Operator overloading

- almost all C operators can be overloaded
  - new meaning can be defined when one operand is a user-defined (class) type
  - define operator + for object of type T
    T T:operator+(int n) { ... }
  - define regular + for object(s) of type T
    T operator +(T f, int n) ( ... )
  - can’t redefine operators for built-in types
    int operator +(int n, int m) ( ... ) is ILLEGAL

- 3 examples
  - complex numbers
  - IO streams (very briefly)
  - subscripting

Complex numbers

- a complex number is a pair of doubles
  (real part, imaginary part)
- supports arithmetic operations like +, -, *

- a basically arithmetic type for which operator overloading makes sense
  - complex added as explicit type in 1999 C standard
  - in C++, can create it as needed

- also illustrates
  - friend declarations
  - implicit coercions
  - default constructors
Class complex, version 1

class complex {
    private:
        double re, im;
    public:
        complex(double r, double i) { re = r; im = i; }
        friend complex cadd(complex, complex);
        friend complex cmul(complex, complex);
    };

    complex cadd(complex x, complex y)
    {
        complex temp(0, 0); // initial values required
        temp.re = x.re + y.re;
        temp.im = x.im + y.im;
        return temp;
    }

    • this uses ordinary (non-class) functions to
      manipulate complex numbers
    • friend declaration permits cadd() to access
      private representation info
    • awkward notation: for c = a + b * c, write
      c = cadd(a, cmul(b, c));

Version 2: constructors, overloading

class complex {
    private:
        double re, im;
    public:
        complex(double r, double i) { re = r; im = i; }
        complex(double r) { re = r; im = 0; }
        complex() { re = im = 0; }
        friend complex operator +(complex, complex);
        friend complex operator *(complex, complex);
    };

    complex operator +(complex x, complex y)
    {
        return complex(x.re + y.re, x.im + y.im);
    }

    complex a, b, c;
    c = a + b * c;

    • operator overloading gives natural notation
    • multiple constructors permit different kinds of
      initializations
    • no such thing as an uninitialized complex
      - C runtime error is a C++ compile-time error
Version 3: coercions, default args

class complex {
    private:
        double re, im;
    public:
        complex(double r = 0, double i = 0)
        {
            re = r; im = i;
        }
        friend complex operator +(complex, complex);
        friend complex operator *(complex, complex);
    };

    complex a(1.1, 2.2), b(3.3), c(4), d;
    c = 2 * a + b * c;

    • note coercion of 2 -> 2.0 -> complex(2.0)

    • default arguments achieve same results as
      overloaded function definitions

    • normally write initializers as
      complex(double r = 0, double i = 0) : re(r), im(i) {}}

Version 4: change of representation

• polar coordinates (r,θ) instead of (re,im)
• private part changes but external does not have to

class complex {
    private:
        double r, theta;  // polar coordinates
    public:
        complex(double re = 0, double im = 0)
        {
            r = sqrt(re*re+im*im);
            theta = atan2(im, re); /* or whatever */
        }
        friend complex operator +(complex, complex);
        friend complex operator *(complex, complex);
    };

    • friend functions that depend on private part have to
      change
Notes on operator overloading

- applies to all operators except . and ?:
  - operator ( ) left-side function calls
  - operator , simulates lists
  - operator -> smart pointers
- works well for algebraic and arithmetic domains
  - complex, bignums, vectors & matrices, ...
- BUT DON'T GET CARRIED AWAY:
- you can't change precedence or associativity of existing operators
  - e.g., if use ^ for exponentiation, precedence is still low
- you can't define new operators
- meanings should make sense in terms of existing operators
  - e.g., don’t overload - to mean + and vice versa

Simple vector class  (v0.c)

- based on overloading operator [ ]

class ivec {
    int *v;         // pointer to an array
    int size;       // number of elements
public:
    ivec(int n) { v = new int[size = n]; }
    int operator [ ](int n) {      // checked access
        assert(n >= 0 && n < size);
        return v[n];
    }
    int elem(int n) { return v[n]; }   // unchecked
};

main()
{
    ivec iv(10);            // declaration
    int i;
    i = iv.elem(10);        // unchecked access
    i = iv[10];              // checked access
What about lvalue access?

• vector element as target of assignment

```c
main()
{
    ivec iv(10);       // declaration
    iv[10] = 1;         // checked access
    iv.elem(10) = 2;    // unchecked access
}
```

$ g++ v1.c
v1.c:22: non-lvalue in assignment
v1.c:23: non-lvalue in assignment
$ CC v1.c
"v1.c", line 22: Error: The left operand cannot be assigned to.
"v1.c", line 23: Error: The left operand cannot be assigned to.

• need a way to access object, not a copy of it
• in C, use pointers
• in C++, use references

References (swap.c)

• attaching a name to an object
• a way to get "call by reference" (var)
  parameters without using pointers

```c
void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

• a way to access an object without copying it

```c
stack s;
stack t = s; // may not want to copy

f(s);        // ...
return s;    // ...

stack s, t;
t = s; // want to control the assignment
```
Lvalue access \( (\nu2.c) \)

```cpp
class ivec {
    int *v;         // pointer to an array
    int size;       // number of elements
public:
    ivec(int n) { v = new int[size = n]; }
    int& operator [] (int n) {
        assert(n >= 0 && n < size);
        return v[n]; }
    int& elem(int n) { return v[n]; } // unchecked
};
ivec iv(10);            // declaration
iv.elem(10) = 2;        // unchecked access
iv[10] = 1;             // checked access
```

- reference gives access to object so it can be changed

Non-zero origin arrays

```cpp
class ivec {
    int *v;         // pointer to an array
    int size;       // number of elements
    int orig;       // origin; default 0
public:
    ivec(int n) { v = new int[size = n]; orig = 0; } // elems are 0 .. n-1
    ivec(int o, int e) { v = new int[size = e-o]; orig = o; } // elems are o .. e+1
    int& operator [] (int n) {
        assert(n >= orig && n < size+orig);
        return v[n-orig]; }
    int& elem(int n) { return v[n-orig]; } // unchecked
};
main()
{
    ivec iv(2000, 2010);    // declaration
    iv.elem(2000) = 2;      // unchecked access
    iv[2010] = 1;           // checked access
}
```
Iostream library (very quick sketch only)

- how can we do I/O of user-defined types with non-function syntax

- C printf can be used in C++
  - no type checking
  - no mechanism for I/O of user-defined types

- Java System.out.print(arg) or equivalent
  - type checking only in trivial sense:
    - calls toString method for object
  - bulky, notationally clumsy
    - one call per item

- can we do better?

- Iostream library
  - overloads << for output, >> for input
  - permits I/O of sequence of expressions
  - type safety for built-in and user-defined types
  - natural integration of I/O for user-defined types
    - same syntax and semantics as for built-in types

Basic use

- overload operator << for output, >> for input
  - very low precedence
  - left-associative, so
  - \( \text{cout} \ll e1 \ll e2 \ll e3 \)
  - is parsed as
    \( (((\text{cout} \ll e1) \ll e2) \ll e3) \)

- take an \([\text{io}]\text{stream}&\) and a data item
- return the reference

```cpp
#include <iostream.h>
ostream&
operator<<(ostream& o, const complex& c) {
  o << "(" << c.real() << ", " << c.imag() << ")";
  return o;
}
```

- iostreams cin, cout, cerr already open
  - correspond to stdin, stdout, stderr
Input with iostreams

#include <iostream.h>

main()
{
    char name[100];
    double val;

    while (cin >> name >> val) {
        cout << name << " = " << val << "\n";
    }
}