CS 280
Programming Language Concepts

Spring 2025

Operator Overloading,
Generic Functions, and Propagating Values to Objects

Topics

- Overview of Short Assignment 3
- Operator Overloading
- Generic Functions
- Propagating Values
  - Copy and Move Constructors
  - Copy and Move Assignment Operators

Short Assignment 3

- Objective of the SA 3:
  - Writing a program that recognizes and counts the number of three types of words. Those are the *Special Words*, *Keywords*, and *Identifiers.*
  - See the posted problem statement on Canvas.
- Overview of the SA 3
  - The program accepts one or more command line arguments for a file name and optional input flags.
    - Reads from the file words until the end of file.
  - Program recognizes and counts three types of words:
    - A *Special Word* is defined as a word that starts with either $, @ or % followed by zero or more letters, underscores, and digits.
    - *Keyword* is defined as a one of a given list of words.
    - *Identifier* is defined as a word that starts by a letter and can be followed by zero or more letters or digits.
  - Program detects and displays error messages of invalid *Identifiers* or *Special Words.*

Short Assignment 3: Example

- Input file:

Line number	File contents
1	# Simple array constructs.
2	@fred = ("How", "are", "you", "today?");
3	print "\@fred contains (@fred).\n";
4	
5	$mike = $fred[1];
6	print " $mike $fred [3]\n";
7	
8	# The array name in a scalar context gives the size.
9	$fredsize = @fred;
10	print ' @fred has ', " $fredsize elements.\n";
11	
12	# The $#name gives the max subscript (size less one).
13	print "Max sub is $#fred\n";
14	
End of File	

Short Assignment 3: Example (Cont'd)

- Given the shown input file contents, the generated results are as follows:
  - See the posted testing case file for further examples.

```
Invalid Identifier Word at line 1: constructs.
Invalid Special Word at line 5: $fred[1];
Invalid Identifier Word at line 8: size.
Invalid Special Word at line 9: @fred;
Invalid Identifier Word at line 10: elements.\n";
Invalid Special Word at line 12: $#name
Invalid Identifier Word at line 12: one).
Invalid Special Word at line 13: $#fred\n";
Total number of words: 61
Number of Keywords: 0
Number of Identifiers: 25
Number of Special Words Starting with $: 5
Number of Special Words Starting with @: 3
Number of Special Words Starting with %: 0
```

Operator Overloading

- Format
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded.
  - `operator+` would be used to overload the addition operator (+)
- No new operators can be created
  - Use only existing operators
- Not possible to change precedence/associativity or syntax of operators

Overloadable operators in C++

- operators that can be overloaded:
  - arithmetic + - * / %
  - bitwise ^ & | ~ << >>
  - logical ! && ||
  - relational < > <= >= == !=
  - assignment =
  - compound assignment += -= *= /= %= ^= &= |= <<= >>=
  - increment/decrement ++ --
  - subscript []
  - function call ()
  - address, indirection & *
  - others ->* , -> **new delete**

Operators which CANNOT be overloaded

- ?: (conditional)
- . (member selection)
- .* (member selection with pointer-to-member)
- :: (scope resolution)
- sizeof (object size information)
- typeid (object type information)

Example 1: `Complex` Class

```
#ifndef _Complex_H
#define _Complex_H

using namespace std;
class Complex{
  float re, im; // by default private c = re + im i
  public:
   Complex(float x = 0, float y = 0)
       : re(x), im(y) { }
   Complex operator*(Complex rhs) const;
   float GetRe() {
       return re;
   }
   float GetIm() {
       return im;
   }
   float modulus() const;
   void print() const;
};
#endif
```

`Complex` class Interface in the file *Complex.h*

Implementation of `Complex` Class

```
#include <iostream>
#include <cmath>
#include "Complex.h"

Complex Complex::operator*(Complex rhs) const
{
   Complex prod;
   prod.re = (re * rhs.GetRe() - im * rhs.GetIm());
   prod.im = (re * rhs.GetIm() + im * rhs.GetRe());
   return prod;
}
float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}
void Complex::print() const
{
    std::cout << "(" << re <<"," << im << ")" << std::endl;
}
```

`Complex` class implementation in file *Complex.cpp*

Using the class in a Driver File

```
#include <iostream>
#include "Complex.h"
// A program that uses Complex in file TestComplex.cpp
int main()
{
   Complex c1, c2(1), c3(1,2);
   float x;
   // overloaded  * operator!!
   c1 = c2 * c3 * c2;
   // OK. Now we use an authorized public function
   x = c1.modulus();
   c1.print();
    return 0;
}
```

- What if we want to multiply a complex number with a scalar? Define another member function with the same name but different parameters.

```
Complex operator*(float k) const;
```

Using the class in a Driver File

```
#include <iostream>
#include "Complex.h"

int main()
{
   Complex c1, c2(1), c3(1,2);

   c1 = c2 * c3 * c2;
   c1.print();

   c1 = c1 * 5; // translated to c1.operator*(5)
   c1.print();

   // How about this?
   c1 = 5 * c1;

    return 0;
}
```

A program that uses Complex in file *TestComplex.cpp*

Using the class in a Driver File

```
#include <iostream>
#include "Complex.h"

int main()
{
   Complex c1, c2(1), c3(1,2);

   c1 = c2 * c3 * c2;
   c1.print();

   c1 = c1 * 5; // translated to c1.operator*(5)
   c1.print();

   // How about this?
   c1 = 5 * c1; // CANNOT translate to 5.operator*(c1)

   return 0;
}
```

A program that uses Complex in file *TestComplex.cpp*

Putting the Scalar to the Left

- To support multiplying with a scalar on the left, we must define a new function that is outside the class scope.

```
Complex operator*(float k, Complex c){
   Complex prod;
   prod.re = k * c.re; // Compile Error: cannot access re
   prod.im = k * c.im; // Compile Error: cannot access im
   return prod;
}
```

  - Note that this function has access errors: an outside function cannot access the private members of a class! We can solve this in two ways.

  - **Solution 1: Setter/Getter Functions**

```
// add the following functions to the class
   void setRe(float x) { re = x; }
   void setIm(float x) { im = x; }
```

  - **Solution 2: Friend Functions**

Putting the Scalar to the Left

- **Declare the outside function as the friend of this class**. It can then access the private members of the class.
  - Note that the "friend" keyword is not used here. It is only used inside the class (see the previous slide).

```
class Complex
{
    ...
    friend Complex operator*(float k, Complex rhs);
    ...
};
```

Friend Classes

A class may declare another class as a friend as well. In that case all member functions of the "be friended" class can access the private members of its friend class

```
class A
{
...
};

class B
{
...
friend A;
};
```

"A" can access private members of "B" (but not vice versa!!!)

Operator Overloading Rules

- Binary operator can be defined either by: 1) member function taking one argument, or 2) global function taking two arguments.
- For any binary operator @, a@b can be interpreted as a.operator@(b) or operator@(a, b).
- Unary operator can be defined either by: 1) member function taking no arguments, or 2) global function taking one argument.
- For any unary operator @, @a can be interpreted as a.operator@() or operator@(a)
- For any postfix unary operator @, a@ can be interpreted as a.operator@(**int**) or operator@(a, **int**) (where second argument only exists to distinguish postfix operators from prefix ones)

Operator Overloading Rules (cont'd)

- If member and global functions both defined, argument matching rules determine which is called
- **Assignment, function-call, subscript, and member-selection operators must be overloaded as member functions**
- **If first operand of overloaded operator not object of class type, must use global function**
- For most part, operators can be defined quite arbitrarily for user-defined types for example, no requirement that "++x", "x += 1", and "x = x + 1" be equivalent
- Of course, probably not advisable to define operators in very counter intuitive ways, as will inevitably lead to bugs in code.

Generic Functions

- We want to be able to describe an algorithm without having to specify the data types of the items being manipulated. Such an algorithm is often referred to as a **generic algorithm.**
  - **Generic algorithm:** An algorithm in which the actions or steps are defined but the data types of the items being manipulated are not.
  - C++ supports generic algorithms by providing two mechanisms: *function overloading* and *template functions.*

- Function template is a family of functions parameterized by one or more parameters. The syntax for template function has general form:

```
template <parameter_list> function_declaration
```

Generic Functions

- Syntax for template function has general form:

  `template` `<parameter_list> function_declaration`

  - `Parameter_list`: parameters on which template function depends
  - `function_declaration`: function declaration or definition
  - type parameter designated by **class** or **typename** keyword
  - template parameter designated by **template** keyword
  - template parameter must use **class** keyword
  - non-type (integral constant) parameter designated by its type (e.g., **int, or bool**)

- Examples:

```
// declaration of function template
template <class T> T max(T x, T y);
// definition of function template
template <class T> T max(T x, T y)
{return x > y ? x : y;}
```

Generic Functions

- In order to explicitly identify particular instance of template, use syntax: function<parameters> when the function is called, i.e.,

  function_name <type> (parameters);

  - For function template declaration:

  ```
  template <class T> T max(T x, T y);
  ```

  - `max<int> (v1, v2);//` refers to `int max(int, int)`
  - `max<double> (v1, v2); //`refers to `double max(double, double)`

- Compiler only creates code for function template when it is instantiated (i.e., used). Therefore, definition of function template must be visible in place where it is instantiated; consequently, function template definitions usually appear in header file.
  - template code only needs to pass basic syntax checks, unless actually instantiated.

Generic Functions

- Overload resolution proceeds (in order) as follows:
  1. Search for an exact match with zero or more trivial conversions on (non-template) functions; if found call it.
  2. Search for function template from which function that can be called with exact match with zero or more trivial conversions can be generated; if found, call it.
  3. Try ordinary overloading resolution for functions; if function found, call it; otherwise, call is error.
  - In each step, if more than one match found, call is ambiguous and is error.
  - **Template function only used in case of exact match (unless explicitly forced)**

Generic Functions

- Example:

```
template <class T> T max(T x, T y) {
return x > y ? x : y;
}
double x, y, z;
int i, j, k;
// ...
z = max(x, y); // calls max<double>
k = max(i, j); // calls max<int>
z = max(i, x); // ERROR: no match
z = max <double>(i, x); // calls max<double>
```

Propagating Values: Copying and Moving

- Reference Types
  - lvalue reference to object of type T is denoted by T&
  - rvalue reference to object of type T is denoted by T&&
- Initializing reference called reference binding
  - lvalue and rvalue references differ in their binding properties (i.e., to what kinds of objects reference can be bound)
    - in most contexts, lvalue references usually needed
    - rvalue references used in context of **move constructors and move assignment operators (to be discussed later)** to refer to an unnamed object
- example:

```
int x;
int& y = x; // y is lvalue reference to int variable
int&& tmp = 3; // tmp is rvalue reference to int constant object
```

- *Unnamed objects* are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of *unnamed objects* are return values of functions or type-casts.

Propagating Values: Copying and Moving Operations

- Copy operation: Propagating the value of the source object source to the destination object destination by copying.
  - A copy operation does not modify the value of the source object.
  - For more details, see
    - https://en.cppreference.com/w/cpp/language/copy_constructor
- Move operation: Propagating the value of the source object to the destination object destination by moving.
  - A move operation is not guaranteed to preserve the value of the source object. After the move operation, the source object has a value that is valid but typically unspecified. The operation is permitted to modify the source object.
  - For more details, see
    - https://en.cppreference.com/w/cpp/language/move_constructor

Copy Constructor

- For class T, constructor taking lvalue reference to T as first parameter that can be called with one argument known as copy constructor
  - used to create object by copying from already-existing one
  - copy constructor for class T typically is of form T(**const** T&)
- defaulted copy constructor performs member-wise copy of its data members (and bases), where copy performed using:
  - copy constructor for class types
  - bitwise copy for built-in types
  - defaulted copy constructor automatically provided (i.e., implicitly defined) as public member if none of following user declared: move constructor, move assignment operator, copy assignment operator, destructor.

Copy Constructor

```
class Vector { // Two-dimensional vector class.
public:
  // ... (e.g., default constructor)
  Vector(const Vector& v) { // Copy constructor.
  x_ = v.x_; y_ = v.y_;
  }
// ...
private:
  double x_; // The x component of the vector.
  double y_; // The y component of the vector.
};


Vector v;
Vector w(v); // calls Vector(const Vector&)
Vector u = v; // calls Vector(const Vector&)
```

Move Constructor

- For class T, constructor taking rvalue reference to T as first parameter that can be called with one argument known as move constructor
  - used to create object by moving from already-existing object
  - move constructor for class T typically is of form T(T&&)
    - Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc.) rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state. For more details, see

      https://en.cppreference.com/w/cpp/language/move_constructor
- if *no move constructor is* specified (and no destructor, copy constructor, or copy/move assignment operator specified), move constructor is *automatically provided* that moves each data member (using move for class and bitwise copy for built-in type)

Move Constructor

```
class Vector { // Two-dimensional vector class.
public:
  // ...
  Vector(Vector && v) { // Move constructor.
  x_ = v.x_; y_ = v.y_;
  }
// ...
private:
  double x_; // The x component of the vector.
  double y_; // The y component of the vector.
};

Vector x(); // declares function x that returns Vector
Vector y = x(); // calls Vector(Vector&&) if move is defined
```

```
class Vector { // Two-dimensional vector class.
public:
  Vector () { // Default constructor.
  x_ = 0.0; y_ = 0.0;}
  Vector(const Vector& v) { // Copy constructor.
  x_ = v.x_; y_ = v.y_;}
  Vector(Vector && v) { // Move constructor.
  x_ = v.x_; y_ = v.y_;}
  Vector(double x, double y) { // Another constructor.
  x_ = x; y_ = y;
   // ...
private:
  double x_; // The x component of the vector.
  double y_; // The y component of the vector.
};
Vector u; // calls Vector(); u set to (0,0)
Vector v(1.0 , 2.0); // calls Vector(double, double)
Vector w(v); // calls Vector(const Vector&)
Vector z = u; // calls Vector(const Vector&)
Vector x(); // declares function x that returns Vector
Vector y = x(); // calls Vector(Vector&&) if move not elided
```

```cpp
class Complex {// Copy/Move Assignment Operator Example: Complex
public:
  Complex(double x = 0.0, double y = 0.0) :
       x_(x), y_(y) {}
  Complex(const Complex& a) : x_(a.x_), y_(a.y_) {}
  Complex(Complex&& a) : x_(a.x_), y_(a.y_) {}
  Complex& operator=(const Complex& a) { // Copy assign
   if (this != &a) {
     x_ = a.x_; y_ = a.y_;
   }
   return *this;
  }
  Complex& operator=(Complex&& a) { // Move assign
       x_ = a.x_; y_ = a.y_;
       return *this;
  }
private:
 double x_; // The real part.
 double y_; // The imaginary part.
};
int main() {
  Complex z(1.0, 2.0);
  Complex v(1.5, 2.5);
  v = z; // v.operator=(z)
  v = Complex(0.0, 1.0); // v.operator=(Complex(0.0, 1.0))
}
```

Special member Functions

- *Special member functions* are member functions that are implicitly defined as member of classes under certain circumstances. There are six:

Member function	typical form for class C:
Default constructor	C::C();
Destructor	C::~C();
Copy constructor	C::C (const C&);
Copy assignment	C& operator= (const C&);
Move constructor	C::C (C&&);
Move assignment	C& operator= (C&&);

  - The six *special members functions* described above are members implicitly declared on classes under certain circumstances.