

Ad Hoc Polymorphism

In C++, there are 3 primary forms of polymorphism: *parametric*, *subtype*, and *ad hoc* polymorphism, or function/operator overloading. We have seen several examples of method/function overloading, and a couple of examples of operator overloading.

```
String();
String(const char* s);
String(const String& s);
String(char s);
```

What happens if you have the following?

```
String (const char* s = 0);
```

The compiler automatically selects the proper function depending on the type signature of the function (name, return type, and type and order of arguments in the parameter list. This means that for every function definition, there must be an unambiguous decision at compile time to determine the correct function to invoke at run-time.

```
class String {
    char* str;
    int len;
public:
    ...
    friend bool operator==(const String& s1, const String& s2)
    { return strcmp(s1.str, s2.str) == 0 ? true : false; }
    friend bool operator<(const String& s1, const String& s2)
    { return strcmp(s1.str, s2.str) < 0 ? true : false; }
};
```

Note that by just defining the above two relational operators, you do not need to define the other relational operators as either member or friend functions, but can rely on template versions of the functions instead. Why?

5. The following operators may only be implemented as class member functions

```
operator= // assignment operator
operator-> // arrow (member access) operator
operator[] // array index operator
operator() // method/function call operator
```

All others can be implemented either as member functions or as friend functions. They are very often implemented as friend functions.

```
class Date {
public:
    ...
    friend ostream& operator<<(ostream&, Date&);
};
```

6. Overloading operator[] is typically used to access elements of a collection indexed by an integer value. However, a problem is that it is unclear as to what value to return when an invalid index is provided:

```
template<class T> class Array {
    T array[SIZE];
    int n;
public:
    ...
    T& operator[](int i) {
        assert(i >= 0 && i < n); // is this the best we can do?
        return array[i]
    }
};
```

Returning a T& instead of just a T means that the operator[] can be used on the LHS of an assignment (as an lvalue):

```
Array<double> a;
a[0] = 3.14;
```

Operator Overloading

Function overloading is a form of ad hoc polymorphism. Operator overloading is a particular type of function overloading.

1. Overloaded operators require a class argument as the first argument. Declaring an operator as a member function of a class guarantees that a class type is the first argument since the first argument will be a this pointer.

2. Precedence and associativity do not change, so using operator^ to denote exponentiation will not do what you want:

$$2 \wedge 31 - 1 == 2 \wedge (31 - 1)$$

3. You cannot overload the following operators nor can you introduce new kinds of operators:

```
. .* :: ?: // can't be overloaded
x ::= y // cant' define Pascal-like assignment
```

4. Special care must be taken when operators have both prefix and postfix forms:

```
BigInteger x = 1234567890;
int y = x++;
int z = ++x;

class BigInteger { // unlimited precision integers
    ...
public:
    ...
    BigInteger& operator++(); // prefix increment
    BigInteger& operator++(int); // postfix increment requires dummy int type
    // same for operator--
    ...
};
```

7. Overloading the "function call" operator() leads to very interesting usage. For example, you can define objects that represent functions so that functions are treated as "first-class objects". For example, you could implement factorial as an object that memorizes previous factorial computations, or you might implement a generic polynomial class, so that specific polynomial "objects" could be instantiated and then evaluated with various coefficient values.

A polynomial of degree n has the form: $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, where $a_n \neq 0$. We might then implement a class to represent a n th-degree polynomial with real coefficients as follows:

```
class Polynomial {
private:
    int _n;
    Array<double> _coeff;
public:
    Polynomial(int n) : _n(n), _coeff(n + 1) {}
    double& operator[](int i) { return _coeff[i]; }

    double operator()(double x) {
        double r = 0;
        for (int i = _n; i >= 0; i--)
            r = r * x + _coeff[i]; // Horner's rule: n multiplications, n additions
        return r;
    }
    ... // E.g., overloaded arithmetic operators: +, -, *, /
};
```

Define and compute the polynomial $2 - 3x^2 + x^3$ as:

```
Polynomial p(3);
p[0] = 2; p[1] = 0; p[2] = -3; p[3] = 1;
double f = p(4); // p.operator()(4)
```

Horner's Rule (a worthwhile side-note)

A polynomial of degree n has the form: $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, where $a_n \neq 0$

Standard functions, such as $\sin(x)$ and $\cos(x)$ are computed using routines that evaluate polynomials, so fast evaluation of polynomials is important to numerical computing.

Question: how do you compute a value for the polynomial efficiently? I.e., minimize the multiplications, which are expensive operations. That is, replace multiplication by addition when possible.

Knuth, *Fundamental Algorithms Vol. II* is a source of good information on efficient evaluation of polynomials.

Horner's rule (1819), rearranges the calculation to minimize the number of multiplications that are required:

$$p(x) = (\dots(a_n x + a_{n-1})x + \dots)x + a_0$$

The calculation requires n multiplications and n additions, minus one addition for each coefficient that is zero. The alternative method of evaluating an n th-degree polynomial of the above form, starting with the lowest term, requires $2n-1$ multiplications.

The Polynomial class can now be written so that it takes the $n+1$ coefficients as constructor arguments:

```
class Polynomial {
private:
    int _n;
    Array<double> _coeff;
public:
    Polynomial(int n, ...) : _n(n), _coeff(n + 1) {
        va_list ap;
        va_start(ap, n);
        for (int i = 0; i < n + 1; i++)
            _coeff[i] = va_arg(ap, double);
        va_end(ap);
    }

    double& operator[](int i) { return _coeff[i]; }

    double operator()(double x) {
        double r = 0;
        for (int i = _n; i >= 0; i--)
            r = r * x + _coeff[i]; // Horner's rule: n multiplications, n additions
        return r;
    }
};
```

Define and compute the polynomial $2 - 3x^2 + x^3$ with $x = 4$ as:

```
Polynomial p(3, 2.0, 0.0, -3.0, 1.0);
double f = p(4);
```

QUESTION: How would you define a Polynomial consisting of complex or integer coefficients?

Using a Variable Number of Arguments

Suppose that you want an error message function that takes a format string and a variable number of arguments.

```
error_msg("length of '%s' = %d\n", str, strlen(str)); // where str is a char*
error_msg("unable to open file descriptor %d", fd);
```

Use Ellipsis in the type signature to denote a variable argument list:

```
void error_msg(const char* format, ...)
```

The following functions are defined in `<stdarg.h>` and are used to process the variable arguments

```
void va_start(va_list, lastarg);
type va_arg(va_list, type);
void va_end(va_list);
```

```
void error_msg(const char* format, ...)
```

```
{
    va_list ap;
    va_start(ap, format); // initialize arg pointer using last known arg

    for (char* p = format; *p; p++) {
        if (*p == '%') {
            switch (**p) {
                case 'd': int d = va_arg(ap, int); break;
                case 's': const char* s = va_arg(ap, const char*); break;
                ...
            }
        }
        va_end(ap);
    }
}
```

Type Conversion Operators

```
class String {
    char* _str;
    int _len;
public:
    ...
    operator const char*() const { return _str; }
};
```

However, you should not use type conversion operators as a way to get a pointer to the internal implementation of an object. In the case of the String class, obtaining a `const char*` is useful when using C functions to process a C++ String object, and the C function requires a `const char*`. For example:

```
// Assume get_token returns a String object
String token = get_token();
...
// explicit type conversion is effected by "casting" to a const char*
printf("Token is %s\n", (const char*) token);
```

You should be very careful about the following style of type conversion, as it violates the encapsulation of the String class by extracting the `char*` pointer to its internal representation. Note that making it `const` helps reduce inadvertent misuse.

```
String token = get_token();
const char* s = token; // implicit type conversion to sneak a pointer to _str
```

Type conversion operators are however quite useful when the conversion are between related types:

```
void plot(PolarCoord pc)
...
CartesianCoord cc (0.5, 1.0);
plot(cc); // CC object should be safely convertible to a PC object
```