LIVEWIRE
FOR LIVE CAREERS

C++ Part 5: STL containers & Iterators, Exception and Error Handling, Libraries & Makefiles

STL CONTAINERS

- The Standard Template Library (STL), has different types of containers for different needs
- two categories of containers based on their functioning: sequences & associations
- A *Sequence Container* is a Container that stores objects of the same type in a linear arrangement.
- Sequence containers: array, vector, list, forward_list, deque.

Container	Function
basic_string	stores and manipulates sequences of characters
Array (since C++11)	static contiguous array
vector	dynamic contiguous array
Deque	double-ended queue
forward_list (since C++11)	singly-linked list
List	doubly-linked list

- A container is a holder object that stores a collection of other objects (its elements) implemented as class templates.
- *Container adaptors* are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements
- container adaptors: stack, queue and priority_queue

Container Adaptors	Representation
stack	LIFO stack
queue	FIFO queue
priority_queue	Priority queue

- *Associative containers* implement ordered associative arrays as template classes.
- these are set, map, multiset, multimap
- Each of these differ only on constraints placed on their elements
- *Unordered associative containers*, as their name implies, do not order their elements.

Associative Containers	Representation
set	set
multiset	Multiple-key set
map	Map
multimap	Multiple-key map

Unordered associative containers:	Representation
unordered_set	Unordered Set
unordered_multiset	Unordered Multiset
unordered_map	Unordered Map
unordered_multimap	Unordered Multimap

TERMINOLOGY

- Containers - ways of clumping together variables of the same type so that sorting, searching, etc., becomes easy

- Maps - ways of setting up a mapping between 2 sets of values.

- Algorithms- Set of standard functions that work on containers.

Algorithm

STL algorithms are generic: every algorithm works on a variety of containers, including built-in types, and many work on all containers.

There's an algorithm called reverse(). Here's how you use it to reverse a string.

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main() {
  string s = "a long string";
// now reverse the whole string from the beginning to the end
  reverse(s.begin(), s.end());
  cout << s << endl;
}
```

Algorithm

and here's how you reverse a vector

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

int main() {
  vector<int> s(3);
  s[0]=1; s[1]=2; s[2]=3;
  reverse(s.begin(), s.end());
  cout << s[0] << s[1] << s[2] << endl;
}
```

Accessing Container Elements - Iterators

- The elements of containers are accessed using iterators
- An iterator is an object that can iterate over elements in an STL container and provide access to individual elements
- Iterators to sequence containers and associative containers use the functions *begin(), end(), rend()* and *rbegin()* to traverse the containers.
- Forward_list and unordered assosiative containers do not support *rbegin()* and *rend().*
- Syntax to define iterators for a class

Container <Type>::iterator

Container <Type>::const_iterator

Iterator Classes

- Each STL container defines what class of iterators it can return.
- InputIterator

If *iter* is an InputIterator, you can use:

  - ++iter and iter++ to increment it, i.e., advance the pointer to the next element
  - *iter to dereference it, i.e., get the element pointed to
  - == and != to compare it another iterator (typically the "end" iterator)

- Example:

```
vector<int> v;
        vector<int>::iterator iter;

        v.push_back(1);
        v.push_back(2);
        v.push_back(3);

        for (iter = v.begin(); iter != v.end(); iter++)
          cout << (*iter) << endl;
```

- **OutputIterator**

If *iter* is an OutputIterator, you can use:

- *++iter* and *iter++* to increment it, i.e., advance the pointer to the next element
- **iter = ...* to store data in the location pointed to

- subclasses of OutputIterator
  - Insert iterators
  - Ostream iterators

Insert Iterators

- lets you "point" to some location in a container and insert elements
- Example: *iter = value;

You create an insert iterator with one of the following:

- back_inserter<container> returns an OutputIterator pointing to the end of the container. Output to this iterator gets added to the end of the container, using the container's push_back() operation.
- front_inserter<container> returns an OutputIterator pointing to the front of the container. Output to this iterator gets added to the front of the container, using the container's push_front() operation.
- inserter<container, iterator> returns an OutputIterator pointing to the location pointed to by iterator of the container. Output to this iterator gets added to the container from that point forward, using the container's insert() operation.

- Ostream OutputIterator: let you "point" to an output stream and insert elements into it, i.e., write to the output stream.
- Example:

```
ostream_iterator<int> outIter( cout, " " );

 ...

 copy( v.begin(), v.end(), outIter );
```

- Istream InputIterator
- copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
- The code fragment above constructs an istream iterator that reads integers from cin and copies them into a vector v.
- We use a *back_inserter* to add the elements to the end of the vector

- **ForwardIterator:** combines *InputIterator* and *OutputIterator*
- Used to read and write to container
- Also supports saving and reuse.

Example:

```
iterSaved = iter;
iter++;
cout << "Previous element is " << (*iterSaved) << endl;
cout << "Current element is " << (*iter) << endl;
```

- BidirectionalIterator: you can use:
- all ForwardIterator operations
- *--iter* and *iter--* to decrement it, i.e., advance the pointer to the previous element
- RandomAccessIterator:

If *iter1* and *iter2* are RandomAccessIterator's, you can use:

- all BidirectionalIterator operations
- standard pointer arithmetic, i.e., *iter + n, iter - n, iter += n, iter -= n,* and *iter1 - iter2* (but **not** iter1 + iter2)
- all comparisons, i.e., iter1 > iter2, iter1 < iter2, iter1 >= iter2, and iter1 <= iter2

```
// Simple shapes w/ STL
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
public:
  void draw() { cout <<
"Circle::draw\n"; }
  ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
  void draw() { cout <<
"Triangle::draw\n"; }
  ~Triangle() { cout << "~Triangle\n"; }
};
```

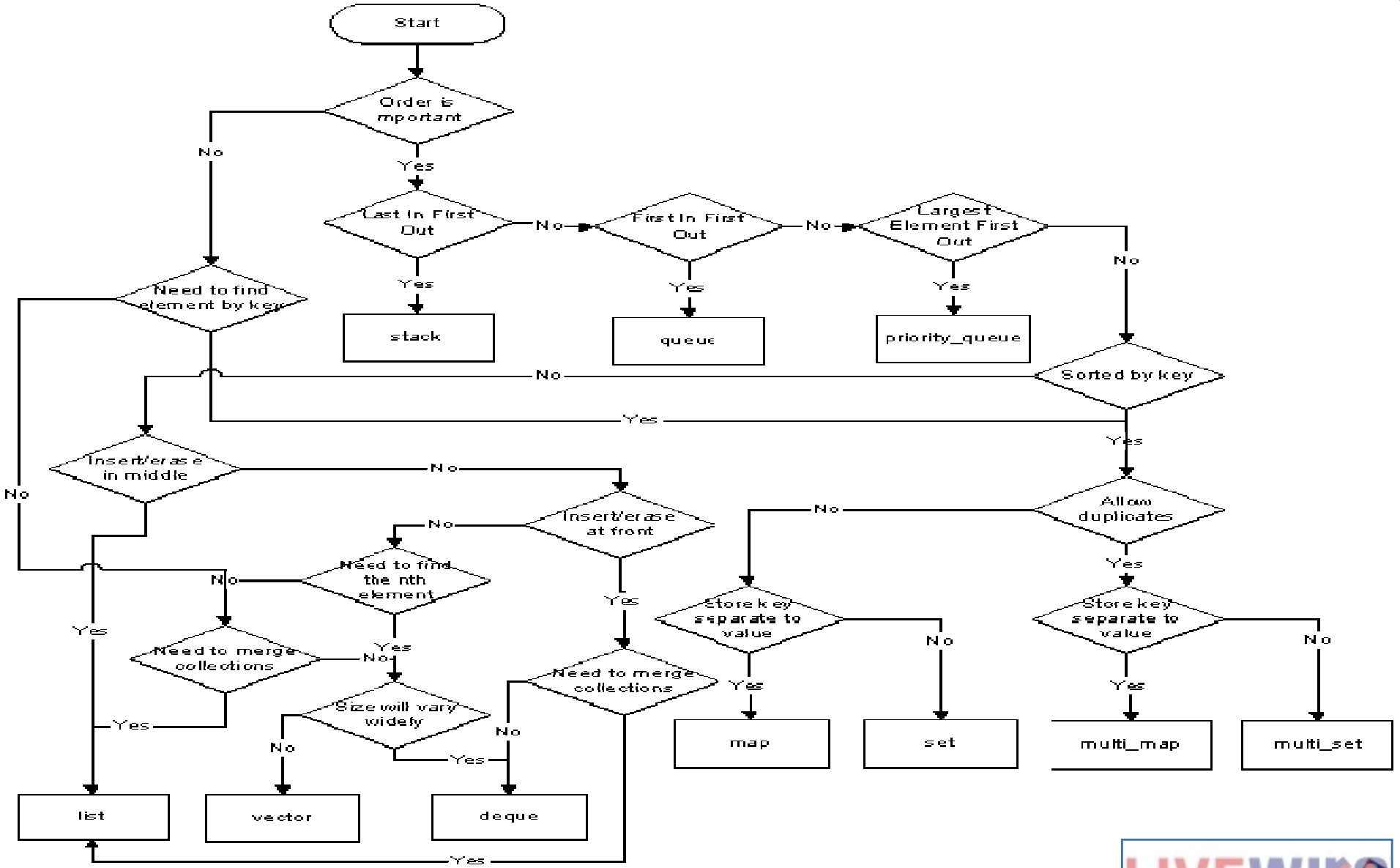
```
class Square : public Shape {
public:
  void draw() { cout <<
"Square::draw\n"; }
  ~Square() { cout << "~Square\n";
}
};
typedef std::vector<Shape*>
Container;
typedef Container::iterator Iter;
int main() {
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);


 for(Iter i = shapes.begin();
     i != shapes.end(); i++)
   (*i)->draw();
 // ... Sometime later:
 for(Iter j = shapes.begin();
     j != shapes.end(); j++)
   delete *j;
}
```

Sequence Containers

Sequence containers hold elements of a single type as a linear sequence. That is, they are ordered in the same way as they are added to the container

Array/String	Array class
vector	Vector
deque	Double ended queue
forward_list	Forward list
list	List

Use of STL Containers



Container Functions

- The containers are defined in headers named after the names of the containers, e.g. vector is defined in header <vector>.
- All containers have begin(), end(), size(), max_size(), empty(), and swap() methods.
- Some of the *modifier functions* in containers are assign(), insert(), erase(), clear(), push_back(), push_front(), swap() etc
- Functions used to *access elements* of stack front(), back(), at() and operator[]
- *List operations*: available only for list and forward list. Example functions are remove(), removeif(), sort(), merge(), unique(), reverse()
- Associative container operations: count(), find(), equal_range(), lower_bound(), upper_bound()

```
//example using the set class
template
#include <cassert>
#include <set>
using namespace std;

int main() {
set<int> intset;
for(int i = 0; i < 25; i++)
for(int j = 0; j < 10; j++)
// Try to insert duplicates:
intset.insert(j);
assert(intset.size() == 10);
}
```

Arrays, String, Vector

If you want to store a determined/undetermined number of objects and you're never going to delete any, then

- Use array if the size is known at compile time.
- Use string if you only need to work with character types and need a string, not just a general-purpose container.
- Use vector in all other cases. It's the default replacement for a C array, and it works like one, but doesn't overflow. You can set its size beforehand as well with reserve().

With all three of these you can use the data() member function to get a pointer to the first element of the container.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  // create a vector to store int
  vector<int> vec;
  int i;
  // display the original size of vec
  cout << "vector size = " <<
vec.size() << endl; /* displays the
size of the vector. */
// push 5 values into the vector
  for(i = 0; i < 5; i++)
  {
    vec.push_back(i); /* inserts
value at the end of the vector,
expanding its size as needed.*/
  }
// display extended size of vec
  cout << "extended vector size = "
<< vec.size() << endl;
// access 5 values from the vector
  for(i = 0; i < 5; i++){
    cout << "value of vec [" << i
<< "] = " << vec[i] << endl;
  }
```

```
// use iterator to access the values
   vector<int>::iterator v = vec.begin(); /*begin( ) returns an iterator to the start of the vector. */
   while( v != vec.end()) /* end( ) returns an iterator to the end of the vector. */
  {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```

List

- implemented as a doubly linked list
- Each element has its own segment of memory and refers to its predecessor and its successor
- Do not provide random access
- A step to the next or previous element is possible
- Used to store an undetermined number of objects that will be added or deleted often.
- Has a higher memory usage.

Example:

```
#include <iostream>
 #include <list>


 using namespace std;


 int main() {
    list<char> coll; //list container
for character elements


    // append elements from 'a' to 'z'
    for (char c = 'a'; c <= 'z'; ++c) {
        coll.push_back(c);
    }

/*   print all elements
     * - while there are elements
     * - print and remove the first
element
     */
     while ( !coll.empty()) {
         cout << coll.front() << ' ';
         coll.pop_front();
     }
     cout << endl;
}
```

Forward List

- singly-linked lists
- sequence containers allow constant time insert and erase operations anywhere within the sequence.
- ordered by association of each element of a link to next element in the sequence.
- only iterated forwards.
- more efficient than list

Example:

```
#include <iostream>
#include <forward_list>
int main()
{
    std::forward_list<int>
forward_list;
    forward_list.push_front(3);
    forward_list.push_front(2);
    forward_list.push_front(1);
auto first = forward_list.begin();
    auto last = first;
last++;
    last++;
    std::cout << "The first element is
" << *first << std::endl;
    std::cout << "The last element is "
<< *last << std::endl;
    std::cout << "Iterate the
forward_list:" << std::endl;
    for (auto it =
forward_list.begin(); it !=
forward_list.end(); ++it)
        std::cout << *it << std::endl;
    return 0;
}
```

DEQUE

- Abbreviation for "double-ended queue.“(pronounced “deck”)
- Dynamic array that can grow in both directions.
- Inserting elements at the end and at the beginning is fast
- Used to insert elements not just from the back but from the front also.
- Not guaranteed to store all its elements in contiguous storage locations

Example:

```
// a simple deque example
#include <iostream>
#include <deque>
using namespace std;
int main()
{
   // deque container for floating-point elements declaration
   deque<float> elem, elem1;
   // insert the elements each at the front
• cout<<"push_front()\n";
•    for(int i=1; i<=10; ++i)
•    // insert at the front
elem.push_front(i*(1.2));
// print all elements separated by a space
   for(i=0; i<elem.size(); ++i)
      cout<<elem[i]<<' ';
   cout<<endl;
// insert the elements each at the back
   cout<<"\npush_back()\n";
   // insert at the back
   for(i=1; i<=10; ++i)
      elem1.push_back(i*(1.2));
// print all elements separated by a space
   for(i=0; i<elem1.size(); ++i)
   cout<<elem1[i]<<' ';
   cout<<endl;
   return 0;
}
```

Container Adaptors

The container adapters are classes that provide a subset of a container's functionality but may provide additional functionality that makes it easier to use containers for certain scenarios.

- **Stack**. A stack can be instantiated either with a *vector*, a *list* or a *deque*. The member functions **empty**, **size**, **top**, **push** and **pop** are accessible to the user.

```
stack<vector<int> > s1;
stack<list<int> >   s2;
stack<deque<int> >  s3;
```

```
s1.push(1); s1.push(5);
cout << s1.top() << endl;
s1.pop();
cout << s1.size() << endl;
s1.empty()? cout << "empty" : cout << "not empty";
```

- **top** returns the element on the top of the stack,
- **pop** removes the top element from the stack.
- For comparison of two stacks, operator **==** and operator **<** are defined.

Queue

➢ **Queue**. A queue can be instantiated with a *list* or a *deque*.

```
queue<list<int> >  q1;
queue<deque<int> > q2;
```

- Its public member functions are **empty**, **size**, **front**, **back**, **push** and **pop**.
- **front** returns the next element from the queue,
- **pop** removes this element.
- **back** returns the last element pushed to the queue with **push**.
- As with the stack, two queues can be compared using operator **==** and operator **<**.

Priority queue

- A priority queue can be instantiated with a *vector* or a *deque*. A priority queue holds the elements added by **push** sorted by using a function object **comp** of type *Compare*.

```
// use less as compare object
priority_queue<vector<int>, less<int> > pq1;

// use greater as compare object
priority_queue<deque<int>, greater<int> > pq2;

vector v(3, 1);
// create a priority_queue out of a vector, use less as compare object
priority_queue<deque<int>, less<int> > pq3 (v.begin(), v.end() );
```

- **top** returns the element with the highest priority,
- **pop** removes this element.
- The element with the highest priority is determined by the sorting order imposed by **comp**.

Note, that a priority queue internally is implemented using a heap.

Associative Containers

These are associative containers, meaning that elements have associations with each other used for determining uniqueness or mappings:

set	collection of unique keys, sorted by keys
map	collection of key-value pairs, sorted by keys, keys are unique
multiset	collection of keys, sorted by keys
multimap	collection of key-value pairs, sorted by keys

Set and Multiset

- Set and multiset containers sort their elements automatically according to a certain sorting criterion. The difference between the two is that multisets allow duplicates, whereas sets do not.
- Use **set** to store only unique values.
- Use **multiset** to allow duplicate elements
- To use set or multiset include header file <set>

```
//example using the set class template
#include <cassert>
#include <set>
using namespace std;
int main() {
set<int> intset;
for(int i = 0; i < 25; i++)
for(int j = 0; j < 10; j++)
// Try to insert duplicates:
intset.insert(j);
assert(intset.size() == 10);//confirming if the size of the set is 10.
}
```

Example: Multiset

```
//program to iterate through a multiset.
#include <iostream>
#include <set>
int main()
{
    std::multiset<int> multiset;
    multiset.insert(2);
    multiset.insert(2);
    multiset.insert(1);
    multiset.insert(1);
    multiset.insert(3);
    multiset.insert(3);
    auto first = multiset.begin();
    auto last = multiset.end();
    --last;
    std::cout << "The first element is " << *first << std::endl;
    std::cout << "The last element is " << *last << std::endl;
    std::cout << "Iterate the multiset:" << std::endl;
    for (auto it = multiset.begin(); it != multiset.end(); ++it)
        std::cout << *it << std::endl;
    return 0;
}
```

Map and Multimap

- A map is an Associative array
- Takes input of key-value pairs
- Input of another value with the same key overwrites the original value.

- A multimap is a map that allows for insertion of multiple values for the same key.
- A key lookup, returns a container with all the values in it.

- Map and Multimap , both, includes header <map>

Example:Multimap

```
#include <iostream>
 #include <map>
 #include <string>
 using namespace std;
 int main() {
    //type of the collection
    typedef multimap<int, string> IntStringMMap;
    IntStringMMap coll; //set container for int/string values
    //insert some elements in arbitrary order
    //- a value with key 1 gets inserted twice
coll.insert(make_pair(5,"tagged"));
    coll.insert(make_pair(2,"a"));
    coll.insert(make_pair(1,"this"));
```

```
coll.insert(make_pair(4,"of"));
  coll.insert(make_pair(6,"strings"));
    coll.insert(make_pair(1,"is"));
coll.insert(make_pair(3,"multimap"));
   /* print all element values
    *- iterate over all elements
    *- element member second is the value
    */
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' ';
    }
    cout << endl;
 }
```

Example: Map

```
#include <iostream>
 #include <map>
 #include <string>
 using namespace std;
 int main() {
    //type of the collection
    typedef map<int, string> IntStringMMap;
    IntStringMMap coll; //set container for int/string values
    //insert some elements in arbitrary order
    //- a value with key 1 gets inserted twice
coll.insert(make_pair(5,"tagged"));
    coll.insert(make_pair(2,"a"));
coll.insert(make_pair(1,"this"));
    coll.insert(make_pair(4,"of"));
```

```
coll.insert(make_pair(6,"strings"));
    coll.insert(make_pair(1,"is"));
    coll.insert(make_pair(3,"map"));
   /* print all element values
    *- iterate over all elements
    *- element member second is the value
    */
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' ';
    }
    cout << endl;
}
```

ERRORS AND WARNINGS

A programming error falls under one of three categories:

- compile-time error
- run-time error
- logic error

- Compile-time Errors : results from the programmer's misuse of the language
- Example: syntax error, undeclared variable, missing semicolon, mismatched curly braces and parentheses, and simple typographical errors.

- Run-time Errors : violations of the language that cannot be detected at compile time
- Example:
- Segmentation violation: This error indicates that the program tried to de-reference a pointer containing a bad value.
- Stack overflow: The program tried to use too many temporary variables. Sometimes this means the program is too big or using too many big temporary arrays, but most of the time this is due to infinite recursion problems.
- Divide by 0: Divide by 0 is an obvious error.
- Logic Errors : neither the compiler nor the run-time system is able to detect the problem

```
#include <iostream>
using namespace std;
int main()
{
double degreesF = 0, degreesC = 0;
// Define the relationship between F and C
degreesC = 5.0 / 9 * (degreesF - 32);
 // Prompt user for degrees F
cout << “Enter the temperature in degrees F: “;
 // Read in the user’s input
 cin >> degreesF;
// Report the result
 cout << degreesC << endl;
 }
```

EXCEPTION

- Type of Runtime Error
- built upon three keywords: *try, catch*, and *throw*.
- throw: A program throws an exception when a problem shows up using keyword throw.
- catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem using keyword catch
- try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

- Syntax of try/catch block

```
try
{
// protected code
}catch( ExceptionName e1 )
{
// catch block
}catch( ExceptionName e2 )
{
// catch block
}catch( ExceptionName eN )
{
// catch block
}
```

```
#include <iostream>
using namespace std;
 int main()
{
Try
 {
throw 10;
}
catch (char *excp)
{
cout <<"Caught "<< excp;
}
catch (...)
{
 cout <<"Default Exception\n";
}
return 0;
}
```

C++ Standard Exceptions

- The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions called *std::exception* defined in the <exception> header
- Contains virtual member function *what()* that returns a null-terminated character sequence (of type char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class myexception : public exception
{
virtual const char* what() const throw()
{
return “My exception happened”;
}
} myex;
int main()
{
try
{
throw myex;
}
catch (exception& e)
{
cout << e.what() <<’\n’;
}
```

Exceptions in Dynamic Memory Allocation

- C++ provides two standard mechanisms to check if the allocation was successful:
- Handling exceptions: Using this method, an exception of type *bad_alloc* is thrown when the allocation fails. If this exception is thrown and it is not handled by a specific handler, the program execution is terminated.This exception method is the method used by default by new, and is the one used in a declaration like:

```
foo = new int[5];  // if allocation fails, an exception is thrown
```

- *Nothrow*: when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution normally.

```
int * foo;
foo = new (nothrow) int[5];
if (foo == nullptr) {  // error assigning memory. Take measures. }
```

Using Gnu Compiler

- G++ compiler
- Using command prompt

SYNTAX: g++ filename.cpp -o filename.

g++ hello.C -o hello

- Pre-processing: (C-preprocessor cpp.exe)

cpp hello.c > hello.i

- Compilation:

g++ -S hello.i

- Assembly (as.exe)

as -o hello.o hello.s

- Linker: (ld.exe)

ld -o hello.exe hello.o

Compiling with multiple source files

- g++ file1.C file2.C -o myprog
- creates the executable file myprog
- The same result can be achieved using the following three commands:
- g++ -c file1.C
- g++ -c file2.C
- g++ file1.o file2.o -o myprog

CREATING LIBRARIES

- C++ library consist of 2 types or files.
  - Header files
  - Precompiled binary
- two types of libraries: *static libraries* and *dynamic libraries*
- static libraries typically have a .lib extension
- dynamic libraries typically have a .dll

Creating static libraries

- Creating the object file:

```
g++ -c AddNumbers.cpp -o AddNumbers.o
```

- Archiving the object files:

```
ar rcs libAddNumbers.a AddNumbers.o
```

g++ flags:

c: Disables the linker invocation.

o: A custom output file.

ar flags:

r: Insert the files member... into archive (with replacement).

c: Indicates you are creating an archive file.

s: Write an object-file index into the archive, change is made to the archive.

- The library name must start with the three letters lib and have the suffix .lib/.a. The lib prefix is required by the linker to find the static library.

Creating a shared library

- g++ -fpic -c AddNumbers.cpp -o AddNumbers.o

The *-fpic* option tells g++ to create position independent code which is needed for shared libraries.

- g++ -shared -o libAddNumbers.dll AddNumbers.o

the library name must start with the three letters lib and have the suffix .so/.dll.

Linking Library Files

- To link static library with main program

g++ -L. -static main.cpp -l AddNumbers -o AddNumbersClient_static

g++ flags:

-L: Indicates the location of the library you want to reference (. = current directory).

-l: Specifies the specific library you want to attach.

The first three letters lib as well as the suffix .lib are not specified for the name of the library.

- To link against the shared library,

g++ -L . -l AddNumbers -o AddNumbersClient_shared

The first three letters lib as well as the suffix .so/.dll are not specified for the name of the library

Specifying Library Path using g++

- To specify a directory to search for (binary) libraries, you just use

-L: -L /data[...]/lib

- To specify the actual library name, you use

-l: -l foo # (links libfoo.a or libfoo.so)

- To specify a directory to search for include files (different from libraries!) you use

-I: -I /data[...]/lib

THE MAKEFILE

- makefile handles the details of compilation
- A makefile typically consists of many entries. Each entry has:
  - a target (usually a file)
  - the dependencies (files which the target depends on)
  - commands to run, based on the target and dependencies.
- SYNTAX:

<target>: [<dependency >]*

[<TAB><command><endl>]+

- Example:

Movie.o: Movie.cpp Movie.h Vector.h

g++ -Wall -c Movie.cpp

```
[File: calcl/makefile.gnu]
#
# Makefile for the Free Software Foundations g++
compiler for the calculator program named calc.cpp
#
CC=g++
CFLAGS=-g -Wall
all: calc
calc: calc.cpp
$(CC) $(CFLAGS) -o calc calc.cpp
clean:
rm calc
```

Executing makefile

- By default, when make looks for the makefile, it tries the following names, in order: GNUmakefile, makefile and Makefile.
- make -f altmake

tells make to use the file altmake as the makefile.