C++ Overview (2)

COS320
Heejin Ahn
(heejin@cs.princeton.edu)
Topics

• Last time
  – Heap memory allocation
  – References
  – Classes
  – Inheritance

• Today
  – Operator overloading
  – I/O streams
  – Templates
  – STL
  – C++11
Operator Overloading

• A new meaning can be defined when one operand of an operator is a user-defined (class) type

• Member vs. nonmember operators
  – Class member operator
    • ex) T T::operator+(const T& rhs)
    • ex) T T::operator+(int num)
    • Pros: can access private data member w/o friend declaration
  – Nonmember (global) operator
    • ex) T operator+(const T& lhs, const T& rhs)
    • ex) T operator+(int num, const T& rhs)
    • Pros: we can define operators when type of lhs is not modifiable (ex. ostream) or primitive (ex. int)

• Operators are just functions – these are valid
  – ex) lhs.operator+(rhs)
  – ex) operator+(lhs, rhs)
class Complex {
    double re, im;

public:
    Complex(double re=0, double im=0) : re(re), im(im) {}  

    friend Complex operator +(const Complex &lhs, const Complex &rhs);
    …

    // Assignment operators
    const Complex &operator+=(const Complex &rhs) {
        re += rhs.re;
        im += rhs.im;
        return *this;
    }
    …
};

// Binary arithmetic & relational operators
Complex operator +(const Complex &lhs, const Complex &rhs) {
    return Complex(lhs.re+rhs.re, lhs.im+rhs.im);
}
Complex operator ===(const Complex &lhs, const Complex &rhs) { … }
Complex operator <=(const Complex &lhs, const Complex &rhs) { … }
Complex Number Class

class Complex {
public:
    // Unary operators
    Complex operator-() const
    { return Complex(-re, im); }
    const Complex &operator++() { // Prefix
        ++re;
        return *this;
    }
    Complex operator++(int) { // Postfix (int arg is dummy)
        Complex tmp = *this;
        ++re;
        return tmp;
    }

    // I/O operators
    ostream &operator<<(ostream &out, const Complex &c)
    { c.print(out); return out; }
    istream &operator>>(istream &in, Complex &c)
    { ... }

    void print(ostream &out=cout) const;
};
Complex c1(3, 5), c2(2, 7);

Complex c3 = c1 + c2; // operator+

bool isEqual = c1 == c2; // operator==
bool isLess = c1 < c2; // operator<

c1 += c2; // operator+=

Complex c4 = -c3; // operator- (unary)
++c4; //operator++ (prefix)
c4++; // operator++ (postfix)

// cout << thing is similar to printf("..", thing);
std::cout << c4; // operator<<
More Operator Overloading

• I/O operators
  – ostream &operator <<(const T& t)
  – Now we can do std::cout << t;

• Type casting operators
  – operator double() const
  – operator int() const

• Subscripting operator
  – You may need to overload these if you make your own vector class
  – const ElemT &operator[](int index) const
  – ElemT &operator[](int index)

• Operator overloading should be used judiciously
Templates

• Specifies a class or a function that is the same for several types
• Evaluated in compile time, not run time

• e.g., vector template in STL defines a class of vectors that can be
  instantiated for any particular type
  – vector<int>
  – vector<string>
  – vector<vector<int>>

• Templates vs. inheritance:
  – Use inheritance when behaviors are different for different types
    • ex) Drawing different Shapes is different
  – Use template when behaviors are the same, regardless of types
    • ex) Accessing the n-th element of a vector is the same, no matter what type
      the vector is

from [1]
// vector class example
// This is just for demonstration. Use std::vector instead in your code.

template <typename T>
class vector {
    T *array; // pointer to array
    int size; // number of elements

public:
    vector(int n=1) { array = new T[size = n]; }
    T& operator [](int n) { return array[n]; }
    const T& operator[](int n) const { return array[n]; }
};

vector<int> iv(100); // vector of ints
vector<Complex> cv(20); // vector of Complexes
vector<vector<int>> vvi(10); // vector of vector of ints
// Assumes v.size() > 0
// Wouldn't compile if Object does not provide '<'

template <typename Object>
const Object &findMax(const vector<Object> &v) {
    int maxIndex = 0;
    for (int i = 0; i < v.size(); i++) {
        if (v[maxIndex] < v[i])
            maxIndex = i;
    }
    return v[maxIndex];
}

vector<int> vec {2, 7, 4, 3}; // C++11-style vector initialization
int max = findMax<int>(vec);

vector<MyClass> classVec {MyClass, MyClass, MyClass};
// This doesn’t compile because MyClass does not have ‘<‘ operator
MyClass maxClass = findMax<MyClass>(classVec); (X)
Templates

• Templates are classes/functions wannabe, not actual classes/functions
  – Will not even be compiled if not used

• In general, all template implementation (including member functions) should be in header files
  – Templates should be accessible in compile time, not link time
  – There are workarounds to place methods in source files, but this is the simplest

• Code bloat
  – If you use vector template class for 4 different types, compiler will generate 4 different versions of vector class internally
Templates

• Multiple template parameters
  – \texttt{map\langle typename Key, typename Value\rangle}

• Template nontype parameters
  – \texttt{template\langle typename Object, int size\rangle class Buffer \{ ... \}}
  – \texttt{Buffer<string, 1024> buf;}

• Default template parameters
  – \texttt{template <typename Object=char, int size=4096> class Buffer \{ ... \}}
  – \texttt{Buffer<> buf;}
Function Objects

• Objects to be called as if they were ordinary functions
• Also called functors
• C++ equivalent of C function pointers
• Lots of predefined function objects in STL
  <functional> header
// Object type should have weight() method to compile

template <typename Object>
class LessThanByWeight {
public:
    bool operator()(const Object &lhs, const Object &rhs) const {
        return lhs.weight() < rhs.weight();
    }
};

template <typename Object, typename Comparator>
const Object &findMax(const vector<Object> &v, Comparator lessThan) {
    int maxIndex = 0;
    for (int i = 0; i < v.size(); i++) {
        if (lessThan(v[maxIndex], v[i])) maxIndex = i;
    }
    return v[maxIndex];
}

vector<SomeObject> vec { ... };
SomeObject &maxObj = findMax(vec, LessThanByWeight<SomeObject>());

// Template parameters can take function objects too
std::priority_queue<int, std::vector<int>, LessThanByWeight>> myQueue;
Template Specialization

• Override the default template implementation to handle a particular type in a different way

• Example
  – For this struct template
    template <typename T1, typename T2> void foo() { ... }
  – Full template specialization
    • template<> void foo<int, bool>() { ... }
  – Partial template specialization
    • template<typename T2> void foo<int, T2>() { ... }
    • template<typename T> void foo<T, T*>(()) { ... }
template<typename T> string tostr(T t) {
    stringstream ss;
    ss << t;
    return ss.str();
}

template<> string tostr<bool>(bool val) {
    return val ? "true" : "false";
}

template<> string tostr<float>(float val) {
    char buf[64];
    snprintf(buf, sizeof(buf), "%.8e", val);
    return string(buf);
}

template<> string tostr<string>(string val) { return val; }
Template Metaprogramming (TMP)

• Uses of the C++ template system to perform computation at compile-time
• We are not going to cover this in detail

```cpp
template <int n>
struct factorial {
    enum { value = n * factorial<n-1>::value };
};

template <>
struct factorial<0> {
    enum { value = 1 };
};

// Usage examples:
// factorial<0>::value would yield 1;
// factorial<4>::value would yield 24.
```
I/O Streams

- $\ll$ : output operator
- $\gg$ : input operator

**Properties**

- Very low precedence
- Left-associative, so these two are the same
  - $\text{cout} \ll e1 \ll e2 \ll e3$
  - $((\text{cout} \ll e1) \ll e2) \ll e3$
- Takes a reference to iostream and data item
- Returns the reference so can use same iostream for next expression

**I/O streams**

- istream: input stream
- ostream: output stream
- iostream: input/output stream

*From [1]*
I/O Streams

• Predefined streams in `<iostream>` header
  – istream cin – standard input stream (stdin)
  – ostream cout – standard output stream (stdout)
  – ostream cerr – standard error stream (stderr)

• cout << “Hello World!” << endl;
  – They have to be std::cout and std::endl. You can omit ‘std::’ if you use ‘using namespace std;’

• Stream error state
  – Test state: eof(), bad(), fail(), good()
  – Clear state: clear()
I/O Streams

• File I/O streams
  – declared in `<fstream>
  – ifstream: input file stream
  – ofstream: output file stream
  – fstream: input/output file stream

• String I/O streams (You can use strings like streams)
  – declared in `<sstream>
  – istringstream: input string stream
  – ostringstream: output string stream
  – stringstream: input/output string stream

• Headers
  – `<iostream> - istream, ostream, cout, cin, ...
  – `<fstream> - ifstream, ofstream, fstream, ...
  – `<sstream> - istringstream, ostringstream, stringstream, ...
I/O Stream Class Hierarchy

- ios_base
- ios
- istream
  - ifstream
  - istringstream
- ostream
  - ofstream
  - ostringstream
- iostream
  - fstream
  - stringstream
I/O Stream Class Hierarchy

• Multiple inheritance and typedefs

```cpp
template<class CharT, class Traits=std::char_traits<CharT>>
class basic_ios : public ios_base { ... }

typedef basic_ios<char> ios;
typedef basic_ios<wchar_t> wios;

template<class CharT, class Traits=std::char_traits<CharT>>
class basic_ostream : virtual public std::basic_ios<CharT, Traits> { ... }

typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;

template<typename CharT, typename Traits=std::char_traits<CharT>>
class basic_iostream : public basic_istream<CharT, Traits>, public basic_ostream<CharT, Traits> { ... }

typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
...```

Stream Manipulators

• Stream format manipulators
  – setw, setprecision, setfill, setw
  – left, right, internal
  – dec, hex, oct
  – showbase, showpos, showpoint
    • noshowbase, noshowpos, noshowpoint
  – fixed, scientific
  – boolalpha, skipws, uppercase
    • noboolalpha, noskipws, nouppercase
  – ...

• Stream input manipulators
  – ws

• Stream output manipulators
  – endl, flush
    • endl outputs a newline and flushes the stream
class Person {
public:
    Person(const string &name, double salary=0.0) : name(name), salary(salary) {} 
    void print(ostream &out=cout) const {
        out << left << setw(15) << name << " " << right << fixed << setprecision(2) 
            << setw(12) << salary;
    }
    ... 
};

ostream &operator<<(ostream &out, const Person &p) {
    p.print(out);
    return out;
}

// In some function
vector<Person> arr;
arr.push_back(Person(“Pat”, 40000.11));
arr.push_back(Person(“Sandy”, 125443.10));
for (int i = 0; i < arr.size(); i++)
    cout << arr[i] << endl;
template <typename Object>
void readData(istream &in, vector<Object> &items) {
  items.resize(0);
  Object x;
  string junk; // to skip over bad data

  while (!(in >> x).eof()) {
    if (in.fail()) {
      in.clear();
      in >> junk;
      cerr << "Skipping " << junk << endl;
    } else
      items.push_back();
  }
}

// In some function..
vector<string> vec;
readData<string>(cin, vec);
// Read each line from input.txt and write it to output.txt

// ifstream / ofstream
// Declared in <fstream>

istream& getline(istream &is, string &str); // declared in <iostream>

string line;
ifstream fin("input.txt");
ofstream fout("output.txt");

if (fin.is_open()) {
    while (getline(fin, line))
        fout << line << '\n';
    myfile.close();
} else
    cout << "Unable to open file" << endl;

return 0;
StringStream I/O

- Use a string like a stream
- Declared in `<stringstream>`
- stringstream output is C++ equivalent of C `sprintf/snprintf`

```cpp
// C-style string generation
char buf[100];
snprintf(buf, 100, "The half of %d is %d", 60, 60/2);
printf("%s", buf);

// C++-style string generation
stringstream ss;
ss << "The half of " << 60 << is << 60/2;
cout << ss.str();
```
Standard Template Library (STL)

- General purpose library of data structures including containers, and algorithms using templates
- Generic: every algorithm works on a variety of containers, including built-in types

- Containers: can contain objects of any type
  - Simple: pair
  - Sequences: vector, list, slist, stack, queue, deque
  - Sorted associative: set, map, multiset, multimap, ...
  - Others: priority_queue, bitset, ...

- Iterators: generalization of pointer for uniform access to items in a container
Standard Template Library (STL)

• Algorithms
  – Finding and counting
    • find_if, count_if, search, all_of, any_of, find, ...
  – Modifying sequence
    • copy, copy_if, swap, replace, fill, generate, remove, reverse, ...
  – Sorting
    • sort, stable_sort, nth_element, ...
  – Binary search
    • binary_search equal_range, lower_bound, upper_bound, ...
  – ...

• Function objects
  – Function wrappers
    • function, mem_fn, ...
  – Bind
    • bind, is_bind_expression, ...
  – Arithmetic / comparisons / logical / bitwise operations
    • plus, minus, equal_to, greater, less, less_equal, logical_and, bit_and, ...
Containers

#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <string>
using namespace std;

int main() {
    vector<int> vec;
    vec.push_back(3); vec.push_back(4);

    list<double> lst;
    lst.push_back(3.14); lst.push_front(6.28);

    set<string> s;
    s.insert("foo"); s.insert("bar"); s.insert("foo");

    multiset<string> ms;
    ms.insert("foo"); ms.insert("bar"); ms.insert("foo");

    print(vec); print(lst); print(s); print(ms);
    return 0;
}
Iterators

template <typename Container>
void print(const Container &c, ostream &out=cout) {
    typename Container::const_iterator it;
    for (it = c.begin(); it != c.end(); ++it)
        out << *itr << " ";
    out << endl;
}

- `begin()`: iterator pointing to the first element
- `end()`: iterator referring to the past-the-end element
- `++it` and `it++` advance the iterator `it` to the next location
Iterators

• Have const and non-const forms
  – iterator begin() / iterator end();
  – const_iterator begin() const / const_iterator end() const;

• Reverse iterators
  – rbegin() / rend()

• Dereferencing iterators: *it
  – for map, it->first is key and it->second is value

• Iterator hierarchy
  – Bidirectional iterator
    • Can be incremented or decremented (++it, --it)
    • list, map, ...
  – Random access iterator
    • Can access elements at an arbitrary offset position (it += k)
    • vector
#include <iostream>
#include <map>
#include <utility> // std::pair, std::make_pair

using namespace std;

int main() {
    map<string, pair<int, int>> m;
    m["apple"] = pair<int, int>(1, 3);
    m["banana"] = make_pair(2, 4);

    for (map<string, pair<int, int>>::iterator it = m.begin();
    it != m.end(); ++it)
    cout << it->first << " : " << "(" << (it->second).first << ", "
    << (it->second).second << ")" << endl;

    return 0;
}
// Sort element in a container
vector<int> v {3, 67, 45, 6, 99};

// sort the whole vector using default operator, which is less(<)
sort(v.begin(), v.end());

// sort only first half using the function object greater<int>
sort(v.begin(), (v.end() - v.begin()) / 2, greater<int>());

// Find the string with length 9
template <int len>
class StrLength { // Function object
public:
    bool operator()(const string &s) const { return s.length() == len; }
};

vector<string> v {"strawberry", "apple", "banana"}
vector<string>::iterator it = find_if(v.begin(), v.end(), StrLength<9>());

// Print the vector to cout
copy(v.begin(), v.end(), ostream_iterator<string>(cout, "\n"));
C++11

• C++ standard approved by ISO on August 2011
• Formerly known as C++0x
• Biggest extension since C++98
• Features – we are going to cover only handful of them
  – Initializer lists (for STL containers)
  – Template alias
  – Rvalue references
  – Variadic templates
  – Lambdas
  – auto
  – range-for
  – Smart pointers: shared_ptr / unique_ptr / weak_ptr
  – nullptr
  – ...


C++11 Additions

- **nullptr**
  - Type-safe and unambiguous replacement for NULL and 0 pointer values

- **auto**
  - Infers the type of x from the type of the initializing value
  - auto x = val;
  - replaces
  - `VeryLongTypeNameLikeWhatYouOftenSeeInJava x = val;`

- **range-for**
  - for (v : whatever) ...
  - replaces
  - for (.... it = whatever.begin(); it != whatever.end() ++it) ...

- **Now >> is possible**
  - C++03: `vector<vector<int>> v;`
  - C++11: `vector<vector<int>> v;`
auto, range-for

// C++03
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
  cout << *it << endl;
for (std::map<int, string>::iterator it = m.begin(); it != m.end(); ++it)
  cout << it->first << ": " << it->second << endl;

// C++11: using auto
for (auto it = v.begin(); it != v.end(); ++it)
  cout << *it << endl;
for (auto it = m.begin(); it != m.end(); ++it)
  cout << it->first << ": " << it->second << endl;

// C++11: using auto with ranged-for
for (auto &e : v)
  cout << e << endl;
for (auto &kv : m)
  cout << kv.first << ": " << kv.second << endl;
C++11 Additions

• Initializer lists for STL containers
  – `std::vector<int> v {34, 23};`
  – `std::vector<int> v = {34, 23};`
  – `std::map<int, string> m = {{1, "hello"}, {5, "world"}};`

• Smart pointers
  – Helps memory management – you don’t need to delete raw pointer manually, which is very error-prone
  – `shared_ptr`: shared ownership (reference counting)
  – `unique_ptr`: unique ownership
  – `weak_ptr`: no ownership
shared_ptr

- Reference-counted ownership of its contained raw pointer
- If the number of users reach 0, deletes the pointer

```cpp
#include <memory>
...

class MyClass {
    public:
        ~MyClass() { cout << "~MyClass" << endl; }
    }

void func() {
    vector<shared_ptr<MyClass>> vec;
    {
        shared_ptr<MyClass> t(new MyClass());
        vec.push_back(t);
    } // "~MyClass" would have been printed here if 't' was not in 'vec'
    cout << "after the block" << endl;
} // "~MyClass" is printed here; now # of users is 0
```
GDB: The GNU Project Debugger

- Standard debugger for GNU operating system
- Supports many programming languages
  - Ada, C, C++, Objective-C, Free Pascal, Fortran, Java, ...
- If you haven’t used it, learn it!
  - There are many tutorials on the internet
  - And it’s not that difficult after all

- Some important commands
  - b(breakpoint), p(print)
  - u(go up), d(go down) (stack frame)
  - r(run), c(continue), ctrl+c(stop), s(step into), n(step over)
  - And most of all, h(help)
Helpful Sites

- http://www.cplusplus.com/
- http://stackoverflow.com/
- And
- http://www.google.com
References
