Topic: Operator overloading

- Operator functions
- Comparison `<` `>` `==` `>=` `<=` `!=`
- Assignment `=`
- Insertion `<<`
- Concatenation `+` `+=`
- Subscripting `[ ]`
- Friends

Many standard operators are already overloaded

`+` `-` `*` `/`
`<` `>` `==` `!=` `<=` `>=`
`=` `<<` `>>`
`sizeof`
`return`

- Operator overloading is a weak kind of polymorphism
Comparing instances of a class

- A programmer-defined class has no standard relational operators built in.
- A relational operator might work like this:

```cpp
void main()
{
    parts a, b;
    a.input();
    b.input();
    if (a > b) cout << "a";
    else cout << "b";
    cout << is greater";
}
```

- The relational operator could select any member or members for comparison.

Overloading a relational operator

- We could overloaded the > operator by defining an operator function:

```cpp
bool parts::operator>(parts pt2)
{
    return (strcmp(name, pt2.name) > 0);
}
```

- The left operand is the object used to call the function; the right is the parameter (pt2).
- If a and b are of type parts, a > b generates a call to parts::operator>.
### Why overload the assignment operator?

- `strings s1("Hello"), s2; s2 = s1;`
- Normal assignment uses member-by-member (shallow) copy, may leave two pointers pointing at same item

  ![Diagram](image)

- Overloading = for deep copy copies data items pointed to by members of the class

### Constructors for a string class

```cpp
class strings
{
public:
    strings()
    {
        content = new char[1];
        *content = '\0';
    }
    strings(char* s)
    {
        content = new char[strlen(s) + 1];
        strcpy(content, s);
    }
    ~strings() { delete[] content; }
private:
    char* content;
};

void main()
{
    strings s1 = "overloading";
}
```
Assignment operator for a `strings` class

```cpp
class strings
{
public:
    strings();
    strings(char* s);
    ~strings() { delete[] content; };  // Assignment operator accepting <strings>
    void operator=(const strings& str); // object.
private:
    char* content;
}

void strings::operator=(const strings& str)  // Uses deep copy, preventing
{  // double-delete in destructor
    if (content) delete[] content;
    content = new char[strlen(str.content)+1];
    strcpy(content,str.content);
}

void main()
{
    strings s1 = "overloading",s2;
    s2 = s1;                                // [see strovlod.cpp]
}
```

Another idea: use a copy constructor (why?)

Overloading the inserter (<<)

An inserter may be defined for any type. To give inserter
access to private members of your class, declare it as a friend:

```cpp
class widgets {
public:
    widgets(int a0,int b0) { a = a0; b = b0; };
    friend ostream& operator<<(ostream&, widgets);
private:
    int a,b;
};

Overload the inserter as follows:

```cpp
ostream& operator<<(ostream& os,widgets y)
    { os << "(\ " << y.a << ",\ " << y.b << "); return os; }

Use the inserter as follows:

```cpp
widgets my_widget(3, 4);
cout << my_widget;
```
Friend classes and functions

- A class’s friends have access to its nonpublic members
- Friendship is given, not taken
- Friends are declared in class declaration but are not members
- A class’s friend may be:
  - global function (e.g., <<)
  - member function of another class
  - another class

String concatenation operators

```c
strings operator+(char c);
strings operator+(char* s);
strings operator+(strings s);
void operator+=(char c);
void operator+=(char* s);
void operator+=(strings s);
```

Examples:

```c
strings first = "Scott ",
  last = "Valdes",
  name = first + last;
name += ", Esq.";
```
The subscripting operator 

```cpp
char& strings::operator[](int index) // Subscript operator for assignment.
{
    char& c = content[index];
    return c;
}
char strings::operator[](int index) const // Subscript operator for retrieval.
{
    return content[index];
}

void main()
{
    strings s1, s2;
    s1 = "Cup";
    s2 = s1;
    cout << "First letter=" << s2[0] << endl;
    s2[0] = 'P';
    cout << "s1=" << s1 << endl
    << "s2=" << s2 << endl;
}
```

Output:

```
First letter=C
s1=Cup
s2=Pup
```

Binary operators have two operands

- If `operator>` is a member function and `a` is an instance of its class, then `a > b` means `a.operator>(b)`
- If `operator>` is a global function, then `a > b` means `operator>(a,b)`
- …and operator functions must be defined accordingly
Operator overloading

• An operator function must have at least one operand that is an instance of a class
• Name of function is operator plus operator’s spelling (e.g., $operator+$)
• Standard C++ operators only
• Advantages: clarity, conciseness, reuse of familiar notation in new contexts
• Stream I/O is simple for all types