Overloading

- Overloading: when you have two or more function definitions for the same function name

- Example 1 (overloaded constructors):
  ```cpp
  student()
  student(std::string name)
  student(int id, std::string name)
  ```

- Example 2 (overloaded functions):
  ```cpp
  double average(double n1, double n2)
  double average(double n1, double n2, double n3)
  ```

Overloading

- The function definitions must have:
  - Different numbers of parameters OR
  - Different types of parameters

- The compiler figures out which function based on the number and types of parameters

- Tip: don’t overload based on const vs non-const or call-by-value vs call-by-reference (this is compiler specific)

Operators

- Examples of operators: +, -, /, %, ==, [], ++, --, <, >, !=

- Sometimes operators work well on different types of variables (operator overloading) eg.
  - `2 + 2` and `2.0 + 2.0`
  - `3 == 3` and `3.0 == 3.0`

- Other times, operators should work but don’t (see next slide)

Operators

You would like equality (==) to work for student objects eg.

```cpp
student s1(1, "Alice");
student s2(1, "Alice");
if( s1 == s2 ) {
    std::cout << "true" << std::endl;
} else {
    std::cout << "false" << std::endl;
}
```

But it doesn’t! Compiling the code above gives you the following error:

```
student.cpp:9: error: no match for 'operator==' in 's1 == s2'
```

Operator Overloading

- What we need to do is to overload the equality operator for student objects

- Let’s define equality for student objects to be: two student objects are equal if
  - Their ids are the same and
  - Their names are the same
Operators

• Binary operators eg. + and == can be rewritten as:
  +(x,y) [ same as x+y ]
  ==(x,y) [ same as x==y ]

• Unary operators eg. – can be rewritten as:
  -(x) [ same as –x ]

Operator Overloading

This code overloads the equality operator for student objects (in student.cpp):

```cpp
bool operator ==(const student& s1, const student& s2) {
    if(( s1.get_id() == s2.get_id() ) &&
        ( s1.get_name() == s2.get_name() )) {
        return true;
    } else {
        return false;
    }
}
```

Note that this is not a member function of the student class. We'll see how to define member functions that overload operators in a bit.

operator.hpp

```cpp
class student {
public:
    // etc.
private:
    // etc.
};
```

Because this is not a member function, the overloaded operator is outside the class definition:

```cpp
student s1(1, "Alice");
student s2(1, "Alice");
if( s1 == s2 ) {
    std::cout << "true" << std::endl;
} else {
    std::cout << "false" << std::endl;
}
```

This code should now work and print out “true”

Operator Overloading

• Now let's redefine the + operator for student objects
• We'll do something silly and say that s1 + s2 produces a new student object with:
  – Id equal to the sum of s1’s id and s2’s id
  – Name equal to s1’s name concatenated with s2’s name

```cpp
const student operator +(const student& s1, const student& s2) {
    return student( s1.get_id() + s2.get_id(),
                    s1.get_name() + s2.get_name() );
}
```

In student.cpp
Operator Overloading

In student.hpp

```cpp
class student {
public:
    // etc.
private:
    // etc.
};
```

```cpp
int student::num_students = 0;
bool operator ==(const student& s1, const student& s2);
const student operator +(const student& s1, const student& s2);
```

Once again, this overloaded operator is outside the class definition:

Student s3 should have:
- Id 2
- Name “AliceAlice”

```cpp
const student operator +(const student& s1, const student& s2) {
    return student( s1.get_id() + s2.get_id(),
                    s1.get_name() + s2.get_name() );
}
```

This returns a const student object.

- Returning a student object allows you to do the following: (s1+s2).print();
- The const means that (s1+s2) cannot be modified. The following will not compile:
  (s1+s2).set_name("Bob");

We can also overload unary operators.
Example: overloading the negation operator for student objects to just negate the id

```cpp
const student operator -(const student& s1) {
    return student( s1.get_id() * -1,
                    s1.get_name());
}
```

But note that the following will work:

```cpp
s3 = (s1+s2);
s3.set_name("Bob");
```

Explanation:
- An anonymous object is created with (s1+s2). This object cannot be changed.
- s3 is created by the default constructor.
- The anonymous object’s properties are copied over to s3

Operator Overloading

Overloading as Member Functions

- We will now make the binary operator + and unary operator – member functions
- Binary operators only need 1 parameter when turned into a member function (the calling object becomes the first parameter)
- Unary operators don’t need any parameters (assumed to apply to the calling object)
Overloading as Member Functions

const student student::operator +(const student& s2) const {
    return student(id + s2.id,
                   name + s2.name);
}

const student student::operator -() const {
    return student(id * -1, name);
}

These are from the calling object

You can use the + operator just like before:

student s1(1, "Alice");
student s2(1, "Alice");
(s1 + s2).print();

s1 is the calling object  s2 is the parameter to the + operator

Updated version of student.hpp

```cpp
class student {
public:
    // etc.
    const student operator +(const student& s2) const;
    const student operator -() const;
private:
    // etc.
};

bool operator ==(const student& s1, const student& s2);
```