DM841

Discrete Optimization

Lecture 4

Elements of C++

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

C++ for optimization

- Several classes needed, we start bottom-up
- Input class (+ auxiliary classes)
- Output class
- Solver class/functions (we will see)
- For greedy solvers, no tool needed (implement from scratch)
- Exercise for tomorrow (?)
  - enumeration solver for a Patient Assignment problem
  - we will provide the input and output classes and the driver
  - you write the solver

Review of C++ basic concepts

- Encapsulation (functions in the class as opposed to C)
- Constructors
- Public and private parts
- Templates
- STL: `vector`

Problem: Bus Driver Scheduling

Problem 22 of CSPLib (www.csplib.org)

- Formulation
  - Tasks/Works
  - Shifts (with costs)
  - Select subset of shifts such that all tasks are covered and the total cost is minimized
  - It is a set covering problem
- Input data format
- Source code: Input class
- Redundant data
- Output data format (vector of Boolean vs. vector of shifts)

Instance File

```
10 12 3
4 3 0 2 3
6 3 1 2 4
5 3 1 4 5
6 3 4 5 6
7 4 2 3 4 5
3 4 6 7 8 9
5 4 3 4 5 8
3 3 2 3 1
5 3 0 8 9
3 3 3 7 8
4 3 3 4 9
6 4 0 2 4 6
```

(Some functions C-like: `string.c_str()` needed to transform a string in char array as in C)

Source code

In `BusDriverScheduling/InputOutput`:
In implementation order:

1. Header file `BDS_Input.hh`, getters
2. Source file `BDS_Input.cc`
3. Driver `BDS_Driver.cc` (main for testing)
4. Header file `BDS_Output.hh`, setters
5. Source file `BDS_Output.cc`

Passing Objects

- In C++ objects are passed:
  - by value `F(A x)`
  - by reference `F(A& x)`
- In java objects are passed by reference, `F(A& x)`

In C++: `F(const A& x)` pass the object but do not change it.
If `F(A& x) const` the function does not change anything
Compare:

```
bool operator[](unsigned i) const { return selected_shifts[i]; }
bool& operator[](unsigned i) { return selected_shifts[i]; }
```

Inheritance

- General idea: extension of a class
- Example with `A` and `B`
- Access level `protected`
- Main (most proper) use: conceptual *is-a* hierarchy
- Hidden fields: syntax with `::` (double colon), eg `A::a1`
- Hidden methods (rewritten)
- Types of inheritance: public, private, and protected (for us: only public)
- Invocation of constructors with inheritance: use of `:`
- Compatibility between base class and derived class (asymmetric)
- Example: `Person` and `Student` (with ID number and grades)
  - Person + driver
  - Student + driver
  - `Makefile` and definition of variables in `Makefile`

```
class A
{
public:
   A(int p1, double p2) { a1 = p1; a2 = p2; }
   int M1() const { return a1; }
   double a2;
protected: //not private
   int a1;
};

class B : public A
{
public:
   B(int p1, double p2, unsigned p3) : A(p1,p2) { b1 = p3; }
   unsigned B1() const { return b1; }
   void SetA1(int f) { a1 = f; }
private:
   unsigned b1;
};

int main()
{
   A x(1,3.4);
   B y(-4,2.3,10);

   y.SetA1(-23);
   cout << y.a2 << " " << y.M1() << endl;
   return 0;
}
```

Virtual functions

- Compatibility between base class and derived class (asymmetric)
- Parameters by value or by reference
- Compatibility in case of redefined methods
- Late binding
- Pure virtual functions
- Abstract classes

```
class A {
public:
  A(int p1, double p2) { a1 = p1; a2 = p2; }
  virtual int M1() const { cout << "A::M1"; return a1; }
  double a2;
  virtual int H() = 0;
protected:
  int a1;
};

class B : public A {
public:
  B(int p1, double p2, unsigned p3) : A(p1,p2) { b1 = p3; }
  unsigned B1() const { return b1; }
  void SetA1(int f) { A::a1 = f; }
  int M1() const { cout << "B::M1"; return a2; }
protected:
  unsigned b1;
  vector<float> a1;
};

void F(A& a) {
  cout << a.M1() << endl;
}

int main() {
  A x(1,3.4);
  B y(-4,2.3,10);
  F(y);
  return 0;
}
```

Redefinition M1

```
class A {
int M1() {return a1;}
int a1
}

class B {
int M1() {return a1;}
int a1;
}


A a(,);
B b(,,);
x=b.M1();

cout<<x<<" "<<a.M1()<<endl;
```

Virtual functions

```
void F(A a)

A x(,);
B y(,,);

F(y);
```

It calls method from class A. It copies an object of class B in A by removing what y had more. It doesn't even know that A exist

```
void F(A& a)
```

function for class A
It is not obvious which one of A or B it is going to use.
Eg. Persons (A) and student (B)

Methods are of two types:

- Final (in java) methods
- Virtual methods

If F is a virtual method it calls the last one defined.
Virtual $\rightsquigarrow$ Late binding makes binding between F and M late, ie, at execution time.

Pure virtual functions

We can have that the function is undefined in the parent class:

```
virtual int H() = 0;
```

pure virtual function, not defined but only redefined.

`A` becomes an abstract class hence we cannot define an object of class A. Like interfaces in java. There everything is virtual, here it is mixed.

Why? I might have different subclasses that implement the functions in different ways.

Framework

Framework set of abstract classes used by inheritance and definition of methods. It gives indication about where to put everything.
Like a library. But instead of calling it calls your methods.

- Pure virtual methods are called hot spots.
- Warm spots (keep or redefine), virtual functions
- Cold spots are those already defined
  Hollywood principle: don't call us, we call you.

Example: Enumeration

Full search space examination. Search space defined by complete solutions.

```
class Enumeration
{
public:
  bool Search();
protected:
  virtual void First() = 0;
  virtual void Next() = 0;
  virtual bool Last() = 0;
  virtual bool Feasible() = 0;
};
```

```
#include "Enumeration.hh"

bool Enumeration::Search()
{
  First();
  if (Feasible())
    return true;
  do
    {
      Next();
      if (Feasible())
        return true;
    }
  while (!Last());
  return false;
}
```

Enumeration: Example N-queens

`Next`
Exhaustive generation of permutations

`1234567`

`...`
`1257643`

list in lexicographic order: from smallest number to largest.
which is the next one?
go on until increasing
5 is the first one after which decreasing
then go back and find the first bigger and exchange and reverse completely.

`12 63457`

`Last` if sequence completely decreasing.
`Next` can check and change

Casting

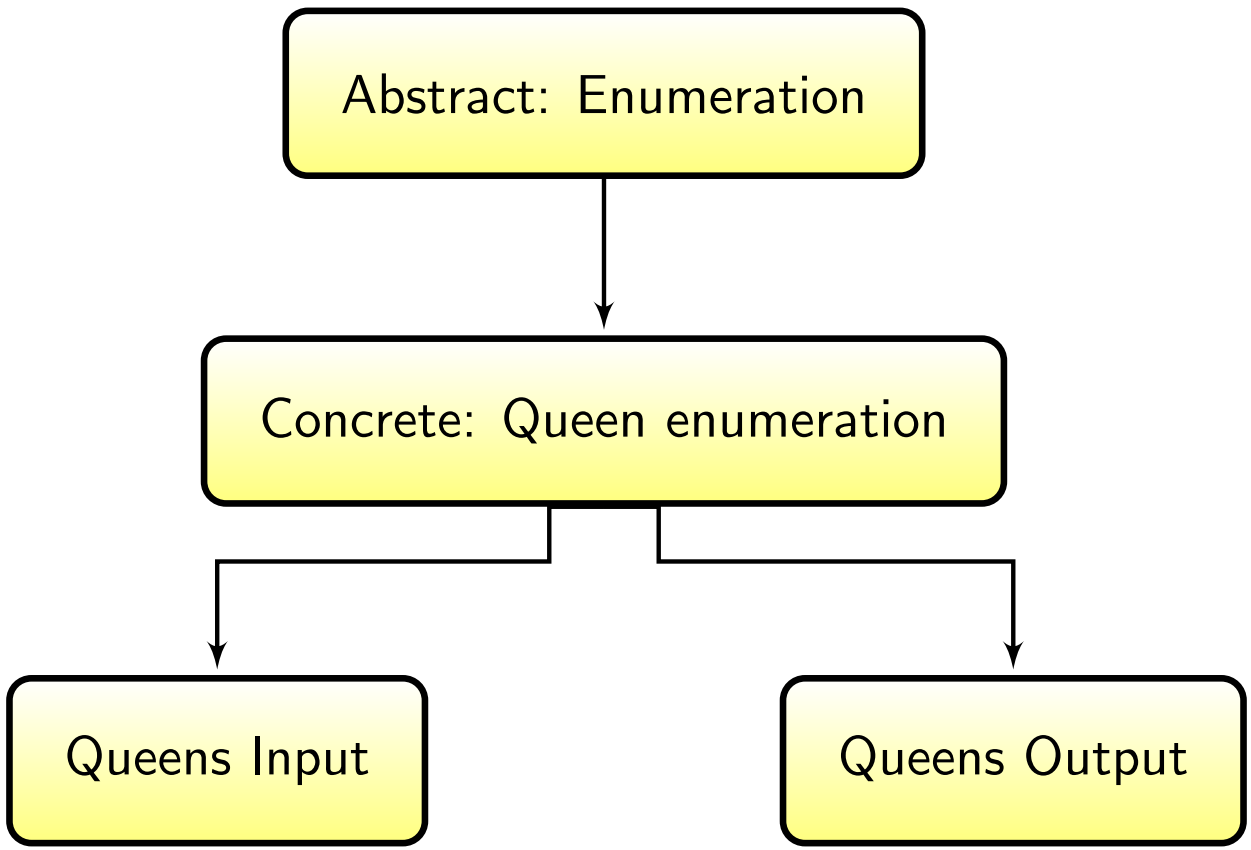
Note:

```
unsigned a,b;
unsigned x = abs(static_cast<int> a - static_cast<int>b);
```

`static_cast<int>` instead of `(int)`

Design

- Separation, modularization: Keep algorithm separated from data. Algorithm is the same, data are always different.



Do everything possible at higher level.

Templates
Example: Enumeration

With templates everything has to be in the header file.

```
template <typename Input, typename Output
        >
class Enumeration
{
public:
   bool Search(const Input& in, Output& out);
   unsigned NumSol() const { return count; }
protected:
   virtual void First(const Input& in, Output&
         out) = 0;
   virtual bool Next(const Input& in, Output&
         out) = 0;
   virtual bool Feasible(const Input& in, const
         Output& out) = 0;
   unsigned count;
};
```

```
template <typename Input, typename Output
        >
bool Enumeration<Input,Output>::Search(
        const Input& in, Output& out)
{
   First(in, out);
   count = 1;
   do
     {
       if (Feasible(in, out))
         return true;
       count++;
     }
   while (Next(in, out));
   return false;
}
```

- Headers
- Namespaces
- Copy constructors
- Destructors