ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Overloaded Operators

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

What is an overloaded operator?

- C++ compiler knows how to handle operators for the basic types for which they are appropriate
  - e.g. the '+' operator with integer or floating point operands

- But cannot apply those operators to new data types which have been defined by the programmer
  - e.g. we can define a class `Employee`, but what does it mean to add 2 `Employee` objects?
  - It means nothing unless we code a meaning
  - For example, perhaps we would like it to mean an Employee object with a salary which is the sum of the salaries of the 2 added

- It is possible to 'overload' an operator to give it a meaning when used with operands of new types.
- The name of the function will be `operator*` where * is the operator we are overloading
  - e.g. To overload the + operator, we will have a function called **`operator+()`**
- Operators can be overloaded as stand-alone functions, and sometimes as member functions of a class.

Overloaded binary operator as a stand-alone function

- The overloaded operator will have 2 arguments
  - The 1st argument is the operand on the left of the operator
  - The 2nd argument is the operand on the right of the operator

- For example, if we have a standalone function with the following signature

```
Employee operator+(Employee emp1, Employee emp2)
```

-> then we can write code like this

```
Employee mary, jo, aisling;
…
mary = jo + aisling;
```

Actually, we could make a few improvements to the function signature as well, see next overhead

-> the function `operator+` would execute, with `jo` used to initialise the var `emp1`, and `aisling` used to initialise the var `emp2`

An overloaded + operator example

```
class Money {
public:
   Money(int = 0, int = 0);
   int getEuro()const;
   int getCent()const;
   void display();
private:
   int euro;
   int cent;
};
```

```
Money operator+(Money op1, Money op2) //mark 1
{
        Money temp(op1.getEuro() + op2.getEuro(),
                op1.getCent() + op2.getCent());
        return temp;
}
```

Ist improvement – pass the class variables by reference to save the overhead of making local copies

```
Money operator+(Money& op1, Money& op2) //mark 2
{
        Money temp(op1.getEuro() + op2.getEuro(),
                op1.getCent() + op2.getCent());
        return temp;
}
```

An overloaded + operator example – cont.

next improvement:
the pass-by-ref vars should be marked as const to make this function more robust

And the return value should be a const value, so that we can't say something like
`(money1 + money2).setEuro(10);`

```
const Money operator+(const Money& op1, const Money& op2) //mark 3
{
   Money temp(op1.getEuro() + op2.getEuro(),
                              op1.getCent() + op2.getCent());
   return temp;
}
```

Using a friend function

```
class Money {
   friend const Money operator+ (const Money&, const Money&)
public:
   Money(int = 0, int = 0);
   int getEuro()const;
   int getCent()const;
   void display();
private:
   int euro;
   int cent;
};
```

Another improvement would be for the `Money` class to declare this stand-alone function as a friend. Then we wouldn't have to use the `getEuro()` and `getCent()` public methods – overhead should be minimised in something as basic as an operator!

```
const Money operator+(const Money& op1, const Money& op2) //mark 4
{
        Money temp(op1.euro + op2.euro, op1.cent + op2.cent);
        return temp;
}
```

Overloaded binary operator as member function of a class

- If a binary operator is overloaded as a member function then it will only have 1 argument
  - The argument is the operand on the right of the operator
  - the operand on the left of the operator is automatically the object on which the method is called is

```
class Money {
public:
        Money(int = 0, int = 0);
        const Money operator+(const Money&) const;
private:
        int euro;
        int cent;
};
```

```
const Money Money::operator+(const Money& rhs) const
{
   return Money(euro + rhs.euro, cent + rhs.cent);
}
```

Overloaded operators - more

- Suppose you wish to define a + operator for the `Money` class where `myMoney + 2` means add 2 euro to `myMoney`
- As a standalone function this would have a LH operand which was a `Money` object, and a RH operand which was an `int`.

```
const Money operator+ (const Money&, int);
```

- As a member function of the `Money` class, it would just have 1 argument, the RH operand of type `int`

```
class Money {
 public:
    …
  const Money operator+(int) const;
  …
};
```

```
const Money operator+(int, const Money&);
```

If the + operator could also be used so that '`2 + myMoney`' meant the same as '`myMoney + 2`' you would need a second stand-alone function like this.

? Could this also be coded as a member function of the Money class?

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Stand-alone or member function?

- If the first (LH) operand is NOT an object of the class, you can only use a stand-alone function.
- If the first (LH) operand is a member of the class, then either is possible.
  - a member function is best.
  - Member function is slightly more efficient as data members of the LH operand are then directly available.
  - The compiler will look first for a member function in the class of the LH operand, and only for a stand-alone function if it does not find one.

Overloaded operators: another example

- Suppose we would like to be able to say:

```
Employee emp;
…
cout << emp;
```

- What type is on the LHS of this operator?
  - `ostream`
- What type is on the RHS?
  - `Employee`
- What will the overloaded operator be called?
  - `operator<<`
- Can it be a member function of `Employee`?
  - No - LHS is not an `Employee` object, we must have a stand-alone function.
- Signature?

```
ostream& operator<<(ostream &str, const Employee &emp);
// remember to return the stream object, so that insertions
   can be chained
```

- Can we make the insertion more efficient?
  - Have `Employee` declare the stand-alone function as a friend