Templates and Generic Programming

Shahram Rahatlou

Computing Methods in Physics
http://www.roma1.infn.it/people/rahatlou/cmp/

Anno Accademico 2018/19

SAPIENZA
UNIVERSITÀ DI ROMA

Today's Lecture

- Templates in C++ and generic programming
  - What is Template?
  - What is a Template useful for?
  - Examples
  - Standard Template Library

- Error handling in C++ with Exceptions

Generic Programming

- Programming style emphasising use of *generics* technique

- Generics technique in computer science:
  - allow one value to take different data types as long as certain contracts are kept
    - For example types having same signature
    - Remember polymorphism

- Simple idea to define a code prototype or "template" that can be applied to different kinds (types) of data

- Template can be *specialised* for different data types

- A range of related functions or types related through templates

C++ Template

- Powerful feature that allows generic programming (but not only) in C++

- Two kinds of template in C++
  - Function template: a function prototype to act in identical manner on all types of input arguments
  - Class template: a class with same behavior for different types of data

- How does template work
  - One prototype written by user
  - Code generated by compiler for different template types and compiled
    - polymorphic code at compile time with no run-time overhead

Function Template

- Functions that perform "identical" operation regardless of type of argument
  - Error at COMPILATION TIME if requested operation not implemented for particular data type

- Template syntax
  - Two keywords used to provide parameters: **`typename`** and **`class`**
    - No difference between the two
    - **`class`** is a generic name here and can refer to a built in type as well

```
template< typename T >

template< typename InputType >

template< class InputType >

template< class InputType, typename OutputType>
```

Example of Function Template

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

#include "Vector3D.h"

template< typename  T >
void printObject(const T& input) {
  cout << "printObject(const T& input): with T = " << typeid( T ).name() << endl;
  cout << input << endl;
}

int main() {

  int i = 456;
  double x = 1.234;
  float  y = -0.23;
  string name("jane");
  Vector3D v(1.2, -0.3, 4.5);

  printObject( i );
  printObject( x );
  printObject( y );
  printObject( name );
  printObject( v );

  return 0;
}
```

typeinfo header needed to use typeid() function

Format of name() depends on each compiler

```
$ g++ -o /tmp/app example1.cpp Vector3D.cc
$ /tmp/app
printObject(const T& input): with T = i
456
printObject(const T& input): with T = d
1.234
printObject(const T& input): with T = f
-0.23
printObject(const T& input): with T =
NSt3__112basic_stringIcNS_11char_traitsIcEENS_9
allocatorIcEEEE
jane
printObject(const T& input): with T = 8Vector3D
(1.2 , -0.3 , 4.5)
```

Understanding Templates

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

#include "Vector3D.h"

template< typename  T >
void printObject(const T& input) {
  cout << "printObject(const T& input): with T = " << typeid( T ).name() << endl;
  cout << input << endl;
}

int main() {

  int i = 456;
  double x = 1.234;
  float  y = -0.23;
  string name("jane");
  Vector3D v(1.2, -0.3, 4.5);

  printObject( i );
  printObject( x );
  printObject( y );
  printObject( name );
  printObject( v );

  return 0;
}
```

Compiler generates actual code for

```
printObject( const int& input )
printObject( const double& input )
printObject( const float& input )
printObject( const string& input )
printObject( const Vector3D& input )
```

Another Template Function

```
// example2.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< class  DataType >
void printArray(const DataType* data, int nMax) {
  cout << "printObject(const T& input): with DataType = "
       << typeid( DataType ).name() << endl;
  for(int i=0; i<nMax; ++i) {
    cout << data[i] << "\t";
  }
  cout << endl;
}

int main() {

  int i[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  const int n1 = 3;
  double x[n1] = { -0.1, 2.2, 12.21};
  string days[] = { "Mon", "Tue", "Wed", "Thur", "Fri", "Sat", "Sun"};

  printArray( i, 10 );
  printArray( x, n1 );
  printArray( days, 7 );

  return 0;
}
```

```
$ g++ -o /tmp/example2 example2.cpp
$ /tmp/example2
printObject(const T& input): with DataType = i
0           1           2           3           4           5           6           7           8           9
printObject(const T& input): with DataType = d
-0.1        2.2         12.21
printObject(const T& input): with DataType = NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE
Mon         Tue         Wed         Thur        Fri         Sat         Sun
```

Typical Error with Template

```
// example3.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< typename  T >
void printObject(const T& input) {
  cout << "printObject(const T& input): with T = " << typeid( T ).name() << endl;
  cout << input << endl;
}

class Dummy {
  public:
    Dummy(const string& name="") {
      name_ = name;
    }
  private:
    string name_;
};

int main() {

  string name("jane");
  Dummy  bad("bad");

  printObject( name );
  printObject( bad );

  return 0;
}
```

No operator<<() implemented for class Dummy!

Error at compilation time because no code can be generated

No prototype to use to generate printArray(const Dummy& input)

```
$ g++ -o /tmp/example3 example3.cpp
example3.cpp:10:8: error: invalid operands to binary expression ('std::__1::ostream' (aka 'basic_ostream<char>') and
      'const Dummy')
  cout << input << endl;
  ~~~~ ^  ~~~~~
example3.cpp:28:3: note: in instantiation of function template specialization 'printObject<Dummy>' requested here
  printObject( bad );
[….] Followed by 100s of other error messages!
```

Compiling Template Code

- Template functions (and classes) are incomplete without specialisation with specific data type
- Template code can not be compiled alone
  - Cannot put template code in source file and into the library
- Remember: code for each specialisation “generated” by compiler at compilation time
- ***Template functions*** *and* ***classes*** *(including member functions) implemented in* ***header files only***
- Data types used must implement the operations used in template function

C++ Template and C Macros

- They might look similar at first glance but fundamentally very different

- Both Templates and Macros are expanded at compile time by compiler and no run-time overhead

- Compiler performs type-checking with template functions and classes
  – Make sure no syntax or type errors in the template code

Class Template

- Class templates are similar to template functions
  - Actual class generated by compiler based on type of parameter provided by user
  - Also referred to as parameterised types

- Class templates extremely useful to implement containers of objects, iterators, and associative maps
  - containers: **`vector<T>`**, **`collection<T>`**, and **`list<T>`** of objects have well defined behaviour independently from particular type **`T`**
    - get n^{th} element regardless of type
  - Iterators: **`vector<T>::iterator`** manipulates objects in a vector of objects of type **`T`**

  - Associative maps: **`map<typename Key, typename Value>`** can be used to relate objects of type **`Key`** to objects of type **`Value`**

Class Template Syntax

```
// example5.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template< typename  T >
class Dummy {
  public:
    Dummy(const T& data);
    ~Dummy();
    void print() const;
  private:
    T* data_;
};

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

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

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

}

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

  double x = 1.23;
  Dummy<double> d2( x );

  d1.print();
  d2.print();

  return 0;
}
```

```
$ g++ -o /tmp/example5 example5.cpp
$ /tmp/example5
Dummy<T>::print() with type T =
NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE, *data_: test
Dummy<T>::print() with type T = d, *data_: 1.23
```

Header and Source Files for Template Classes

```
#ifndef DummyBis_h_
#define DummyBis_h_

template< typename  T >
class DummyBis {
  public:
    DummyBis(const T& data);
    ~DummyBis();
    void print() const;

  private:
    T* data_;
};
#endif
```

```
// example5-bad.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

#include "DummyBis.h"

int main() {
  DummyBis<std::string> d1( std::string("test") );

  double x = 1.23;
  DummyBis<double> d2( x );

  d1.print();
  d2.print();

  return 0;
}
```

```
$ g++ -o /tmp/bad example5-bad.cpp
Undefined symbols for architecture x86_64:
  "DummyBis<std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
>::DummyBis(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&)", referenced from:
      _main in example5-bad-ad84c5.o
  "DummyBis<std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > >::~DummyBis()", referenced from:
      _main in example5-bad-ad84c5.o
  "DummyBis<double>::DummyBis(double const&)", referenced from:
      _main in example5-bad-ad84c5.o
  "DummyBis<double>::~DummyBis
      _main in example5-bad-ad8
  "DummyBis<std::__1::basic_str
      _main in example5-bad-ad8
  "DummyBis<double>::print() const", referenced from:
      _main in example5-bad-ad84c5.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Can't separate into header and source files… compiler **needs** the source code for template class to generate specialized template code!

class Dummy

- All code contained in header file Dummy.h
  - no source file Dummy.cc

- Note that typename, class, and T must be written for each and every function

```
#ifndef Dummy_h_
#define Dummy_h_

#include<iostream>

template< typename  T >
class Dummy {
  public:
    Dummy(const T& data);
    ~Dummy();
    void print() const;

  private:
    T* data_;
};

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


template<class T>
Dummy<T>::~Dummy() {
  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;
}
#endif
```

Template Class in Header Only

- Template classes must be implemented
  in the header file

- Can still separate declaration and implementation
  as long as it stays within header file

- Including header file provides source code to compiler in order to generate code for specialised templates

```
// example5bis.cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

#include "Dummy.h"

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

  double x = 1.23;
  Dummy<double> d2( x );

  d1.print();
  d2.print();

  return 0;
}
```

Standard Template Library (STL)

- Powerful library implementing template-based and reusable components using generic programming

- Provides comprehensive set of common data structures and algorithms to manipulate such data structures
  - Particularly useful in industrial applications
  - STL now part of the Standard C++ Library

- Popular and key components largely used
  - Containers: templated data structures that can store any type of data
  - Iterators provide pointers to individual elements of containers

  - Algorithms to manipulate data structures: insert, delete, sort, copy elements of containers

Example: map<string,int> and iterators

```
// example6.cpp
#include <iostream>
#include <string>
#include <map>

int main() {

  std::map<std::string, int> days;

  days[std::string("Jan")] = 31;
  days[std::string("Feb")] = 28;
  days[std::string("Mar")] = 31;
  days[std::string("Apr")] = 30;
  days[std::string("May")] = 31;
  days[std::string("Jun")] = 30;
  days[std::string("Jul")] = 31;
  days[std::string("Aug")] = 31;
  days[std::string("Sep")] = 30;
  days[std::string("Oct")] = 31;
  days[std::string("Nov")] = 30;
  days[std::string("Dec")] = 31;
  std::map<std::string, int>::const_iterator iter;
```

```
$ g++ -o /tmp/example6 example6.cpp
$ /tmp/example6
key iter->first: Apr value iter->second: 30
key iter->first: Aug value iter->second: 31
key iter->first: Dec value iter->second: 31
key iter->first: Feb value iter->second: 28
key iter->first: Jan value iter->second: 31
key iter->first: Jul value iter->second: 31
key iter->first: Jun value iter->second: 30
key iter->first: Mar value iter->second: 31
key iter->first: May value iter->second: 31
key iter->first: Nov value iter->second: 30
key iter->first: Oct value iter->second: 31
key iter->first: Sep value iter->second: 30
Bad key: december
month? frt
Bad key: frt. try again
month? Jan
Jan has 31 days
month?
```

```
 for( iter = days.begin(); // first elemnt
       iter!= days.end();  // last element
       ++iter  // step
     ) {
   std::cout << "key iter->first: " << iter->first
             << " value iter->second: " << iter->second
             << std::endl;
  }

  // lookup non-exisiting key
  std::string december("december");

  iter = days.find(december);

  if( iter != days.end() ) {
    std::cout << days["Dec"] << std::endl;
  } else {
     std::cout << "Bad key: " << december << std::endl;
  }

  std::string month;
  do {
    /* code */
    std::cout << "month? ";
    std::cin >> month;
    iter = days.find(month);
    if( iter != days.end() ) {
      std::cout << month << " has "
                << iter->second << " days" << std::endl;
    } else {
       std::cout << "Bad key: " << month
                 << ". try again" << std::endl;
    }
  } while(true);


  return 0;
}
```

Both map and iterator classes are template classes!

Same code can be used for any data type

polymorphism at compilation time!

Non-Type Parameter for Class Template

```
template<class T, int maxSize=7>
class Vector {
  public:
    Vector() {
      size_ = maxSize;
      for(int i=0; i<size_; ++i) {
        data_[i] = T();
      }
    }

  private:
    int size_;
    T data_[maxSize];
};
```

- Template can be used also with non-type parameters
- Helpful to instantiate data members in the constructor
- Replace dynamic allocation with automatic objects
  - Easier memory management
  - Faster code since variables are automatic and no new/delete needed!

Class Vector with Template

Vector.h

```
#include <iostream>

template<class T, int maxSize=7>
class Vector {
  public:
    Vector() {
      size_ = maxSize;
      for(int i=0; i<size_; ++i) {
        data_[i] = T();
      }
    }

    ~Vector() {};
    int size() const { return size_; }
    T& operator[](int index);
    const T& operator[](int index) const;
    //friend std::ostream& operator<<(ostream& os,
                                   const Vector<T,maxSize>& vec);

  private:
    int size_;
    T data_[maxSize];
};

template<typename T, int maxSize>
T& Vector<T,maxSize>::operator[](int index){
  return data_[index];
}
```

```
template<typename T, int maxSize>
const T& Vector<T,maxSize>::operator[](int index) const{
  return data_[index];
}

template<typename T, int maxSize>
std::ostream&
operator<<(std::ostream& os, const Vector<T,maxSize>& vec)
{
  os << "vector with " << vec.size() << " elements: " <<
endl;
  for(int i=0; i<vec.size(); ++i) {
    os << "i: " << i
       << " v[i]: " << vec[i] << endl;
  }
  return os;
}
```

```
// example7.cpp
#include <iostream>
#include <string>
using namespace std;

#include "Vector.h"

int main() {
  Vector<string> vstr;
  vstr[0] = "test";
  vstr[1] = "foo";
  cout << vstr << endl;
  Vector<double,1000> v1;
  return 0;
}
```

```
$ g++ -o /tmp/example7 example7.cpp
$ /tmp/example7
vector with 7 elements:
i: 0 v[i]: test
i: 1 v[i]: foo
i: 2 v[i]:
i: 3 v[i]:
i: 4 v[i]:
i: 5 v[i]:
i: 6 v[i]:
```

Exercise

- Implement the old Datum class to be a template class

```
Datum<double> mis1(1.2,-0.3);
Datum<int> count(6,2);
```

Template and Inheritance

- 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 specialisations
  - 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 specialised templates are identical except for the data type
  - Template-class specialisation is equivalent to any regular non-template class

- But remember…
  - Class template not equivalent to base class
  - No base-class pointer mechanism for different specialisations
  - No runtime polymorphism
  - Different specialisations 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 initialise 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

```
$ ./example10
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 specialisations 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 specialisation
  - `template<class T> class MyString : public Dummy<std::string> {};`

- A class-template specialisation can be derived from a class-template specialisation
  - `class Dummy<Car> : public Vector<Object> { };`

- A non-template class can be derived from a class-template specialisation
  - `class Student : public Dummy<std::string> { };`

Static data member

```
#ifndef NewDummy_h_
#define NewDummy_h_

#include<iostream>

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

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

template<class T>
int NewDummy<T>::total_ = 0;


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

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

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

- All code in **`NewDummy.h`**
  - Remember no source file

Template and static

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

#include "NewDummy.h"

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

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

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

  cout << "NewDummy<int>::total(): " << NewDummy<int>::total() << endl;

  return 0;
}
```

```
$ g++ -o /tmp/example11 example11.cpp
$ /tmp/example11
NewDummy<std::string>::total(): 3
NewDummy<double>::total(): 2
NewDummy<int>::total(): 0
```

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

Template and friend Functions

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

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

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

    // member function of Bar friend of all specialisations
    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>`

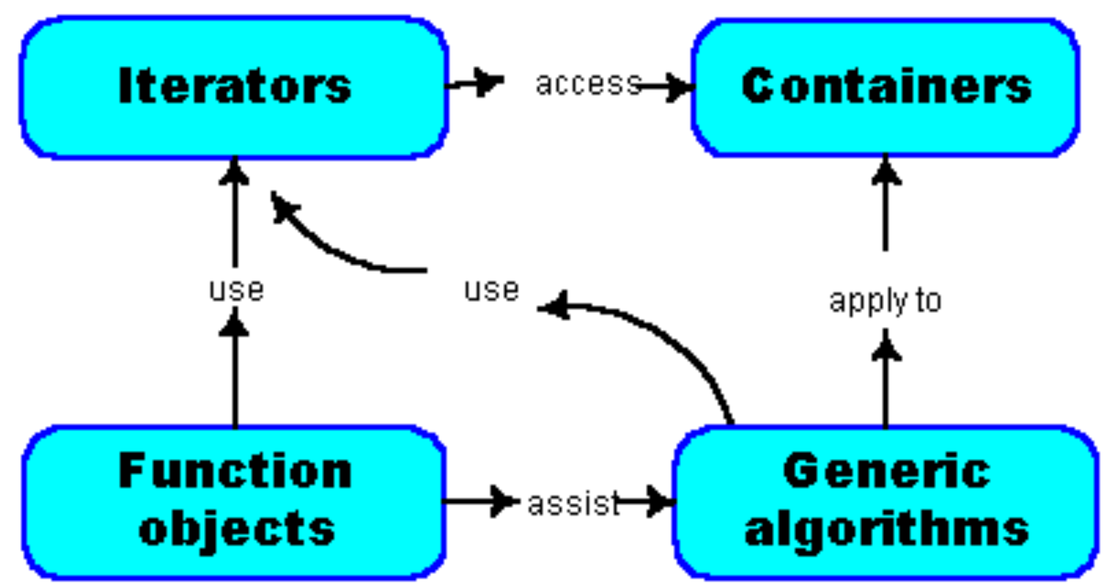
`specialPrint(string)` friend of `Foo<string>` but NOT friend of `Foo<int>`

`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

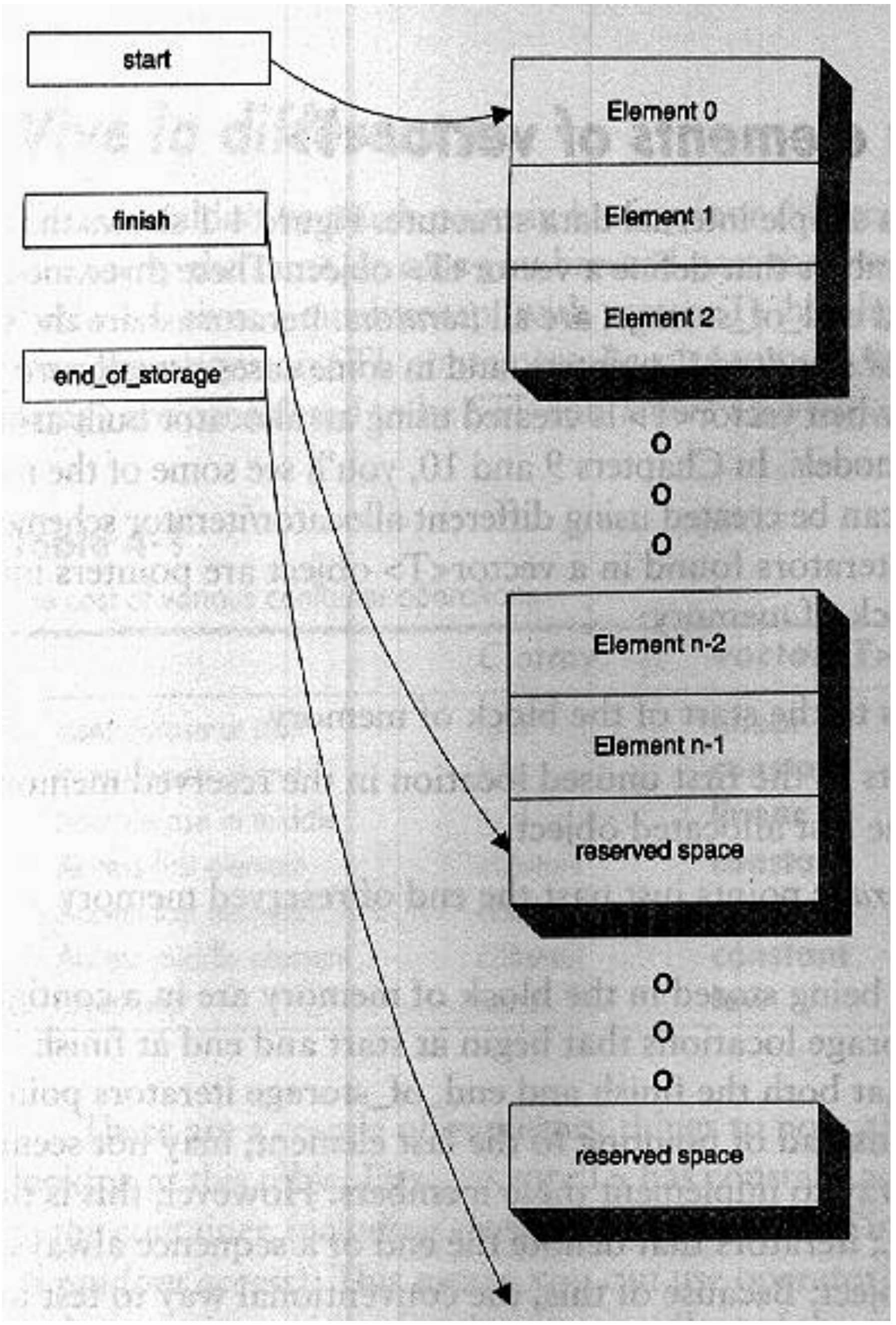


- Algorithms: methods used to manipulate container items
  - Finding, sorting, reverting items
- Iterators: generalization of pointer to provide access to items in a container

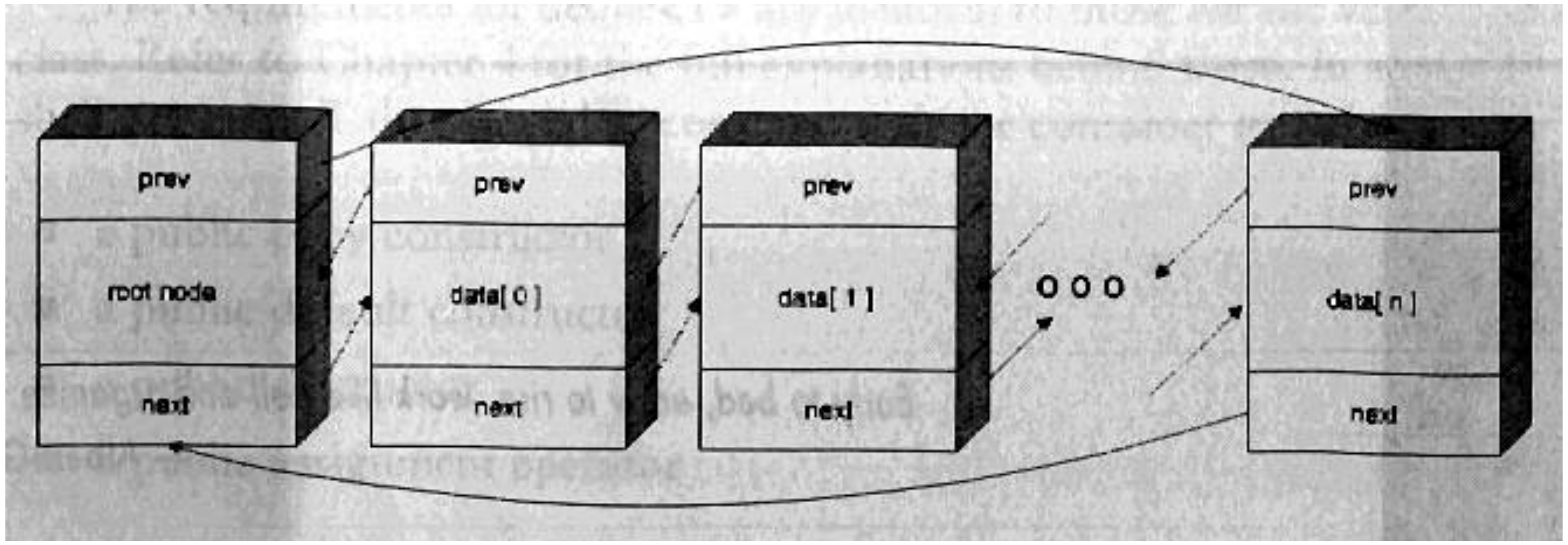
containers

- 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

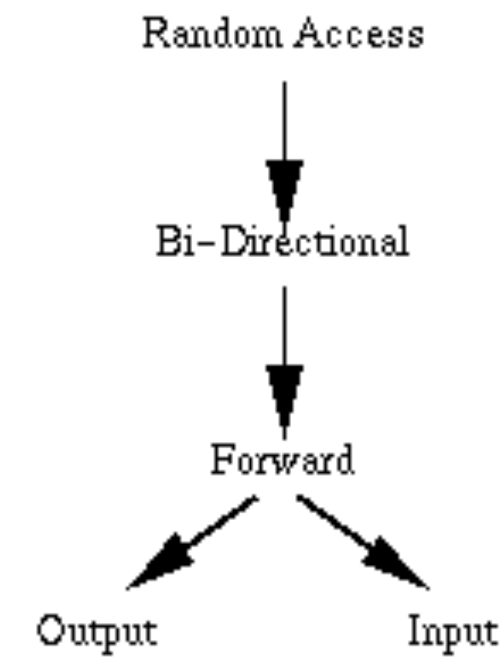
- Inserting an object of type `T` corresponds to copying object into the container

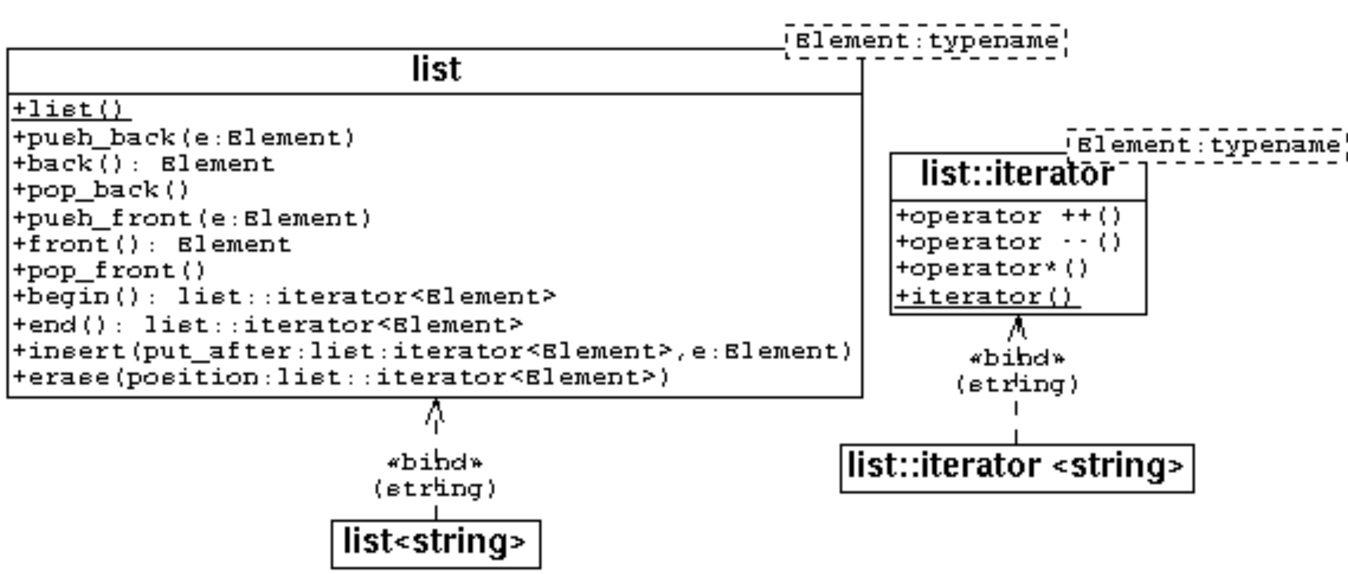
- Sequential containers require a proper copy constructor and assignment operator (=) for class `T`
  - Default implementations is fine as long as non-trivial data members are used

- 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

- 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

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

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 used 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 optimise performance and minimise 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 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

Error Handling in C++

Exception Handling: What does it mean?

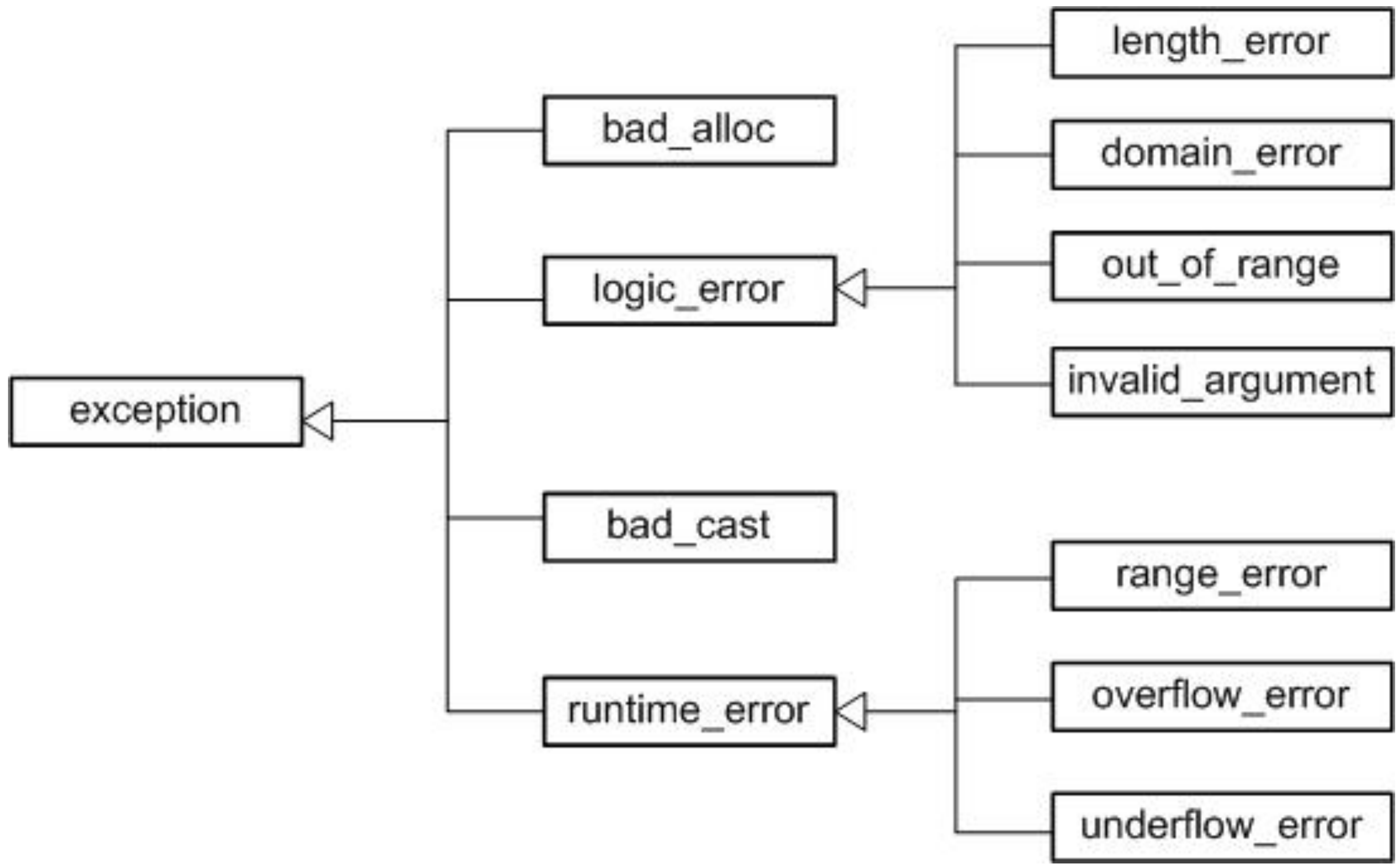
- Under normal circumstances applications should run successfully to completion

- 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

```
// example13.cpp
#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 (double) i1/i2;
}

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

  cout << "Compute ratio a/b of 2 integers (ctrl-C to end)"
       << endl;
  do{
    cout << "a? ";
    cin >> i1;
    cout << "b? ";
    cin >> i2;

    try {
      cout << "a/b =  " << ratio(i1,i2) << endl;

    } catch(std::runtime_error& ex) {
       cout << ex.what() << ": denominator is 0"  << endl;
    }

  } while (true);
  return 0;
}
```

throw an exception when error condition occurs

exception is a C++ object!

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

```
$ g++ -o /tmp/example13 example13.cpp
$ /tmp/example13
Compute ratio a/b of 2 integers (ctrl-C to end)
a? 3
b? 7
a/b =  0.428571
a? 7
b? 0
a/b =  error in ratio: denominator is 0
a?
```

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

Exceptions Defined by Users

- New exceptions can be implemented by users
  - Inherit from existing exceptions and specialise for use case relevant for your application

```
// example14.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("division 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 << "Compute ratio a/b of 2 integers (ctrl-C to end)" << endl;
  do{
    cout << "a? ";
    cin >> i1;
    cout << "b? ";
    cin >> i2;
    try {
      cout << "a/b =  " << ratio(i1,i2) << endl;

    } catch(MyError& ex) {
       cout << "error occured..." << ex.what() << endl;
    }
  } while (true);
  return 0;
}
```

```
$ g++ -o /tmp/example14 example14.cpp
$ /tmp/example14
Compute ratio a/b of 2 integers (ctrl-C to end)
a? 3
b? 0
a/b =  error occurred...division by zero
a?
```