More on Templates
Standard Template Library
`exception` in C++

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

http://www.roma1.infn.it/people/rahatlou/programmazione++/

Corso di Programmazione++

Roma, 15 June 2008

Today's Lecture

- More on Template
  - Inheritance
  - static data members
  - friend and Template
  - example: auto_ptr<T>
  - Standard template library

- Error handling in applications
  - Typical solutions
    - advantages and disadvantages
  - C++ exception
    - What is it?
    - How to use it

Template and Runtime Decision

- Fundamental difference between Template and Inheritance

- All derived classes share common functionalities
  - Can point to any derived class object via base-class pointer

- No equivalent of base-class pointer for class-template specializations
  - Dummy<string> and Dummy<double> are different classes
  - No polymorphism at run time!

Template and Inheritance

- Inheritance provides run-time polymorphism
- Templates provide compile-time polymorphism
  - Code generated by compiler at compilation time using the Template class or function and the specified parameter
  - All specialized templates are identical except for the data type
  - Template-class specialization is equivalent to any regular non-template class
- But remember…
  - Class template NOT EQUIVALENT to base class
  - No base-class pointer mechanism for different specializations
  - No runtime polymorphism
  - Different specializations are different classes with no inheritance relation

Difference between Template and Inheritance

```
int main() {
  Person* p = 0;
  int value = 0;
  while(value<1 || value>10) {
    cout << "Give me a number [1,10]: ";
    cin >> value;
  }
  cout << flush; // write buffer to output
  cout << "make a new derived object..." << endl;
  if(value>5) p = new Student("Susan", 123456);
  else        p = new GraduateStudent("Paolo", 9856, "Physics");
  cout << "call print() method ..." << endl;
  p->print();
  delete p;
  return 0;
}
```

Same base-class pointer used to initialize data based on user input

one call to ::print()

no if statement

no checking for null pointer

```
int main() {
  Dummy<std::string>* d1 = 0;
  Dummy<double>* d2 = 0;

  int value = 0;
  while(value<1 || value>10) {
    cout << "Give me a number [1,10]: ";
    cin >> value;
  }
  cout << flush;

  if(value>5) d1 = new Dummy<std::string>( "string" );
  else        d2 = new Dummy<double>( 1.1 );

  if( d1 != 0 ) d1->print();
  if( d2 != 0 ) d2->print();

  return 0;
}
```

Need as many pointers as possible outcomes of input by user

No base-class pointer → No polymorphism

Check specific pointers to be non-null before calling DIFFERENT ::print() methods

```
$ ./example0
Give me a number [1,10]: 3
Dummy<T>::print() with type T = d, *data_: 1.1
$ ./example0
Give me a number [1,10]: 7
Dummy<T>::print() with type T = Ss, *data_: string
```

Template and Inheritance

- Can use specializations as any other class
  - But can't inherit from a class template
- A class template can be derived from a non-template class
  - `template<class T> class GenericPerson : public Person { };`
- A class template can be derived from a class-template specialization
  - `template<class T> class MyString : public Dummy<std::string> {};`
- A class-template specialization can be derived from a class-template specialization
  - `class Dummy<Car> : public Vector<Object> { };`
- A non-template class can be derived from a class-template specialization
  - `class Student : public Dummy<std::string> { };`

Template and static

- All specializations of a class template have their copy of own static data
  - Treat class-template specialization as equivalent to normal non-template class

```
// example1.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

#include "Dummy.h"

int main() {
  Dummy<std::string> d1( "d1" );
  Dummy<std::string> d2( "d2" );
  Dummy<std::string> d3( "d3" );

  Dummy<double> f1( 0.1 );
  Dummy<double> f2( -56.45 );

  cout << "Dummy<std::string>::total(): " << Dummy<std::string>::total() << endl;
  cout << "Dummy<double>::total(): " << Dummy<double>::total() << endl;
  cout << "Dummy<int>::total(): " << Dummy<int>::total() << endl;

  return 0;
}
```

```
$ g++ -Wall -o example1
example1.cpp
$ ./example1
Dummy<std::string>::total(): 3
Dummy<double>::total(): 2
Dummy<int>::total(): 0
```

Static data with Dummy<T>

```
template< typename  T >
class Dummy {
  public:
    Dummy(const T& data);
    ~Dummy();
    void print() const;
    static total() { return total_; }

  private:
    T* data_;
    static int total_;
};
```

All code in **Dummy.h**

Remember no source file!

```
template<class T>
int Dummy<T>::total_ = 0;


template<class T>
Dummy<T>::Dummy(const T& data) {
  data_ = new T(data);
  total_++;
}


template<class T>
Dummy<T>::~Dummy() {
  total_--;
  delete data_;
}

template<class T>
void
Dummy<T>::print() const {
  std::cout << "Dummy<T>::print() with type T =
"
       << typeid(T).name()
       << ", *data_: " << *data_
       << std::endl;
}
```

Template and friend Functions

- All usual rules for friend methods and classes are still valid
- You can declare functions to be friends of
  - all specializations of a template-class or specific specializations
  - Your Favorite combination of template classes and functions

```
template< typename  T >
class Foo {
  public:
    Foo(const T& data);
    ~Foo();
    void print() const;
    // friend of all specializations
    friend void nicePrint();

    // friend of specialization with same type
    friend void specialPrint( const Foo<T>& obj);

    // member function of Bar friend of all specializations
    friend void Bar::printFoo();

    // member function of Dummy with same type
    friend void Dummy<T>::print(const Foo<T> & f )

  private:
    T* data_;
};
```

`nicePrint()` friend of `Foo<int>` and `Foo<string>`

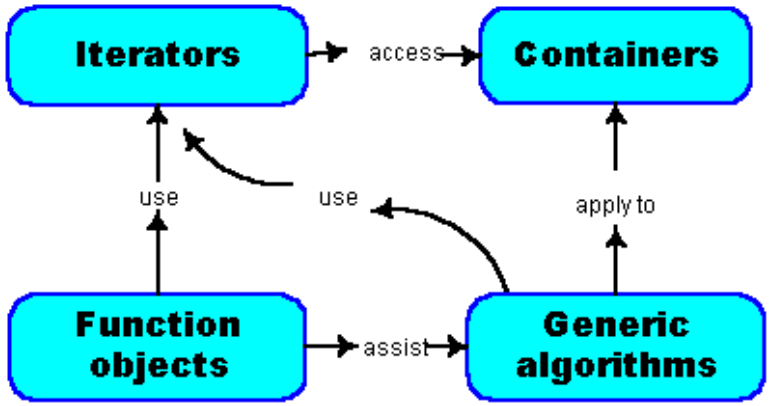
`specialPrint(string)` friend of `Foo<string>` but NOT friend of `Foo<int>`

`Bar::printFoo()` friend of `Foo<int>` and `Foo<string>`

`Dummy<int>::print(int)` friend of `Foo<int>` but NOT friend of `Foo<string>`

Standard Template Library

- Library of container classes, algorithms, and iterators
  - Covers many of basic algorithms and data structures of common use
  - Very efficient through compile-time polymorphism achieved by using Template
- Containers: classes whose purpose is to contain any type of objects
  - Sequence containers: vector, list, seq, deque
  - Associative containers: set, multiset, map, multimap

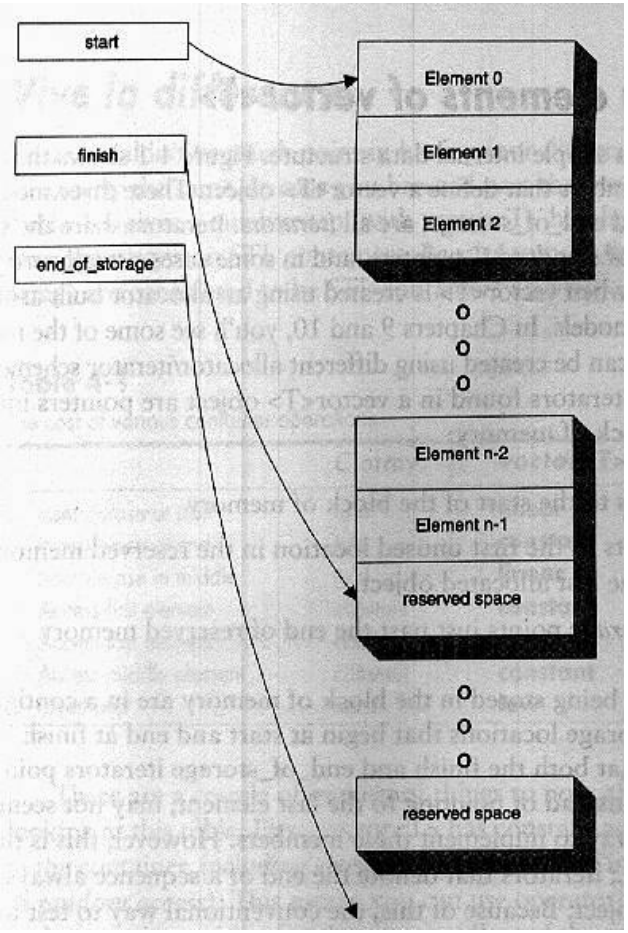


- Algorithms: methods used to manipulate container items
  - Finding, sorting, reverting items
- Iterators: generalization of pointer to provide access to items in a container

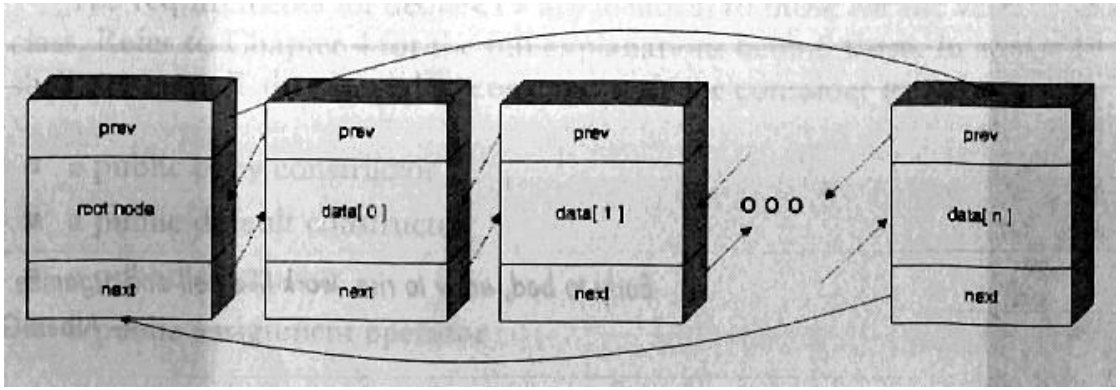
containers

- Address different needs with different perfmance
- Vector: fast random access. Rapid insertion and deletion at the end of vector
- List: rapid insertion and deletion anywehere
  - No sequential storage of data

vector



list

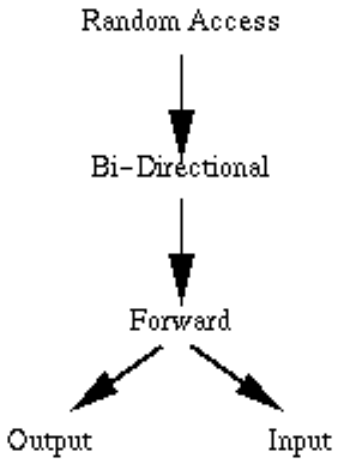


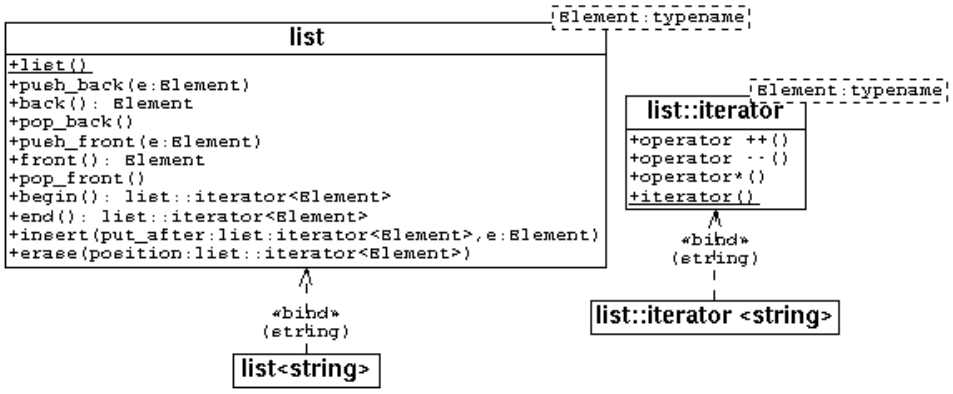
Requirements for type `T` objects in containers

- Any C++ type and class can be used but a minimum set of functionality required
- Inserting an object of type `T` corresponds to copying object into the container
- Sequential containers require a proper copy constructor and assignment operator (=) for class `T`
  - Default implementations is fine as long as non-trivial data members are used
- Associative containers often perform comparison between elements
  - Class `T` should provide equality (==) and less-than (<) operators

iterators

- Allows user to traverse through all elements of a container regardless of its specific implementation
  - Allow pointing to elements of containers
- Hold information sensitive to particular containers
  - Implemented properly for each type of container
  - Five categories of iterators





Iterator Type	Behavioral Description	Operations Supported
random access (most powerful)	Store and retrieve values Move forward and backward Access values randomly	`* = ++ -> == != --` `+ - [] < > <= >= += -=`
bidirectional	Store and retrieve values Move forward and backward	`* = ++ -> == != --`
forward	Store and retrieve values Move forward only	`* = ++ -> == !=`
input	Retrieve but not store values Move forward only	`* = ++ -> == !=`
output (least powerful)	Store but not retrieve values Move forward only	`* = ++`

`iterator` Operations

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

Fig. 23.10 | Iterator operations for each type of iterator.

23.1.3 Introduction to Algorithms

The STL provides algorithms that can be used generically across a variety of containers. STL provides many algorithms you will use frequently to manipulate containers. Inserting

- Predefined iterator typedef's found in class definitions
- **`iterator`**
  - Forward read-write
- **`const_iterator`**
  - Forward read-only
- **`reverse_iterator`**
  - Bacward read-write
- **`const_reverse_iterator`**
  - backward read-only

Using iterators

```
vector<Student> v1; // declare vector


// create iterator from container
vector<Student>::const_iterator iter;

// use of iterator on elements of vector
for(  iter = v1.begin();
      iter != v1.end();
      ++iter) {
       cout << iter->name() << endl;
       (*iter).print();
}
```

- Two member functions **`begin()`** and **`end()`** returning iterators to beginning and end of container
  - **`begin()`** points to first object
  - **`end()`** is slightly different. Points to NON-EXISTING object past last item

Algorithms

- Almost 70 different algorithms provided by STL to be usedu generically with variety of containers
- Algorithms use iterators to interact with containers
  - This feature allows decoupling algorithms from containers!
  - Implement methods outside specific containers
  - Use generic iterator to have same functionality of many containers
- Many algorithms act on range of elements in a container identified by pair of iterators for first and last element to be used
- Iterators used to return result of an algorithm
  - Points to element in the container satisfying the algorithm

Non-modifying Algorithms

Non-modifying sequence operations:

for_each	Apply function to range (template function)
find	Find value in range (function template)
find_if	Find element in range (function template)
find_end	Find last subsequence in range (function template)
find_first_of	Find element from set in range (function template)
adjacent_find	Find equal adjacent elements in range (function template)
count	Count appearances of value in range (function template)
count_if	Return number of elements in range satisfying condition (function template)
mismatch	Return first position where two ranges differ (function template)
equal	Test whether the elements in two ranges are equal (function template)
search	Find subsequence in range (function template)
search_n	Find succession of equal values in range (function template)

Sorting:

sort	Sort elements in range (function template)
stable_sort	Sort elements preserving order of equivalents (function template)
partial_sort	Partially Sort elements in range (function template)
partial_sort_copy	Copy and partially sort range (function template)
nth_element	Sort element in range (function template)

Binary search (operating on sorted ranges):

lower_bound	Return iterator to lower bound (function template)
upper_bound	Return iterator to upper bound (function template)
equal_range	Get subrange of equal elements (function template)
binary_search	Test if value exists in sorted array (function template)

Min/max:

min	Return the lesser of two arguments (function template)
max	Return the greater of two arguments (function template)
min_element	Return smallest element in range (function template)
max_element	Return largest element in range (function template)

Merge (operating on sorted ranges):

merge	Merge sorted ranges (function template)
inplace_merge	Merge consecutive sorted ranges (function template)
includes	Test whether sorted range includes another sorted range (function template)
set_union	Union of two sorted ranges (function template)
set_intersection	Intersection of two sorted ranges (function template)
set_difference	Difference of two sorted ranges (function template)
set_symmetric_difference	Symmetric difference of two sorted ranges (function template)

Modifying algorithms

`swap()` allows fast and non-expensive copy of elements between containers

Commonly used to optimize performance and minimize unnecessary copy operations

Modifying sequence operations:

copy	Copy range of elements (function template)
copy_backward	Copy range of elements backwards (function template)
swap	Exchange values of two objects (function template)
swap_ranges	Exchange values of two ranges (function template)
iter_swap	Exchange values of objects pointed by two iterators (function template)
transform	Apply function to range (function template)
replace	Replace value in range (function template)
replace_if	Replace values in range (function template)
replace_copy	Copy range replacing value (function template)
replace_copy_if	Copy range replacing value (function template)
fill	Fill range with value (function template)
fill_n	Fill sequence with value (function template)
generate	Generate values for range with function (function template)
generate_n	Generate values for sequence with function (function template)
remove	Remove value from range (function template)
remove_if	Remove elements from range (function template)
remove_copy	Copy range removing value (function template)
remove_copy_if	Copy range removing values (function template)
unique	Remove consecutive duplicates in range (function template)
unique_copy	Copy range removing duplicates (function template)
reverse	Reverse range (function template)
reverse_copy	Copy range reversed (function template)
rotate	Rotate elements in range (function template)
rotate_copy	Copy rotated range (function template)
random_shuffle	Rearrangle elements in range randomly (function template)
partition	Partition range in two (function template)
stable_partition	Parition range in two - stable ordering (function template)

Comments and Criticism to STL

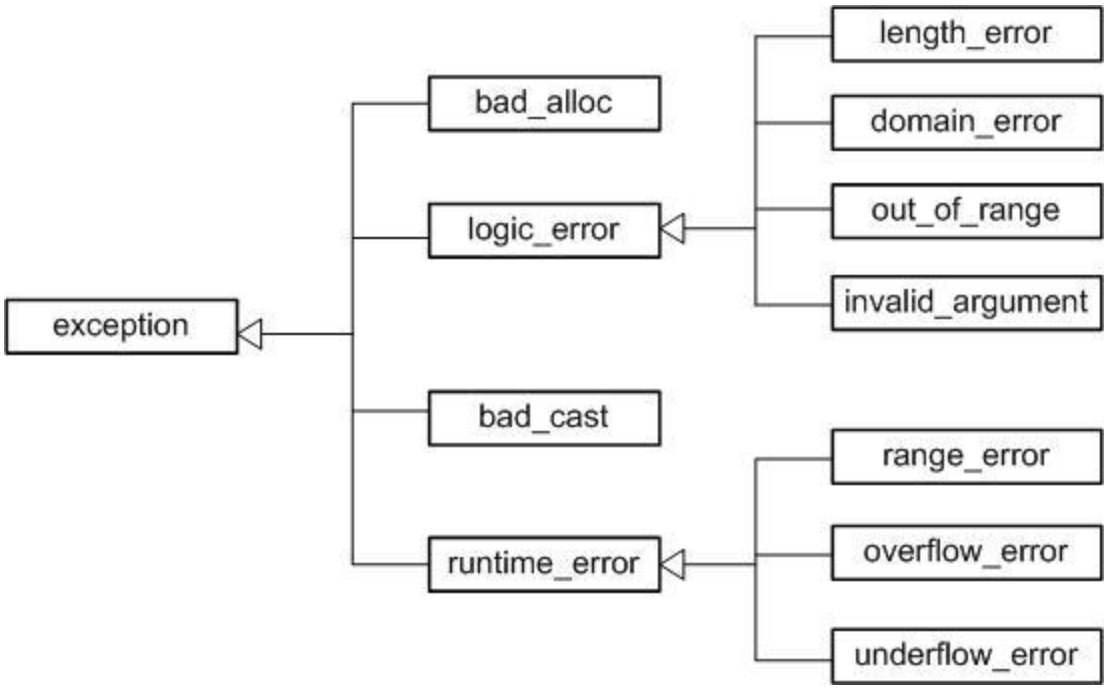
- Heavy use of template make STL very sensitive to changes or capabilities of different compilers
- Compilation error messages can be hard to decipher by developer
  - Tools being developed to provide indention and better formatting of improved error messages
- Generated code can be very large hence leading to significant increase in compilation time and memory usage
  - Careful coding necessary to prevent such problems
- Common problem with invalid pointers when element deleted from a container
  - Iterator not update hence pointing to non-existing element

Error Handling in C++

Exception Handling: What does it mean?

- Under normal circumstances applications should run successfully to completion
- Exceptions: special cases when errors occur
  - 'exception' is meant to imply that such errors occur rarely and are an exception to the rule (successful running)
  - Warning: exceptions **SHOULD NEVER** be used as replacement for conditionals!
- C++ Exceptions provide mechanism for error handling and writing fault-tolerant applications
  - errors can occur deep into the program or in third party software not under our control
- Applications use exceptions to decide if terminate or continue execution

Hierarchy of C++ STL Exceptions



C++ Exceptions

```
#include <iostream>
#include <stdexcept>
using std::cin;
using std::cout;
using std::endl;
using std::runtime_error;

double ratio(int i1, int i2) {
  if(i2 == 0) throw std::runtime_error("error in ratio");
  return i1/i2;
}

int main() {
  int i1 = 0;
  int i2 = 0;

  cout << "enter two numbers (ctrl-D to end): ";
  while( cin >> i1 >> i2 ) {

    try {
      cout << "ratio: " << ratio(i1,i2) << endl;

    } catch(std::runtime_error& ex) {
       cout << "error occured..." << ex.what() << endl;
    }

    cout << "enter two numbers (ctrl-Z to end): ";
  }
  return 0;
}
```

include code that can throw exception in a try{} block

throw an exception when error condition occurs

exception is a C++ object!

```
$ g++ -Wall -o example3 example3.cpp
$ ./example3
enter two numbers (ctrl-D to end): 7876 121
ratio: 65
enter two numbers (ctrl-D to end): 34 14
ratio: 2
enter two numbers (ctrl-D to end): 56 0
error occured...error in ratio
enter two numbers (ctrl-D to end):
```

use catch() {} to catch possible exceptions thrown within the try{} block

Exceptions Defined by Users

```
// example4.cpp
#include <iostream>
#include <stdexcept>
using std::cin;
using std::cout;
using std::endl;
using std::runtime_error;

class MyError : public std::runtime_error {
  public:
    MyError() : std::runtime_error("dividing by zero") {}
};

double ratio(int i1, int i2) {
  if(i2 == 0) throw MyError();
  return i1/i2;
}

int main() {
  int i1 = 0;
  int i2 = 0;

  cout << "enter two numbers (ctrl-Z to end): ";
  while( cin >> i1 >> i2 ) {

    try {
      cout << "ratio: " << ratio(i1,i2) << endl;

    } catch(MyError& ex) {
       cout << "error occured..." << ex.what() << endl;
    }

    cout << "enter two numbers (ctrl-Z to end): ";
  }
  return 0;
}
```

New exceptions can be implemented by users

Inherit from existing exceptions and specialize for use case relevant for your application

```
$ g++ -Wall -o example4 example4.cpp
$ ./example4
enter two numbers (ctrl-Z to end): 6 5
ratio: 1
enter two numbers (ctrl-Z to end): 5 0
error occured...dividing by zero
enter two numbers (ctrl-Z to end):
```