + input.hr;        ---------     6  6 15

  return ttemp;
}

timeType t1 (3, 50, 45);
timeType t2 (2, 15, 30);
```

Binary Operator Overloading

- Pro
  - Convenient to use
  - Changes long and complex codes into short and simple codes.
- Con
  - Nil

Summary

- We had just discussed about,
  - "this" pointer
  - Friend function
  - Operator Overloading
  - Binary Operator Overloading