CS213/293 Data Structure and Algorithms 2025

Lecture 2: Advanced Features in C++

Instructor: Ashutosh Gupta

IITB India

Compile date: 2025-07-31

Topic 2.1

OS, Compiler, Makefile, Debugger for CS293

OS

We use Linux.

We strongly recommend for you to install linux in your personal machine.

Very hard for us to support the other operating systems.

Compiler

We will be using g++ compiler.

While we compile our code, we may give the following options.

- `-g` : Insert debug information in the binary
- `-Wall -Wextra` : Enables warnings
- `-std=c++20` : Use c++20 version of C++

Exercise 2.1

What do the following options do?

```
-Wshadow -Wpedantic -Werror -fsanitize=address,undefined
```

Makefile in labs

We will be providing Makefile along with the lab problems.

A Makefile consists of build rules.

```
<File To Be Built>: <File Dependencies>
      <Action For Building The File>
```

Example 2.1

The following rule builds a.out and depends on file hello.cpp.

```
a.out : hello.cpp
    g++ hello.cpp
```

Example: a detailed Makefile

```
# Compiler
CXX = g++

# Compiler flags
CXXFLAGS= -g -Wall -std=c++20

# Source files
SOURCES = linear.cpp main.cpp

# Object files
OBJECTS = $(SOURCES:.cpp=.o)

# Executable
EXEC = linear
```

```
# Default target
all: build runtests

# Build target
build: $(EXEC)

# Link object files to create the executable
$(EXEC): $(OBJECTS)
   $(CXX) $(CXXFLAGS) -o $(EXEC) $(OBJECTS)

# Compile source files to object files
%.o: %.cpp *.h
   $(CXX) $(CXXFLAGS) -c $< -o $@

runtests: .....
```

The Makefile in the labs may not be identical but they will be roughly similar.

Runnning make

Please download demo.tar from the wesbite and untar the file.

```
$ tar -xvf demo.tar
$ cd demo/linear
```

Try the following commands.

```
$ make clean          // Deletes the compiler generated files
$ make main.o         // Calls compiler to generate main.o
$ make linear         // Runs all the rules to generate linear
$ make build          // Does the same thing as linear
$ make                // Runs all rule
```

make does not run the action if the target was built after the modifications in the dependencies.

VSCode and debugger

We are giving configuration files such that you can run the VSCode debugger for your code.

To Run VSCode on the problem folder go to the problem folder and launch VSCode as follows.

```
$ cd demo/linear
$ code .
```

Follow the instructions in `./demo/linear/README.md` to launch the debugger.

Please get used to the programming environment.

Topic 2.2

Containers

What are containers?

A collection of C++ objects

- `int a[10]; //Array`
- `vector<int> b;`

Exercise 2.2
Why the use of the word ‘containers’?

More container examples

- `array`
- `vector<T>`
- `set<T>`
- `map<T,T>`
- `unordered_set<T>`
- `unordered_map<T,T>`

In math, sets are unordered?

Set in C++ $\neq$ Mathematical set

Why do we need containers?

Collections are everywhere

- CPUs in a machine
- Incoming service requests
- Food items on a menu
- Shopping cart on a shopping website

Not all collections are the same

Example: using a container

Source: http://www.cplusplus.com/reference/set

```
#include <iostream>
#include <set>
int main () {
  std::set<int> s;
  for(int i=5; i>=1; i--)   // s: {50,40,30,20,10}
    s.insert(i*10);
  s.insert(20);     // no new element inserted
  s.erase(20);     // s: {50,40,30,10}

  if( s.contains(40) )
    std::cout << "s has 40!\n";

  for( int i : s )  // printing elements of a container
    std::cout << i << '\n';
  return 0;
}
```

Why do we need many kinds of containers?

- Expected properties and usage patterns define the container

  For example,

  - Unique elements in the collection
  - Arrival/pre-defined order among elements
  - Random access vs. sequential access
  - Only few additions(small collection) and many membership checks
  - Many additions (large collection) and a few sporadic deletes

Different containers are

efficient to use/run

in varied usage patterns

Choose a container

Exercise 2.3

Which container should we use for the following collections?

- CPUs in a machine
- Incoming service requests
- Food items on a menu
- Shopping cart on a shopping website

Some examples of containers

`set<T>`

- Unique element
- insert/erase/contains interface
- collection has implicit ordering among elements

`map<T,T>`

- Unique key-value pairs
- insert/erase interface
- collection has implicit ordering among keys
- Finding a key-value pair is not the same as accessing it
- Throws an exception if accessed using a non-existent key

Containers are abstract data types

The containers do not provide details on the implementation. They provide an interface with guarantees.

In computer science, we call the libraries abstract data types. The guarantees are called axioms of abstract data type.

Commentary: Defining the axioms is not a simple matter. We need to answer the following questions.

Why do we need exactly these five axioms?
Are these sufficient?
Are any of them redundant, i.e., implied by others?
Do they contradict each other?

These kind of questions will be answered in CS228.

Example 2.2

Axioms of abstract data type set.

- `std::set<int> s; s.contains(v) == false`
- `s.insert(v); s.contains(v) == true`
- `x = s.contains(u); s.insert(v); s.contains(u) == x`, where $u! = v$.
- `s.erase(v); s.contains(v) == false`
- `x = s.contains(u); s.erase(v); s.contains(u) == x`, where $u! = v$.

Example: map<T,T>

Source: http://www.cplusplus.com/reference/map

```
#include <iostream>
#include <string>
#include <map>
int main () {
  std::map<std::string,int> cart;
  //Set some initial values:
  cart["soap"] = 2;
  cart["salt"] = 1;
  cart.insert( std::make_pair( "pen", 10 ) );
  cart.erase("salt");
  //access elements
  std::cout << "Soap: " << cart["soap"] << "\n";
  std::cout << "Hat: " << cart["hat"] << "\n";
  std::cout << "Hat: " << cart.at("hat") << "\n";
}
```

Commentary: When we run `cart["hat"]`, C++ modifies the content of cart and maps "hat" to 0 (default value of int). Therefore, the run `cart.at("hat")` succeeds without exception. If we delete the second last statement containing `cart["hat"]` in the program, the last statement will throw an exception. It is a strange situation, where mere reading a data structure is modifying it and changing the behavior of the data structure.

Exercise 2.4 What will happen at the last two calls?

Exceptions in Containers

If containers cannot return an appropriate value, they throw exceptions.

Callers must be ready to catch the exceptions and respond accordingly.

Example 2.3

Read operation `cart.at("shoe")` throws an exception if the cart does not value for key `"shoe"`.

STL: container libraries with unified interfaces

Since the containers are similar

`http://www.cplusplus.com/reference`

C++ in flux

Once C++ was set in stone. Now, modern languages have made a dent!

Major revisions in history!!

- c++98
- c++11
- c++17
- c++20 (we will use this compiler!)

Topic 2.3

Exceptions

What to do if an unexpected event occurs?

Example 2.4

Often our programs face unexpected events.

- Divide by zero
- Open a non-existent file
- Network device is failed

A solution: Stop the program and throw an exception!

Exceptions: something unexpected happened!

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

int foo(int x) {
  try
  {
    throw 20; // something has gone wrong!!
  }
  catch (int e) // type of e must match the type of thrown value!
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

Exceptions: catch matches the types!

```
int foo(int x) {
  try{
    if( x >  0 ){
      throw 20; // something has gone wrong!!
    }else{
      throw "C'est la vie!"; // Another thing has gone wrong!
    }
  }
  catch (int e){ // type of e is matched!
    cout << "An int exception occurred. " << e << '\n';
  }
  catch (string e){ // type of e is matched!
    cout << "A string exception occurred. " << e << '\n';
  }
  return 0;
}
```

Exceptions in the callee

```
int bar(){
  ...
  throw 20; // something has gone wrong!!
  ...
}

int foo(int x) {
  try{
    bar();
  }
  catch (int e){ // type of e is matched!
    cout << "An int exception occurred. " << e << '\n';
  }
}
```

Why write exceptions instead of handling the ”unexpected” cases?

To avoid cumbersome code!

If no catch is written, the exception flows to the top, and the program fails.

Exceptions provide a succinct mechanism to handle all possible errors, with a few catches.

Topic 2.4

Smart Pointers

Problem of memory leak

```
void memoryLeak() {
  int* ptr = new int(42); // Dynamically allocate an integer
  // Forgot to delete ptr -> Memory leak occurs!
}

int main() {
  while (true) {  // Infinite loop to simulate long-running process
    memoryLeak(); // Each call leaks memory
  }
  return 0;
}
```

Fixing memory leak

```
void memoryLeakFixed() {
  int* ptr = new int(42); // Dynamically allocate an integer

  delete ptr; // People tend to forget writing this
}

int main() {
  while (true) {
    memoryLeakFixed();
  }
  return 0;
}
```

Smart pointers

```
#include<memory>

void memorySmart() {
  std::shared_ptr<int> ptr = std::make_shared<int>(1);

  // Auto deletes the memory when there are zero references
}

int main() {
  while (true) {
    memorySmart();
  }
  return 0;
}
```

Reference counting

```
#include<memory>

void memorySmart() {
  std::shared_ptr<int> ptr = std::make_shared<int>(1);

  ptr = std::make_shared<int>(2); // Memory containing 1 is deleted here!
}
```

Reference counting across function calls

```
#include<memory>

std::shared_ptr<int> memorySmart() {
  std::shared_ptr<int> ptr = std::make_shared<int>(1);
  return ptr;
}

std::shared_ptr<int> memoryCaller() {
  auto p =  memorySmart();
  std::cout << p;
}
```

Exercise 2.5

a. What is the output of the above program?

b. How do you print the integer that is stored in the address?

Default initialization

```
#include<memory>

int main() {
  std::shared_ptr<int> p;
  int* s;
  std::cout << p << "\n";
  std::cout << s << "\n";
  return 0;
}
```

Exercise 2.6
What is the output of the above program?

Pointer Cycles are bad for smart pointer

```
class Node {
  Node(int value) : value(value) {}
  int value;
  std::shared_ptr<Node> nextNode;
};

void circular() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  node1->nextNode = node2;
  node2->nextNode = node1;
}

int main(){
  while (true) circular(); // Will cause memory leak
}
```

Cycles can be broken using weak pointer!

```
class Node {
  Node(int value) : value(value) {}
  int value;
  std::weak_ptr<Node> nextNode; // breaks the reference cycle
};

void circular() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  node1->nextNode = node2;
  node2->nextNode = node1;
}

int main(){
  while (true) circular(); // Will not cause memory leak
}
```

unique pointer

```
class Node {
  Node(int value) : value(value) {}
  int value;
  ....
};
int main(){
  unique_ptr<Node> node1(new Node(1));
  std::cout << node1->value;

  // No two pointers point at the same memory
  // will give compile time error
  unique_ptr<Node> node2 = node1;
  std::cout << node2->value;

  return 0;
}
```

Topic 2.5

Stack vs Vector

Why stack? Is stack faster than vector?

```
#include <iostream>
#include <vector>
#include <stack>
#include <chrono>
void test_vector(int n) {
  std::vector<int> v;
  for(int i = 0; i<n; ++i)
    v.push_back(i);
  while (!v.empty())
    v.pop_back();
}
void test_stack(int n) {
  std::stack<int> s;
  for(int i = 0; i < n;++i)
    s.push(i);
  while (!s.empty())
    s.pop();
}
```

```
using namespace std::chrono;
int test_timing() {
  int n = 10'000'000;

  auto start = high_resolution_clock::now();
  test_vector_stack(n);
  auto end = high_resolution_clock::now();
  std::cout << "Vector push/pop time: "
    << duration<double>(end - start).count()
    << " s\n";
  start = high_resolution_clock::now();
  test_stack_stack(n);
  end = high_resolution_clock::now();
  std::cout << "Stack push/pop time: "
    << duration<double>(end - start).count()
    << " s\n";
    return 0;
}
```

Topic 2.6

Array vs. Vector

Vector

- Variable length
- Primarily stack-like access
- Allows random access
- Difficult to search
- Overhead of memory management

Array

- Fixed length
- Random access
- Difficult to search
- Low overhead

Let us create a test to compare the performances

```
#include <iostream>
#include <vector>
#include "rdtsc.h"
using namespace std; // unclear!! STOP ME!
int local_vector(size_t N) {
  vector<int> bigarray; //initially empty vector
  //Fill vector up to length N
  for(unsigned int k = 0; k<N; ++k)
    bigarray.push_back(k);
  //Find the max value in the vector
  int max = 0;
  for(unsigned int k = 0; k<N; ++k) {
    if( bigarray[k] > max )
      max = bigarray[k];
  }
  return max;
} // 3N memory operations
```

Let us create a test to compare the performance (2)

```
// call local_vector M times
int test_local_vector( size_t M, size_t N ) {
  unsigned sum = 0;
  for(unsigned int j = 0; j < M; ++j ) {
    sum = sum + local_vector( N );
  }
  return sum;
}
//In total, 3MN memory operations
```

Let us create a test to compare the performance (3)

```
// assumes the 64-bit machine
int main() {
  ClockCounter t; // counts elapsed cycles
  size_t MN = 4*32*32*32*32*16;
  size_t N = 4;
  while( N <= MN  ) {
    t.start();
    test_local_vector( MN/N , N );
    double diff = t.stop();
    //print average time for 3 memory operations
    std::cout << "N = " << N << " : "<< (diff/MN);
    N = N*32;
  }
}
```

Exercise 2.7

Write the same test for arrays.

Topic 2.7

Tutorial Problems

Exercise: What is the difference between at and ..[..] accesses?

Exercise 2.8

What is the difference between “at” and “..[..]” accesses in C++ maps?

Exercise: smart pointers

Exercise 2.9

C++ does not provide active memory management. However, smart pointers in C++ allow us the capability of a garbage collector. The smart pointer classes in C++ are

- `shared_ptr`
- `unique_ptr`
- `weak_ptr`
- `auto_ptr`

Write programs that illustrate the differences among the above smart pointers.

Exercise: const

Exercise 2.10

Why do the following four writes cause compilation errors in the C++20 compiler?

```
class Node {
public:
  Node() : value(0) { }
  const Node& foo( const Node* const x) const {
    value = 3;          // Not allowed because ______________
    x[0].value = 4;     // Not allowed because ______________
    x = this;           // Not allowed because ______________
    return x[0];
  }
  int value;
};
int main() {
  Node x[3], y;
  auto& z = y.foo(x);
  z.value = 5; // Not allowed because ______________
}
```

Topic 2.8

Problems

True or False

Exercise 2.11

Mark the following statements True / False and also provide justification.

1. A unique pointer can be used to provide a non-owning reference to an object that is managed by a shared pointer.
2. In C++, if we refer to objects using only `shared_ptr`, there is no possibility of memory leak.
3. Code `A& x = new A();` will give compilation error. Assume class A is defined and has constructor A().
4. For `std::map<T,U> m` in C++, `m.at(x)` cannot throw exception for any x.

Rust ownership

Exercise 2.12

What is the difference between C++ smart pointers and the ownership model of Rust?

Exercise: named requirements

Exercise 2.13

Some of the containers have named requirements in their description. For example, "std::vector (for T other than bool) meets the requirements of Container, AllocatorAwareContainer (since C++11), SequenceContainer, ContiguousContainer (since C++17), and ReversibleContainer.".

What are these? Can you describe the meaning of these? How are these conditions checked?

Exercise: auto in exception (2024 student suggestion!)

Exercise 2.14

Can we write auto within the catch parameter?

```
int foo(int x) {
  try{
    throw 20; // something has gone wrong!!
  }
  catch (auto e){ // type of e is matched!
    cout << "An int exception occurred. " << e << '\n';
  }
  return 0;
}
```

Topic 2.9

Extra slides: weak pointers

An illustrative example of weak pointer usage (continued)

```cpp
#include <iostream>
#include <memory>
class Node {
public:
  Node(int value) : value(value) {std::cout << "Node " << value << " created." << std::endl; }
  // Functions to set/get the next node/weak ref to previous node/shared ref to previous node
  void setNext    ( std::shared_ptr<Node> next ) { nextNode = next;     }
  void setWeakPrev( std::shared_ptr<Node> next ) { prevWeakNode = next; }
  void setPrev    ( std::shared_ptr<Node> next ) { prevNode = next;     }
  std::shared_ptr<Node> getNext()     const      { return nextNode;            }
  std::shared_ptr<Node> getPrev()     const      { return prevNode;            }
  std::shared_ptr<Node> getWeakPrev() const      { return prevWeakNode.lock(); }
  // Function to display the value of the node
  void display() const { std::cout << "Node value: " << value << std::endl; }
private:
  int value;
  std::shared_ptr<Node> nextNode;
  std::shared_ptr<Node> prevNode;
  std::weak_ptr<Node>   prevWeakNode;
};
void print_list( std::weak_ptr<Node> current ) {
  for (int i = 0; i < 5; ++i) {
    auto current_ref = current.lock();
    if (current_ref) {
      current_ref->display();
      current = current_ref->getNext();
    } else {
      std::cout << "Next node is nullptr." << std::endl; break;
    }
  }
}
```

An example of weak pointer usage (2)

```
// Creating a doubly linked list via shared_ptr/weak_ptr
std::weak_ptr<Node> shared_test() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  auto node3 = std::make_shared<Node>(3);
  // Create a circular reference
  node1->setNext(node2);
  node2->setNext(node3);
  node2->setPrev(node1); // shared pointer pointing to previous node is causing a reference cycle
  node3->setPrev(node2);
  return node1;
}
std::weak_ptr<Node> weak_test() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  auto node3 = std::make_shared<Node>(3);
  node1->setNext(node2);
  node2->setNext(node3);
  node2->setWeakPrev(node1); // weak pointer pointing to previous node breaks cyclic reference counting
  node3->setWeakPrev(node2);
  return node1;
}
int main() {
  std::cout << "Testing shared pointer:" << std::endl;
  auto current = shared_test();
  print_list(current);
  std::cout << "Testing weak pointer:" << std::endl;
  current = weak_test();
  print_list(current);
  return 0;
}
```

Topic 2.10

Measuring time

What is efficient?

Use less resources.

Programs consume memory, running time, and the time of programmers.

We should be able to measure the running time of programs.

Measuring the runtime!

```
#include <iostream>
#include <vector>
#include "rdtsc.h"          // Declares ClockCounter class

int main() {
  ClockCounter time;        // Counts number of CPU cycles
  time.start();             // Start the counter

  std::vector<int> v;       // Do some task

  auto t = time.stop();     // Measure
  std::cout << t << "\n";// Report
}
// Open terminal
// Go to inside folder ./measure-time
// Run: $g++ measure.cpp
//      $./a.out
```

Repeated Measuring to check the variations in the measure

```
#define REPEAT 10000000
int main() {
  ClockCounter time;
  for (unsigned i=0; i < REPEAT; i++ ) {
    time.start();

    std::vector<int> v;

    auto t = time.stop();
    std::cout << i << " " << t << "\n";
  }
}

// Let us use Makefile located in the folder,
// which can compile and run using one command
// Run:  $make measure
//       $make plot
```

Take average or repeated experiments to reduce noise

```
int main() {
  unsigned long long t = 0;
  ClockCounter time;
  for (unsigned i=0; i < REPEAT; i++ ) {
    time.start();

    std::vector<int> v;

    t += time.stop();
  }
  std::cout << (t*1.0)/REPEAT << "\n";
}

// Run:  $make measure
//       $make show
```

Average time to insert an element in a vector!

```
int main() {
  unsigned long long t = 0;
  ClockCounter time;
  for (unsigned i=0; i < REPEAT; i++ ) {
    std::vector<int> v;

    time.start();

    v.push_back(0);

    t += time.stop();
  }
  std::cout << (t*1.0)/REPEAT << "\n";
}

// Run:  $make measure
//       $make show
```

Average time for long running tasks

```
#define RUN_LENGTH 100
int main() {
  unsigned long long t = 0;
  ClockCounter time;
  for (unsigned i=0; i < REPEAT; i++ ) {
    std::vector<int> v;

    time.start();

    for(unsigned j=0;j<RUN_LENGTH;j++) v.push_back(j);

    t += time.stop();
  }
  std::cout << (t*1.0)/(REPEAT*RUN_LENGTH) << "\n";
}

// Run:   $make measure
//        $make show
```

Let us change the length of the tasks!

```
int main() {
  ClockCounter time;
  for (unsigned size=0; size < RUN_LENGTH; size++ ) {
    unsigned long long t = 0;
    for (unsigned i=0; i < REPEAT; i++ ) {
      std::vector<int> v;

      time.start();

      for(unsigned j=0;j<size;j++) v.push_back(j);

      t += time.stop();
    }
    std::cout << size <<" "<< (t*1.0)/(REPEAT*size)<<"\n";
  }
}
// Run:  $make measure
//       $make plot
```

Measuring the runtime while avoiding local disturbances

Try:

```
make measure-o0
make measure-o1
make measure-o2
make measure-o3
```

```
int main() {
  ClockCounter time;
  std::map<int,float> m;
  for(unsigned size = 1; size < RUN_LENGTH; size++) m[size]=0.0;
  for(unsigned l = 0; l < REPEAT; l++) {              // Global average
    for(unsigned size = 1; size<RUN_LENGTH; size++) {
      for( unsigned i = 0; i < REPEAT; i++){          // Local average
        std::vector<int> v;
        time.start();
        for(int j=size; j>=1; j--) v.push_back(j); // TASK
        m[size] += time.stop();                       // Collect times
      }
    }
  }
  for(unsigned size = 1; size < RUN_LENGTH; size++)
    std::cout << size <<" "<< (m[size]*1.0)/(REPEAT*REPEAT*size)<<"\n";
  return 0;
}
```

Exercise 2.15

Modify the above program to compute the average insertion time of the kth element.

End of Lecture 2