Intermediate Programming

Day 36

Announcements

Please complete:

- Enrolled student survey
- Teacher course evaluations

Outline

- Exercise 13-1
- Iterators
- Review questions

Exercise 13-1 (part 3)

- Too many integers

exceptionExercise.cpp

```
...
int main( int argc , char ** argv )
{
        ...
        std::vector< int > numbers;
        numbers = readFile( argv[1] );
        ...
}
```

exceptionExercise.cpp

```
...
int main( int argc , char ** argv )
{
        ...
        vector< int > numbers;
        try{ numbers = readFile( argv[1] ); }
        catch( out_of_range &e )
        {
                cerr << "Too many numbers in file" << endl;
                return 1;
        }
        ...
}
```

Exercise 13-1 (part 4)

- File does not exist

exceptionExercise.cpp

```
...
vector< int > readFile( char* filename )
{
	ifstream fin( filename );
	vector< int > numbers(10);
	...
}

int main( int argc , char ** argv )
{
	...
	vector< int > numbers;
	try{ numbers = readFile( argv[1] ); }
	catch( out_of_range &e )
	{
		cerr << "Too many numbers in file" << endl;
		return 1;
	}
	...
}
```

exceptionExercise.cpp

```
...
vector< int > readFile( char* filename )
{
	ifstream fin( filename );
	vector< int > numbers(10);
	if( !fin.is_open() ) throw ios_base::failure( "Couldn't open file" );
	...
}
int main( int argc , char ** argv )
{
	...
	vector< int > numbers;
	try{ numbers = readFile( argv[1] ); }
	catch( out_of_range &e )
	{
		cerr << "Too many numbers in file" << endl;
		return 1;
	}
	catch( ios_base::failure &e )
	{
		cerr << e.what() << endl;
		return 1;
	}
	...
}
```

Exercise 13-1 (part 5)

- Not an `int`

exceptionExercise.cpp

```
...
int main( int argc , char ** argv )
{
        ...
        vector< int > numbers;
        try{ numbers = readFile( argv[1] ); }
        catch( out_of_range &e )
        {
                cerr << "Too many numbers in file" << endl;
                return 1;
        }
        catch( ios_base::failure &e )
        {
                cerr << e.what() << endl;
                return 1;
        }
        ...
}
```

exceptionExercise.cpp

```
...
int main( int argc , char ** argv )
{
        ...
        vector< int > numbers;
        try{ numbers = readFile( argv[1] ); }
        catch( out_of_range &e )
        {
                cerr << "Too many numbers in file" << endl;
                return 1;
        }
        catch( ios_base::failure &e )
        {
                cerr << e.what() << endl;
                return 1;
        }
        catch( invalid_argument &e )
        {
                cerr << e.what() << endl;
                return 1;
        }
        ...
}
```

Exercise 13-1 (part 6)

- Bad array index

exceptionExercise.cpp

```
...
vector< int > readFile( char* filename )
{
    ifstream fin( filename );
    vector< int > numbers(10);
    if( !fin.is_open() ) throw ios_base::failure( "Couldn
    ...
    numbers.at(index) = n;
    ...
}
```

exceptionExercise.cpp

```
...
vector< int > readFile( char* filename )
{
    ifstream fin( filename );
    vector< int > numbers(10);
    if( !fin.is_open() ) throw ios_base::failure( "Couldn't open file" );
    ...
    try{ numbers.at(index) = n; }
    catch( out_of_range &e )
    {
        numbers.resize( numbers.size()+1 );
        numbers.at(index) = n;
    }
    ...
}
```

Outline

- Exercise 13-1
- Iterators
- Review questions

Iterators

- In our code, we often work with containers of things:

myVec.h

```
template< typename T >
class MyVec
{
	size_t _size;
	T *_values;
public:
	MyVec( int size ) : _values( new T[size] ) , _size(size) {}
	~MyVec( void ) { delete[] _values; }
	size_t size( void ) const { return _size; }
	T& operator[] ( size_t i ){ return _values[i]; }
	const T& operator[] ( size_t i ) const { return _values[i]; }
};
```

```
#include <iostream>
#include "myVec.h"
using namespace std;

void Print( const MyVec< int > &v )
{
	for( size_t i=0 ; i<v.size() ; i++ ) cout << v[i] << endl;
}

int main( void )
{
	MyVec< int > v( 3 );
	v[0] = 0 , v[1] = 3 , v[2] = 5;
	Print( v );
	return 0;
}
```

```
>> ./a.out
0
3
5
>>
```

Iterators

- In our code, we often work with containers of things:

myNode.h

```
template< typename T >
class MyNode
{
public:
	T value;
	MyNode *next;
	MyNode( T v , MyNode *n=nullptr ) : next(n) , value(v) { }
};
```

m

```
#include <iostream>
#include "myNode.h"
using namespace std;

void Print( const MyNode< int > &l )
{
	for( const MyNode< int > *i=&l ; i!=NULL ; i=i->next )
		cout << i->value << endl;
}

int main( void )
{
	MyNode< int > n1( 0 ) , n2( 3 ) , n3( 5 );
	n1.next = &n2 , n2.next = &n3;
	Print( n1 );
	return 0;
}
```

```
>> ./a.out
0
3
5
>>
```

Iterators

- When working with containers of things, we don't want to special-case the type-specific ways for running through the elements of the container

main.cpp

```
#include <iostream>
#include “myVec.h"
using namespace std;

void Print( const MyVec< int > &v )
{
	for( size_t i=0 ; i<v.size() ; i++ )
		cout << v[i] << endl;
}
…
```

main.cpp

```
#include <iostream>
#include "myNode.h"
using namespace std;

void Print( const MyNode< int > &n )
{
	for( const MyNode< int > *i=&n ; n!=NULL ; i=i->next )
		cout << i->value << endl;
}
…
```

Iterators

- In our code, we often work with lists of values:
  - We unify the iteration by defining an auxiliary "pointer-like" object / *iterator* for walking through the list
  - We need to:
    - Get an iterator that "points" to the beginning of the list
    - Get an iterator that "points" just past the end of the list
    - Dereference the iterator
    - Advance the iterator
    - Check if two iterators are different

main.cpp

```
…
template< typename Container >
void Print( const Container &c )
{
	for( PointerLikeObject p=c.begin() ; p!=c.end() ; ++p )
		cout << *p << endl;
}
…
```

Iterators

- In C++, when we have a container class, we define the iterator as a public <u>nested</u> class called:
  - iterator if we want to be able to modify the values of the reference
  - const_iterator if we do not

container.h

```
template< typename T >
class Container
{
public:
	…
	class iterator
	{
		…
	};
	class const_iterator
	{
		…
	};
};
…
```

Iterators

- In C++, when we have a container class, we define the iterator as a **public** nested class called:
- The iterator must overload:
  - The dereference operator
  - The (pre-)increment operator
  - The inequality operator

container.h

```
template< typename T >
class Container
{
public:
	…
	class iterator
	{
		public:
			T &operator * ( );
			iterator &operator ++ ();
			bool operator != ( const iterator &i ) const;
		…
	};
	class const_iterator{ … };
};
…
```

Iterators

- In C++, when we have a container class, we define the iterator as a public nested class called:
- The iterator must overload:
  - The dereference operator
  - The (pre-)increment operator
  - The inequality operator
- The container must define:
  - A begin/cbegin method
  - An end/cend method

container.h

```
template< typename T >
class Container
{
public:
	…
	class iterator{ … };
	class const_iterator{ … };
	…
	iterator begin( void );
	iterator end( void );
	const_iterator cbegin( void ) const;
	const_iterator cend( void ) const;
};
…
```

Iterators

- Putting these together, we can define generic code:

container.h

```
template< typename T >
  Container


  c:
  …
  class iterator{ … };
  class const_iterator{ … };
  …
  iterator begin( void );
  iterator end( void );
  const_iterator cbegin( void ) const;
  const_iterator cend( void ) const;
};
…
```

main.cpp

```
…
template< typename C >
void Print( const C &c )
{
	for( typename C::const_iterator i=c.cbegin() ; i!=c.cend() ; ++i )
		cout << *i << endl;
}
…
```

Iterators

- Putting these together, we can define generic code:

main.cpp

```
…
template< typename C >
void Print( const C &c )
{
	for( typename C::const_iterator i=c.cbegin() ; i!=c.cend() ; ++i )
		cout << *i << endl;
}
…
```

container.h

```
template< typename T >
 Container

c:
	…
	class iterator{ … };
	class const_iterator{ … };
	…
	iterator begin( void );
	iterator end( void );
	const_iterator cbegin( void ) const;
	const_iterator cend( void ) const;
};
…
```

<u>Note</u>:
The keyword **typename** is needed to let the compiler know that **const_iterator** is a class / type, not a static member.

Iterators: **MyVec**

- constructor

myVec.h

```
template< typename T >
class MyVec
{
	T *_values;
	size_t _size;
public:
	MyVec( int size );
	~MyVec( void );
	size_t size( void ) const;
	T &operator[] ( size_t i );
	const T &operator[] ( size_t i ) const;
	…
```

```
	…
	class const_iterator
	{
		const T *_ptr;
	public:
		const_iterator( const T *ptr ) : _ptr( ptr ){ }
	};
};
```

Iterators: **MyVec**

- dereference

myVec.h

```
template< typename T >
class MyVec
{
        T *_values;
        size_t _size;
public:
        MyVec( int size );
        ~MyVec( void );
        size_t size( void ) const;
        T &operator[] ( size_t i );
        const T &operator[] ( size_t i ) const;
        …
```

```
        …
        class const_iterator
        {
                const T *_ptr;
        public:
                const_iterator( const T *ptr ) : _ptr( ptr ){ }
                const T &operator * ( ) const { return *_ptr; }
        };
};
```

Iterators: **MyVec**

- pre-increment

myVec.h

```
template< typename T >
class MyVec
{
	T *_values;
	size_t _size;
public:
	MyVec( int size );
	~MyVec( void );
	size_t size( void ) const;
	T &operator[] ( size_t i );
	const T &operator[] ( size_t i ) const;
	…
```

```
	…
	class const_iterator
	{
		const T *_ptr;
	public:
		const_iterator( const T *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return *_ptr; }
		const_iterator &operator ++ () { _ptr++ ; return *this; }
	};
};
```

Iterators: **MyVec**

- inequality

myVec.h

```
template< typename T >
class MyVec
{
	T *_values;
	size_t _size;
public:
	MyVec( int size );
	~MyVec( void );
	size_t size( void ) const;
	T &operator[] ( size_t i );
	const T &operator[] ( size_t i ) const;
	…
```

```
	…
	class const_iterator
	{
		const T *_ptr;
	public:
		const_iterator( const T *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return *_ptr; }
		const_iterator &operator ++ () { _ptr++ ; return *this; }
		bool operator != ( const const_iterator& i ) const
		{
			return _ptr!=i._ptr;
		}
	};


};
```

Iterators: **MyVec**

- beginning / ending iterators

myVec.h

```
template< typename T >
class MyVec
{
	T *_values;
	size_t _size;
public:
	MyVec( int size );
	~MyVec( void );
	size_t size( void ) const;
	T &operator[] ( size_t i );
	const T &operator[] ( size_t i ) const;
	…
```

```
	…
	class const_iterator
	{
		const T *_ptr;
	public:
		const_iterator( const T *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return *_ptr; }
		const_iterator &operator ++ () { _ptr++ ; return *this; }
		bool operator != ( const const_iterator& i ) const
		{
			return _ptr!=i._ptr;
		}
	};
	const_iterator cbegin( void ) const { return const_iterator( _values ); }
	const_iterator cend( void ) const { return const_iterator( _values+_size ); }
};
```

Iterators: MyNode

- constructor

```
myNode.h
template< typename T >
class MyNode
{
public:
	MyNode< T > *next;
	T value;
	MyNode( T v , MyNode< T > *n=nullptr );
	…
```

```
	…
	class const_iterator
	{
		const MyNode< T > *_ptr;
	public:
		const_iterator( const MyNode< T > *ptr ) : _ptr( ptr ){ }
	};
};
```

Iterators: **MyNode**

- dereference

myNode.h

```
template< typename T >
class MyNode
{
public:
	MyNode< T > *next;
	T value;
	MyNode( T v , MyNode< T > *n=nullptr );
	…
```

```
	…
	class const_iterator
	{
		const MyNode< T > *_ptr;
	public:
		const_iterator( const MyNode< T > *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return _ptr->value; }
	};
};
```

Iterators: **MyNode**

- pre-increment

myNode.h

```
template< typename T >
class MyNode
{
public:
	MyNode< T > *next;
	T value;
	MyNode( T v , MyNode< T > *n=nullptr );
	…
```

```
	…
	class const_iterator
	{
		const MyNode< T > *_ptr;
	public:
		const_iterator( const MyNode< T > *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return _ptr->value; }
		const_iterator &operator ++ () { _ptr=_ptr->next ; return *this; }
	};
};
```

Iterators: **MyNode**

- inequality

myNode.h

```
template< typename T >
class MyNode
{
public:
	MyNode< T > *next;
	T value;
	MyNode( T v , MyNode< T > *n=nullptr );
	…
```

```
	…
	class const_iterator
	{
		const MyNode< T > *_ptr;
	public:
		const_iterator( const MyNode< T > *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return _ptr->value; }
		const_iterator &operator ++ () { _ptr=_ptr->next ; return *this; }
		bool operator != ( const const_iterator& i ) const
		{
			return _ptr!=i._ptr;
		}
	};


};
```

Iterators: **MyNode**

- beginning / ending iterators

myNode.h

```
template< typename T >
class MyNode
{
public:
	MyNode< T > *next;
	T value;
	MyNode( T v , MyNode< T > *n=nullptr );
	…
```

```
	…
	class const_iterator
	{
		const MyNode< T > *_ptr;
	public:
		const_iterator( const MyNode< T > *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return _ptr->value; }
		const_iterator &operator ++ () { _ptr=_ptr->next ; return *this; }
		bool operator != ( const const_iterator& i ) const
		{
			return _ptr!=i._ptr;
		}
	};
	const_iterator cbegin( void ) const { return const_iterator( this ); }
	const_iterator cend( void ) const { return const_iterator( nullptr ); }
};
```

Iterators: **MyVec**

- When the iterator is a pointer, things can be made simpler

myVec.h

```
template< typename T >
class MyVec
{
	T *_values;
	size_t _size;
public:
	MyVec( int size );
	~MyVec( void );
	size_t size( void ) const;
	T& operator[] ( size_t i );
	const T& operator[] ( size_t i ) const;
	…
```

```
	…
	class const_iterator
	{
		const T *_ptr;
	public:
		const_iterator( const T *ptr ) : _ptr( ptr ){ }
		const T &operator * ( ) const { return *_ptr; }
		const_iterator &operator ++ () { _ptr++ ; return *this; }
		bool operator != ( const const_iterator& i ) const
		{
			return _ptr!=i._ptr;
		}
	};
	const_iterator cbegin( void ) const { return const_iterator( _values ); }
	const_iterator cend( void ) const { return const_iterator( _values+_size ); }
};
```

Iterators: **MyVec**

- When the iterator is a pointer, things can be made simpler

myVec.h

```
template< typename T >
class MyVec
{
	T *_values;
	size_t _size;
public:
	MyVec( int size );
	~MyVec( void );
	size_t size( void ) const;
	T& operator[] ( size_t i );
	const T& operator[] ( size_t i ) const;
```

```
…
class const_iterator
{
	const T *_ptr;
public:
	const_iterator( const T *ptr ) : _ptr( ptr ){ }
	const T &operator * ( ) const { return *_ptr; }
	const_iterator &operator ++ () { _ptr++ ; return *this; }
	bool operator != ( const const_iterator &i ) const
	{
		return _ptr!=i._ptr;
	}
};
…st { return const_iterator( _values ); }
… { return const_iterator( _values+_size ); }
```

```
	…
	typedef const T *const_iterator;
	const_iterator cbegin( void ) const { return _values; }
	const_iterator cend( void ) const { return _values+_size; }
};
```

Iterators

- We can define a single (templated) function for processing contents of different types of containers.

main.cpp

```cpp
#include <iostream>
#include "myVec.h"
#include "myNode.h"

template< typename Container >
void Print( const Container &c )
{
	for( typename Container::const_iterator it=c.cbegin() ; it!=c.cend() ; ++it )
		std::cout << *it << std::endl;
}

int main( void )
{
	MyVec< int > v( 3 );
	v[0] = 0 , v[1] = 3 , v[2] = 5;
	std::cout << "Printing MyVec" << std::endl;
	Print( v );

	MyNode< int > n1( 0 ) , n2( 3 ) , n3( 5 );
	n1.next = &n2 , n2.next = &n3;
	std::cout << "Printing MyNode" << std::endl;
	Print( n1 );

	return 0;
}
```

```
>> ./main
Printing MyVec
0
3
5
Printing MyNode
0
3
5
>>
```

Outline

- Exercise 13-1
- Iterators
- Review questions

Review questions

1. Why use iterators?

Iterators unify the manner in which we step through the elements in a container

Review questions

2. When won't a pointer work for representing an iterator?

When data is not stored sequentially in memory

Review questions

3. What are the bare minimum operators that need to be overloaded by an iterator?

Inequality, dereference, and (pre-)increment

Review questions

4. Given a container how/where should the `iterator` and `const_iterator` classes be specified?

As a `public` nested subclasses of the container

Review questions

5. In addition to defining the `iterator` and `const_iterator` classes, what else should the container do to support iteration?

Define `begin`/`cbegin` and `end`/`cend` member functions

Review questions

6. What might go wrong if we don't also define a `const_iterator` for a container?

We won't be able to iterate over the contents of a `const` object of that container class

Exercise 13-2

- Website -> Course Materials -> ex13-2