

# CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 23  
November 28, 2018

PS6: Who prints the blockchain?

STL Iterators

STL Algorithms

Name Visibility

# PS6: Who prints the blockchain?

## OO-design problem

In PS6, we need a function `print` that prints a blockchain.

Which class does `print` belong in? Possibilities:

- ▶ Class `Blockchain`, because Blockchain is semantically meaningful.
- ▶ Class `Block`, because to print a blockchain requires knowledge of how the chain is represented and how to go from one block to the next. That knowledge is only available in `Block`.

## An analog of class demo 13-BarGraph

A `Row` is represented by a linked list of `Cells`.

This is analogous to a `Blockchain` being represented by a linked list of `Blocks`.

The `Row` print function reaches inside the `Cell` in order to iterate down the list of `Cells`.

This is possible because `Row` is a friend class of `Cell`.

Note: There is a comment in `row.cpp` that says,  
// Design decision: print `Cell` data directly; no delegation of print

## An analog to STL containers

Iterators (see next section) are like pointers and can be used by a client to iterate through a container such as a vector or list.

One could define a class `iterator` inside of `Blockchain` to allow one to iterate through a chain of blocks.

The `Blockchain::print()` function could then simply do

```
for(Block::iterator it=begin(); it!=end(); ++it) out<<*it;
```

Unfortunately, this would result in the blocks being printed in reverse order from what I specified in the assignment. You would need a backwards iterator, which doesn't work for singly linked lists.

In addition, iterators still do not overcome the problem of a `Blockchain` function needing knowledge of the structure of a `Block`.

## A compromise

The compromise I chose for my own solution is to give `Block` two print functions:

- ▶ `print()` prints a single block.
- ▶ `printChain()` prints the whole chain of blocks. An easy recursive solution prints the chain in the right order.
- ▶ `printChain()` delegates the printing of a single block to `Block::print()`.

`Blockchain::print()` delegates the printing of the whole blockchain to `Block::printChain()`.

# STL Iterators



# Containers

A container stores a collection of objects of arbitrary type `T`.

The basic containers in STL are:

- ▶ `vector` – a dynamic array
- ▶ `deque` – a double-ended queue
- ▶ `list` – a doubly linked list
- ▶ `map` – an associative array of key/value pairs with unique keys
- ▶ `set` – a sorted collection of unique values
- ▶ `multimap` – an associative array of key/value pairs with duplicate keys allowed
- ▶ `multiset` – a sorted collection of values with multiplicity

# Iterators

Iterators are like generalized pointers into containers.

Most pointer operations `*`, `->`, `++`, `==`, `!=`, etc. work with iterators.

- ▶ `begin()` returns an iterator pointing to the first element of the vector.
- ▶ `end()` returns an iterator pointing past the last element of the vector.

## Iterator example

Here's a program to store and print the first 10 perfect squares.

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

int main() {
    vector<int> tbl(10);
    for (unsigned k=0; k<10; k++) tbl[k] = k*k;
    vector<int>::iterator pos;
    for (pos = tbl.begin(); pos != tbl.end(); pos++)
        cout<< *pos<< endl;
}
```

## Using iterator inside a class

```
#include <iostream>
#include <vector>
using namespace std;
class Squares : vector<int> {
public:
    Squares(unsigned n) : vector<int>(n) {
        for (unsigned k=0; k<n; k++) (*this)[k] = k*k; }
    ostream& print(ostream& out) const {
        const_iterator pos;    // must be const_iterator
        for (pos=begin(); pos!=end(); pos++) out<< *pos<< endl;
        return out; }
};
int main() {
    Squares sq(10);
    sq.print(cout);
}
```

## Using subscripts and size()

```
#include <iostream>
#include <vector>
using namespace std;
class Squares : vector<int> {
public:
    Squares(unsigned n) {
        for (unsigned k=0; k<n; k++) push_back(k*k); }
    ostream& print(ostream& out) const {
        for (unsigned k=0; k<size(); k++) out<< (*this)[k]<< endl;
        return out; }
};
int main() {
    Squares sq(10);
    sq.print(cout);
}
```

# STL Algorithms

# Algorithms

STL has algorithms as well as data structures.

You must `#include <algorithm>`.

Commonly used: `copy`, `fill`, `swap`, `max`, `min`, `max_element`, `min_element`, but there are many many more.

We'll look at `sort` in greater detail.

## STL sort algorithm

`sort` works only on randomly-accessible containers such as `vector`. (`list` has its own sort method.)

`sort` takes two iterator arguments to designate the sort range.

It can also take an optional third “comparison” argument to define the sort order.



## Reverse sort example

```
class Squares : vector<int> {  
public:  
    Squares(unsigned n) {  
        for (unsigned k=0; k<n; k++) push_back(k*k);}   
  
    // decreasing order; *** must be static ***  
    static bool cmp( const int& x1, const int& x2 ) {  
        return x1 > x2; }  
  
    void rsort() { sort(begin(), end(), cmp); }  
  
    ostream& print(ostream& out) const {  
        for (unsigned k=0; k<size(); k++) out<< (*this)[k]<< endl;  
        return out; }  
};
```

## Reverse sort example (cont.)

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

class Squares : vector<int> {
    ...
};

int main() {
    Squares sq(10);
    sq.rsort();
    sq.print(cout);
}
```

## `pair<T1, T2>`

A `pair<T1, T2>` is an ordered pair of elements of type `T1` and `T2`, respectively.

Class `pair<T1, T2>` has public data members `first` and `second`.

Example:

```
pair<string, double> item("book", 49.95);  
                                // makes pair <"book", 49.95>  
cout<< item.first;             // prints "book"  
cout<< item.second;           // prints 49.95
```

## `map<Key, Val>`

`map<Key, Val>` associates a value with each key.

More precisely, it is an ordered collection of elements of type `pair<Key, Val>`.

You must `#include <map>`.

Can use standard subscript notation to access `map` contents, where subscript is the key.

Can also use a `map` iterator, which returns a pointer to a `pair`.

## Using a map<Key,Val>

Example:

```
typedef map<string,double> myMap; // alias for convenience
myMap::iterator pos;
myMap m;                          // a map from strings to doubles
m["dog"];                         // puts pair <"dog",0.0> into m
m["bird"]=5.2;                    // puts pair <"bird",5.2> into m
pos = m.find("cat");              // returns m.end() for not found
cout<< (pos==m.end())<< endl;    // prints 1 (true)
pos = m.find("bird");             // pos points to <"bird",5.2>
if (pos!=m.end()) {
    cout<< pos->first<< endl;    // prints "bird"
    cout<< pos->second<< endl;  // prints 5.2; }
}
```

## Copying from one container to another

Two ways to copy multiple elements in one statement.

Suppose `m` is a map and `v` a vector of pairs compatible with `m`.

1. `v.assign(m.begin(), m.end());`
2. Supply `m.begin()` and `m.end()` as arguments to the `v` constructor.

## Copying from `map` to vector of pairs

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
using namespace std;
int main() {
    map<string,double> m;
    m["dog"]=3; m["cat"]=2;
    // construct p from m
    vector<pair<string,double> > p(m.begin(),m.end());
    // declare iterator
    vector<pair<string,double> >::const_iterator pos;
    // print p
    for (pos=p.begin(); pos!=p.end(); ++pos)
        cout<< pos->first<< " "<< pos->second<< endl;
}
```

## string class

The standard `string` class tries to make strings behave like other built-in data types.

Like `vector<char>`, strings are growable, but they are not implemented using `vector`, and they support many special string operations.

They can be assigned (`=`, `assign()`), compared (`==`, `!=`, `<`, `<=`, `>`, `>=`, `compare()`), concatenated (`+`), read and written (`>>`, `<<`), searched (`find()`, ...), extracted (`[]`, `substr()`), modified (`+=`, `append()`, ...), and more.

Their length can be found (`size()`, `length()`).

`s.c_str()` or `s.data()` returns a copy of `s` as a C string.

You must `#include <string>`.



# Name Visibility

## Private derivation (default)

`class B : A { ... };` specifies **private** derivation of `B` from `A`.

A class member inherited from `A` become **private** in `B`.

Like other private members, it is inaccessible outside of `B`.

If **public** in `A`, it can be accessed from within `A` or `B` or via an instance of `A`, but not via an instance of `B`.

If **private** in `A`, it can only be accessed from within `A`.

It cannot even be accessed from within `B`.

## Private derivation example

Example:

```
class A {  
private:  int x;  
public:   int y;  
};  
class B : A {  
    ... f() {... x++; ...} // privacy violation  
};  
//----- outside of class definitions -----  
A a; B b;  
a.x    // privacy violation  
a.y    // ok  
b.x    // privacy violation  
b.y    // privacy violation
```

## Public derivation

`class B : public A { ... };` specifies **public** derivation of `B` from `A`.

A class member inherited from `A` retains its privacy status from `A`.

If **public** in `A`, it can be accessed from within `B` and also via instances of `A` or `B`.

If **private** in `A`, it can only be accessed from within `A`.  
It cannot even be accessed from within `B`.

## Public derivation example

Example:

```
class A {  
private:  int x;  
public:   int y;  
};  
class B : public A {  
    ... f() {... x++; ...} // privacy violation  
};  
//----- outside of class definitions -----  
A a; B b;  
a.x    // privacy violation  
a.y    // ok  
b.x    // privacy violation  
b.y    // ok
```

## The protected keyword

`protected` is a privacy status between `public` and `private`.

Protected class members are inaccessible from outside the class (like `private`) but accessible within a derived class (like `public`).

Example:

```
class A {  
protected: int z;  
};  
class B : A {  
    ... f() {... z++; ...} // ok  
};
```

## Protected derivation

`class B : protected A { ... };` specifies **protected** derivation of `B` from `A`.

A **public** or **protected** class member inherited from `A` becomes **protected** in `B`.

If **public** in `A`, it can be accessed from within `B` and also via instances of `A` but not via instances of `B`.

If **protected** in `A`, it can be accessed from within `A` or `B` but not from outside.

If **private** in `A`, it can only be accessed from within `A`.  
It cannot be accessed from within `B`.

## Surprising example 1

Link to [surprising-1.cpp](#).

```
1  class A {  
2  protected:  
3      int x;  
4  };  
5  class B : public A {  
6  public:  
7      int f() { return x; }           // ok  
8      int g(A* a) { return a->x; }   // privacy violation  
9  };
```

Result:

```
tryme1.cpp: In member function 'int B::g(A*)':  
tryme1.cpp:3: error: 'int A::x' is protected  
tryme1.cpp:9: error: within this context
```



## Surprising example 2: Contrast the following

Link to [surprising-2a.cpp](#).

```
1  class A { };
2  class B : public A {};    // <-- public derivation
3  int main() { A* ap; B* bp;
4      ap = bp; }
```

Result: OK.

---

Link to [surprising-2b.cpp](#).

```
1  class A { };
2  class B : private A {};   // <-- private derivation
3  int main() { A* ap; B* bp;
4      ap = bp; }
```

Result:

tryme2.cpp: In function 'int main()':

tryme2.cpp:4: error: 'A' is an inaccessible base of 'B'

## Surprising example 3

Link to [surprising-3.cpp](#).

```
1  class A { protected: int x; };  
2  class B : protected A {};  
3  int main() { A* ap; B* bp;  
4      ap = bp; }
```

Result:

```
tryme3.cpp: In function 'int main()':  
tryme3.cpp:4: error: 'A' is an inaccessible base of 'B'
```