Data Structures

Advances in C++ (3)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Operator overloading**
- Exception handling

Recall our MyVector class

```
class MyVector
{
private:
  int n;
  double* m;
public:
  MyVector() : n(0), m(NULL) { };
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector() { delete [] m; }
  void print() const;
};
```

```
MyVector::MyVector(int n, double m[])
{
  this->n = n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = m[i];
}
MyVector::MyVector(const MyVector& v)
{
  this->n = v.n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = v.m[i];
}
void MyVector::print() const
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

Comparing `MyVector` objects

- When we have many vectors, we may need to **compare** them.
- For vectors u and v:
  - $u = v$ if their dimensions are equal and $u_i = v_i$ for all i.
  - $u < v$ if their dimensions are equal and $u_i < v_i$ for all i.
  - $u \leq v$ if their dimensions are equal and $u_i \leq v_i$ for all i.
- How to add **member functions** that do comparisons?
  - Naturally, they should be **instance** rather than static functions.

Member function `isEqual()`

```
class MyVector
{
private:
  int n;
  double* m;
public:
  MyVector() : n(0), m(NULL) { };
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector() { delete [] m; }
  void print() const;
  bool isEqual(const MyVector& v) const;
};
```

```
bool MyVector::isEqual(const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else
  {
    for(int i = 0; i < n; i++)
    {
      if(this->m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```

`isEqual()` is fine, but …

- Adding the instance function **`isEqual()`** is fine.
  - But it is not intuitive.
  - If we can write **`if(a1 == a2)`**, it will be great!
- Of course we cannot:
  - The compiler does not know what to do to this statement.
  - We need to define **==** for **`MyVector`** just as we define member functions.
- In fact, **==** has been **overloaded** for different data types.
  - We may compare two **`int`**s, two **`double`**s, one **`int`** and one **`double`**, etc.
  - We will now define how **==** should compare two **`MyVector`**s.
- This is **operator overloading**.

Operator overloading

- Most operators (if not all) have been overloaded in the C++ standard.
  - E.g., the division operator `/` has been overloaded.
  - Divisions between integers is just different from divisions fractional values!
- Overloading operators for self-defined classes are **not required**.
  - Each overloaded operator can be replaced by an instance function.
  - However, it often makes programs **clearer** and the class **easier to use**.
- Some **restrictions**:
  - Not all operators can be overloaded (see your textbook).
  - The number of operands for an operator cannot be modified.
  - New operators cannot be created.

Overloading an operator

- An operator is overloaded by "implementing a **special instance function**".
  - It cannot be implemented as a static function.
- Let ***op*** be the operator to be overloaded, the "special instance function" is always named

```
operatorop
```

  - The keyword `operator` is used for overloading operators.
- Let's overload `==` for `MyVector`.

Overloading ==

- Recall that we defined **`isEqual()`**:

```
class MyVector
{
private:
  int n;
  double* m;
public:
  MyVector() : n(0), m(NULL) { };
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector() { delete [] m; }
  void print() const;
  bool isEqual(const MyVector& v) const;
};
```

```
bool MyVector::isEqual(const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else
  {
    for(int i = 0; i < n; i++)
    {
      if(this->m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```

Overloading ==

- To overload ==, simply do this:

```
class MyVector
{
private:
  int n;
  double* m;
public:
  MyVector() : n(0), m(NULL) { };
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector() { delete [] m; }
  void print() const;
  bool operator==(const MyVector& v) const;
};
```

```
bool MyVector::operator==(const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else
  {
    for(int i = 0; i < n; i++)
    {
      if(this->m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```

- So easy!

Invoking overloaded operators

- We are indeed implementing instance functions with special names.
- Regarding **invoking** these instance functions:

```
int main() // without operator overloading
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);

  double d2[4] = {1, 2, 3, 4};
  const MyVector a2(4, d2);
  const MyVector a3(a1);

  a1.isEqual(a2) ? cout << "Y\n" : cout << "N\n";
  a1.isEqual(a3) ? cout << "Y\n" : cout << "N\n";

  return 0;
}
```

```
int main() // with operator overloading
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);

  double d2[4] = {1, 2, 3, 4};
  const MyVector a2(4, d2);
  const MyVector a3(a1);

  a1 == a2 ? cout << "Y\n" : cout << "N\n";
  a1 == a3 ? cout << "Y\n" : cout << "N\n";

  return 0;
}
```

Invoking overloaded operators

- Interestingly, we may also do:

```
int main() // with operator overloading
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);

  double d2[4] = {1, 2, 3, 4};
  const MyVector a2(4, d2);
  const MyVector a3(a1);

  a1.operator==(a2) ? cout << "Y\n" : cout << "N\n";
  a1.operator==(a3) ? cout << "Y\n" : cout << "N\n";

  return 0;
}
```

- Other comparison operations (**<**, **!=**, etc.) can all be overloaded similarly.

Parameters for overloaded operators

- The number of parameters is **restricted** for overloaded operators.
  - The **types of parameters** are not restricted.
  - The **return type** is not restricted.
  - **What is done** is not restricted.
- Always avoid unintuitive implementations!

```
class MyVector
{
  // ...
  bool operator==(const MyVector& v) const;
  bool operator==(MyVector v) const;
  void operator==(int i) const
  {
    cout << "...\n";
  } // no error but never do this!
  bool operator==(int i, int j); // error
};
```

Overloading the indexing operator

- Another natural operation that is common for vectors is indexing.
  - Given vector v, we want to know/modify the element v_i.
- For C++ arrays, we use the **indexing operator `[]`**.

```
class MyVector
{
  // ...
  double operator[](int i) const;
};
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1); // terminate the program!
             // required <cstdlib>
  return m[i];
}
```

More are needed for `[]`

- Compiling the program with the main function below results in an error!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1); // non-const
  cout << a1[3] << endl; // good
  a1[1] = 4; // error!

  return 0;
}
```

- Error: **`a1[1]`** is just a **literal**, not a variable.
  - A literal cannot be put at the LHS in an assignment operation!
  - Just like **`3 = 5`** results in an error.

Another overloaded []

- Let's overload `[]` into another version:

```
class MyVector
{
  // ...
  double operator[](int i) const;
  double& operator[](int i);
};
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
double& MyVector::operator[](int i)
{
  if(i < 0 || i >= n) // same
    exit(1);          // implementation!
  return m[i];
}
```

- The second implementation returns a **reference** of a member variable.
  - Modifying that reference modifies the variable.

Two different []

- Now the program runs successfully!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);
  cout << a1[1] << endl; // 2
  a1[1] = 4; // good
  cout << a1[1] << endl; // 4

  return 0;
}
```

- Which `[]` is invoked?
  - The **`const`** after the function prototype is the key.

```
class MyVector
{
  // ...
  double operator[](int i) const;
  double& operator[](int i);
};
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
double& MyVector::operator[](int i)
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
```

Default assignment operator

- When we do an assignment, what do we expect?
- In fact, the assignment operator has been overloaded!
  - The compiler adds a **default assignment operator** into each class.
  - It simply **copies each instance variable** to its corresponding one.
  - Just like the default copy constructor.
- What may be wrong when we run the main function with the default assignment operator?

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  double d2[4] = {1, 2, 3, 4};
  MyVector a1(5, d1);
  MyVector a2(4, d2);

  a2.print();
  a2 = a1; // dangerous!
  a2.print();

  return 0;
}
```

Overloading the assignment operator

- The assignment operator may need to be manually overloaded.
- Our first implementation:

```
class MyVector
{
  // ...
  void operator=(const MyVector& v);
};
```

```
void MyVector::operator=(const MyVector& v)
{
  if(this->n != v.n)
  {
    delete [] this->m;
    this->n = v.n;
    this->m = new double[this->n];
  }
  for(int i = 0; i < n; i++)
    this->m[i] = v.m[i];
}
```

- How about **`a1 = a1`**?

Overloading the assignment operator

- Our second implementation:

```
class MyVector
{
  // ...
  void operator=(const MyVector& v);
};
```

```
void MyVector::operator=(const MyVector& v)
{
  if(this != &v)
  {
    if(this->n != v.n)
    {
      delete [] this->m;
      this->n = v.n;
      this->m = new double[this->n];
    }
    for(int i = 0; i < n; i++)
      this->m[i] = v.m[i];
  }
}
```

- How about **a1 = a2 = a3**?

Overloading the assignment operator

- Our third implementation:

```
class MyVector
{
  // ...
  MyVector& operator=(const MyVector& v);
};
```

- To avoid **(a1 = a2) = a3**, we may return **const MyVector&**.

```
MyVector& MyVector::operator=(const MyVector& v)
{
  if(this != &v)
  {
    if(this->n != v.n)
    {
      delete [] this->m;
      this->n = v.n;
      this->m = new double[this->n];
    }
    for(int i = 0; i < n; i++)
      this->m[i] = v.m[i];
  }
  return *this;
}
```

Preventing assignments and copying

- In some cases, we **disallow** assignments between objects of a certain class.
  - To do so, overload the assignment operator as **private** or **protected**.
- In some cases, we disallow creating an object by **copying** another object.
  - To do so, implement the copy constructor as **private** or **protected**.
- The copy constructor, assignment operator, and destructor form a group.
  - Either you need **none** of them, or you need **all** of them.

Self-assignment operators

- For vectors, it is often to do arithmetic and assignments.
  - Given vectors u and v of the same dimension, the operation u += v makes u_i become $u_i + v_i$ for all i.
- Let's overload **+=**:
  - Why returning **`const MyVector&`**?
- Returning **`MyVector&`** allows **`(a1 += a3)[i]`**.
- Returning **`const MyVector&`** disallows **`(a1 += a3) = a2`**.

```
class MyVector
{
  // ...
  const MyVector& operator+=(const MyVector& v);
};
const MyVector& MyVector::operator+=(const MyVector& v)
{
  if(this->n == v.n)
  {
    for(int i = 0; i < n; i++)
      this->m[i] += v.m[i];
  }
  return *this;
}
```

Arithmetic operators

- Overloading an arithmetic operator is not hard.
- Consider the addition operator **+** as an example.
  - Take **`const MyVector&`** as a parameter.
  - Add each pair of elements one by one.
  - Do not modify the parameter object.
  - Return **`const MyVector`** to allow **`a1 + a2 + a3`** but disallow **`(a1 + a2) = a3`**.

Overloading the addition operator

- Let's try to do it.

```
class MyVector
{
  // ...
  const MyVector operator+(const MyVector& v);
};
const MyVector MyVector::operator+(const MyVector& v)
{
  MyVector sum(*this); // creating a local variable
  sum += v; // using the overloaded +=
  return sum;
}
```

- Why not returning **`const MyVector&`**?
  - Hint: What will have to **`sum`** after the function call is finished?

Overloading the addition operator

- We may overload it for another parameter type:

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);
  MyVector a2(5, d1);

  a1 = a1 + a2; // good
  a1.print();
  a1 = a2 + 4.2; // good
  a1.print();

  return 0;
}
```

```
class MyVector
{
  // ...
  const MyVector operator+(const MyVector& v);
  const MyVector operator+(double d);
};
const MyVector MyVector::operator+(const MyVector& v)
{
  MyVector sum(*this); // creating a local variable
  sum += v; // using the overloaded +=
  return sum;
}
const MyVector MyVector::operator+(double d)
{
  MyVector sum(*this);
  for(int i = 0; i < n; i++)
    sum[i] += d;
  return sum;
}
```

Instance function vs. global function

- One last issue: addition is **commutative**, but the program below does not run!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);

  a1 = 4.2 + a1; // bad!
  a1.print();

  return 0;
}
```

- We cannot let a double variable invoke our "instance function **`operator+`**".
- We should overload **+** as a **global function**.

A global-function version

- To overload **+** as global functions, we need to handle the three combinations:

```
const MyVector operator+(const MyVector& v, double d)
{
  MyVector sum(v);
  for(int i = 0; i < v.n; i++) // What do we need for this?
    sum[i] += d; // pairwise addition
  return sum;
}
const MyVector operator+(double d, const MyVector& v)
{
  return v + d; // using the previous definition
}
const MyVector operator+(const MyVector& v1, const MyVector& v2)
{
  MyVector sum(v1);
  return sum += v2; // using the overloaded +=
}
```

A global-function version

- Now all kinds of addition may be performed:

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);
  MyVector a3(a1);

  a3 = 3 + a1 + 4 + a3;
  a3.print();

  return 0;
}
```

- Each operator needs a separate consideration.

Outline

- Operator overloading
- **Exception handling**

Exceptions

- **Exceptions** are those thing that are not expected to happen.
  - When one writes a program, that typically refers to **logic** or **run-time** errors.
- Consider the following example:

```
#include <iostream>
using namespace std;

void f(int a[], int n)
{
  int i = 0;
  cin >> i;
  a[i] = 1; // logic error?
}
```

```
int main()
{
  int a[5] = {0};
  f(a, 5);
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

Exceptions

- Some **checks** can be helpful:

```
#include <iostream>
using namespace std;

bool f(int a[], int n)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    return false;
  a[i] = 1;
  return true;
}
```

```
int main()
{
  int a[5] = {0};
  f(a, 5);
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

- The client can check the value returned by **`f()`** to do appropriate responses.

Exceptions

- Some **checks** may not be enough.
  - Even if the function has **multiple reasons** to return **`false`**, the client will not see the reason.
  - We **cannot send messages** to the client about the error (do not print out an error message on the screen!).
  - We **cannot enforce** the client to respond to the returned value.
- C++ (and many other modern languages) offers **exception handling**.
  - A mechanism for handling **logic** or **run-time error**.
  - A function can report the occurrence of an error by **throwing an exception**.
  - One **catches an exception** and then respond accordingly.

Try and catch

- In C++, we use a **try** block and **catch** blocks.

```
try
{
  // statements that may throw exceptions
}
catch(ExceptionClass identifier) // this kind?
{
  // responses
}
catch(AnotherExceptionClass identifier) // that kind?
{
  // other responses
}
```

- A **try** block must be followed by **at least one** **catch** block.
- Try to include only statements that may throw exceptions in a **try** block.

Try and catch

- When a statement (function or method) in a **`try`** block causes an exception:
  - Control **ignores the rest statements** in the **`try`** block.
  - Control passes to the catch block **corresponding to the exception** (if any).
  - After the catch block executes, control passes to statements after this try-catch block.
- If there is no applicable catch block for an exception, **abnormal program termination** usually occurs.
- If an exception occurs in the middle of a try block, the **destructors** of all (static) objects local to that block are called.
  - This is to ensure that all resources allocated in that block are released.
  - It is suggested **not to dynamically allocate** anything inside a try block.

Example: string::replace()

- The **replace()** function of C++ strings is easy to use.

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

void g(string& s, int i)
{
  s.replace(i, 1, ".");
}
```

```
int main()
{
  string s = "12345";
  int i = 0;
  cin >> i;
  g(s, i);
  cout << s << endl;
  return 0;
}
```

- It is defined to be able to **throw** an **out_of_range exception**.
  - **out_of_range** is a class defined in **<stdexcept>**.
  - If we do not respond to the exception, the program terminates abnormally.

Example: string::replace()

- Let's try and catch!

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

void g(string& s, int i)
{
  try {
    s.replace(i, 1, ".");
  }
  catch(out_of_range e) {
    cout << "...\n";
  }
}
```

```
int main()
{
  string s = "12345";
  int i = 0;
  cin >> i;
  g(s, i);
  cout << s << endl;
  return 0;
}
```

Example: string::replace()

- We may also try and catch in the client:

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

void g(string& s, int i)
{
  s.replace(i, 1, ".");
}
```

```
int main()
{
  string s = "12345";
  int i = 0;
  cin >> I;
  try {
    g(s, i);
  }
  catch(out_of_range e) {
    cout << "...\n";
  }
  cout << s << endl;
  return 0;
}
```

- A thrown exception will be passed to callers until one catches it.
  - If no one catches it, the program terminates abnormally.

Standard exception classes

- In the C++ standard library, we have the following standard exception classes:
  - Inheritance and polymorphism!

```
try {
  g(s, i);
}
// this also works
catch(logic_error e) {
  cout << "...\n";
}
```

  - Include **<stdexcept>** to use them.

```
exception
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        range_error
        overflow_error
        underflow_error
```

Throwing an exception

- We may also **throw an exception** by ourselves.

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(logic_error)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    throw logic_error("...");
  a[i] = 1;
}
```

```
int main()
{
  int a[5] = {0};
  f(a, 5);
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

- This **enforce** the client to **catch** the exception!

Throwing an exception

- Let the client **catch** the exception:

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(logic_error)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    throw logic_error("...");
  a[i] = 1;
}
```

```
int main()
{
  int a[5] = {0};
  try {
    f(a, 5);
  }
  catch(logic_error e) {
    cout << e.what();
  }
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

- **`what()`** returns the message generated when throwing an exception.

Modifying the function header

- Functions that throw an exception have a **`throw` clause** at the end of their headers.
  - This restricts the exceptions that a function can throw.
  - Omitting a **`throw`** clause allows a function to throw any exception.
- To allow multiple types of exceptions, write like:

```
void f(int a[], int n) throw(type1, type2)
```

- The documentation of a function (or method) should indicate any exception it might throw.

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(logic_error)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    throw logic_error("...");
  a[i] = 1;
}
```

Defining your own exception classes

- C++ Standard Library supplies a number of exception classes.
- You may also want to define your own exception class.
  - This helps your program communicate better to your clients.
  - Your own exception classes should inherit from standard exception classes for a standardized exception working interface.

```
#include <stdexcept>
using namespace std;

class MyException : public exception
{
public:
  MyException(const string& msg = "")
    : exception(msg.c_str()) {}
};
```

Defining your own exception classes

- Let's use our own exception class:

```
#include <iostream>
#include <stdexcept>
using namespace std;

void f(int a[], int n) throw(MyException)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    throw MyException("...");
  a[i] = 1;
}
```

```
int main()
{
  int a[5] = {0};
  try {
    f(a, 5);
  }
  catch(MyException e) {
    cout << e.what();
  }
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

Applying these techniques

- Operator overloading may be applied to **`ArrayBag`** and **`LinkedBag`**:
  - One may “add” two bags to create one bag.
  - One may “subtract” one bag from another bag.
  - One may “compare” two bags.
- Exception handling may be applied to **`ArrayBag`** and **`LinkedBag`**:
  - When the bag size limit is reached, throw an exception.
  - When a non-existing item is asked to be removed, throw an exception.