C++ Reference

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

Kaedon Cleland-Host

April 25, 2021

Contents

1 Fundamentals **3**
- 1.1 g++ Compiler . . . 3
  - 1.1.1 Flags . . . 3
  - 1.1.2 Compiling and Linking . . . 3
- 1.2 Header Files (.h) . . . 4
- 1.3 Source Files (.cpp or .hpp) . . . 4
- 1.4 Object Files (.o) . . . 4
- 1.5 Executable (.exe) . . . 4
- 1.6 Multiple File Compilation . . . 4

2 Types **5**
- 2.1 Casting . . . 5
- 2.2 Integer (int) . . . 5
- 2.3 Double (double) . . . 5
- 2.4 Boolean (bool) . . . 5
- 2.5 Float (float) . . . 5
- 2.6 Characters (char) . . . 5
- 2.7 Strings (string) . . . 6
- 2.8 Vectors (vector) . . . 7
- 2.9 Pairs (pair) . . . 7
- 2.10 Maps (map) . . . 8
  - 2.10.1 Range Based Loop on Maps . . . 8
- 2.11 Sets (set) . . . 9
- 2.12 MultiMaps (multimap) and MultiSets (multiset) . . . 9
- 2.13 Unordered Map and Sets . . . 9
- 2.14 Arrays ([]) . . . 10
  - 2.14.1 Passing Arrays . . . 10

3 Declarations **11**
- 3.1 Variable Declarations . . . 11
  - 3.1.1 Pointers . . . 11
  - 3.1.2 References . . . 11
  - 3.1.3 Constants (const) . . . 11

4 Statements **12**
- 4.1 If Else . . . 12
- 4.2 Switch . . . 12
- 4.3 While and Do While . . . 12
- 4.4 Continue . . . 12
- 4.5 Break . . . 12

5 STD Features **13**
- 5.1 Algorithms (algorithm) . . . 13
- 5.2 Math (cmath) . . . 15
- 5.3 Streams . . . 15
  - 5.3.1 IO Streams (iostream) . . . 16
  - 5.3.2 String Streams (sstream) . . . 16
  - 5.3.3 File Streams (fstream) . . . 16

6 Exceptions **17**
6.1 Try Catch Block . . . 17
6.2 Throw Expression . . . 17
6.3 List of Exceptions . . . 17

7 Iterators and Lambdas **18**
7.1 Iterators . . . 18
7.2 Lambdas . . . 18

8 Dynamic Memory **19**
8.1 New . . . 19
8.2 Delete . . . 19
8.3 Dynamic Arrays . . . 19

9 Object Oriented Programming **20**
9.1 Classes and Structs . . . 20
9.1.1 Members . . . 20
9.1.2 Member Functions . . . 20
9.1.3 Constructors . . . 21
9.2 Encapsulation . . . 22
9.2.1 Friend Functions . . . 22
9.2.2 Overloading Operators . . . 23
9.3 Rule of Three . . . 24
9.3.1 Copy and Swap Idiom . . . 24
9.4 Stack Example . . . 24

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.

```
#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.

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

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

int main(){
  std::cout << "Hello World" << std::endl;
}
```

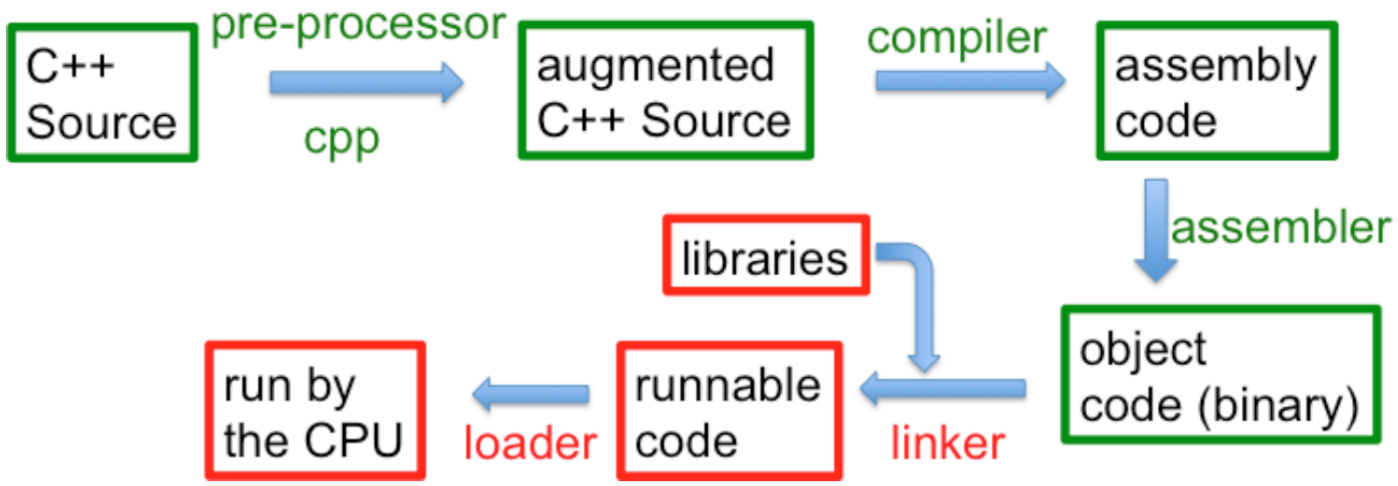
1.4 Object Files (.o)

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.

```
#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 vector is a list of elements of a certain type. A vector can expand with new data.

```
#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(vec2)	Swap vectors
.clear()	Removes all elements from the vector

2.9 Pairs (pair)

A pair is an class that store two instances of two possibly different types.

```
#include <utility>

int main() {
  std::pair<string,int> p1("hi", 1);
  std::pair<string,int> p2, p3;
  p2 = std::make_pair("x", 2);
  p3={"bye", 3};
}
```

.first	The first element
.second	The second element
std::**make_pair**(a,b)	creates a pair from two elements

2.10 Maps (map)

A map is a set of pairs that maps elements to keys. Each key can only have one element associated with it.

```
#include <map>

int main() {
  std::map<string, string> map1{ {"jill", "555-2323"},
                                 {"bill", "555-1212"} };
  map1["jill"] = "123-4567"; // asigns to the "jill" key
  map1["bob"] = "123-4567"; // asigns and creates the "bob" key
}
```

Map Iterators

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

Map Operations

.size()	Return size of the map (std::size_type)
.empty()	Test if the map is empty
.find(key)	Get iterator to pair with key
.count(key)	Return number of pairs with key

Map Modifiers

.insert(pair)	Adds a pair to the map
.erase(key)	Erase pair from the map
.swap(map2)	Swap maps
.clear()	Removes all elements from the map

2.10.1 Range Based Loop on Maps

A dynamic range based loop can be constructed for maps using the following code:

```
#include <map>

int main() {
  const std::map<std::string, int> name_to_age {
      {"Josh", 31},
      {"Janice", 25},
      {"Jacob", 15},
      {"Jenn", 20}
  };

  for (auto [name, age] : name_to_age) {
      std::cout << name << " is " << age << std::endl;
  }
  std::cout << std::endl;
}
```

2.11 Sets (set)

A set is an ordered list of unique elements.

```
#include <set>

int main() {
  set<string> set1{"brother", "father", "sister"};
  pair<set<string>::iterator,bool> set2;
  set1.insert("element");
}
```

Set Iterators

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

Set Operations

.size()	Return size of set (std::size_type)
.empty()	Test if set is empty
.find(val)	Get iterator to val
.count(val)	Return number of val in the set (0 or 1)

Set Modifiers

.insert(val)	Adds val from the set
.erase(val)	Erase pair from the set
.swap(set2)	Swap set
.clear()	Removes all elements from the set

2.12 MultiMaps (multimap) and MultiSets (multiset)

Multimaps are an alternate version of the map class that allows for multiple elements with the same key.

```
#include <map>

int main() {
  std::multimap<string, string> mutlimap1{ {"jill", "555-2323"},
                                           {"jill", "555-1212"} };
}
```

Multisets are an alternate version of the set class that does not require unique elements.

```
#include <set>

int main() {
  std::multiset<string, string> mutliset1{ "jill", "jill", "bill" };
}
```

2.13 Unordered Map and Sets

Unordered maps and sets are versions of set, multiset, map, and multimap such that the the elements are ordered by the hash of each key. This allows for much faster lookup of individual elements.

```
#include <set>

int main() {
  unordered_set<string> set1{"brother", "father", "sister"};
}
```

2.14 Arrays ([])

An array is a pointer to the first element in a list of elements. They can represent a list of any type.

```
const size_t size = 5;
int ary1[size]{8,5,6,7,4};
char ary2[]{'a', 'b', 'c', 'd'};  // fixed size, 4 chars
long ary3[size]{}; // 0 initialized, each element initialized!
long ary4[size];  // uninitialized, no work done.
ary1[0] // first value of ary1
ary1[1] // second value of ary1
```

Since arrays are pointers to the first element you can add numbers to the array to increment the address. This allows for the algorithms library to be applied to arrays.

2.14.1 Passing Arrays

When passing an array to another function, the size of the array is not explicitly stored in the array. Instead it can be inferred from the compiler or must be passed along in functions.

```
// pass as array of unknown size, basically a pointer
string to_string(int ary[], size_t size){
  cout << "sizeof array (empty []):"<< sizeof(ary) << endl;
  ostringstream oss;
  for (size_t  i=0; i<size; i++){
    oss << ary[i] <<", ";
    // oss << *(ary + i) <<", "; // same thing!
  }
  string result = oss.str();
  return result.substr(0,result.size()-2);
}

// pass as pointer directly, SAME THING as above
int sum (int *ary, size_t size){
  int sum=0;
  cout << "sizeof array (ptr):"<< sizeof(ary) << endl;
  for(size_t i=0; i<size; i++){
    sum += *(ary + i);
  }
  return sum;
}

// pass the bounds as pointers (size implicit)
int max_val (int *first, int *one_past_last){
  int result = 0;
  cout << "sizeof bounds: "<< sizeof(first) << endl;
  for(int *i= first; i<one_past_last; ++i){
    if ( *i > result)
      result = *i;
  }
  return result;
}

// pass as reference to an array, sizeof works here
long prod(const int (&ary)[5]){
  int result=1;
  cout << "sizeof array (ref, fixed):"<< sizeof(ary) << endl;
  // compiler knows size, can do range-based for
  for(auto &element : ary)
    result = result * element;
  return result;
}

// let template deduce size, we don't have to say!!
template<typename Type, size_t Size>
long squares(const Type (&ary)[Size]){
  Type result=0;
  cout << "sizeof array (template size):"<<sizeof(ary)<<endl;
  for(auto element : ary)
    result = result + (element * element);
  return result;
}
```

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.

```
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.

```
*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.

```
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.

```
&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.

```
int const variable; // const int
int const * const variable; // const pointer to a const int
int * const variable; // const pointer to an int
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.

```
if (condition1){
    // Runs when condition1 is true
}else if (condition2){
    // Run when condition1 is false but condition2 is true
}else{
    // Runs when condition1 and condition2 are both false
}
```

4.2 Switch

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

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

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.

```
while (condition) {
  // code block to be executed
}
do{
  // code block to be executed
}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_intersection**(begin,end,begin2,end2,result,lessthan)	constructs the intersection 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)

```
#include <cmath>
std::sin(x) // Computes sin
std::asin(x) // Computes arc sin
std::atan2(y,x) // Computes arc tan and accounts for the direction
std::pow(x,y) // Computes x to the power of y
std::exp(x) // Computes e to the power of x
std::log(x) // Computes the natural log of x
std::abs(x) // Computes absolute value
std::ceil(x) // Round up
std::floor(x) // Round down
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.

```
#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.

```
#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.

```
#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.

```
#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.

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

6.3 List of Exceptions

```
#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 *.

```
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.

```
#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.

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

Chapter 8

Dynamic Memory

Dynamic memory allows for memory to be allocated at run-time. This allows for objects to grow in size or for the amount of memory allocated to change when the program is run.

8.1 New

The new command allocates new memory from the os. It returns a pointer to the new memory.

```
long *lptr1, *lptr2;
MyClass *mcptr, *mcptr2;

lptr1 = new long;   // uninitialized
lptr2 = new long{1234567}; // initialized
mcptr = new MyClass; // default constructor
mcptr2 = new MyClass(123456, 123, "3param ctor");
```

8.2 Delete

The delete command releases the memory of a pointer back to the os. This is necessary to avoid a memory leak. A memory leak will not cause an error or prevent the program from running, but it will waste memory while could eventually cause slow downs and crashes.

```
delete lptr1;
delete lptr2;
delete mcptr;
delete mcptr2;
```

8.3 Dynamic Arrays

New and Delete can also be applied to arrays to created an array with a size defined at run-time. Note that the delete[] must be called for every copy of the array as well.

```
// dynamic array
size_t size;
cout << "How big to make the array:";
cin >> size;
// not an array type, only a pointer
// long *ary = new long[size];   // not initialized
int *ary = new int[size]{}; // initialize all!
fill(ary, size, 10);
dump(cout, ary, size);

cout << "1:"<<ary[0]<<endl;
cout << "n-1:"<<ary[size-1]<<endl;
cout << "Size:"<<sizeof(ary)<<endl; //pointer,not array type!

// if you make it, you must delete it. Note []
delete [] ary;
```

Chapter 9

Object Oriented Programming

Object oriented programming is programming that focuses on the creation of objects which contain data and methods to modify or act on the data.

9.1 Classes and Structs

Classes and structs allow you to make new types. The only difference between a class and a struct is classes make objects private by default and structs make objects public by default. Structs are often used for simple data types when all the members are public.

9.1.1 Members

Members are elements in the class or struct that store data. They are typically private and it is common notation to add an underscore to the end of variable names.

```
struct Clock{
  int minutes;
  int hours;
  string period;
};
```

9.1.2 Member Functions

```
struct Clock{
  int minutes;
  int hours;
  string period;

  void add_minutes(int);
};

string clk_to_string(const Clock &);
```

The implementation file:

```
#include "15.2-clock.h"

void Clock::add_minutes(int min){
  auto temp = minutes + min;
  if (temp >= 60) {
    minutes = temp % 60;
    hours = temp / 60 + hours;
  } else {
    minutes = temp;
  }
}

string clk_to_string(const Clock &c){
  ostringstream oss;
  oss << "Hours:"<<c.hours
      <<", Minutes:"<<c.minutes
      <<", Period:"<<c.period;
  return oss.str();
}
```

9.1.3 Constructors

The default constructor will initialize all the data members to their default value. If a constructor simply needs to initialize a few member values as copies of parameters, then you can use an inline definition. Constructors can also be created to convert between other types; this is called a converting constructor.

```
struct Clock{
  int minutes = 0;
  int hours = 0;
  string period;

  Clock()=default;
  Clock(int m, int h, string s) : minutes(m), hours(h), period(s) {};
  Clock(string s);
  // explicit Clock(string s);  // no implicit conversion

  void add_minutes(int);
};

string clk_to_string(const Clock &);
void split(const string &, vector<string> &, char);
```

The implementation file:

```
#include "16.2-clock.h"

// string constructor
Clock::Clock(string s){
  // format is hr:min:period
  vector<string> fields;
  split(s, fields, ':');
  hours = stol(fields[0]);
  minutes = stol(fields[1]);
  period = fields[2];
}

// add to minutes, correct hours if overflow
void Clock::add_minutes(int min){
  auto temp = minutes + min;
  if (temp >=60){
    minutes = temp % 60;
    hours = temp / 60 + hours;
  }
  else
    minutes = temp;
}

// convert clock to string
string clk_to_string(const Clock &c){
  ostringstream oss;
  oss << "Hours:"<<c.hours<<", Minutes:"
      <<c.minutes<<", Period:"<<c.period;
  return oss.str();
}

// split string based on sep, ref return of vector<string>
void split(const string &s, vector<string> &elems, char sep) {
    istringstream iss(s);
    string item;
    while(getline(iss, item, sep))
      elems.push_back(item);
}
```

9.2 Encapsulation

Encapsulation is the bundling of data and methods in classes to only make visible what is necessary. This reduces the complexity that the user of a class has to interact with and reduces the chance of bugs. In C++ this is primarily achieved by marking members private if they do not need to be accessed by the user.

9.2.1 Friend Functions

Friend functions are non-member functions that have access to the private members of a class.

```
class Stack{
private:
    char *ary_ = nullptr;
    int sz_ = 0;
    int top_ = -1;

public:
    // ...
    friend ostream& operator<< (ostream&, const Stack &);
    friend void swap (Stack&, Stack&);
};

void swap (Stack&, Stack&);
ostream& operator<< (ostream&, const Stack&);
```

The implementation file:

```
#include "stack.h"

void swap(Stack &s1, Stack &s2){
    // have to indicate that in here we are using the stl swap
    std::swap (s1.top_, s2.top_);
    std::swap (s1.sz_, s2.sz_);
    std::swap (s1.ary_, s2.ary_);
}

ostream& operator<<(ostream &out, const Stack &s){
    out << "(bottom) ";
    copy(s.ary_, s.ary_ + s.top_+1,
     ostream_iterator<char>(out, ","));
    out << " (top)";
    return out;
}
```

9.2.2 Overloading Operators

Operators are the functions called by many symbols in c++. You can define how these operators function for your class by overriding them. You can overload any of the following operators:

```
+ - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> && || ++ -- , ->* ->.
```

Expression	As member function	As non-member function	Example
@a	(a).operator@ ( )	operator@ (a)	`!std::cin` calls `std::cin.operator!()`
a@b	(a).operator@ (b)	operator@ (a, b)	`std::cout << 42` calls `std::cout.operator<<(42)`
a=b	(a).operator= (b)	cannot be non-member	Given `std::string s;`, `s = "abc";` calls `s.operator=("abc")`
a(b...)	(a).operator()(b...)	cannot be non-member	Given `std::random_device r;`, `auto n = r();` calls `r.operator()()`
a[b]	(a).operator[](b)	cannot be non-member	Given `std::map<int, int> m;`, `m[1] = 2;` calls `m.operator[](1)`
a->	(a).operator-> ( )	cannot be non-member	Given `std::unique_ptr<S> p;`, `p->bar()` calls `p.operator->()`
a@	(a).operator@ (0)	operator@ (a, 0)	Given `std::vector<int>::iterator i;`, `i++` calls `i.operator++(0)`
in this table, @ is a placeholder representing all matching operators: all prefix operators in @a, all postfix operators other than -> in a@, all infix operators other than = in a@b			

```
class Clock{
private:
    int minutes_ = 0;
    int hours_ = 0;
    string period_;

public:
    \\...

    // overloaded operator+ method
    Clock operator-(const Clock&);

    // friend operators
    friend Clock operator+(const Clock &, const Clock&);
    friend ostream &operator<<(ostream&, const Clock&);
};
```

The implementation file:

```
#include "17.4-clock.h"

// overloaded operator- as a member
Clock Clock::operator-(const Clock &c2){
    Clock new_c;
    new_c.minutes_ = minutes_ - c2.minutes_;
    new_c.hours_ = hours_ - c2.hours_;
    new_c.period_ = period_;
    new_c.adjust_clock(0,0,"");
    return new_c;
}

// overload operator+ as a friend
// declared a friend in the header
Clock operator+(const Clock &c1, const Clock &c2){
    Clock new_c;
    new_c.minutes_ = c1.minutes_ + c2.minutes_;
    new_c.hours_ = c1.hours_ + c2.hours_;
    new_c.period_ = c1.period_;
    new_c.adjust_clock(0, 0, c1.period_);
    return new_c;
}

// overload operator<< as a friend
// declared a friend in the header
ostream & operator<<(ostream &out, const Clock &c){
    out << "Hours:"<<c.hours_<<", Minutes:"
    <<c.minutes_<<", Period:"<<c.period_;
    return out;
}
```

9.3 Rule of Three

When dynamically allocating memory you will probably need to redefine three important functions.

- Copy constructor
  The copy constructor is a constructor that takes a const reference to another instance of the class. It should construct a copy of the parameter. By default this will apply a member to member copy.

```
1 Stack::Stack(const Stack &s);
```

- Assignment operator
  The asignment operator copies an instance of the class and then returns a reference to the class. By default this will apply a member to member copy.

```
1 Stack& Stack::operator=(const Stack &s);
```

- Destructor
  The destructor is responsible for calling delete on all dynamically allocated memory that the class is using.

```
1 Stack::~Stack();
```

If you need one then you need all three. You very rarely need to impliment these functions unless you are doing dynamic memory. Additionally, you can set the copy and assignment operator to delete which tells the compiler that these functions should not be called.

9.3.1 Copy and Swap Idiom

The copy swap idiom is the idea that you don't need to write the code for copying a class twice for the assignment operator. Instead create a parameter copy and then simply swap the copy with the current class.

```
1 Stack& Stack::operator=(Stack s){ // Copy created here
2     swap(*this, s); // Swap
3     return *this; // Destructor is called on s
4 }
```

9.4 Stack Example

stack.h :

```
#pragma once

#include<iostream>
using std::ostream;
#include<initializer_list>
using std::initializer_list;

class Stack{
private:
    char *ary_ = nullptr;
    int sz_ = 0;
    int top_ = -1;
    void grow();

public:
    Stack()=default;
    // Stack(size_t sz);
    Stack(initializer_list<char>);

    Stack(const Stack &s);   // copy
    ~Stack();                // destructor
    Stack& operator=(const Stack); // copy not ref

    char top();
    void pop();
    void push(char);
    bool empty();
    // bool full();
    void clear();
    friend ostream& operator<< (ostream&, const Stack &);
    friend void swap (Stack&, Stack&);
};

void swap (Stack&, Stack&);
ostream& operator<< (ostream&, const Stack&);
```

stack.cpp :

```
#include<iostream>
using std::cout; using std::endl;
#include<string>
using std::string;
#include<iterator>
using std::ostream_iterator;
#include<algorithm>
using std::copy;
#include<initializer_list>
using std::initializer_list;
#include<stdexcept>
using std::overflow_error; using std::underflow_error;

#include "stack.h"

void Stack::grow(){
  char *new_ary;

  if (sz_ == 0){
    new_ary = new char[1]{};
    sz_ = 1;
    // ary_ empty, just assign
    ary_ = new_ary;
  } else {
    // use {} to init to default
    new_ary = new char[sz_ * 2]{};
    copy(ary_, ary_+sz_, new_ary);
    sz_ *= 2;
    // stl swap, not Stack swap
    std::swap (new_ary, ary_);
    delete [] new_ary;
  }
}

```

```
35  void Stack::push(char element){
36    if (top_ >= (sz_ - 1) ){
37      // never throws, grows!
38      grow();
39      cout << "Grew to size:"<<sz_<<endl;
40    }
41    ary_[++top_] = element;
42  }
43
44  void swap(Stack &s1, Stack &s2){
45      std::swap (s1.top_, s2.top_);
46      std::swap (s1.sz_, s2.sz_);
47      std::swap (s1.ary_, s2.ary_);
48  }
49
50  // calls copy ctor for s (local var)
51  // swap means s points to old memory
52  // and dtor called on s (now old mem)
53  Stack& Stack::operator=(Stack s){
54      swap(*this, s);
55      return *this;
56  }
57
58  Stack::Stack(const Stack &s){
59      sz_ = s.sz_;
60      top_ = s.top_;
61      ary_= new char[s.sz_];
62      copy(s.ary_, s.ary_+s.sz_, ary_);
63  }
64
65  Stack::~Stack(){
66      delete [] ary_;
67  }
68
69  Stack::Stack(initializer_list<char> c){
70    sz_ = c.size();
71    ary_ = new char[sz_];
72    size_t indx = 0;
73    top_ = sz_ - 1;
74
75    for (auto e : c)
76      ary_[indx++] = e;
77  }
78
79  char Stack::top(){
80    if (top_ < 0)
81      throw underflow_error("top, empty stack");
82    return ary_[top_];
83  }
84
85  void Stack::pop(){
86    if (top_ < 0)
87      throw underflow_error("pop, empty stack");
88    --top_;
89  }
90
91  bool Stack::empty(){
92    return top_ < 0;
93  }
94
95  void Stack::clear(){
96    top_ = -1;
97  }
98
99  ostream& operator<<(ostream &out, const Stack &s){
100     out << "(bottom) ";
101     copy(s.ary_, s.ary_ + s.top_+1,
102       ostream_iterator<char>(out, ","));
103     out << " (top)";
104     return out;
105 }
```