C++ Reference
from the context of the course
CSE 232: Introduction to Programming II

Kaedon Cleland-Host

March 11, 2021

Contents

Chapter 1

Fundamentals

1.1 g++ Compiler

A good windows c++ compiler is available here.

1.1.1 Flags

-Wall

The **-Wall** flag stands for Warnings all and enables additional compiler warnings.

-std

The **-std** is used to set the version of the C++ standard.

-c

The **-c** flag will compile the source file into object files but will not link.

-o

The **-o** flag specifies the file name of the output.

-g

The **-g** flag will enable the debugging output.

1.1.2 Compiling and Linking

To compile a single **.cpp** file to **.exe** file:

```
1   g++ helloworld.cpp −Wall −std=c++17
```

Compiling

To compile a Sources file (**.cpp**) to an Object file (**.o**):

```
1   g++ helloworld.cpp −c −Wall −std=c++17
```

Linking

To link an Object file (**.o**) into an Executable (**.exe**):

```
1   g++ helloworld.o −Wall −std=c++17 −o output.exe
```

1.2 Header Files (.h)

Header files contain declarations of classes and functions.

```cpp
#pragma once // Ensures that this file is only included once

//Example declaration
long function1(long p1, long p2=2);

//Templates are implimented in the Header File
template <typename tmplt_type>
void template1 (tmplt_type &first, tmplt_type &second) {
  tmplt_type temp;
  temp = first;
  first = second;
  second = temp;
}
```

1.3 Source Files (.cpp or .hpp)

Sources files contain the definitions of classes and functions.

```cpp
#include <iostream> // STD header

#include "library.h" // Inclusion of a header file

int main(){
  std::cout << "Hello World" << std::endl;
}
```
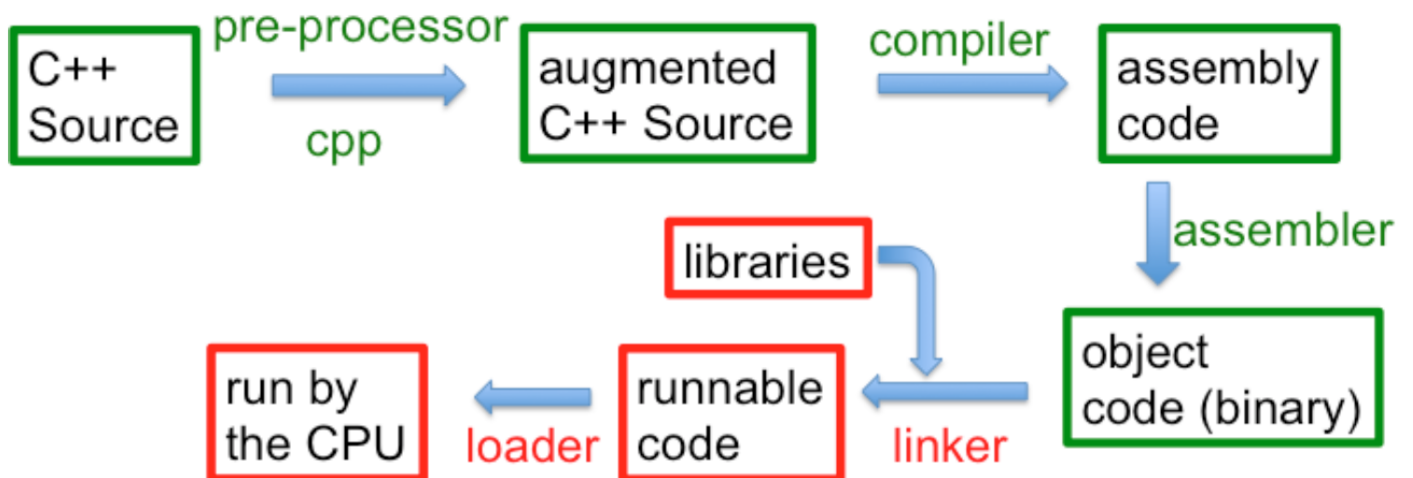
1.4 Object Files (.o)

Object files are assembly code that cannot be run on its own. Objects files must be linked with other object files and libraries to produce an executable.

1.5 Executable (.exe)

Executable is the final machine code that can be run.

1.6 Multiple File Compilation

Chapter 2

Types

2.1 Casting

Static cast requests the explicit converse between types.

```
1    static_cast <int >(" 1")
```

2.2 Integer (int)

An integer stores a positive or negative number with no decimal places.

```
1    int  variable = 5;
```

Fixed Width Integers

When a larger integer is needed you can manually specify a larger integer with the following types. This is recommend over the use of longs.

std::int8_t	1 byte signed integer	-128 to 127
std::int16_t	2 byte signed integer	-32,768 to 32,767
std::int32_t	4 byte signed integer	-2,147,483,648 to 2,147,483,647
std::int64_t	8 byte signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

2.3 Double (double)

A double stores a floating point number in 8 bytes.

```
1    double  variable = 2.75;
```

2.4 Boolean (bool)

A bool stores a single bit.

```
1    bool  variable = true;
```

2.5 Float (float)

A float stores a floating pointer number in 4 bytes.

```
1    float  variable = 3.14;
```

2.6 Characters (char)

A char stores a single character as a single byte.

```
1    char  variable = 'a';
```

2.7 Strings (string)

A string is a sequence of zero or more chars. It requires including a header from STD.

```cpp
#include <string>

int main() {
    std::string variable = "hello";
}
```

String Functions

std::stoi()	String to int
std::stod()	String to double
std::stol()	String to long
std::stof()	String to float

String Iterators

.begin()	Return iterator to beginning
.end()	Return iterator to end
.rbegin()	Return reverse iterator to beginning
.rend()	Return reverse iterator to end

String Capacity

.size()	Return length of string (std::size_type)
.length()	Return length of string (std::size_type)
.clear()	Clear string
.empty()	Test if string is empty

String Modifiers

operator+=	Append to string
.append(str)	Append string to string
.push_back(c)	Append character to string
.insert(pos, str)	Insert into string right before pos
.erase(pos, len)	Erase characters from string
.replace(pos, len, str2)	Replace portion of string
.swap(str)	Swap string values
.pop_back()	Delete last character

String operations

.find(content)	Find content in string
.rfind(content)	Find last occurrence of content in string
.find_first_of(s)	Find character in string that matches any of s
.find_last_of(s)	Find character in string from the end that matches any of s
.find_first_not_of(s)	Find absence of character in string that matches any of s
.find_last_not_of(s)	Find non-matching character in string from the end that matches any of s
.substr(pos, len)	Generate substring
.compare()	Compare strings

2.8 Vectors (vector)

A string is a sequence of zero or more chars. It requires importing a library from STD.

```
#include <vector>

int main() {
    std::vector<int> variable(5); // Vector with initial size 5
    std::vector<int> variable(5, 1546); // Vector with initial size 5 and default value 1546
    std::vector<char> variable = {'h','5','b'}; // Vector with size 3
}
```

Vector Iterators

.begin()	Return iterator to beginning
.end()	Return iterator to end
.rbegin()	Return reverse iterator to beginning
.rend()	Return reverse iterator to end

Vector Capacity

.size()	Return length of vector (std::size_type)
.empty()	Test if vector is empty
.front()	Returns first element
.back()	Returns last element

Vector Modifiers

.push_back(e)	Append element to end of vector
.pop_back()	Delete last element
.insert(pos, val)	Insert into vector right before pos
.erase(pos)	Erase element from vector
.erase(first, last)	Erase range elements from vector
.swap(str2)	Swap vectors
.clear()	Removes all elements from the vector

Chapter 3

Declarations

3.1 Variable Declarations

3.1.1 Pointers

Pointers are variables that store the memory address of other variables. To create a pointer to a type add a * after the type.

```
1  int* pointer = &variable; // Creates a pointer to an int
```

Dereferencing pointers

To dereference a pointer add a * in front of the variable name. This will recall the value that the pointer is pointing at.

```
1  *pointer;
```

3.1.2 References

References are simply variables that point to the same place in memory as another variable. To create a reference simply add a & after the type.

```
1  int& pointer = variable; // Creates a reference to an int
```

Memory Address of a variable

To recall the memory address of a variable add a & in front of the variable name. This is used to assign to pointers.

```
1  &variable;
```

3.1.3 Constants (const)

The const modifier indicates that the value of a variable is not to be changed. This can include pointers so read variable declarations right to left.

```
1  int const variable; // const int
2  int const * const variable; // const pointer to a const int
3  int * const variable; // const pointer to an int
4  int const * variable; //  pointer to a const int
```

Chapter 4

Statements

4.1 If Else

The if statment will check a condition and if the condition is true it will run a block of code otherwise it will move on to the next block. If can be chained with elseif and else to add additional possibilities if the first condition is false.

```
1  if (condition1){
2      // Runs when condition1 is true
3  }else if (condition2){
4      // Run when condition1 is false but condition2 is true
5  }else{
6      // Runs when condition1 and condition2 are both false
7  }
```

4.2 Switch

The switch will run a code block if the expression matches a case.

```
1  switch(expression) {
2    case x:
3      // code block
4      break;
5    case y:
6      // code block
7      break;
8    default:
9      // code block
10 }
```

4.3 While and Do While

While will execute a code block until a condition is met checking for the condition before each execution. A do while is the same as a while loop but it will check for the condition after executing the code block.

```
1  while (condition) {
2    // code block to be executed
3  }
4  do{
5    // code block to be executed
6  }while (condition)
```

4.4 Continue

Continue will skip the rest of the current iteration of the loop and then continue looping.

4.5 Break

Break will skip the rest of the current iteration of the loop and exit the loop.

Chapter 5

STD Features

5.1 Algorithms (algorithm)

The algorithms library from the std includes a large number of useful templates for sorting or manipulating data. Most of the algorithms listed here are from the algorithm header.

```
1  #include <algorithm>
```

The accumulate function is from the numeric header.

```
1  #include <numeric>
```

Non-Modifying Sequence

std::**accumulate**(begin,end,init,add)	Adds all the elements up
std::**all_of**(begin,end,cond)	tests if all elements are true
std::**any_of**(begin,end,cond)	tests if any elements are true
std::**none_of**(begin,end,cond)	tests if all elements are false
std::**for_each**(begin,end,func)	calls func for each element
std::**find**(begin,end,value)	returns iterator to first match or end
std::**find_if**(begin,end,cond)	returns iterator to first true element or end
std::**find_if_not**(begin,end,cond)	returns iterator to first false element or end
std::**search**(begin,end,begin2,end2,equiv)	returns iterator to first element of the first occurence of the second sequence or end
std::**search_n**(begin,end,n,value)	returns iterator to first element of the first instance of n elements matching value
std::**find_end**(begin,end,begin2,end2,equiv)	returns iterator to first element of the last occurrence of the second sequence or end
std::**find_first_of**(begin,end,begin2,end2,equiv)	returns iterator to first element that matches any element in the second sequence
std::**adjacent_find**(begin,end,equiv)	returns iterator to first element of the first pair of matching elements
std::**count**(begin,end,value)	returns number of elements equal to value
std::**count_if**(begin,end,cond)	returns number of true elements
std::**mismatch**(begin,end,begin2,equiv)	returns iterator to first mismatch
std::**equal**(begin,end,begin2,equiv)	tests if two sequences are equal

Modifying Sequence

std::**copy**(begin,end,result)	copies elements to another sequence
std::**copy_n**(begin,end,n,result)	copies first n elements to another sequence
std::**copy_if**(begin,end,result,cond)	copies true elements to another sequence
std::**move**(begin,end,result)	moves elements to another sequence
std::**swap**(a,b)	swaps the values of a and b
std::**swap_ranges**(begin,end,begin2)	swaps the values of two sequences
std::**iter_swap**(a,b)	swaps the values pointed at by a and b
std::**transform**(begin,end,result,func)	constructs output of func on each element of the sequence
std::**transform**(begin,end,begin2,result,func)	constructs output of func on each element of two sequences
std::**replace**(begin,end,find,replace)	replaces each instance of find with replace
std::**replace_if**(begin,end,cond,replace)	replaces each true element with replace
std::**replace_copy**(begin,end,result,find,replace)	copies then replaces each instance of find with replace
std::**replace_copy_if**(begin,end,result,cond,replace)	copies then replaces each true element with replace
std::**fill**(begin,end,value)	assigns value to all elements in the sequence
std::**fill_n**(begin,end,n,value)	assigns value to n elements in the sequence
std::**generate**(begin,end,func)	assigns value returned by func to all elements in the sequence
std::**generate_n**(begin,end,func)	assigns value returned by func to n elements in the sequence
std::**remove**(begin,end,value)	removes instances of value from the sequence
std::**remove_if**(begin,end,cond)	removes true elements from the sequence
std::**remove_copy**(begin,end,result,value)	copies sequence with instances of value removed
std::**remove_copy_if**(begin,end,result,cond)	copies sequence with true elements removed
std::**unique**(begin,end,equiv)	removes all but the first element from every consecutive group of equivalent elements
std::**unique_copy**(begin,end,result,equiv)	copies sequence with all but the first element from every consecutive group of equivalent elements removed
std::**reverse**(begin,end)	reverses sequence
std::**reverse_copy**(begin,end,result)	copies reversed sequence
std::**rotate**(begin,end,middle)	rolls elements such that middle becomes the first element
std::**rotate_copy**(begin,end,middle,result)	copies rolled sequence such that middle is the first element
std::**random_shuffle**(begin,end)	randomly shuffles sequence

Sorting

std::**sort**(begin,end,lessthan)	sorts elements in range in ascending order
std::**stable_sort**(begin,end,lessthan)	sort but preserves order of equivalent elements
std::**is_sorted**(begin,end,lessthan)	tests if sequence is sorted

Sorted

std::**merge**(begin,end,begin2,end2,result,lessthan)	combines the elements of two sorted sequences
std::**includes**(begin,end,begin2,end2,lessthan)	tests if the sorted sequence contains all the elements of the second sorted sequence
std::**set_union**(begin,end,begin2,end2,result,lessthan)	constructs the union of the two sorted sequences
std::**set_difference**(begin,end,begin2,end2,result,lessthan)	constructs the set of elements in the first sorted sequence that are not in the second sorted sequence
std::**set_difference**(begin,end,begin2,end2,result,lessthan)	constructs the set of elements in one sorted sequence but not the other

Min/Max

std::**min**(a,b,lessthan)	returns the min of a and b
std::**max**(a,b,lessthan)	returns the max of a and b
std::**minmax**(a,b,lessthan)	returns the a pair of a and b with the min first
std::**min_element**(begin,end,lessthan)	returns a interator to the min element
std::**max_element**(begin,end,lessthan)	returns a interator to the max element
std::**minmax_element**(begin,end,lessthan)	returns a pair of interators to the min and max elements

5.2 Math (cmath)

```
1  #include <cmath>
2  std::sin(x) // Computes sin
3  std::asin(x) // Computes arc sin
4  std::atan2(y,x) // Computes arc tan and accounts for the direction
5  std::pow(x,y) // Computes x to the power of y
6  std::exp(x) // Computes e to the power of x
7  std::log(x) // Computes the natural log of x
8  std::abs(x) // Computes absolute value
9  std::ceil(x) // Round up
10 std::floor(x) // Round down
11 std::round(x) // Round to nearest integer
```

5.3 Streams

Streams are buffers of memory that temporary stores information before it is passed to a program or device. Streams are either input streams or output streams.

Status Functions

.**good**()	returns true if the stream is not in an error state
.**bad**()	returns true if the stream is in an error state
.**fail**()	returns true if the previous operation failed
.**eof**()	returns true if the stream reached the end of file token
.**clear**()	clear error state flags

Format Codes

std::**boolalpha**	use alphanumerical bool values
std::**noboolalpha**	don't use alphanumerical bool values
std::**skipws**	skip white space
std::**noskipws**	don't skip white space
std::**dec**	use decimal base
std::**hex**	use hexadecimal base
std::**dex**	use octal base
std::**uppercase**	generate upper case letters
std::**nouppercase**	don't generate upper case letters
std::**fixed**	use fixed floating-point notation
std::**scientific**	use scientific floating-point notation
std::**left**	use left justification
std::**right**	use right justification

Input Streams

An input stream take data from a device or its definition and through the extraction operator can be read from. If reading from a buffer is not successful the stream will go into a fail state.

Input Functions

.**peek**()	read next character without removing
.**get**(c)	returns a single element from the buffer
.**getline**(var)	read from buffer until '\n'
.**ignore**(n, delim)	ignore the next n character or until delim
.**putback**(c)	put c back in the stream
.**peek**()	read next character without removing

Output Streams

An output stream allows data to be written to another program or devices using the insertion operator. **Output Functions**

std::**endl**	add '\n' and flush buffer
std::**flush**	flush buffer
.**put**(c)	sends a single element to the buffer
.**getline**(var)	read from buffer until '\n'
.**ignore**(n, delim)	ignore the next n character or until delim
.**putback**(c)	put c back in the stream
.**peek**()	read next character without removing

5.3.1 IO Streams (iostream)

IO streams are how c++ communicates with the console. cin is console input which allows for functions to take input from the console. cout is console output which allows for printing to the console. String stream and file streams are also streams so they inherit all the functions used in this section.

```cpp
#include <iostream>
std::cout << "Text or Variable"; // Writes to console
std::cout << std::setprecision(5); // Sets decimal precision
std::cout << "Text or Variable" << std::endl; // Writes to console and goes to the next line.
int variable;
std::cin >> variable; // Reads from console and converts to integer.
std::cin >> std::noskipws; // Sets cin to not skip whitespace
while (!cin.eof()) { // Read until end of file
    std::string current_line;
    std::getline(cin, current_line); // Read one line of input from console
}
```

5.3.2 String Streams (sstream)

String streams are stream generated from or to strings. They allow for more elegant string conversion to variable types. There are two types of streams input and output. Input streams are created from a string and can be read from into variables. Output streams can be written to from variables and then converted back to strings.

```cpp
#include<iostream>
#include<string>
#include<sstream>

int main(){
    std::string input_str = "Homer 36";
    std::string name;
    long age;

    // std::istringstream iss(input_str);
    std::istringstream iss;
    iss.str(input_str);
    iss >> name;
    iss >> age;

    std::ostringstream oss;
    oss << name << " is " << age << std::endl;
    std::cout << oss.str() << std::endl;
    oss.str("");
}
```

5.3.3 File Streams (fstream)

File streams are streams generated from or pointing to files in storage.

```cpp
#include <iostream>
#include <fstream>
#include <string>

void ofstreamExample() {
    std::ofstream myfile;
    myfile.open("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    return 0;
}

void ifstreamExample() {
    std::string line;
    std::ifstream myfile("example.txt");
    if (myfile.is_open())
    {
        while ( getline (myfile,line) )
        {
            cout << line << '\n';
        }
        myfile.close();
    }
    else cout << "Unable to open file";
}
```

Chapter 6

Exceptions

6.1 Try Catch Block

The try catch block with run a block of code until an exception is thrown. When an exception is thrown in the try block the corresponding catch block is called and passed the exception object.

```cpp
#include<stdexcept>
try {
    C = my_str.at(indx);                     // thows out_of_range
    char_long = stol( string(1,C) );         // throws invalid_argument
    if (char_long == 0)
      throw runtime_error("division by zero");
    cout << my_str.size() / char_long << endl;
  } catch (out_of_range& e) {
    cout << "In the out of range catcher" << endl;
    cout << e.what() << endl;
  } catch (invalid_argument& e) {
    cout << "in the invalid_arg catcher" << endl;
    cout << e.what() << endl;
  }
```

6.2 Throw Expression

The throw keyword will throw an exception. This can be used to indicate when a problem has occurred.

```cpp
#include<stdexcept>
throw std::runtime_error("division by zero");
```

6.3 List of Exceptions

```cpp
#include<stdexcept>
std::runtime_error; // General error
std::out_of_range; // Out of an index range
std::invalid_argument; // Bad value
```

Chapter 7

Iterators and Lambdas

7.1 Iterators

Iterators are pointers to elements in an iteratable type. The ++ operator can be used to iterate to the next iterator. All iteratable types have a .begin() and .end() which return iterators at the beginning and end + 1. You can dereference an iterator with *.

```cpp
for (auto pos=v.begin(), end=v.end(); pos != end; ++pos){
    auto element = *pos;
    element += 1;
  }
```

Back inserter is an extension of iterators that allows for elements to be added to a vector or strings beyond there current length.

```cpp
#include<iterator>
vector<int>v;
back_inserter(v);
```

Forward

Forward iterators are limited to one direction in which to iterate through a range (forward). All standard containers support at least forward iterator types.

Bidirectional

Bidirectional iterators are like forward iterators but can also be iterated through backwards. Such as with −−.

Random Access

Random-access iterators implement all the functionality of bidirectional iterators, and also have the ability to access ranges non-sequentially: distant elements can be accessed directly by applying an offset value to an iterator without iterating through all the elements in between. These iterators have a similar functionality to standard pointers (pointers are iterators of this category).

7.2 Lambdas

Lambdas are single use declarations of functions for use when passing functions. The brackets indicate any variables from the current scope to capture into the function implementation and the parentheses indicates the signature of the function.

```cpp
std::sort(v.begin(), v.end(),
  [&variable] (const auto& a, const auto& b){
      return (a+b>variable);
    });
```