DigiPen
INSTITUTE OF TECHNOLOGY

CS170#05.2

Operator Overloading

Vadim Surov

Outline

- Introduction
- Example Without Overloading
- What Do We Want?
- Traditional Method
- Operator Function
- Syntax Of Operator Overloading
- Example
- Usage
- More Examples
- Summary

Introduction

- Some operators in C (and C++) behave differently depending on the operand types

3 + 4 is 7 (int)
3.1 + 4.1 is 7.2 (double)
3.1f + 4.1f is 7.2f (float)
3 + 4.1 is 7.1 (double)

- Note that machine instructions for adding integers are very different from adding floating point numbers!

Introduction

- This functionality (known as *operator overloading*) is built into the compiler
- If the compiler does not have such functionality, then we could write overloaded functions for it:

```
int add(int a, int b);
float add(float a, float b);
double add(double a, double b);
double add(int a, double b);
...
double d = add(3, 4.1);
```

Introduction

- There are many more examples of built-in operator overloading

- For example, the & operator
  - represents a reference when used in a declaration
  - gives the address of the right operand
  - performs bitwise AND with two int operands

- In C++, we can make it so that some operators can "recognise" our user-defined class and perform the correct operations

Example Without Overloading

```
class StopWatch
{
public:
  // Constructors
  StopWatch(void);
  StopWatch(int seconds);
  StopWatch(int hours, int minutes, int seconds);

  void Increment(int seconds = 1);
  void Reset(void);
  int GetSeconds(void) const;
  void Display(void) const;

private:
  int seconds;
};
```

```
// Constructors
StopWatch::StopWatch(void){
  seconds = 0;
}
StopWatch::StopWatch(int seconds) {
  this->seconds = seconds;
}
StopWatch::StopWatch(int hour, int minutes, int seconds) {
  this->seconds = (hour*3600) + (minutes*60) + seconds;
}
void StopWatch::Increment(int seconds){
  this->seconds += seconds;
}
void StopWatch::Reset(void) {
  this->seconds = 0;
}
int StopWatch::GetSeconds() const {
  return seconds;
}
```

```
void StopWatch::Display() const
{
  int h, m, s;
  h = seconds / 3600;
  m = seconds % 3600 / 60;
  s = seconds % 60;
  std::cout.fill('0');
  std::cout << std::setw(2) << h << ':';
  std::cout << std::setw(2) << m << ':';
  std::cout << std::setw(2) << s << std::endl;
}
```

```
StopWatch sw1;
StopWatch sw2(625);
StopWatch sw3(9, 30, 0);

sw1.Display(); // 00:00:00
sw2.Display(); // 00:10:25
sw3.Display(); // 09:30:00

sw1.Increment();  // add 1 sec
sw1.Increment();  // add 1 sec
sw1.Increment(5); // add 5 secs

sw1.Display();    // 00:00:07
```

What Do We Want?

- We want to be able to do something like this:

```
StopWatch sw1(10), sw2(20);
StopWatch sw3 = sw1 + sw2;
```

- The above code does not compile because the compiler does not "know" how to add two StopWatch objects

Traditional Method

- We could write a function that does this job, e.g.

```
StopWatch AddStopWatch(const StopWatch& sw1,
                       const StopWatch& sw2)
{
  // Construct a new SW from two
  StopWatch sw(sw1.GetSeconds()+sw2.GetSeconds());

  // Return the result
  return sw;
}
```

- And use it this way:

```
StopWatch sw3 = AddStopWatch(sw1, sw2);
```

Method 1 (contd)

- But this does not scale well if you want to add multiple StopWatch objects together:

```
StopWatch sw1(10), sw2(20), sw3(30), sw4(40), sw5(50);
StopWatch tempwatch = AddStopWatch(sw1, sw2);
tempwatch = AddStopWatch(tempwatch, sw3);
tempwatch = AddStopWatch(tempwatch, sw4);
StopWatch sw6 = AddStopWatch(tempwatch2, sw5);
```

- Or you could do this (not much better):

```
StopWatch sw6 =
   AddStopWatch(
      AddStopWatch(
         AddStopWatch(
            AddStopWatch(sw1, sw2), sw3), sw4), sw5);
```

Operator Function

- What we really want to do is this:

```
StopWatch sw6 = sw1 + sw2 + sw3 + sw4
                + sw5;
```

- We can "teach" the compiler to add StopWatch objects by overloading the + operator
- This is done using an **operator function**

Syntax Of Operator Overloading

- Syntax:

```
return-type operatorop(argument-list)
{
   statements
}
```

  - `operator` is a keyword
  - `op` is the operator to overload
  - `argument-list` is the list of arguments to the operator
- Argument list contains at most 2 arguments
  - the conditional operator ?: cannot be overloaded

Example

```
StopWatch operator+(const StopWatch& lhs,
                    const StopWatch& rhs) {
  // lhs is left hand side, rhs is right hand side
  StopWatch sw(lhs.GetSeconds()+rhs.GetSeconds());

  // Return the result
  return sw;
}
```

- Note that this is almost exactly the same as our previous AddStopWatch function (only the name has changed, really)

Usage

- Now this expression:
  `sw1 + sw2`
  is equivalent to calling this function:
  `operator+(sw1, sw2)`
- The left operand corresponds to the first argument
- The right operand corresponds to the second argument

Usage

```
StopWatch sw1(10), sw2(20), sw3(30),
    sw4(40), sw5(50);

// 150 seconds
StopWatch sw6 = sw1 + sw2 + sw3 + sw4 + sw5;
sw6.Display(); // 00:02:30

// functional notation
StopWatch sw7 = operator+(sw1, sw2);
sw7.Display(); // 00:00:30
```

Example 2

- Let's overload the – operator as well:

```
StopWatch operator-(const StopWatch& lhs,
                    const StopWatch& rhs) {
  // check for negative seconds
  int seconds = lhs.GetSeconds()-rhs.GetSeconds();
  if (seconds < 0)
  seconds = 0;
  // return the result
  return StopWatch(seconds);
}
```

- Now you can do this:

```
StopWatch sw3 = sw1 - sw2;
```

Example 3

- What about overloading the * operator so that you can multiply a StopWatch with an integer:

```
StopWatch operator*(const StopWatch& lhs, int rhs)
{
  // create the new StopWatch
  StopWatch sw(lhs.GetSeconds() * rhs);

  // return the result
  return sw;
}
```

- Now you can do this:

```
StopWatch sw2 = sw1 * 2;
```

Example 4

- Can you do this?

```
StopWatch sw2 = 2 * sw1;
```

- No! You need to add this function:

```
StopWatch operator*(int lhs, const StopWatch& rhs)
{
  StopWatch sw(lhs * rhs.GetSeconds());
  return sw;
}
```

- Or better yet, do this:

```
StopWatch operator*(int lhs, const StopWatch& rhs)
{
  return rhs * lhs;
}
```

Summary

- Operator overloading allows user-defined types to be used as operands for operators like in-built types
- This is done by defining an **operator** function using the operator keyword
- In a non-member, non-friend function, the first argument corresponds to the left operand while the second argument corresponds to the right operand