Operator Overloading

Venkatesh

WDC

January 31, 2018

Outline

1. Overloading
2. Operator Overloading
    - Introduction
    - Special Operators
    - Type Conversions
    - Userdefined Literals

Overloading

- Two or more declarations for the `same name` in the same `scope`
- Only Function and Function Templates can be overloaded
- Variable and Type declarations cannot be overloaded

Non Overloadable Declarations

- Functions differ only in return Type
- static and non-static member functions with same name and parameter-type list

Example

```
class X {
static void f();
void f();                    // ill-formed
void f() const;              // ill-formed
void f() const volatile;     // ill-formed
void g();
void g() const;              // OK: no static g
void g() const volatile;     // OK: no static g
};
```

Non Overloadable Declarations...

- Member functions with same parameter-type list, some are with a ref-qualifier

Example

```
class Y {
void h() &;
void h() const &; // OK
void h() && ;    // OK, all declarations have a ref-qualifier
void i()  &;
void i() const;  // ill-formed, prior declaration of i
                 // has a ref-qualifier
};
```

Non Overloadable Declarations...

- Declarations that differ only in the type specifiers are equivalent
- Note : For any type T "pointer to T", "pointer to const T" are distinct

Example

```
typedef const int cInt;

int f (int);
int f (const int);          //  redeclaration of f(int)

int f (int) { /* ... */ }   //  definition of f(int)
int f (cInt) { /* ... */ }  // error: redefinition of f(int)
```

Non Overloadable Declarations...

- Functions with same name in base and derived classes

Example

```
struct B {
    int f(int);
};
struct D : B {
    int f(const char*); // hides B::f(int)
};
void h(D* pd) {
    pd->f(1);        // error : D::f(const char*) hides B::f(int)
    pd->B::f(1);     // OK
    pd->f("Ben");    // OK, calls D::f
}
```

Access Rules

- overloaded member functions can have different access rules

Example

```
class buffer {
private:
    char* p;
    int size;
protected:
    buffer(int s, char* store) { size = s; p = store; }
public:
    buffer(int s) { p = new char[size = s]; }
};
```

Overloaded operators

syntax

```
return-type operator symbol(params)
```

symbol: one of

```
new delete new[] delete[]
+   -   *     /     %     ^     &     |     ~
!   <     >   +=    -=    *=    /=    %=
^=  &=  |=  <<    >>    >>=   <<=     ==    !=
<=  >=  &&    ||    ++    --     ,    ->*
=  ->  ()  []
```

Constraints

- Both unary and binary forms of + − ∗ & can be overloaded
- We cannot introduce new tokens as operators
- Precedence, grouping, number of operands cannot be changed
- Semantics/Identity can be changed

operators cannot be overloaded

```
.   .*  ::  ?:
#   ##    preprocessing symbols
sizeof  alignof  typeid
```

Operator Overloading Rules

- Either `non-static member` function or non-member function
- Atleast one paramter type is a class/enum
- Cannot have default arguments
- `= & (unary) , (comma)` predefined for each type, can be changed
- `= [] () ->` must be non-static member functions

Operator Overloading

Example

```
class X {
public:
    X(int);
    void operator+(int);
};
void operator+(X,X);
void operator+(X,double);
void f(X a) {
    a+1;    // same as  a.operator+(1)
    1+a;    // ::operator+(X(1),a)
    a+1.0;  // ::operator+(a,1.0)
    std::string s = "a" + "b"   // error : both are const char *
}
```

Operator overload Lookup

overload resolution

No preference is given to members over nonmembers

How to Resolve Operators in Namespaces

Example

```
#include <iostream>

int main() {
    std::string s = "hello wolrd";
    std::cout << s; // << is defined in namespace std
    return 0;
}

// std::cout.operator<<(s) or operator(std::cout, s)
```

Operators in Namespaces

Consider a binary operator @, x@y is resolved like this:
x is of type X and y is of type Y.
Look for declarations of operator@

- if X is a class, check for members of X or base of X ; and
- context surrounding x@y ; and
- if X is defined in namespace N, then in N ; and
- if Y is defined in namespace M, then in M

Assignment operator

- `operator=` is a non-static member function with exactly one parameter
- implicitly declared for a class if not declared by the user
- Any assignment operator can be virtual

Function call

- `operator()` is a non-static member function with an arbitrary number of parameters

Function call operator

Example

```
class Action {
public:
    Action();
    int operator()(int);
    pair<int,int> operator()(int,int);
    double operator()(double);
};
void f(Action act)
{
    int x = act(2);
    auto y = act(3,4);
    double z = act(2.3);
};
```

Lambda Functions

- shorthand for defining and using a function object
- By default, `operator()` is const, it doesn't modify the captured variables

Subscripting : operator[]

- non-static member function with exactly one parameter

Example

```
struct Assoc {
    vector<pair<string,int>> vec; // vector of {name,value} pairs
    const int& operator[] (const string&) const;
    int& operator[](const string&);
};
Assoc values;
values[string("key")];
```

Dereferencing: operator->

- non-static member function taking no parameters

Example

```
class Ptr {
    X* operator-> ();
};
void g(Ptr p) {
    p->m = 7;               // (p.operator->())->m = 7
    X* q1 = p->;            // syntax error
    X* q2 = p.operator->(); // OK
}
```

Increment and Decrement

Example

```
struct X {
    X& operator++();     // prefix ++
    X operator++(int);   // postfix ++
};
struct Y { };
Y& operator++(Y&);       // prefix ++b
Y operator++(Y&, int);   // postfix b++

void f(X a, Y b) {
    ++a; // a.operator++();
    a++; // a.operator++(0);
    ++b; // operator++(b);
    b++; // operator++(b, 0);
}
```

Conversion Operators

- conversion from a user-defined type to a built-in type
- X::operator T() where T is a type name, defines a conversion from X to T

Example

```
class unique_ptr {
public:
    explicit operator bool() const noexcept;
};
void use(unique_ptr<Record> p, unique_ptr<int> q) {
    if (p) {
        // use it
    }
    bool b = p;      // error ; suspicious use
    int x = p + q;  // error ; we definitly don't want this
}
```

UserDefined Literals

syntax

operator "" identifier(parameter-declaration-clause)

```
/* identifier is literal suffix identifier */
parameter-declaration-clause is one of :
    const char*
    unsigned long long int
    long double
    char
    const char*, std::size_t
```

Example

```
long double operator "" _km(long double);
```

References

- The C++ Programming Language [4th Edition] - Bjarne Stroustrup

Thank You