Advanced C++ Programming

CIS29

**Joseph Bentley**

**DeAnzaCollege**

Computer Information System

December 2022

Contents

[Review 3](#_Toc62935497)

[Classes, Constructors, and Destructors 3](#_Toc62935498)

[Maybe You Haven’t Covered This 14](#_Toc62935499)

[Some C++ 11/14/17/20 Features 21](#_Toc62935500)

[auto type 21](#_Toc62935501)

[the decltype operator 21](#_Toc62935502)

[the constexpr specifier 22](#_Toc62935503)

[nullptr 22](#_Toc62935504)

[Uniform initialization/Brace/List initialization 23](#_Toc62935505)

[Range-based for loop 23](#_Toc62935506)

[Defaulted and deleted constructors 24](#_Toc62935507)

[The override specifier 24](#_Toc62935508)

[R-value references 25](#_Toc62935509)

[Default class member initializer 28](#_Toc62935510)

[The generic size function 29](#_Toc62935511)

[Binary File I/O 30](#_Toc62935512)

[istream member functions 30](#_Toc62935513)

[ostream member functions 33](#_Toc62935514)

[Cast operators 41](#_Toc62935515)

[Static Cast 41](#_Toc62935516)

[Const Cast 41](#_Toc62935517)

[Reinterpret Cast 43](#_Toc62935518)

[Dynamic Cast 44](#_Toc62935519)

[The string class 45](#_Toc62935520)

[Constructors 45](#_Toc62935521)

[Iterator Functions 47](#_Toc62935522)

[Capacity Functions 49](#_Toc62935523)

[Access Functions 52](#_Toc62935524)

[Modifier Functions 52](#_Toc62935525)

[Search Functions 56](#_Toc62935526)

[Operation Functions 59](#_Toc62935527)

[Non-member Functions 61](#_Toc62935528)

[Member Operators 62](#_Toc62935529)

[Non-member Operators 63](#_Toc62935530)

[Member Constant 64](#_Toc62935531)

[The stringstream classes 65](#_Toc62935532)

[The istringstream class 65](#_Toc62935533)

[The ostringstream class 69](#_Toc62935534)

[The stringstream class 71](#_Toc62935535)

[I/O Manipulators 72](#_Toc62935536)

[std manipulators 72](#_Toc62935537)

[Write your own manipulator 84](#_Toc62935538)

[Data at the Bit Level 87](#_Toc62935539)

[Data Storage at the bit level 87](#_Toc62935540)

[Bitwise Operators 94](#_Toc62935541)

[Practical Applications 100](#_Toc62935542)

[Multiple Inheritance 104](#_Toc62935543)

[Multiple Inheritance with Virtual Base Classes 105](#_Toc62935544)

[Exception Handling 108](#_Toc62935545)

[When are Exception Handling Methods Appropriate? 108](#_Toc62935546)

[Previous Error Handling Methods 110](#_Toc62935547)

[Exception Handling Basics 113](#_Toc62935548)

[Namespaces 136](#_Toc62935549)

[Libraries 143](#_Toc62935550)

[Creating a Library 143](#_Toc62935551)

[Using a Library 143](#_Toc62935552)

[Types of Linking 143](#_Toc62935553)

[Examples 144](#_Toc62935554)

[Using the Curl Library 185](#_Toc62935555)

[Templates 190](#_Toc62935556)

[Function Templates 190](#_Toc62935557)

[Class Templates 200](#_Toc62935558)

[Hash Tables 214](#_Toc62935559)

[Standard Template Library 219](#_Toc62935560)

[Containers 219](#_Toc62935561)

[array 220](#_Toc62935562)

[vector 224](#_Toc62935563)

[list 231](#_Toc62935564)

[forward\_list 238](#_Toc62935565)

[deque 246](#_Toc62935566)

[queue 253](#_Toc62935567)

[priority\_queue 257](#_Toc62935568)

[stack 260](#_Toc62935569)

[set 263](#_Toc62935570)

[multiset 269](#_Toc62935571)

[map 273](#_Toc62935572)

[multimap 279](#_Toc62935573)

[unordered\_set 282](#_Toc62935574)

[unordered\_multiset 287](#_Toc62935575)

[unordered\_map 292](#_Toc62935576)

[unordered\_multimap 298](#_Toc62935577)

[bitset 301](#_Toc62935578)

[STL Algorithms 306](#_Toc62935579)

[Lambda Expressions / Functions 312](#_Toc62935580)

[Smart Pointers 319](#_Toc62935581)

[unique\_ptr 319](#_Toc62935582)

[shared\_ptr 321](#_Toc62935583)

[Assignment 9 - Reference Counting and a Linked List 325](file:///Z:\deanza\cis29\CIS29.docx#_Toc62935584)

[Programming Style 331](#_Toc62935585)

# Review

## Classes, Constructors, and Destructors

### Example 1 – Card and Deck class (old code)

1. #include <iostream>
2. #include <cstdlib> // needed for rand() function
3. using namespace std;
4. const char\* const value\_name[13] =
5. {"two","three","four","five","six","seven","eight","nine","ten",
6. "jack","queen","king","ace"};
7. const char\* const suit\_name[4] =
8. {"clubs","diamonds","hearts","spades"};
9. const unsigned short DeckSize = 52;
10. class Card
11. {
12. public:
13. enum suitType { clubs, diamonds, hearts, spades };
14. Card ();
15. void assign(unsigned short);
16. int get\_value(void) const
17. {
18. return value;
19. }
20. int get\_suit(void) const
21. {
22. return suit;
23. }
24. void print(void) const;
25. private:
26. unsigned short value;
27. suitType suit;
28. };
29. Card::Card() : value(0), suit(clubs)
30. {}
31. void Card::assign(unsigned short x)
32. {
33. value = x % 13;
34. suit = (suitType) (x % 4);
35. }
36. void Card::print(void) const
37. {
38. cout << (value\_name[value]) << " of "
39. << (suit\_name[suit]) << endl;
40. }
41. class Deck
42. {
43. public:
44. Deck();
45. void print(void) const;
46. private:
47. Card card[DeckSize];
48. void shuffle(void);
49. };
50. Deck::Deck()
51. {
52. unsigned short i;
53. for (i = 0; i < DeckSize; i++) card[i].assign(i);
54. shuffle();
55. }
56. void Deck::shuffle(void)
57. {
58. unsigned short i;
59. unsigned short k;
60. Card temp;
61. cout << "I am shuffling the Deck\n";
62. for (i = 0; i < DeckSize; i++)
63. {
64. k = rand() % DeckSize;
65. temp = card[i];
66. card[i] = card[k];
67. card[k] = temp;
68. }
69. }
70. void Deck::print(void) const
71. {
72. for (unsigned short i = 0; i < DeckSize; i++) card[i].print();
73. }
74. int main(void)
75. {
76. Deck poker;
77. poker.print();
78. return 0;
79. }

\*\*\*\*\* Output \*\*\*\*\*

I am shuffling the Deck

four of diamonds

ten of clubs

jack of hearts

jack of diamonds

six of diamonds

nine of clubs

…

eight of clubs

Review questions

Line 5: what does “const char\* const” mean?

Line 9: why not **#define DeckSize 52** ?

Line 14: enum suitType { clubs, diamonds, hearts, spades };

Is this a declaration or a definition?

Does it have to be placed inside the class definition?

What are the implications/constraints/requirements of placing it inside the class definition?

Line 17: What’s this?

Line31: What’s this?

Line 38: Is this a 4-letter word? (suitType)

How else can you write this line?

What is the relationship between Card and Deck?

Lines 57-62: What if you write the Deck constructor as …

Deck::Deck()

{

for (unsignedshort i = 0; i < DeckSize; i++) {

card[i].assign(i);

}

shuffle();

}

What’s the difference?

Scope?

How many constructor calls take place when line 90 is executed?

Why are there no destructors in this example?

### Example 2 – Card and Deck class (revised)

1. #include <iostream>
2. #include <cstdlib> // needed for rand() function
3. #include <string>
4. using namespace std;
5. const unsigned short DeckSize = 52;
6. class Card
7. {
8. public:
9. enum suitType { clubs, diamonds, hearts, spades };
10. static const string value\_name[13];
11. static const string suit\_name[4];
12. Card ();
13. Card (int);
14. int get\_value(void) const
15. {
16. return value;
17. }
18. suitType get\_suit(void) const
19. {
20. return suit;
21. }
22. private:
23. int value;
24. suitType suit;
25. static int default\_card\_initializer;
26. };
27. int Card::default\_card\_initializer = 0;
28. const string Card::value\_name[13] =
29. {"two","three","four","five","six","seven",
30. "eight","nine","ten","jack","queen","king","ace"};
31. const string Card::suit\_name[4] =
32. {"clubs","diamonds","hearts","spades"};
33. Card::Card()
34. : value(default\_card\_initializer % 13),
35. suit(static\_cast<suitType>(default\_card\_initializer % 4))
36. {
37. ++default\_card\_initializer;
38. }
39. Card::Card(int x)
40. : value(x % 13),
41. suit(static\_cast<suitType>(x % 4))
42. {}
43. ostream& operator<<(ostream& out, const Card& card)
44. {
45. out << (Card::value\_name[card.get\_value()])
46. << " of "
47. << (Card::suit\_name[card.get\_suit()]);
48. return out;
49. }
50. class Deck
51. {
52. public:
53. Deck();
54. const Card\* get\_card() const
55. {
56. return card;
57. }
58. Card get\_card(int index) const
59. {
60. return card[index];
61. }
62. private:
63. Card card[DeckSize];
64. void shuffle();
65. friend ostream& operator<<(ostream& out, const Deck& deck);
66. };
67. Deck::Deck()
68. {
69. shuffle();
70. }
71. void Deck::shuffle()
72. {
73. int k;
74. Card temp;
75. cout << "I am shuffling the Deck\n";
76. for (int i = 0; i < DeckSize; i++)
77. {
78. k = rand() % DeckSize;
79. temp = card[i];
80. card[i] = card[k];
81. card[k] = temp;
82. }
83. }
84. ostream& operator<<(ostream& out, const Deck& deck)
85. {
86. for (Card c : deck.card) // range-based for loop
87. out << c << endl;
88. return out;
89. }
90. int main(void)
91. {
92. Deck poker;
93. cout << poker << endl;
94. }

### Example 3 – Card and Deck class (another revision)

1. #include <iostream>
2. #include <cstdlib> // needed for rand() function
3. #include <string>
4. using namespace std;
5. class Card
6. {
7. public:
8. enum suitType { clubs, diamonds, hearts, spades };
9. static const string value\_name[13];
10. static const string suit\_name[4];
11. Card ();
12. Card (int);
13. int get\_value(void) const
14. {
15. return value;
16. }
17. suitType get\_suit(void) const
18. {
19. return suit;
20. }
21. private:
22. int value;
23. suitType suit;
24. static int default\_card\_initializer;
25. friend ostream& operator<<(ostream& out, const Card& card);
26. };
27. int Card::default\_card\_initializer = 0;
28. const string Card::value\_name[13] =
29. {
30. "two","three","four","five","six","seven",
31. "eight","nine","ten","jack","queen","king","ace"
32. };
33. const string Card::suit\_name[4] =
34. {"clubs","diamonds","hearts","spades"};
35. Card::Card()
36. : value(default\_card\_initializer % 13),
37. suit(static\_cast<suitType>(default\_card\_initializer % 4))
38. {
39. ++default\_card\_initializer;
40. }
41. Card::Card(int x)
42. : value(x % 13), suit(static\_cast<suitType>(x % 4))
43. {}
44. ostream& operator<<(ostream& out, const Card& card)
45. {
46. out << (Card::value\_name[card.value])
47. << " of "
48. << (Card::suit\_name[card.suit]);
49. return out;
50. }
51. class Deck
52. {
53. public:
54. Deck();
55. Deck(const Deck&);
56. ~Deck() { delete [] cards; cards = 0;}
57. Deck& operator= (const Deck&);
58. const Card\* get\_cards() const
59. {
60. return cards;
61. }
62. Card get\_cards(int index) const
63. {
64. return cards[index];
65. }
66. private:
67. static const unsigned short DeckSize;
68. Card\* cards;
69. void shuffle();
70. friend ostream& operator<<(ostream& out, const Deck& deck);
71. };
72. const unsigned short Deck::DeckSize = 52;
73. Deck::Deck() : cards(new Card[DeckSize])
74. {
75. shuffle();
76. }
77. Deck::Deck(const Deck& anotherDeck)
78. : cards(new Card[DeckSize])
79. {
80. for (auto i = 0; i < DeckSize; ++i)
81. {
82. cards[i] = anotherDeck.cards[i];
83. }
84. }
85. Deck& Deck::operator=(const Deck& anotherDeck)
86. {
87. if (cards) delete [] cards;
88. cards = new Card[DeckSize];
89. for (auto i = 0; i < DeckSize; ++i)
90. {
91. cards[i] = anotherDeck.cards[i];
92. }
93. return \*this;
94. }
95. void Deck::shuffle()
96. {
97. int k;
98. Card temp;
99. cout << "I am shuffling the Deck\n";
100. for (auto i = 0; i < DeckSize; i++)
101. {
102. k = rand() % DeckSize;
103. temp = cards[i];
104. cards[i] = cards[k];
105. cards[k] = temp;
106. }
107. }
108. ostream& operator<<(ostream& out, const Deck& deck)
109. {
110. for (auto i = 0; i < Deck::DeckSize; ++i)
111. out << deck.cards[i] << endl;
112. return out;
113. }
114. int main(void)
115. {
116. Deck poker;
117. cout << poker << endl;
118. }

\*\*\*\*\* Output \*\*\*\*\*

I am shuffling the Deck

four of diamonds

ten of clubs

jack of hearts

jack of diamonds

six of diamonds

nine of clubs

ace of diamonds

…

Review questions

Lines 63 - 65: copy constructor, destructor, overloaded assignment operator – why?

Line 83: syntax

Line 91: auto

Lines 97-106: how to write an overloaded assignment operator

Lines 27 and 108: Do you have to have friends?

### Example 4 – Adding Matrices

1. #include <iomanip>
2. #include <iostream>
3. #include <cstdlib> // for rand()
4. using namespace std;
5. class Matrix
6. {
7. private:
8. int\*\* element;
9. int rows;
10. int cols;
11. void alloc();
12. void release();
13. public:
14. Matrix(int = 0, int = 0); // also default constructor
15. Matrix(const Matrix&); // copy constructor
16. ~Matrix();
17. Matrix operator+(const Matrix&) const;
18. Matrix& operator=(const Matrix&);
19. friend ostream& operator<<(ostream&, const Matrix&);
20. };
21. int main()
22. {
23. Matrix A(3,4), B(3,4), C;
24. cout << A << endl;
25. cout << B << endl;
26. cout << C << endl;
27. C = A + B;
28. cout << C << endl;
29. }
30. Matrix::Matrix(int r, int c) : rows(r), cols(c)
31. {
32. cout << "Constructor called for object " << this <<endl;
33. alloc();
34. // initialize Matrix elements with random numbers 0-9
35. for (int i = 0; i < rows; i++)
36. for (int j = 0; j < cols; j++)
37. element[i][j] = rand()%10;
38. }
39. Matrix::Matrix(const Matrix& arg) : rows(arg.rows), cols(arg.cols)
40. {
41. cout << "\nIn copy constructor for object " << this;
42. cout << ", argument: " << &arg << endl;
43. alloc();
44. for (int i = 0; i < rows; i++)
45. for (int j = 0; j < cols; j++)
46. element[i][j] = arg.element[i][j];
47. }
48. Matrix::~Matrix()
49. {
50. cout << "\n~~ Destructor called for object: " << this << endl;
51. release();
52. }
53. void Matrix::alloc() // allocate heap memory for elements
54. {
55. cout << "Allocate memory for Matrix " << this << " elements\n";
56. element = new int\*[rows];
57. for (int i = 0; i < rows; i++)
58. element[i] = new int[cols];
59. }
60. void Matrix::release()
61. {
62. cout << "I got rid of Matrix " << this << "'s elements\n";
63. for (int i = 0; i < rows; i++)
64. delete [] element[i];
65. delete [] element;
66. }
67. Matrix Matrix::operator+(const Matrix& arg) const
68. {
69. cout << "\nExecuting operator+ for object: " << this;
70. cout << ", argument: " << &arg << endl;
71. if (rows != arg.rows || cols != arg.cols)
72. {
73. cerr << "Invalid Matrix addition\n";
74. return (\*this);
75. }
76. Matrix temp(rows,cols);
77. for (int i = 0; i < rows; i++)
78. for (int j = 0; j < cols; j++)
79. temp.element[i][j] = element[i][j] + arg.element[i][j];
80. cout << temp << endl;
81. return temp;
82. }
83. Matrix& Matrix::operator=(const Matrix& arg)
84. {
85. cout << "\nExecuting operator= for object: " << this;
86. cout << ", argument: " << &arg << endl;
87. // Make sure rows and cols match the argument
88. if (rows != arg.rows || cols != arg.cols)
89. {
90. release();
91. rows = arg.rows;
92. cols = arg.cols;
93. alloc();
94. }
95. for (int i = 0; i < arg.rows; i++)
96. for (int j = 0; j < arg.cols; j++)
97. element[i][j] = arg.element[i][j];
98. return \*this;
99. }
100. ostream& operator<<(ostream& out, const Matrix& m)
101. {
102. out << "\nMatrix values for object: "<< &m << endl;
103. out << "----------------\n";
104. for (int i = 0; i < m.rows; i++)
105. {
106. for (int j = 0; j < m.cols; j++)
107. out << setw(4) << m.element[i][j];
108. out << endl;
109. }
110. out << "----------------";
111. return out;
112. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Constructor called for object 0xffffcb80

Allocate memory for Matrix 0xffffcb80 elements

Constructor called for object 0xffffcb70

Allocate memory for Matrix 0xffffcb70 elements

Constructor called for object 0xffffcb60

Allocate memory for Matrix 0xffffcb60 elements

Matrix values for object: 0xffffcb80

----------------

3 3 2 9

0 8 2 6

6 9 1 1

----------------

Matrix values for object: 0xffffcb70

----------------

3 5 8 3

0 6 9 2

7 7 2 8

----------------

Matrix values for object: 0xffffcb60

----------------

----------------

Executing operator+ for object: 0xffffcb80, argument: 0xffffcb70

Constructor called for object 0xffffcb00

Allocate memory for Matrix 0xffffcb00 elements

Matrix values for object: 0xffffcb00

----------------

6 8 10 12

0 14 11 8

13 16 3 9

----------------

In copy constructor for object 0xffffcb90, argument: 0xffffcb00

Allocate memory for Matrix 0xffffcb90 elements

~~ Destructor called for object: 0xffffcb00

I got rid of Matrix 0xffffcb00's elements

Executing operator= for object: 0xffffcb60, argument: 0xffffcb90

I got rid of Matrix 0xffffcb60's elements

Allocate memory for Matrix 0xffffcb60 elements

~~ Destructor called for object: 0xffffcb90

I got rid of Matrix 0xffffcb90's elements

Matrix values for object: 0xffffcb60

----------------

6 8 10 12

0 14 11 8

13 16 3 9

----------------

~~ Destructor called for object: 0xffffcb60

I got rid of Matrix 0xffffcb60's elements

~~ Destructor called for object: 0xffffcb70

I got rid of Matrix 0xffffcb70's elements

~~ Destructor called for object: 0xffffcb80

I got rid of Matrix 0xffffcb80's elements

## Maybe You Haven’t Covered This

### Conversion Operators

### Example 5 - Conversion of a user-defined type to a primitive type

1. #include <iostream>
2. using namespace std;
3. class B
4. {
5. int b;
6. public:
7. B(int i) : b(i) {}
8. operator int() const;
9. };
10. B::operator int() const
11. {
12. cout << "\* B:: operator int() called\n";
13. return b;
14. }
15. int main()
16. {
17. B eight(8);
18. cout << eight << endl;
19. cout << eight + 5 << endl;
20. cout << 5 + eight << endl;
21. cout << (eight > 3) << endl;
22. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

\* B:: operator int() called

8

\* B:: operator int() called

13

\* B:: operator int() called

13

\* B:: operator int() called

1

What would happen if operator int() was not defined?

### Example 6 - More Conversions of a user-defined type

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. class Day; // forward declaration
5. class Number
6. {
7. int n;
8. public:
9. Number(int i) : n(i)
10. {
11. cout << "Number(int) ctor called\n";
12. }
13. operator int() const;
14. operator Day() const;
15. };
16. Number::operator int() const
17. {
18. cout << "\* Number::operator int() called\n";
19. return n;
20. }
21. const string Days[7] =
22. {
23. "Sunday","Monday","Tuesday","Wednesday","Thursday",
24. "Friday","Saturday"
25. };
26. class Day
27. {
28. string dow;
29. public:
30. Day(int n) : dow(Days[n%7])
31. {
32. cout << "Day(int) ctor called\n";
33. }
34. operator Number() const; // convert Day to Number
35. void operator!() const
36. {
37. cout << "dow = " << dow << endl;
38. }
39. };
40. Day::operator Number() const
41. {
42. cout << "\*\* Day:: operator Number() called\n";
43. for (int i = 0; i < 7; i++)
44. if (dow == Days[i]) return Number(i);
45. return Number(-1);
46. }
47. Number::operator Day() const // Why is this function here?
48. {
49. cout << "\*\*\* Number::operator Day() called\n";
50. return n; //Day(n);
51. }
52. void somefunction(Day)
53. {
54. cout << "somefunction called\n";
55. }
56. int main()
57. {
58. Number N1(65);
59. cout << "N1 = " << N1 << endl;
60. Day d1(1);
61. !d1;
62. // Day d2(N1); Why is this an ambiguity?
63. Number N2(d1);
64. cout << "N2 = " << N2 << endl;
65. !Day(Number(d1)+2);
66. somefunction(N1);
67. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Number(int) ctor called

\* Number::operator int() called

N1 = 65

Day(int) ctor called

dow = Monday

\*\* Day:: operator Number() called

Number(int) ctor called

\* Number::operator int() called

N2 = 1

\*\* Day:: operator Number() called

Number(int) ctor called

\* Number::operator int() called

Day(int) ctor called

dow = Wednesday

\*\*\* Number::operator Day() called

Day(int) ctor called

somefunction called

### Explicit Constructors

The keyword ***explicit*** is used to specify that a constructor may only be used for object instantiation and not for automatic conversion. Here’s an example that demonstrates the effect.

### Example 7 – Explicit constructors

1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. public:
6. A(int); // non-explicit ctor
7. };
8. class B
9. {
10. public:
11. explicit B(int); // explicit ctor
12. };
13. A::A(int)
14. {
15. cout << "A ctor called for object " << this << endl;
16. }
17. B::B(int) // do not repeat keyword explicit
18. {
19. cout << "B ctor called for object " << this << endl;
20. }
21. void funkA(A object)
22. {
23. cout << "funkA called\n";
24. }
25. void funkB(B object)
26. {
27. cout << "funkB called\n";
28. }
29. void funkAB(A obj)
30. {
31. cout << "funkAB(A) called\n";
32. }
33. void funkAB(B obj)
34. {
35. cout << "funkAB(B) called\n";
36. }
37. int main()
38. {
39. A objA(2); // instantiate an A object
40. B objB(3); // instantiate a B object
41. funkA(objA); // call funkA() with an exact argument match
42. funkA(9); // call funkA() with an non-exact match
43. funkB(objB); // call funkB() with an exact argument match
44. // funkB(16); // error: cannot convert int to a B object
45. funkAB(6); // compile error if B(int) is not explicit
46. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

A ctor called for object 0x6dfefd

B ctor called for object 0x6dfefc

funkA called

A ctor called for object 0x6dfefe

funkA called

funkB called

A ctor called for object 0x6dfeff

funkAB(A) called

### typedef and using

The keyword, typedef, originally from C, is used to define a type.

C++ 11 introduced the keyword, using to act like typedef.

### typeid operator

The typeid operator returns an identifier of a type, a variable or an expression. The return of the typeid is a class type, called type\_info. You can use the name() member function of the type\_info class to display a literal description of the type.

### Example 8 – typedef, using, typeid

1. #include <iostream>
2. #include <typeinfo> // for typeid
3. using namespace std;
4. int main()
5. {
6. typedef int number;
7. number n;
8. typedef long long int bignumber;
9. bignumber biggie;
10. typedef double(\*ptr2arrayof10)[10];
11. double d[13][10];
12. ptr2arrayof10 p = d;
13. using Word = unsigned int;
14. Word seven = 7U;
15. using pint = int\*;
16. pint addr\_n = &n;
17. using Int4 = int[4];
18. Int4 iota4 = {1,2,3,4};
19. cout << "typeid(int).name()=" << typeid(int).name() << endl;
20. cout << "typeid(bignumber).name()=" << typeid(bignumber).name()
21. << endl;
22. cout << "typeid(biggie).name()=" << typeid(biggie).name()
23. << endl;
24. cout << "typeid(p).name()=" << typeid(p).name() << endl;
25. cout << "typeid(ptr2arrayof10).name()="
26. << typeid(ptr2arrayof10).name() << endl;
27. cout << "typeid(seven).name()=" << typeid(seven).name()
28. << endl;
29. cout << "typeid(Word).name()=" << typeid(Word).name() << endl;
30. cout << "typeid(pint).name()=" << typeid(pint).name() << endl;
31. cout << "typeid(addr\_n).name()=" << typeid(addr\_n).name()
32. << endl;
33. cout << "typeid(Int4).name()=" << typeid(Int4).name() << endl;
34. cout << "typeid(iota4).name()=" << typeid(iota4).name()
35. << endl;
36. }

\*\*\*\*\*\* Code::Blocks / NetBeans / Eclipse / Linux / Mac Xcode \*\*\*\*\*\*

typeid(int).name()=i

typeid(bignumber).name()=x

typeid(biggie).name()=x

typeid(p).name()=PA10\_d

typeid(ptr2arrayof10).name()=PA10\_d

typeid(seven).name()=j

typeid(Word).name()=j

typeid(pint).name()=Pi

typeid(addr\_n).name()=Pi

typeid(Int4).name()=A4\_i

typeid(iota4).name()=A4\_i

\*\*\*\*\*\* MS Visual Studio 2019 \*\*\*\*\*\*

typeid(int).name()=int

typeid(bignumber).name()=\_\_int64

typeid(biggie).name()=\_\_int64

typeid(p).name()=double (\*)[10]

typeid(ptr2arrayof10).name()=double (\*)[10]

typeid(seven).name()=unsigned int

typeid(Word).name()=unsigned int

typeid(pint).name()=int \*

typeid(addr\_n).name()=int \*

typeid(Int4).name()=int [4]

typeid(iota4).name()=int [4]

# Some C++ 11/14/17/20 Features

## auto type

Using the auto keyword, a variable’s type may be automatic assigned. The new usage of the auto keyword negates the former ansi-C storage class meaning.

## the decltype operator

The decltype operator is similar to auto, it returns the type of an expression.

### Example 1 – auto type and decltype

1. #include <iostream>
2. #include <typeinfo> // for typeid
3. using namespace std;
4. int main()
5. {
6. auto v1 = 7; // v1 is type int
7. auto mygrade ='a'; // mygrade is type char
8. auto pi = 31.4; // pi is type double
9. auto cstring = "have a nice day"; // pointer to const char
10. auto ptr2char = &mygrade; // pointer to char
11. auto z = "zebra"[0]; // z is type char
12. cout << typeid(v1).name() << endl;
13. cout << typeid(mygrade).name() << endl;
14. cout << typeid(pi).name() << endl;
15. cout << typeid(cstring).name() << endl;
16. cout << typeid(ptr2char).name() << endl;
17. cout << typeid(z).name() << endl;
18. typedef decltype(7) myint;
19. myint x;
20. cout << typeid(x).name() << endl;
21. decltype(7) y;
22. cout << typeid(y).name() << endl;
23. // Somewhat practical
24. int array[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
25. cout << typeid(array).name() << endl;
26. cout << typeid(array[1]).name() << endl;
27. cout << typeid(\*array).name() << endl;
28. cout << typeid(&array).name() << endl;
29. }

\*\*\*\*\*\* Code::Blocks / NetBeans / Linux \*\*\*\*\*\*

i

c

d

PKc

Pc

c

i

i

A3\_A4\_i

A4\_i

A4\_i

PA3\_A4\_i

\*\*\*\*\*\* MS Visual Studio 2017 \*\*\*\*\*\*

int

char

double

char const \*

char \*

char

int

int

int [3][4]

int [4]

int [4]

int (\*)[3][4]

## the constexpr specifier

The constexpr specifier declares that a function or variable is const at compile time.

Examples

constexpr float pi = 3.14;

constexpr float areaOfCircle(float radius)

{

return pi \* radius \* radius;

}

constexpr float area1 = areaOfCircle(1);

const float two = 2.f;

constexpr float area2 = areaOfCircle(two);

float three = 3.f;

constexpr float area32 = areaOfCircle(three); **// ERROR**

## nullptr

nullptr is a pointer constant with conversions to any pointer type. It is used as a replacement for the macro, NULL or a 0 pointer.

char\*ptr = nullptr;

void somefunk(type\* ptr = nullptr);

if (p == nullptr) …

## Uniform initialization/Brace/List initialization

int I{7}; // instead of int I = 7;

int zero{}; // same as int zero = 0;

string s{“apple pie”};

SomeClass object{19}; // instead of SomeClass object(19);

AnotherClass obj{thing,23,2.5,’a’}; // instead of AnotherClass obj(thing,23,2.5,’a’);

## Range-based for loop

### Example 2 – Range-based for loop

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int array[5] = {2,3,5,7,11};
6. for (int i : array)
7. cout << i << " ";
8. cout << endl;
9. for (auto i : array)
10. cout << i << " ";
11. cout << endl;
12. for (auto i : array)
13. i = 13;
14. for (auto i : array)
15. cout << i << " ";
16. cout << endl;
17. for (auto& i : array)
18. i = 13;
19. for (auto i : array)
20. cout << i << " ";
21. cout << endl;
22. for (auto value : {9,8,7,6} ) // note initializer list
23. {
24. cout << value << " ";
25. }
26. cout << endl;
27. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

2 3 5 7 11

2 3 5 7 11

2 3 5 7 11

13 13 13 13 13

9 8 7 6

## Defaulted and deleted constructors

The default specifier with the default constructor causes the compiler to generate it. The delete specifier is used to disable a constructor.

class ABC

{

int a,b,c;

Public:

ABC() = **default**; // same as ABC(){}

ABC(int, int, int);

ABC(const ABC&) = **delete**; // disable copy constructor

…

};

## The override specifier

The keyword override specifier is a way to ensure that a virtual function in a derived class overrides the analogous function in the base class.

class Base

{

…

public:

virtual void funk1(int);

virtual void funk2(float);

virtual void funk3(string);

…

};

class Derived : public Base

{

…

public:

virtual void funk1(int); // overrides funk1 in Base class

// funk2 is not overridden

virtual void funk3(string) override; // funk3 is overridden

virtual void funk4(char) override; // ERROR

…

};

## R-value references

R-value references permits a reference to bind to an r-value – a temporary or a literal. This is useful for the *move constructor* or the *move assignment operator*, avoiding the expense of copying an object for this purpose.

### Example 3 – R-value References

1. #include <iostream>
2. #include <utility> // for move
3. using namespace std;
4. void increment(int& value)
5. {
6. cout << "increment with lvalue reference argument" << endl;
7. ++value;
8. }
9. void increment(int&& value)
10. {
11. cout << "increment with rvalue reference argument" << endl;
12. ++value;
13. }
14. int main()
15. {
16. int i = 1;
17. // Increment a variable
18. increment(i);
19. cout << "i=" << i << endl;
20. // Increment an expression
21. increment(i + 5);
22. // Increment a literal constant
23. increment(3);
24. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

increment with lvalue reference argument

i=2

increment with rvalue reference argument

increment with rvalue reference argument

### Move Semantics

With the use of rvalue references in C++11, the move constructor and the move assignment operator was added as a replacement for the copy constructor and the overloaded assignment operator.

### Example 4 – Move Semantics

1. #include <iostream>
2. #include <cstring>
3. #include <utility> // for move
4. using namespace std;
5. class Student
6. {
7. char\* name;
8. public:
9. Student(); // default constructor
10. Student(const char\* n);
11. Student(const Student& obj); // copy constructor
12. Student(Student&& obj); // move constructor
13. ~Student(); // destructor
14. Student& operator=(const Student& obj); // assignment operator
15. Student& operator=(Student&& obj); // move assignment
16. const char\* getName() const
17. {
18. return name ? name : "";
19. }
20. };
21. ostream& operator<<(ostream& out, const Student& obj)
22. {
23. return out << "object=" << &obj << " name=" << obj.getName();
24. }
25. Student::Student() : name(nullptr)
26. {
27. cout << "> In default constructor: " << \*this << endl;
28. }
29. Student::Student(const char\* n)
30. : name(new char[strlen(n)+1])
31. {
32. strcpy(name,n);
33. cout << "> In Student(const char\* n) ctor: " << \*this << endl;
34. }
35. Student::Student(const Student& obj)
36. : name(new char[strlen(obj.name+1)])
37. {
38. strcpy(name,obj.name);
39. cout << "> In copy constructor: " << \*this << endl;
40. }
41. Student::Student(Student&& obj)
42. : name(new char[strlen(obj.name+1)])
43. {
44. strcpy(name,obj.name);
45. cout << "> In move constructor: " << \*this << endl;
46. delete [] obj.name;
47. obj.name = nullptr;
48. }
49. Student::~Student()
50. {
51. cout << "~ Student destructor " << \*this << endl;
52. if (name) delete [] name;
53. name = nullptr;
54. }
55. Student& Student::operator=(const Student& obj)
56. {
57. delete [] name;
58. name = new char[strlen(obj.name+1)];
59. strcpy(name,obj.name);
60. cout << "= In assignment operator: " << \*this << endl;
61. return \*this;
62. }
63. Student& Student::operator=(Student&& obj)
64. {
65. delete [] name;
66. name = obj.name;
67. cout << "= In move assignment operator: " << \*this << endl;
68. obj.name = nullptr;
69. return \*this;
70. }
71. Student create()
72. {
73. cout << "In create()\n";
74. return Student("Temporary");;
75. }
76. int main()
77. {
78. cout << "Executing line => Student j(\"Joe\");" << endl;
79. Student j("Joe");
80. cout << "j = " << j << endl;
81. cout << "\nExecuting line => Student h(j);" << endl;
82. Student h(j);
83. cout << "\nExecuting line => h = j;" << endl;
84. h = j;
85. cout << "\nExecuting line => j = create();" << endl;
86. j = create();
87. cout << "j = " << j << endl;
88. cout << "\nExecuting line => Student k(move(j));" << endl;
89. Student k(move(j));
90. cout << "k = " << k << endl;
91. cout << "j = " << j << endl;
92. cout << "\nThat's all folks!!!" << endl;
93. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Executing line => Student j("Joe");

> In Student(const char\* n) ctor: object=0x61fe00 name=Joe

j = object=0x61fe00 name=Joe

Executing line => Student h(j);

> In copy constructor: object=0x61fdf8 name=Joe

Executing line => h = j;

= In assignment operator: object=0x61fdf8 name=Joe

Executing line => j = create();

In create()

> In Student(const char\* n) ctor: object=0x61fe08 name=Temporary

= In move assignment operator: object=0x61fe00 name=Temporary

~ Student destructor object=0x61fe08 name=

j = object=0x61fe00 name=Temporary

Executing line => Student k(move(j));

> In move constructor: object=0x61fdf0 name=Temporary

k = object=0x61fdf0 name=Temporary

j = object=0x61fe00 name=

That's all folks!!!

~ Student destructor object=0x61fdf0 name=Temporary

~ Student destructor object=0x61fdf8 name=Joe

~ Student destructor object=0x61fe00 name=

## Default class member initializer

Non-static class data members may contain a default initializer in the class definition. This default initializer can be overridden in a contructor initialization list or in the body of a constructor.

### Example 5 –Default class member initializer

1. #include <iostream>
2. using namespace std;
3. class DMI
4. {
5. int a = 0;
6. int b = 1;
7. int c = 2;
8. public:
9. DMI();
10. int geta() const { return a; }
11. int getb() const { return b; }
12. int getc() const { return c; }
13. };
14. DMI::DMI() : a(5), b(6) { b = 8; c = 9; }
15. ostream& operator<<(ostream& out, const DMI& obj)
16. {
17. out << obj.geta() << ' ' << obj.getb() << ' ' << obj.getc();
18. return out;
19. }
20. int main()
21. {
22. DMI object;
23. cout << object << endl;
24. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

5 8 9

##### Explanation

Each member of the DMI class has a default member initializer. Class member initialiations are overridden as follows:

* a is overridden by the constructor initializer
* b is overridden by the constructor initializer, and then overridden in the body of the constructor
* c is overridden in the body of the constructor

## The generic size function

The generic size function was introduced in C++ 17. It is used to return the size of an array (number of elements) or a C++ container. It requires the <iterator> header file.

### Example 6 – The size function

Note: this example must be compiled using a C++17 compiler.

1. #include <iostream>
2. #include <iterator>
3. #include <vector>
4. using namespace std;
5. int main()
6. {
7. int a[5];
8. int b[] = {1,2,3};
9. vector<int> v{3,4,5,6};
10. cout << size(a) << endl;
11. cout << size(b) << endl;
12. cout << size(v) << endl;
13. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

5

3

4

# Binary File I/O

## istream member functions

### read

Read a specified number of characters from an input stream and stores them in a char array. The array is not null-terminated.

istream& read (char\* s, streamsize[[1]](#footnote-1) n);

### peek

Returns the next character to be read without extracting it from the input stream.

int peek();

### seekg

Sets the next read position in the input stream.

stream& seekg (streampos[[2]](#footnote-2) pos);

istream& seekg (streamoff[[3]](#footnote-3) offset, ios\_base::seekdir way);

ios\_base::seekdir can be one of three constants

|  |  |
| --- | --- |
| **Constant** | **Meaning** |
| beg | Beginning of the input stream |
| cur | Current position in the input stream |
| end | End of the input stream |

### tellg

Returns the next read position in the input stream.

streampos tellg();

### Example 1 – istream member functions

Input file

HAVE A NICE DAY

have a nice day

This is line 3.

And that's all folks!!!

1. #include <iostream>
2. #include <fstream>
3. #include <cstdlib>
4. using namespace std;
5. int main()
6. {
7. char buffer[32];
8. const char\* filename = "c:/temp/ex1data.txt";
9. ifstream fin(filename);
10. if (!fin) {
11. cerr << "Unable to open input file " << filename << endl;
12. exit(1);
13. }
14. fin.read(buffer, 9); // Read the first 9 bytes of the file
15. cout << '/' << buffer << '/' << endl;
16. buffer[9] = 0; // Null terminate the buffer
17. cout << '/' << buffer << '/' << endl << endl;
18. cout << "fin.tellg() = " << fin.tellg() << endl;
19. cout << "fin.peek() = " << fin.peek() << endl;
20. cout << "static\_cast<char>(fin.peek()) = " <<

static\_cast<char>(fin.peek()) << endl << endl;

1. // Reposition to byte 1
2. // fin.seek(1); ERROR
3. fin.seekg(static\_cast<streampos> (1));
4. cout << "fin.tellg() = " << fin.tellg() << endl << endl;
5. // Create a streampos object
6. streampos pos = fin.tellg();
7. // pos++; ERROR
8. // pos = pos + 5; // throws a warning
9. pos = 2;
10. fin >> buffer;
11. cout << "buffer = " << buffer << endl;
12. cout << "fin.tellg() = " << fin.tellg() << endl << endl;
13. fin.seekg(-2, ios\_base::cur);
14. fin.read(buffer, 25);
15. buffer[25] = 0;
16. cout << "buffer = " << buffer << endl << endl;
17. fin.seekg(0, ios\_base::beg);
18. fin.read(buffer, sizeof (buffer) - 1);
19. buffer[sizeof (buffer) - 1] = 0;
20. cout << "buffer = " << buffer << endl;
21. }

\*\*\*\*\*\* Output: NetBeans on Windows \*\*\*\*\*\*

/HAVE A NI���/

/HAVE A NI/

fin.tellg() = 9

fin.peek() = 67

static\_cast<char>(fin.peek()) = C

fin.tellg() = 1

buffer = AVE

fin.tellg() = 4

buffer = VE A NICE DAY

have a nic

buffer = HAVE A NICE DAY

have a nice da

\*\*\*\*\*\* Output: MS Visual Studio 2017 \*\*\*\*\*\*

/HAVE A NI╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠F çöL ²/

/HAVE A NI/

fin.tellg() = 9

fin.peek() = 67

static\_cast<char>(fin.peek()) = C

fin.tellg() = 1

buffer = AVE

fin.tellg() = 4

buffer = VE A NICE DAY

have a nice

buffer = HAVE A NICE DAY

have a nice day

\*\*\*\*\*\* Output: Code::Blocks on Windows \*\*\*\*\*\*

/HAVE A NI/

/HAVE A NI/

fin.tellg() = 13

fin.peek() = 67

static\_cast<char>(fin.peek()) = C

fin.tellg() = 1

buffer = AVE

fin.tellg() = 8

buffer = NICE DAY

have a nice day

buffer = HAVE A NICE DAY

have a nice day

## ostream member functions

### write

Write a specified number of characters to an output stream

ostream& write (const char\* s, streamsize n);

### seekp

Sets the next write position in the output stream.

ostream& seekp (streampos pos);

ostream& seekp (streamoff off, ios\_base::seekdir way);

### tellp

Returns the next write position in the output stream.

streampos tellp();

### Example 2 – ostream member functions

1. #include <iostream>
2. #include <fstream>
3. #include <cstdlib>
4. #include <cstring>
5. using namespace std;
6. int main()
7. {
8. const char\* filename = "ex2data.bin";
9. ofstream fout(filename);
10. if (!fout)
11. {
12. cerr << "Unable to open output file " << filename << endl;
13. exit(1);
14. }
15. fout.write("Have a nice day",strlen("Have a nice day."));
16. int age = 35;
17. double gpa = 3.5;
18. fout.write(reinterpret\_cast<char\*>(&age),sizeof(int));
19. fout.write(reinterpret\_cast<char\*>(&gpa),sizeof(gpa));
20. cout << fout.tellp() << endl;
21. fout.seekp(0,ios::end);
22. cout << fout.tellp() << endl;
23. fout.seekp(sizeof("Have a ")-1,ios::beg);
24. cout << fout.tellp() << endl;
25. fout.write("good",4);
26. cout << fout.tellp() << endl;
27. fout.close();
28. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

28

28

7

11

### Example 3 – binary file I/O: a practical example

This example demonstrates reading text file, storing each record in a struct and writing it out as a binary file. The “processing” requirement is to read the binary file and give all teachers a 5% raise and give Joe Bentley a 10% raise. The binary file will be updated to reflect the changes.

Input Text File

AGUILAR, RICARDO L ANIMAL CONTROL OFFICER 70644.00

ALLISON, JOHN L ANIMAL TEACHERCONTROL OFFICER 64392.00

AYALA, ARTHUR ANIMAL CONTROL OFFICER 70644.00

BATINICH, JACLYN M VETERINARY ASST 66948.00

BENTLEY, JOE TEACHER 95000.00

CABALLERO, JORGE ANIMAL CONTROL OFFICER 45924.00

CRAYTON, MARSTINE L SUPVSR OF ANIMAL CONTROL OFFICERS 73992.00

DEL RIO, JOSE A SUPVSR OF ANIMAL CONTROL OFFICERS 89124.00

…

1. #include <iostream>
2. #include <iomanip>
3. #include <fstream>
4. #include <cstdlib>
5. #include <cstring>
6. using namespace std;
7. const int NumRecords = 27;
8. const int SizeOfName = 23;
9. const int SizeOfJobtitle = 39;
10. struct SalaryData {
11. char name[SizeOfName];
12. char jobtitle[SizeOfJobtitle];
13. float salary;
14. };
15. void printSalaryData(const SalaryData& record);
16. void rtrim(char\* text);
17. void readAndPrintBinaryFile(const char\* binaryfilename);
18. void processBinaryFile(const char\* binaryfilename);
19. void readTextFileAndWriteToBinaryFile(const char\* textfilename,
20. const char\* binaryfilename);
21. int main()
22. {
23. const char\* textfilename = "c:/temp/ex3data.txt";
24. const char\* binaryfilename = "c:/temp/ex3data.bin";
25. readTextFileAndWriteToBinaryFile(textfilename, binaryfilename);
26. processBinaryFile(binaryfilename);
27. readAndPrintBinaryFile(binaryfilename);
28. }
29. void readTextFileAndWriteToBinaryFile(const char\* textfilename,
30. const char\* binaryfilename)
31. {
32. ifstream fin(textfilename);
33. if (!fin)
34. {
35. cerr << "Unable to open input text file " << textfilename
36. << endl;
37. exit(1);
38. }
39. ofstream fout(binaryfilename, ios::binary);
40. if (!fout)
41. {
42. cerr << "Unable to open input text file " << textfilename
43. << endl;
44. exit(2);
45. }
46. char buffer[80];
47. SalaryData temp;
48. for (int i = 0; i < NumRecords; ++i)
49. {
50. fin.getline(buffer, sizeof (buffer));
51. strtok(buffer, "\r");
52. strncpy(temp.name, buffer, SizeOfName);
53. temp.name[SizeOfName - 1] = 0;
54. rtrim(temp.name);
55. strncpy(temp.jobtitle, buffer + 23, SizeOfJobtitle);
56. temp.jobtitle[SizeOfJobtitle - 1] = 0;
57. rtrim(temp.jobtitle);
58. temp.salary = atof(buffer + 61);
59. printSalaryData(temp);
60. fout.write(reinterpret\_cast<const char\*>(&temp),
61. sizeof (SalaryData));
62. }
63. cout << "--------------------------------------------------\n";
64. }
65. void printSalaryData(const SalaryData& record)
66. {
67. cout << fixed << setprecision(2);
68. cout << left << setw(SizeOfName + 1) << record.name
69. << setw(SizeOfJobtitle + 1) << record.jobtitle
70. << right << setw(10) << record.salary << endl;
71. }
72. void rtrim(char\* text)
73. {
74. size\_t size = strlen(text);
75. for (int i = size - 1; i > 1; --i)
76. {
77. if (!isspace(text[i])) break;
78. else text[i] = 0;
79. }
80. }
81. void readAndPrintBinaryFile(const char\* binaryfilename)
82. {
83. ifstream fin(binaryfilename, ios::binary | ios::in);
84. SalaryData temp;
85. if (fin)
86. {
87. for (int i = 0; i < NumRecords; ++i)
88. {
89. fin.read(reinterpret\_cast<char\*>(&temp),
90. sizeof (temp));
91. printSalaryData(temp);
92. }
93. }
94. else
95. {
96. cerr << "Unable to open binary input file "
97. << binaryfilename << endl;
98. exit(3);
99. }
100. }
101. // Teachers get a 5% raise
102. // Joe Bentley gets a 10% raise
103. void processBinaryFile(const char\* binaryfilename)
104. {
105. // open the binary file for read and write
106. fstream finfout(binaryfilename, ios::binary|ios::in|ios::out);
107. SalaryData temp;
108. if (finfout)
109. {
110. while (!finfout.eof())
111. {
112. finfout.read(reinterpret\_cast<char\*>(&temp),
113. sizeof (temp));
114. if (strstr(temp.name, "BENTLEY"))
115. {
116. temp.salary \*= 1.1;
117. // Backup and rewrite the record
118. finfout.seekp(finfout.tellg() –
119. static\_cast<streampos>(sizeof (SalaryData)));
120. finfout.write(reinterpret\_cast<char\*>(&temp),
121. sizeof (temp));
122. }
123. else if (!strcmp(temp.jobtitle, "TEACHER"))
124. {
125. temp.salary \*= 1.05;
126. // Backup and rewrite the record
127. finfout.seekp(finfout.tellg() –
128. static\_cast<streampos> (sizeof (SalaryData)));
129. finfout.write(reinterpret\_cast<char\*>(&temp),
130. sizeof (temp));
131. }
132. else
133. {
134. }
135. }
136. }
137. else
138. {
139. cerr << "Unable to binary file for processing "
140. << binaryfilename << endl;
141. exit(4);
142. }
143. if (!finfout.good()) finfout.clear();
144. finfout.close();
145. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

AGUILAR, RICARDO L ANIMAL CONTROL OFFICER 70644.00

ALLISON, JOHN L ANIMAL TEACHERCONTROL OFFICER 64392.00

AYALA, ARTHUR ANIMAL CONTROL OFFICER 70644.00

BATINICH, JACLYN M VETERINARY ASST 66948.00

BENTLEY, JOE TEACHER 95000.00

CABALLERO, JORGE ANIMAL CONTROL OFFICER 45924.00

CRAYTON, MARSTINE L SUPVSR OF ANIMAL CONTROL OFFICERS 73992.00

DEL RIO, JOSE A SUPVSR OF ANIMAL CONTROL OFFICERS 89124.00

DIAKHATE, MAMADOU OPERATIONS MANAGER - ANIMAL CONTROL 85008.00

DRAKE, TAURUS L ANIMAL CONTROL INSPECTOR 70644.00

EDGECOMBE, CHERYL K ANIMAL CONTROL INSPECTOR 58644.00

FELTON, DONIELLA M TEACHER 47844.00

FRANCO, ARTURO ANIMAL CONTROL OFFICER 45924.00

GARNER, LINDSAY VETERINARIAN 88080.00

HAMILTON, ARTHUR ANIMAL SHELTER MANAGER 68220.00

HOLCOMB, ALLEN R ANIMAL CONTROL INSPECTOR 77520.00

HOWARD, MARYANN J ANIMAL CONTROL INSPECTOR 64392.00

HUBBS, CARLA A SUPERVISING VETERINARY TECHNICIAN 62820.00

JACOB, VIVISH SUPVSR OF ANIMAL CARE AIDES 84420.00

KELLER, AUDREY A VETERINARIAN 124428.00

LOZANO, RENE P ANIMAL CONTROL OFFICER 67464.00

MARTINIS, JENNIFER ANIMAL CONTROL OFFICER 41832.00

RUSSELL, SUSAN J TEACHER 130008.00

SCHLUETER, JENNIFER L EXEC ADMINISTRATIVE ASST II 59976.00

SILVA, YVONNE ANIMAL CONTROL OFFICER 41832.00

WALTERS, MICHELLE TEACHER 70092.00

YAMAJI, PETER S VETERINARIAN 128136.00

---------------------------------------------------------------

AGUILAR, RICARDO L ANIMAL CONTROL OFFICER 70644.00

ALLISON, JOHN L ANIMAL TEACHERCONTROL OFFICER 64392.00

AYALA, ARTHUR ANIMAL CONTROL OFFICER 70644.00

BATINICH, JACLYN M VETERINARY ASST 66948.00

BENTLEY, JOE TEACHER 104500.00

CABALLERO, JORGE ANIMAL CONTROL OFFICER 45924.00

CRAYTON, MARSTINE L SUPVSR OF ANIMAL CONTROL OFFICERS 73992.00

DEL RIO, JOSE A SUPVSR OF ANIMAL CONTROL OFFICERS 89124.00

DIAKHATE, MAMADOU OPERATIONS MANAGER - ANIMAL CONTROL 85008.00

DRAKE, TAURUS L ANIMAL CONTROL INSPECTOR 70644.00

EDGECOMBE, CHERYL K ANIMAL CONTROL INSPECTOR 58644.00

FELTON, DONIELLA M TEACHER 50236.20

FRANCO, ARTURO ANIMAL CONTROL OFFICER 45924.00

GARNER, LINDSAY VETERINARIAN 88080.00

HAMILTON, ARTHUR ANIMAL SHELTER MANAGER 68220.00

HOLCOMB, ALLEN R ANIMAL CONTROL INSPECTOR 77520.00

HOWARD, MARYANN J ANIMAL CONTROL INSPECTOR 64392.00

HUBBS, CARLA A SUPERVISING VETERINARY TECHNICIAN 62820.00

JACOB, VIVISH SUPVSR OF ANIMAL CARE AIDES 84420.00

KELLER, AUDREY A VETERINARIAN 124428.00

LOZANO, RENE P ANIMAL CONTROL OFFICER 67464.00

MARTINIS, JENNIFER ANIMAL CONTROL OFFICER 41832.00

RUSSELL, SUSAN J TEACHER 136508.41

SCHLUETER, JENNIFER L EXEC ADMINISTRATIVE ASST II 59976.00

SILVA, YVONNE ANIMAL CONTROL OFFICER 41832.00

WALTERS, MICHELLE TEACHER 73596.60

YAMAJI, PETER S VETERINARIAN 128136.00

# Cast operators

## Static Cast

A static\_cast is used to return a variable or expression as a different type. Static casts are

* Often a cast that would occur automatically
* Usually a replacement for a C-style cast
* Sometimes not necessary, but used to provide visibility to a convesion

### Example 1 – static\_cast

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. unsigned ui = 0U;
6. unsigned long ul = 123UL;
7. int i = 0;
8. bool b;
9. float f = 3;
10. // i = rand() % f; // Error
11. i = rand() % static\_cast<int>(f);
12. b = i < ul; // Warning
13. b = static\_cast<unsigned long>(i) < ul;
14. f = NULL; // Warning
15. f = static\_cast<float>(NULL);
16. enum color { red, white, blue };
17. // Assign int value to enum variable
18. // color hue = 1; // Error
19. color hue = static\_cast<color>(1);
20. // Assign enum variable to int type
21. i = hue; // OK
22. // Assign enum value to int type
23. ui = white; // OK
24. int\* ptrI;
25. // ptrI = &f; // Error
26. // ptrI = static\_cast<int\*>(&f); // Error
27. ptrI = reinterpret\_cast<int\*>(&f); // OK
28. }

## Const Cast

A const\_cast is used to add or remove *constness* to an expression. Note, removing constness from a “pointed to” value may result in undefined behavior.

### Example 2 – const\_cast

1. #include <string>
2. #include <iostream>
3. using namespace std;
4. void foo(string& s) { cout << s << endl; }
5. void goo(const string& s) { cout << s << endl; }
6. void delta(string& s) { s = "I am changed"; }
7. int main()
8. {
9. string s1 = "I am volatile";
10. const string s2 = "I am const";
11. foo(s1);
12. // foo(s2); // Error: cannot convert
13. foo(const\_cast<string&>(s2));
14. goo(s1);
15. goo(s2);
16. cout << "Before: s1 = " << s1 << endl;
17. cout << "Before: s2 = " << s2 << endl;
18. delta(s1);
19. delta(const\_cast<string&>(s2));
20. cout << "After: s1 = " << s1 << endl;
21. cout << "After: s2 = " << s2 << endl;
22. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

I am volatile

I am const

I am volatile

I am const

Before: s1 = I am volatile

Before: s2 = I am const

After: s1 = I am changed

After: s2 = I am changed

## Reinterpret Cast

A reinterpret\_cast is used to cast one type to another. It is most commonly used to treat one pointer type as another pointer type, or to treat a pointer type as an integer type and vice versa. Note, this case type may be unsafe and to use it effectively, the sizes of the casted value and the casted type should match.

### Example 3 – reinterpret\_cast

1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main()
5. {
6. int i = 5;
7. double d = 3.14;
8. cout << d << ' ' << static\_cast<int>(d) << ' '
9. << \*(reinterpret\_cast<int\*>(&d)) << endl;
10. cout << "&i=" << &i << ' ' << reinterpret\_cast<long long>(&i)
11. << endl;
12. // write int and double out to a binary file
13. ofstream fout("binaryfile");
14. //fout.write(static\_cast<char\*>(&i), sizeof(i)); // ERROR
15. fout.write(reinterpret\_cast<char\*>(&i), sizeof(i));
16. fout.write(reinterpret\_cast<char\*>(&d), sizeof(d));
17. fout.close();
18. ifstream fin("binaryfile");
19. fin.read(reinterpret\_cast<char\*>(&i), sizeof(i));
20. fin.read(reinterpret\_cast<char\*>(&d), sizeof(d));
21. fin.close();
22. cout << i << ' ' << d << endl;
23. }

\*\*\*\*\*\* Output (Code::Blocks vers 20.03) \*\*\*\*\*\*

3.14 3 1374389535

&i=0x61fe0c 6422028

5 3.14

## Dynamic Cast

A dynamic\_cast is used with inheritance to cast a base class pointer or reference to a derived class pointer or references. This is called downcasting. The dynamic\_cast is used in conjunction with polymorphism to allow the user to execute a member function of a derived class using a pointer or reference of the base class. In order for this to succeed, the base class must be polymorphic (contains a virtual function).

Reference: http://www.bogotobogo.com/cplusplus/upcasting\_downcasting.php

### Example 4 – dynamic\_cast

1. #include <iostream>
2. using namespace std;
3. class Animal
4. {
5. public:
6. virtual ~Animal() {} // Initiate polymorphism via virtual dtor
7. };
8. class Cat : public Animal
9. {
10. };
11. class Dog : public Animal
12. {
13. public:
14. void bark() const
15. {
16. cout << "woof\n";
17. }
18. };
19. int main()
20. {
21. Cat fred;
22. Dog fido;
23. fido.bark();
24. Animal\* ptrAnimal;
25. Dog\* ptrDog;
26. // Call the bark function using an Animal\*
27. ptrAnimal = &fido;
28. // ptrAnimal -> bark();
29. // Call the bark function using an Animal\* cast to a Dog\*
30. dynamic\_cast<Dog\*>(ptrAnimal) -> bark();
31. // Testing a dynamic cast
32. ptrDog = dynamic\_cast<Dog\*>(&fido);
33. cout << "&fido=" << &fido << " ptrDog = " << ptrDog << endl;
34. ptrDog = dynamic\_cast<Dog\*>(&fred);
35. cout << "&fred=" << &fred << " ptrDog = " << ptrDog << endl;
36. }

\*\*\*\*\* Output \*\*\*\*\*

woof

woof

&fido=0x61fdf0 ptrDog = 0x61fdf0

&fred=0x61fdf8 ptrDog = 0

# The string class

The **string** class, part of the C++ “standard”, is an instantiation of the **basic\_string** template for type char, or

typedef basic\_string<char> string;

Access to the class requires the inclusion of the <string> header file.

## Constructors

string();  
string(const char\* str);  
string(const str& str);

string (const string& str, size\_t pos, size\_t len=npos);

string (const char\* s, size\_t n);

string (size\_t n, char c);

template <class InputIterator>

string(InputIterator first,InputIterator last);

### Example 1 – string constructors

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. // default constructor
7. string s1;
9. // c-string argument
10. string s2a("second string");
11. string s2b = "second string";
12. string s2c{"second string"};
14. // copy constructor
15. string s3a(s2a);
16. string s3b = s2a;
18. // substring
19. string s4(s2a,4,5);
21. // c-string buffer
22. string s5a("fifth string",5);
23. string s5b("fifth string",25);
25. // fill constructor
26. string s6(10,'A');
28. // range using iterators
29. string s7(s2a.begin(),s2a.begin()+3);
31. // initializer list
32. string s8{'W','o','w','!'};
34. // move constructor
35. string temp("Bye bye");
36. string s9(move(temp));
37. cout << "s1=" << s1 << endl;
38. cout << "s2a=" << s2a << endl;
39. cout << "s2b=" << s2b << endl;
40. cout << "s2c=" << s2c << endl;
41. cout << "s3a=" << s3a << endl;
42. cout << "s3b=" << s3b << endl;
43. cout << "s4=" << s4 << endl;
44. cout << "s5a=" << s5a << endl;
45. cout << "s5b=" << s5b << endl;
46. cout << "s6=" << s6 << endl;
47. cout << "s7=" << s7 << endl;
48. cout << "s8=" << s8 << endl;
49. cout << "s9=" << s9 << endl;
50. cout << "temp=" << temp << endl;
51. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

s1=

s2a=second string

s2b=second string

s2c=second string

s3a=second string

s3b=second string

s4=nd st

s5a=fifth

s5b=fifth stringBye byes1=

s6=AAAAAAAAAA

s7=sec

s8=Wow!

s9=Bye bye

temp=

## Iterator Functions

#### begin

Returns an iterator pointing to the first character of the string

iterator begin() noexcept[[4]](#footnote-4);

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the character beyond the end of the string

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last character of the string

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the character in front of the first character of the string

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a const iterator pointing to the first character of the string

const\_iterator begin() const noexcept;

#### cend

Returns a const iterator pointing to the character beyond the end of the string

const\_iterator end() const noexcept;

#### crbegin

Returns a const reverse iterator pointing to the last character of the string

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a const reverse iterator pointing to the character in front of the first character of the string

const\_reverse\_iterator rend() const noexcept;

### Example 2 – string iterator functions

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string s1("Have a nice day.");
7. // cout << s1.begin() << endl; ERROR
8. cout << \*s1.begin() << endl;
9. cout << \*(s1.begin()+2) << endl;
10. cout << '/' << \*s1.end() << '/' << endl; **// error on MSVC++**
11. cout << \*(s1.end()-4) << endl;
12. cout << "\*s1.rbegin()=" << \*s1.rbegin() << '/' << endl;
13. cout << "\*(s1.rbegin()+1)=" << \*(s1.rbegin()+1) << '/' << endl;
14. cout << "\*(s1.rbegin()-1)=" << \*(s1.rbegin()-1) << '/' << endl;
15. cout << endl;
16. cout << "\*s1.rend()=" << \*s1.rend() << '/' << endl;
17. cout << "\*(s1.rend()+1)=" << \*(s1.rend()+1) << '/' << endl;
18. cout << "\*(s1.rend()-1)=" << \*(s1.rend()-1) << '/' << endl;
19. cout << endl;
20. \*s1.begin() = 'Z';
21. cout << s1 << endl;
22. // \*s1.cbegin() = 'Z'; ERROR
23. for (string::const\_iterator it = s1.begin(); it != s1.end(); ++it)
24. cout << \*it << '/';
25. cout << endl;
26. for (string::const\_reverse\_iterator it = s1.rbegin(); it != s1.rend(); ++it)
27. cout << \*it << '/';
28. }

\*\*\*\*\*\* Code::Blocks on Windows \*\*\*\*\*\*

H

v

/ /

d

\*s1.rbegin()=./

\*(s1.rbegin()+1)=y/

\*(s1.rbegin()-1)= /

\*s1.rend()= /

\*(s1.rend()+1)= /

\*(s1.rend()-1)=H/

Zave a nice day.

Z/a/v/e/ /a/ /n/i/c/e/ /d/a/y/./

./y/a/d/ /e/c/i/n/ /a/ /e/v/a/Z/

\*\*\*\*\*\* Linux g++ 4.1.2

H

v

//

d

\*s1.rbegin()=./

\*(s1.rbegin()+1)=y/

\*(s1.rbegin()-1)=/

\*s1.rend()=/

\*(s1.rend()+1)=/

\*(s1.rend()-1)=H/

Zave a nice day.

Z/a/v/e/ /a/ /n/i/c/e/ /d/a/y/./

./y/a/d/ /e/c/i/n/ /a/ /e/v/a/Z/

## Capacity Functions

#### size

Returns the length of a string

size\_t size() const noexcept;

#### length

Returns the length of a string

size\_t length() const noexcept;

#### capacity

Returns the size allocated for the string

size\_t capacity() const noexcept;

#### max\_size

Returns the maximum size for any string

size\_t max\_size() const noexcept;

#### reserve

Change the string’s capacity. The function reserves ***at least the size*** requested.

void reserve(size\_t n = 0);

#### clear

Erases a string. Size becomes 0

void clear() noexcept;

#### resize

Resizes a string to n characters

void resize (size\_t n);

void resize (size\_t n, char c);

#### empty

Returns whether the size is empty

bool empty() const noexcept;

#### shrink\_to\_fit

Changes the capacity to the size of the string

void shrink\_to\_fit();

### Example 3 – capacity functions

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string s1 = "Have an exceptionally nice day";
7. cout << s1 << endl;
8. cout << "s1.size()=" << s1.size() << endl;
9. cout << "s1.capacity()=" << s1.capacity() << endl;
10. cout << "s1.max\_size()=" << s1.max\_size() << endl << endl;
11. s1.reserve(50);
12. cout << s1 << endl;
13. cout << "s1.size()=" << s1.size() << endl;
14. cout << "s1.capacity()=" << s1.capacity() << endl << endl;
15. s1.reserve(5);
16. cout << s1 << endl;
17. cout << "s1.size()=" << s1.size() << endl;
18. cout << "s1.capacity()=" << s1.capacity() << endl << endl;
19. s1.reserve(75);
20. cout << s1 << endl;
21. cout << "s1.size()=" << s1.size() << endl;
22. cout << "s1.capacity()=" << s1.capacity() << endl << endl;
23. s1.resize(19);
24. cout << s1 << endl;
25. cout << "s1.size()=" << s1.size() << endl;
26. cout << "s1.capacity()=" << s1.capacity() << endl << endl;
27. s1.shrink\_to\_fit();
28. cout << s1 << endl;
29. cout << "s1.size()=" << s1.size() << endl;
30. cout << "s1.capacity()=" << s1.capacity() << endl << endl;
31. s1.clear();
32. cout << s1 << endl;
33. cout << "s1.size()=" << s1.size() << endl;
34. cout << "s1.capacity()=" << s1.capacity() << endl << endl;
35. cout << boolalpha << s1.empty() << endl;
36. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Have an exceptionally nice day

s1.size()=30

s1.capacity()=30

s1.max\_size()=1073741820

Have an exceptionally nice day

s1.size()=30

s1.capacity()=60

Have an exceptionally nice day

s1.size()=30

s1.capacity()=30

Have an exceptionally nice day

s1.size()=30

s1.capacity()=75

Have an exceptional

s1.size()=19

s1.capacity()=75

Have an exceptional

s1.size()=19

s1.capacity()=19

s1.size()=0

s1.capacity()=19

true

## Access Functions

#### at

Returns character at position

char& at (size\_t pos);

const char& at (size\_t pos) const;

#### back

Returns last character in string

char& back();

const char& back() const;

#### front

Returns first character in string

char& front();

const char& front() const;

### Example 4 – access functions

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string s = "Have a nice day";
7. cout <<s.front() << s.at(3) << s.back() << endl;
8. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Hey

## Modifier Functions

#### assign

Assigns a new value to a string

string& assign(const string& str);

string& assign(const string& str,size\_t subpos, size\_t sublen = npos);

string& assign(const char\* s);

string& assign(const char\* s, size\_t n);

string& assign(size\_t n, char c);

#### append

Appends a value to a string

string& append(const string& str);

string& append(const string& str,size\_t subpos, size\_t sublen = npos);

string& append(const char\* s);

string& append(const char\* s, size\_t n);

string& append(size\_t n, char c);

#### erase

Erases part of a string

string& erase(size\_t pos = 0, size\_t len = npos);

iterator erase(const\_iterator p);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts characters into a string at a specified position

string& insert(size\_t pos, const string& str);

string& insert(size\_t pos, const string& str, size\_t subpos,

size\_t sublen = npos);

string& insert(size\_t pos,const char\* s);

string& insert(size\_t pos, const char\* s, size\_t n);

string& insert(size\_t pos, size\_t n, char c);

iterator insert(const\_iterator p, size\_t n, char c);

iterator insert(const\_iterator p, char c);

#### push\_back

Appends a char to the end of a string

void push\_back (char c);

**replace**

Replaces part of a string with new contents

string& replace(size\_t pos, size\_t len, const string& str);

string& replace(const\_iterator i1, const\_iterator i2, const string& str);

string& replace(size\_t pos, size\_t len, const string& str,size\_t subpos,

size\_t sublen = npos);

string& replace(size\_t pos, size\_t len, const char\* s);

string& replace(const\_iterator i1, const\_iterator i2, const char\* s);

string& replace(size\_t pos, size\_t len, const char\* s, size\_t n);

string& replace(const\_iterator i1,const\_iterator i2, const char\* s,

size\_t n);

string& replace(size\_t pos, size\_t len, size\_t n, char c);

string& replace(const\_iterator i1, const\_iterator i2, size\_t n,char c);

**swap**

Swaps two strings

void swap (string& str);

#### pop\_back

Erases the last character of a string

void pop\_back();

### Example 5 – modifier functions

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string s1 = "Have a nice day";
7. string s2, s3, s4, s5, s6;
8. s2.assign(s1);
9. s3.assign(s1,7,4);
10. s4.assign("Hey");
11. s5.assign(s1.c\_str(),3);
12. s6.assign(5,'x');
13. cout << s2 << endl << s3 << endl << s4 << endl << s5
14. << endl << s6 << endl << endl;
15. s2.append(s1);
16. s3.append(s1,7,4);
17. s4.append("Hey");
18. s5.append(s1.c\_str(),3);
19. s6.append(5,'x');
20. cout << s2 << endl << s3 << endl << s4 << endl << s5
21. << endl << s6 << endl << endl;
22. s2.erase();
23. s3.erase(4);
24. s4.erase(3,2);
25. s5.erase(s5.begin()+1,s5.begin()+4);
26. cout << s2 << endl << s3 << endl << s4 << endl << s5
27. << endl << endl;
28. s2 = s1;
29. s3 = "very ";
30. s2.insert(7,s3);
31. cout << s2 << endl;
32. s2.insert(s2.find("nice"),"VERY ");
33. cout << s2 << endl << endl;
34. s2.push\_back('!');
35. cout << s2 << endl << endl;
36. s2.replace(s2.find("very VERY"),string("excellent").size(),
37. "excellent");
38. cout << s2 << endl << endl;
39. s2.replace(s2.find("excellent"),
40. string("excellent nice").size(),
41. "swell");
42. cout << s2 << endl << endl;
43. s1.swap(s2);
44. cout << s1 << endl << s2 << endl << endl;
45. s1.pop\_back();
46. cout << s1 << endl;
47. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Have a nice day

nice

Hey

Hav

xxxxx

Have a nice dayHave a nice day

nicenice

HeyHey

HavHav

xxxxxxxxxx

nice

Heyy

Hav

Have a very nice day

Have a very nice day

Have a very nice day!

Have a excellent nice day!

Have a swell day!

Have a swell day!

Have a nice day

Have a swell day

## Search Functions

#### find

Locates text in a string.  Returns npos if not found

size\_t find(const string& str, size\_t pos = 0) const;  
size\_t find(const char\* s, size\_t pos = 0) const;

size\_t find(const char\* s, size\_t pos size\_type n) const;      
size\_t find(char c, size\_t pos = 0) const;

#### find\_first\_of

Locates first occurrence of text in a string

size\_t find\_first\_of (const string& str, size\_t pos = 0) const noexcept;

size\_t find\_first\_of (const char\* s, size\_t pos = 0) const;

size\_t find\_first\_of (const char\* s, size\_t pos, size\_t n) const;

size\_t find\_first\_of (char c, size\_t pos = 0) const noexcept;

#### find\_last\_of

Locates last occurrence of text in a string

size\_t find\_last\_of (const string& str, size\_t pos = 0) const noexcept;

size\_t find\_last\_of (const char\* s, size\_t pos = 0) const;

size\_t find\_last\_of (const char\* s, size\_t pos, size\_t n) const;

size\_t find\_last\_of (char c, size\_t pos = 0) const noexcept;

#### find\_first\_not\_of

Locates first occurrence of any characters not in a string

size\_t find\_first\_not\_of (const string& str, size\_t pos = 0) const noexcept;

size\_t find\_first\_not\_of (const char\* s, size\_t pos = 0) const;

size\_t find\_first\_not\_of (const char\* s, size\_t pos, size\_t n) const;

size\_t find\_first\_not\_of (char c, size\_t pos = 0) const noexcept;

#### find\_last\_not\_of

Locates last occurrence of any characters not in a string

size\_t find\_last\_not\_of (const string& str, size\_t pos = 0) const noexcept;

size\_t find\_last\_not\_of (const char\* s, size\_t pos = 0) const;

size\_t find\_last\_not\_of (const char\* s, size\_t pos, size\_t n) const;

size\_t find\_last\_not\_of (char c, size\_t pos = 0) const noexcept;

#### rfind

Locates text in a string.

size\_t rfind(const string& str, size\_t pos = 0) const;  
size\_t rfind(const char\* s, size\_t pos = 0) const;

size\_t rfind(const char\* s, size\_t pos size\_type n) const;      
size\_t rfind(char c, size\_t pos = 0) const;

### Example 6 – search functions

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string hand = "Have a nice day";
7. string nice = "nice";
8. string Nice = "Nice";
9. cout << hand.find(nice) << endl;
10. cout << hand.find("nice") << endl;
11. cout << hand.find(Nice) << endl;
12. cout << nice << " is "
13. << (hand.find(nice) == string::npos ? "not " : "")
14. << "present" << endl;
15. cout << Nice << " is "
16. << (hand.find(Nice) == string::npos ? "not " : "")
17. << "present" << endl << endl;
18. // Find the first 'a'
19. cout << hand.find('a') << endl;
20. // Find the second 'a'
21. cout << hand.find('a',hand.find('a')+1) << endl;
22. // Find the third 'a'
23. cout << hand.find('a',hand.find('a',hand.find('a')+1)+1)
24. << endl;
25. // Find the last 'a'
26. cout << hand.rfind('a') << endl << endl;
27. cout << hand.find\_first\_of(nice) << endl;
28. cout << hand.find\_first\_of("abcde") << endl;
29. cout << hand.find\_first\_of('v') << endl;
30. cout << hand.find\_first\_of('v',3) << endl << endl;
31. cout << hand.find\_last\_of("abcde") << endl;
32. cout << hand.find\_first\_not\_of("abcdefghijklmnopqrstuvwxyz")
33. << endl;
34. cout << hand.find\_last\_not\_of("abcdefghijklmnopqrstuvwxyz")
35. << endl;
36. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

7

7

4294967295

nice is present

Nice is not present

1

5

13

13

3

1

2

4294967295

13

0

11

## Operation Functions

#### c\_str

Returns the null-terminated char array contents of the string. The c\_str and data functions return the same value.

const char\* c\_str() const noexcept;

#### compare

Compares two strings or a string and a cstring

int compare (const string& str) const noexcept;

int compare (size\_t pos, size\_t len, const string& str) const;

int compare (size\_t pos, size\_t len, const string& str,

size\_t subpos, size\_t sublen = npos) const;

int compare (const char\* s) const;

int compare (size\_t pos, size\_t len, const char\* s) const;

int compare (size\_t pos, size\_t len, const char\* s, size\_t n) const;

#### copy

Copies part of a string into a char array. A null is not added to the char array.

size\_t copy (char\* s, size\_t len, size\_t pos = 0) const;

#### substr

Returns part of a string

string substr (size\_t pos = 0, size\_t len = npos) const;

### Example 7 – operation functions

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string Hand = "Have a nice day";
7. string hand = "have a nice day";
8. string Have = "Have";
9. string nice = "nice";
10. cout << Hand.compare(Hand) << endl;
11. cout << Hand.compare(hand) << endl;
12. cout << Hand.compare(Have) << endl;
13. cout << string("ABC").compare("ABD") << endl;
14. cout << Hand.compare(7,4,nice) << endl;
15. cout << Hand.compare(1,string::npos,hand,1,string::npos)<<endl;
16. cout << Have.compare(Have.c\_str()) << endl << endl;
17. char array[16];
18. Hand.copy(array,4);
19. cout << array << endl;
20. cout << Hand.substr(5) << endl;
21. cout << Hand.substr(5,6) << endl;
22. }

\*\*\*\*\*\* Code::Blocks on Windows \*\*\*\*\*\*

0

-1

11

-1

0

0

0

Have╠eÆ╓Ç@

a nice day

a nice

\*\*\*\*\*\* Linux g++ 4.1.2 \*\*\*\*\*\*

0

-1

11

-1

0

0

0

Have

a nice day

a nice

\*\*\*\*\*\* Linux g++ 6.4.0 \*\*\*\*\*\*

0

-32

11

-1

0

0

0

Have

a nice day

a nice

## Non-member Functions

#### getline

Extracts from a input stream into a string

istream& getline (istream& is, string& str, char delim);

istream& getline (istream& is, string& str);

#### swap

Swaps two string

void swap (string& x, string& y);

### Example 8 – Non-member string functions

1. #include <iostream>
2. #include <fstream>
3. #include <string>
4. using namespace std;
5. int main()
6. {
7. string filename = \_\_FILE\_\_; // What’s this?
8. cout << "#1 " << filename << endl << endl;
9. ifstream fin(filename);
10. if (!fin)
11. {
12. cerr << "Unable to open " << filename << endl;
13. exit(1);
14. }
15. string buffer1, buffer2;
16. getline(fin,buffer1);
17. cout << "#2 buffer1 = " << buffer1 << endl;
18. getline(fin,buffer2);
19. cout << "#3 buffer2 = " << buffer2 << endl << endl;
20. swap(buffer1, buffer2);
21. cout << "#4 buffer1 = " << buffer1 << endl;
22. cout << "#5 buffer2 = " << buffer2 << endl << endl;
23. getline(fin,buffer1,**'<'**);
24. cout << "#6 buffer1 = " << buffer1 << '/' << endl;
25. getline(fin,buffer2);
26. cout << "#7 buffer2 = " << buffer2 << endl << endl;
27. getline(fin,buffer1,**'\_'**);
28. cout << "#8 " << buffer1 << endl << endl;
29. cout << "Life is good? " << boolalpha << fin.good() << endl;
30. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

#1 Z:\deanza\cis29\examples\string\_class\ex5-8.cpp

#2 buffer1 = #include <iostream>

#3 buffer2 = #include <fstream>

#4 buffer1 = #include <fstream>

#5 buffer2 = #include <iostream>

#6 buffer1 = #include /

#7 buffer2 = string>

#8 using namespace std;

int main()

{

string filename =

Life is good? true

## Member Operators

#### operator=

Assignment operator: assigns a new value to a string

string& operator= (const string& str);

string& operator= (const char\* s);

string& operator= (char c);

#### operator[]

Index operator: returns the character at the specified location

char& operator[] (size\_t pos);

const char& operator[] (size\_t pos) const;

#### operator+=

Plus-equal operator: concatenates text to an existing string

string& operator+= (const string& str);

string& operator+= (const char\* s);

string& operator+= (char c);

## Non-member Operators

#### operator+

Operator +: returns, by value, the result of two concatenated strings

string operator+ (const string& lhs, const string& rhs);

string operator+ (const string& lhs, const char\* rhs);

string operator+ (const char\* lhs, const string& rhs);

string operator+ (const string& lhs, char rhs);

string operator+ (char lhs, const string& rhs);

#### operator<<

Insertion operator: inserts a string into an output stream

ostream& operator<< (ostream& os, const string& str);

#### operator>>

Extraction operator: extracts a string from an input stream

istream& operator>> (istream& os, const string& str);

### Example 9 – Member and non-member string operators

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string s = "Have a nice day";
7. string s2, s3, s4;
8. s2 = s;
9. s3 = "Hey";
10. s4 = '!';
11. cout << s3[1] << endl;
12. s3[1] = 'a';
13. cout << s3[1] << endl << endl;
14. s2 += s4;
15. cout << s2 << endl;
16. s2 += '\*';
17. cout << s2 << endl << endl;
18. cout << s3 + s4 << endl;
19. cout << s3 + " you" << endl;
20. cout << "you " + s3 << endl;
21. cout << s3 + '?' << endl;
22. cout << '?' + s3 << endl;
23. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

e

a

Have a nice day!

Have a nice day!\*

Hay!

Hay you

you Hay

Hay?

?Hay

## Member Constant

#### npos

npos is a static member constant, equal to the maximum value for type, size\_t. It is used to indicate the location beyond the length of a string, or with use of a find function, the return value, not found.

static const size\_t npos = -1;

# The stringstream classes

The stringstream classes, istringstream, ostringstream, and stringstream, are instantiations of the basic\_string<> and the basic\_istream<> and basic\_ostream<> templates. These classes are the results of inheritance of class templates.

ios\_base

basic\_ios<>

ios / wios

basic\_streambuf<>

streambuf / wstreambuf

basic\_istream<>

istream / wistream

basic\_ostream<>

ostream / wostream

basic\_iostream<>

iostream / wiostream

basic\_istringstream<>

istringstream/wistringstream

basic\_stringstream<>

stringstream/wstringstream

basic\_ostringstream<>

ostringstream/wostringstream

basic\_stringbuf<>

stringbuf/wstringbuf

## The istringstream class

The istringstream class is used to read from a string buffer. A useful technique is to read a string into an istringstream buffer, then use that buffer to parse the input of the entire string.

### Example 1 – Using istringstream for parsing input

1. #include <sstream>
2. #include <iostream>
3. #include <string>
4. using namespace std;
5. int main()
6. {
7. string string1("Have a nice day.");
8. string buffer;
9. istringstream sin(string1);
10. // What is in the istringstream buffer?
11. cout << "sin.str()=" << sin.str() << endl;
12. // read from the istringstream buffer
13. while (sin >> buffer)
14. {
15. cout << buffer << endl;
16. }
17. // Let's get a new istringstream buffer
18. sin.str("Let's get a new istringstream buffer");
19. while (sin >> buffer)
20. {
21. cout << buffer << endl;
22. }
23. // Why didn't this work?
24. // after reading from the istringstream, what is the "state" of the stream?
25. cout << boolalpha << "sin.eof()=" << sin.eof() << endl;
26. cout << "sin.rdstate()=" << sin.rdstate()<< endl;
27. // clear the eofbit
28. sin.clear();
29. cout << boolalpha << "sin.eof()=" << sin.eof() << endl;
30. cout << "sin.rdstate()=" << sin.rdstate()<< endl;
31. cout << "sin.str()="<<sin.str()<<endl;
32. cout << "sin.tellg()=" << sin.tellg() << endl;
33. sin >> buffer;
34. cout << "buffer=" << buffer << " sin.gcount()=" << sin.gcount() << endl;
35. // Why is sin.gcount()= 0?
36. char cbuffer[32];
37. sin.seekg(0);
38. sin.read(cbuffer,4);
39. cout << "sin.gcount()=" << sin.gcount() << endl;
40. getline(sin,buffer);
41. cout << "buffer=" << buffer << " sin.gcount()=" << sin.gcount() << endl;
42. sin.seekg(0);
43. sin.get(cbuffer,sizeof(cbuffer));
44. cout << "cbuffer=" << buffer << " sin.gcount()=" << sin.gcount() << endl;
45. sin.seekg(0);
46. sin.getline(cbuffer,sizeof(cbuffer));
47. cout << "cbuffer=" << buffer << " sin.gcount()=" << sin.gcount() << endl;
48. }

### Example 2 - A practical example

1. #include <fstream>
2. #include <sstream>
3. #include <iostream>
4. #include <string>
5. using namespace std;
6. int main()
7. {
8. ifstream fin("c:/temp/short\_gettysburg\_address.txt");
9. string buffer, word;
10. istringstream sin;
11. while (!fin.eof())
12. {
13. getline(fin,buffer);
14. sin.str(buffer);
15. while (sin >> word)
16. {
17. cout << word << endl;
18. }
19. sin.clear();
20. }
21. }

**\*\*\*\*\*\* Output \*\*\*\*\*\***

Four score and seven years ago our fathers brought forth on this continent, a new

nation, conceived in Liberty, and dedicated to the proposition that all men are

created equal.

Four

score

and

seven

years

ago

our

fathers

brought

forth

on

this

continent,

a

new

nation,

conceived

in

…

## The ostringstream class

The ostringstream class is used to write into a string buffer. This is useful for composing a desired output format.

### Example 3 – Using ostringstream to compose output

1. // ostringstream example
2. #include <iostream>
3. #include <iomanip>
4. #include <sstream>
5. #include <string>
6. using namespace std;
7. void print(double number);
8. int main()
9. {
10. double array[] =
11. {1,1.2,1.23,1.234,123.45,1234.56,12345.67,1234.5678};
12. auto numberOfElements = sizeof(array) / sizeof(double);
13. for (auto element : array)
14. print(element);
15. }
16. void print(double number)
17. {
18. ostringstream sout;
19. cout << left << setw(12) << setprecision(8) << number;
20. sout << setprecision(2) << fixed << '$';
21. if (number > 1000)
22. {
23. int thousands = static\_cast<int>(number) / 1000;
24. sout << thousands << ',';
25. sout << number - thousands\*1000;
26. }
27. else
28. {
29. sout << number;
30. }
31. cout << right << setw(16) << sout.str() << endl;
32. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

1 $1.00

1.2 $1.20

1.23 $1.23

1.234 $1.23

123.45 $123.45

1234.56 $1,234.56

12345.67 $12,345.67

1234.5678 $1,234.57

## The stringstream class

### Example 4 – Using the stringstream class

1. #include <iostream>
2. #include <fstream>
3. #include <sstream>
4. #include <cctype>
5. using namespace std;
6. void rewriteScore(const string&);
7. int main()
8. {
9. ifstream fin("c:/temp/nfl\_scores.txt");
10. string buffer;
11. while (getline(fin,buffer) && buffer.size())
12. rewriteScore(buffer);
13. }
14. void rewriteScore(const string& buffer)
15. {
16. string temp, dummy, winner, loser;
17. int winnerScore, loserScore;
18. stringstream ss;
19. ss.str(buffer);
20. ss >> dummy >> winner >> temp;
21. winner += ' ';
22. winner += temp;
23. ss >> temp;
24. // look for a comma at the end of temp
25. if (isalpha(temp[0]) or temp == "49ers")
26. {
27. winner += ' ';
28. winner += temp;
29. ss >> temp;
30. }
31. // remove the comma from the winner's score string
32. temp.resize(temp.size()-1);
33. winnerScore = stoi(temp);
34. ss >> loser >> temp;
35. loser += ' ';
36. loser += temp;
37. ss >> temp;
38. if (isalpha(temp[0])or temp == "49ers")
39. {
40. loser += ' ';
41. loser += temp;
42. ss >> temp;
43. }
44. loserScore = stoi(temp);
45. ss.clear();
46. ss << winner << " over " << loser << ' ' << winnerScore <<
47. " to " << loserScore;
48. cout << ss.str() << endl;
49. }

Input File

8-Sep Denver Broncos 21, Carolina Panthers 20

11-Sep Green Bay Packers 27, Jacksonville Jaguars 23

11-Sep Baltimore Ravens 13, Buffalo Bills 7

11-Sep Cincinnati Bengals 23, New York Jets 22

11-Sep Houston Texans 23, Chicago Bears 14

11-Sep Minnesota Vikings 25, Tennessee Titans 16

11-Sep Philadelphia Eagles 29, Cleveland Browns 10

11-Sep Oakland Raiders 35, New Orleans Saints 34

11-Sep Kansas City Chiefs 33, San Diego Chargers 27

11-Sep Tampa Bay Buccaneers 31, Atlanta Falcons 24

11-Sep Seattle Seahawks 12, Miami Dolphins 10

11-Sep New York Giants 20, Dallas Cowboys 19

…

\*\*\*\*\*\* Output \*\*\*\*\*\*

Denver Broncos over Carolina Panthers 21 to 20

Green Bay Packers over Jacksonville Jaguars 27 to 23

Baltimore Ravens over Buffalo Bills 13 to 7

Cincinnati Bengals over New York Jets 23 to 22

Houston Texans over Chicago Bears 23 to 14

Minnesota Vikings over Tennessee Titans 25 to 16

Philadelphia Eagles over Cleveland Browns 29 to 10

Oakland Raiders over New Orleans Saints 35 to 34

Kansas City Chiefs over San Diego Chargers 33 to 27

Tampa Bay Buccaneers over Atlanta Falcons 31 to 24

Seattle Seahawks over Miami Dolphins 12 to 10

New York Giants over Dallas Cowboys 20 to 19

…

# I/O Manipulators

## std manipulators

Manipulators are functions or function-like operators that change the state of the I/O stream.

|  |  |  |
| --- | --- | --- |
| **Manipulator** | **I/O** | **Purpose** |
| **Independent Flags** |  | **Turns Setting On** |
| boolalpha | I/O | sets boolalpha flag |
| showbase | O | sets showbase flag |
| showpoint | O | sets showpoint flag |
| showpos | O | sets showpos flag |
| skipws | I | sets skipws flag |
| unitbuf | O | sets unitbuf flag |
| uppercase | O | sets uppercase flag |
| **Independent Flags** |  | **Turns Setting Off** |
| noboolalpha | I/O | clears boolalpha flag |
| noshowbase | O | clears showbase flag |
| noshowpoint | O | clears showpoint flag |
| noshowpos | O | clears showpos flag |
| noskipws | I | clears skipws flag |
| nounitbuf | O | clears unitbuf flag |
| nouppercase | O | clears uppercase flag |
| **Numeric Base Flags** |  |  |
| dec | I/O | sets dec flag for i/o of integers, clears oct,hex |
| hex | I/O | sets hex flag for i/o of integers, clears dec,oct |
| oct | I/O | sets oct flag for i/o of integers, clears dec,hex |
| hexfloat (C++11) | I/O | sets hexadecimal floating point formatting |
| defaultfloat (C++11) | I/O | clears the float field formats |
| **Floating Point Flags** |  |  |
| fixed | O | sets fixed flag |
| scientific | O | sets scientific flag |
| **Adjustment Flags** |  |  |
| internal | O | sets internal flag |
| left | O | sets left flag |
| right | O | sets right flag |
| **Input Only** |  |  |
| ws | I | extracts whitespace |
| **Output Only** |  |  |
| endl | O | inserts a newline **and flushes output stream** |
| ends | O | inserts a null |
| flush | O | flushes stream |
| **Parameterized Manipulators**(these require the ***iomanip*** header file) | | |
| resetiosflags(ios\_base::fmtflags mask) | I/O | clears format flags specified by mask |
| setbase(int base) | I/O | sets integer base (8, 10, or 16) |
| setfill(char\_type ch) | O | sets the fill character to ch |
| setiosflags(ios::base::fmtflags mask) | I/O | sets format flags to mask value |
| setprecision(int p) | O | sets precision of floating point numbers |
| setw(int w) | O | sets output field width to w |
| get\_money (C++11) | I | parses a monetary value |
| put\_money (C++11) | O | formats and outputs a monetary value |
| get\_time (C++11) | I | parses a date/time value |
| put\_time (C++11) | O | formats and outputs a date/time value |
| quoted (C++14) | I/O | Allows input/output of quoted text |

### Example 1 – Input/Output manipulators

The following examples illustrates the use of standard input/output manipulators.

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. void show\_fmtflags(ios\_base& stream);
5. int main()
6. {
7. // save the initial cout flags settings
8. ios\_base::fmtflags cout\_fmtflags = cout.flags();
9. // Display the cout flags
10. show\_fmtflags(cin);
11. show\_fmtflags(cout);
12. show\_fmtflags(cerr);
13. show\_fmtflags(clog);
14. cout << endl;
15. int x = 123;
16. // hex, oct, & dec manipulators
17. cout << "dec: x = " << dec << x << endl;
18. cout << "hex: x = " << hex << x << endl;
19. cout << "oct: x = " << oct << x << endl;
20. show\_fmtflags(cout);
21. cout << endl;
22. // Turn on showpos, uppercase, showpoint, left, hex
23. cout << setiosflags(ios::showpos|ios::uppercase|ios::showpoint|
24. ios::showbase|ios::left|ios::hex);
25. show\_fmtflags(cout);
26. cout << "x = " << x << endl << endl;
27. // Clear the oct flag
28. cout << resetiosflags(ios::oct) << "x = " << x << endl;
29. show\_fmtflags(cout);
30. cout << endl;
31. // Demonstrate the setfill and setw manipulators
32. cout << setfill('$') << setw(10) << "x = " << x << endl;
33. cout << "x = " << x << endl << endl;
34. // Reset cout's flags back to the original settings
35. cout.flags(cout\_fmtflags);
36. // Turn on hex
37. cout << hex << "x = " << x << endl;
38. show\_fmtflags(cout);
39. cout << endl;
40. // Turn on octal
41. cout << oct << "x = " << x << endl;
42. show\_fmtflags(cout);
43. cout << endl;
44. // Demonstrate setprecision
45. cout << setprecision(3) << 1.2 << ' ' << 3.14 << ' ' << 35
46. << ' ' << 3.14159 << endl;
47. // Demonstrate setprecision with showpoint
48. cout << showpoint << 1.2 << ' ' << 3.14 << ' ' << 35 << ' '
49. << 3.14159 << endl;
50. // Demonstrate showpos
51. cout << showpos << 1.2 << ' ' << 3.14 << ' ' << 35 << ' '
52. << 3.14159 << endl;
53. show\_fmtflags(cout);
54. cout << endl;
55. // Back to decimal
56. cout << dec << 1.2 << ' ' << 3.14 << ' ' << 35 << ' '
57. << 3.14159 << endl;
58. show\_fmtflags(cout);
59. cout << endl;
60. // What is truth?
61. cout << true << ' ' << boolalpha << true << endl;
62. show\_fmtflags(cout);
63. }
64. void show\_fmtflags(ios\_base& stream)
65. {
66. cout << (&stream == &cout ? "cout " : "");
67. cout << (&stream == &cerr ? "cerr " : "");
68. cout << (&stream == &clog ? "clog " : "");
69. cout << (&stream == &cin ? "cin " : "");
70. cout << "fmtflags set: ";
71. cout << (stream.flags() & ios::boolalpha ? "boolalpha " : "");
72. cout << (stream.flags() & ios::dec ? "dec " : "");
73. cout << (stream.flags() & ios::fixed ? "fixed " : "");
74. cout << (stream.flags() & ios::hex ? "hex " : "");
75. cout << (stream.flags() & ios::internal ? "internal " : "");
76. cout << (stream.flags() & ios::left ? "left " : "");
77. cout << (stream.flags() & ios::oct ? "oct " : "");
78. cout << (stream.flags() & ios::right ? "right " : "");
79. cout << (stream.flags() & ios::scientific ? "scientific " :"");
80. cout << (stream.flags() & ios::showbase ? "showbase " : "");
81. cout << (stream.flags() & ios::showpoint ? "showpoint " : "");
82. cout << (stream.flags() & ios::showpos ? "showpos " : "");
83. cout << (stream.flags() & ios::skipws ? "skipws " : "");
84. cout << (stream.flags() & ios::unitbuf ? "unitbuf " : "");
85. cout << (stream.flags() & ios::uppercase ? "uppercase " : "");
86. cout << endl;
87. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

cin fmtflags set: dec skipws

cout fmtflags set: dec skipws

cerr fmtflags set: dec skipws unitbuf

clog fmtflags set: dec skipws

dec: x = 123

hex: x = 7b

oct: x = 173

cout fmtflags set: oct skipws

cout fmtflags set: hex left oct showbase showpoint showpos skipws uppercase

x = +123

x = 0X7B

cout fmtflags set: hex left showbase showpoint showpos skipws uppercase

x = $$$$$$0X7B

x = 0X7B

x = 7b

cout fmtflags set: hex skipws

x = 173

cout fmtflags set: oct skipws

1.2 3.14 43 3.14

1.20 3.14 43 3.14

+1.20 +3.14 43 +3.14

cout fmtflags set: oct showpoint showpos skipws

+1.20 +3.14 +35 +3.14

cout fmtflags set: dec showpoint showpos skipws

+1 true

cout fmtflags set: boolalpha dec showpoint showpos skipws

### Example 2 - floatfield manipulators

1. #include <iostream>
2. #include <sstream>
3. using namespace std;
4. int main()
5. {
6. // save the cout format flags
7. ios\_base::fmtflags originalFlags = cout.flags();
8. double f = 1234.5678;
9. cout << "Default output: " << f << endl;
10. cout << "fixed: " << fixed << f << endl;
11. cout << "scientific: " << scientific << f << endl;
12. cout << "hexfloat: " << hexfloat << f << endl;
13. cout << "default: " << defaultfloat << f << endl;
14. // read hexfloat format into a double
15. istringstream("0x1P-1022") >> hexfloat >> f;
16. // display the double in default format
17. cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
18. f = 3.141592654;
19. cout << f << " as hexfloat: " << hexfloat << f << endl;
20. // save hexfloat value into a string
21. ostringstream sout;
22. sout << hexfloat << f << endl;
24. // save the hexfloat value into an input string buffer
25. istringstream sin;
26. sin.str(sout.str());
27. // read the input string buffer into a double
28. sin >> hexfloat >> f;
29. // display f
30. cout << f << endl;
31. // display f in original format
32. cout.flags(originalFlags);
33. cout << f << endl;
34. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

(MS Visual Studio 2017)

Default output: 1234.57

fixed: 1234.567800

scientific: 1.234568e+03

hexfloat: 0x1.34a457p+10

default: 1234.57

Parsing 0x1P-1022 as hex gives 2.22507e-308

3.14159 as hexfloat: 0x1.921fb5p+1

0x1.921fb5p+1

3.14159

(MacBook Xcode 8.33)

Default output: 1234.57

fixed: 1234.567800

scientific: 1.234568e+03

hexfloat: 0x1.34a456d5cfaadp+10

default: 1234.57

Parsing 0x1P-1022 as hex gives 2.22507e-308

3.14159 as hexfloat: 0x1.921fb5452455p+1

0x1.921fb5452455p+1

3.14159

(gnu compiler output)

Default output: 1234.57

fixed: 1234.567800

scientific: 1.234568e+03

hexfloat: 0x1.34a456d5cfaadp+10

default: 1234.57

Parsing 0x1P-1022 as hex gives **0🡨 This looks like a bug**

3.14159 as hexfloat: 0x1.921fb5452455p+1

**0x0p+0🡨 This looks like a bug**

**0 🡨 This looks like a bug**

### Example 3 - get\_money manipulator

1. #include <iostream>
2. #include <sstream>
3. #include <string>
4. #include <iomanip>
5. #include <locale>
6. using namespace std;
7. int main()
8. {
9. istringstream in("$1,234.56 2.22 USD 3.33");
10. locale mylocale("");
11. in.imbue(mylocale);
12. long double v1, v2;
13. string v3;
14. in >> std::get\_money(v1) >> std::get\_money(v2) >> std::get\_money(v3, true);
15. cout << quoted(in.str()) << " parsed as: " << v1 << ' ' << v2 << ' ' << v3 << endl;
16. in.str("$125 .99");
17. in.seekg(0);
18. in >> std::get\_money(v1) >> std::get\_money(v2);
19. cout << quoted(in.str()) << " parsed as: " << v1 << ' ' << v2 << endl;
20. }

(MS Visual Studio 2017, MS Visual Studio 2019 and gnu compiler on Linux and MacBook)

(Does not run on gnu compilers on a PC – 1/28/20)

"$1,234.56 2.22 USD 3.33" parsed as: 123456 222 333

"$125 .99" parsed as: 12500 99

Note: the quoted() function required compilation with ***std=c++14***.

### Example 4 - put\_money manipulator

1. #include <iostream>
2. #include <iomanip>
3. #include <string>
4. using namespace std;
5. int main()
6. {
7. long double value = 123.45;
8. std::cout.imbue(std::locale(""));
9. cout << put\_money(value) << endl;
10. cout << put\_money(value, true) << endl; // use international representation
11. cout << showbase;
12. cout << put\_money(value) << endl;
13. cout << put\_money(value, true) << endl; // use international representation
14. string stringValue = "2345.67";
15. cout << noshowbase;
16. cout << put\_money(stringValue) << endl;
17. cout << put\_money(stringValue, true) << endl; // use international representation
18. cout << showbase;
19. cout << put\_money(stringValue) << endl;
20. cout << put\_money(stringValue, true) << endl; // use international representation
21. }

(MS Visual Studio 2017 / MS Visual Studio 2019)

1.23

1.23

$1.23

USD1.23

23.45

23.45

$23.45

USD23.45

(g++ 7.2.0 on Linux)

1.23

1.23

$1.23

USD 1.23

23.45

23.45

$23.45

USD 23.45

(g++ on MacBook)

1.23

1.23

$1.23

USD 1.23

23.45

23.45

$23.45

USD 23.45

This does not work on Windows gnu compilers – 1/28/20

### Example 5 - get\_time and put\_time manipulators

1. #include <iostream> // cin, cout
2. #include <iomanip> // get\_time
3. #include <ctime> // struct tm
4. #include <string>
5. #include <sstream>
6. #include <locale>
7. using namespace std;
8. int main()
9. {
10. struct tm when;
11. const string monthName[] = { "January","February","March","April","May","June",
12. "July","August","September","October","November","December" };
13. cout << "Please, enter the time (hh:mn): ";
14. cin >> get\_time(&when, "%R"); // extract time (24H format)
15. if (cin.fail()) cout << "Error reading time\n";
16. else {
17. cout << "The time entered is: ";
18. cout << when.tm\_hour << " hours and " << when.tm\_min << " minutes\n";
19. }
20. cout << "Please, enter the date (mm/dd/yy): ";
21. cin >> get\_time(&when, "%D"); // extract date
22. if (cin.fail()) cout << "Error reading date\n";
23. else {
24. cout << "The date entered is: ";
25. cout << monthName[when.tm\_mon] << " " << when.tm\_mday << ", ";
26. cout << when.tm\_year + 1900 << endl;
27. }
28. tm t = {};
29. istringstream ss("2011-February-18 23:12:34");
31. // imbue cout with the “local” locale
32. cout.imbue(locale(""));
34. // get the datetime from an istringstream
35. ss >> get\_time(&t, "%Y-%b-%d %H:%M:%S");
36. if (ss.fail()) {
37. cout << "Parse failed" << endl;
38. }
39. else {
40. cout << put\_time(&t, "%c") << endl;
41. cout << put\_time(&t, "%D %r") << endl;
42. }
43. }

(MS Visual Studio 2017

Please, enter the time (hh:mn): **16:57 🡨 User input**

The time entered is: 16 hours and 57 minutes

Please, enter the date (mm/dd/yy): **09/08/17 🡨 User input**

The date entered is: September 8, 2017

2/18/2011 11:12:34 PM

02/18/11 11:12:34 PM

(g++ on MacBook)

Please, enter the time (hh:mn): **14:22 🡨 User input**

The time entered is: 14 hours and 22 minutes

Please, enter the date (mm/dd/yy): **11/15/17 🡨 User input**

The date entered is: November 15, 2017

Sun Feb 18 23:12:34 2011

02/18/11 11:12:34 PM

(Cygwin compiler on Windows – g++ 7.4.0): not working 1/28/20

Please, enter the time (hh:mn): **16:57 🡨 User input**

The time entered is: 16 hours and 57 minutes

Please, enter the date (mm/dd/yy): **09/08/17 🡨 User input**

The date entered is: September 8, 1917

### Example 6 – quoted manipulator

1. #include <iostream>
2. #include <iomanip>
3. #include <sstream>
4. #include <string>
5. using namespace std;
6. int main()
7. {
8. stringstream ss1;
9. stringstream ss2;
10. string in = "String with spaces, and embedded \"quotes\" too";
11. string out;
12. // write in to a stringstream object
13. ss1 << in;
14. cout << "read in [" << in << "]\n"
15. << "stored as [" << ss1.str() << "]\n";
16. // read from a stringstream object
17. ss1 >> out;
18. cout << "written out [" << out << "]\n";
19. cout << "----------------------------------------------" << endl;
20. // write in to a stringstream object using quoted
21. ss2 << quoted(in);
22. cout << "read in [" << in << "]\n"
23. << "stored as [" << ss2.str() << "]\n";
25. // read from a stringstream object using quoted
26. ss2 >> quoted(out);
27. cout << "written out [" << out << "]\n";
28. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

read in [String with spaces, and embedded "quotes" too]

stored as [String with spaces, and embedded "quotes" too]

written out [String]

----------------------------------------------------

read in [String with spaces, and embedded "quotes" too]

stored as ["String with spaces, and embedded \"quotes\" too"]

written out [String with spaces, and embedded "quotes" too]

## Write your own manipulator

### Example 7 - Write your own manipulator with no arguments

Technique: use a function with a stream argument, passed by reference and return the same stream.

#include <iostream>

using namespace std;

ostream& spaces3(ostream& os)

{

return os << " ";

}

int main()

{

cout <<"Some" <<spaces3 <<"text" <<endl;

}

\*\*\*\*\*\* Output \*\*\*\*\*\*

Some text

### Example 8 - Write your own manipulator with one or more arguments

The following example illustrates a technique for creating a parameterized manipulator by creating a class with the same name.

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. struct prec
5. {
6. prec(int x) : prec\_(x) {}
7. int prec\_;
8. };
9. ostream& operator<<(ostream& out, const prec& obj)
10. {
11. out.precision(obj.prec\_);
12. return out;
13. }
14. class dollar
15. {
16. double amount;
17. public:
18. dollar(double amt) : amount(amt) {}
19. friend ostream& operator<<(ostream& out, const dollar& obj);
20. };
21. ostream& operator<<(ostream& out, const dollar& obj)
22. {
23. out << '$';
24. auto currentFlags = out.flags() ;
25. auto currentPrecision = out.precision();
26. out << fixed << setprecision(2) << obj.amount;
27. out.flags(currentFlags);
28. out.precision(currentPrecision);
29. return out;
30. }
31. class format
32. {
33. int width;
34. int decimalPlaces;
35. public:
36. format(int arg1, int arg2 = -1);
37. friend ostream& operator<<(ostream& out, const format& obj);
38. };
39. format::format(int arg1, int arg2)
40. : width(arg2 == -1 ? 0: arg1),
41. decimalPlaces(arg2 == -1 ? arg1: arg2)
42. { }
43. ostream& operator<<(ostream& out, const format& obj)
44. {
45. out << fixed << setw(obj.width)
46. << setprecision(obj.decimalPlaces);
47. return out;
48. }
49. int main( )
50. {
51. double pi = 3.141592654;
52. cout << prec(4) << pi << endl;
53. cout << prec(6) << pi << endl;
54. cout << prec(0) << pi << endl;
55. cout << dollar(pi) << endl;
56. cout << pi << endl;
57. cout << "-----------------" << endl;
58. // print with a width of 5 and 2 decimal places
59. cout << '/' << format(5,2) << pi << '/' << endl;
60. // print with a width of 12 and 4 decimal places
61. cout << '/' << format(12,4) << pi << '/' << endl;
62. // print with 1 decimal place
63. cout << '/' << format(1) << pi << '/' << endl;
64. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

3.142

3.14159

3

$3.14

3

-----------------

/ 3.14/

/ 3.1416/

/3.1/

# Data at the Bit Level

## Data Storage at the bit level

### Example 1 – Data storage

The following example shows how data is stored in stack memory. Eleven int variables are declared and initialized. The printVariableValueAndAddress() function displays the value of each variable in decimal and hexadecimal and its memory address in hexadecimal and decimal. The printMemoryContents() function displays the memory contents where the eleven variables are stored.

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. void printVariableValueAndAddress(char ch, const int&);
5. void printMemoryContents(unsigned char\*, unsigned char\*);
6. int main()
7. {
8. int a = 1;
9. int b = 12;
10. int c = 123;
11. int d = 1234;
12. int e = 12345;
13. int f = 123456;
14. int g = 1234567;
15. int h = 12345678;
16. int i = 123456789;
17. int j = 1234567890;
18. int k = 12345678901; // Warning!
19. cout << "Var Dec Value Hex Value Hex Address Dec Address"
20. << endl;
21. printVariableValueAndAddress('a', a);
22. printVariableValueAndAddress('b', b);
23. printVariableValueAndAddress('c', c);
24. printVariableValueAndAddress('d', d);
25. printVariableValueAndAddress('e', e);
26. printVariableValueAndAddress('f', f);
27. printVariableValueAndAddress('g', g);
28. printVariableValueAndAddress('h', h);
29. printVariableValueAndAddress('i', i);
30. printVariableValueAndAddress('j', j);
31. printVariableValueAndAddress('k', k);
32. unsigned char\* addr1 = reinterpret\_cast<unsigned char\*> (&k);
33. unsigned char\* addr2 = reinterpret\_cast<unsigned char\*> (&a)+3;
34. printMemoryContents(addr1, addr2);
35. }
36. void printVariableValueAndAddress(char ch, const int& i)
37. {
38. cout << left << showbase;
39. cout << ch << " = " << setw(11) << i << ' ' << setw(12) << hex
40. << i << dec << &i << " " << reinterpret\_cast<long> (&i)
41. << endl;
42. }
43. void printMemoryContents(unsigned char\* addr1,unsigned char\* addr2)
44. {
45. cout << endl << "Addresses / Contents" << endl;
46. cout << hex << setfill('0') << noshowbase << right;
47. for (unsigned char\* addr = addr1; addr <= addr2; addr += 4)
48. {
49. // Memory addresses are stored in a width of 8 and
50. // only the 8 least significant digits are displayed
51. cout << setw(8) << reinterpret\_cast<long>(addr)%0x100000000
52. << ' ';
53. }
54. cout << noshowbase << left << endl;
55. int i = 1;
56. for (unsigned char\* addr = addr1; addr <= addr2; ++addr, ++i)
57. {
58. cout << setw(2) << static\_cast<int> (\*addr);
59. if (i && i % 4 == 0)
60. {
61. cout << ' ';
62. }
63. }
64. cout << endl;
65. }

\*\*\*\*\*\* Output – NetBeans 8.2 (Windows) \*\*\*\*\*\*

Var Dec Value Hex Value Hex Address Dec Address

a = 1 0x1 0xffffcbec 4294953964

b = 12 0xc 0xffffcbe8 4294953960

c = 123 0x7b 0xffffcbe4 4294953956

d = 1234 0x4d2 0xffffcbe0 4294953952

e = 12345 0x3039 0xffffcbdc 4294953948

f = 123456 0x1e240 0xffffcbd8 4294953944

g = 1234567 0x12d687 0xffffcbd4 4294953940

h = 12345678 0xbc614e 0xffffcbd0 4294953936

i = 123456789 0x75bcd15 0xffffcbcc 4294953932

j = 1234567890 0x499602d2 0xffffcbc8 4294953928

k = -539222987 0xdfdc1c35 0xffffcbc4 4294953924

Addresses / Contents

ffffcbc4 ffffcbc8 ffffcbcc ffffcbd0 ffffcbd4 ffffcbd8 ffffcbdc ffffcbe0 ffffcbe4 ffffcbe8 ffffcbec

351cdcdf d2209649 15cd5b70 4e61bc00 87d61200 40e21000 39300000 d2400000 7b000000 c0000000 10000000

\*\*\*\*\*\* Output – Code::Blocks (Windows) \*\*\*\*\*\*

Var Dec Value Hex Value Hex Address Dec Address

a = 1 0x1 0x6dfef4 7208692

b = 12 0xc 0x6dfef0 7208688

c = 123 0x7b 0x6dfeec 7208684

d = 1234 0x4d2 0x6dfee8 7208680

e = 12345 0x3039 0x6dfee4 7208676

f = 123456 0x1e240 0x6dfee0 7208672

g = 1234567 0x12d687 0x6dfedc 7208668

h = 12345678 0xbc614e 0x6dfed8 7208664

i = 123456789 0x75bcd15 0x6dfed4 7208660

j = 1234567890 0x499602d2 0x6dfed0 7208656

k = -539222987 0xdfdc1c35 0x6dfecc 7208652

Addresses / Contents

006dfecc 006dfed0 006dfed4 006dfed8 006dfedc 006dfee0 006dfee4 006dfee8 006dfeec 006dfef0 006dfef4

351cdcdf d2209649 15cd5b70 4e61bc00 87d61200 40e21000 39300000 d2400000 7b000000 c0000000 10000000

Note: memory addresses are only 3 bytes in size

\*\*\*\*\*\* Output – Linux g++ version 7.3.0 \*\*\*\*\*\*

Var Dec Value Hex Value Hex Address Dec Address

a = 1 0x1 0x7ffc74fb91ac 140722271130028

b = 12 0xc 0x7ffc74fb91a8 140722271130024

c = 123 0x7b 0x7ffc74fb91a4 140722271130020

d = 1234 0x4d2 0x7ffc74fb91a0 140722271130016

e = 12345 0x3039 0x7ffc74fb919c 140722271130012

f = 123456 0x1e240 0x7ffc74fb9198 140722271130008

g = 1234567 0x12d687 0x7ffc74fb9194 140722271130004

h = 12345678 0xbc614e 0x7ffc74fb9190 140722271130000

i = 123456789 0x75bcd15 0x7ffc74fb918c 140722271129996

j = 1234567890 0x499602d2 0x7ffc74fb9188 140722271129992

k = -539222987 0xdfdc1c35 0x7ffc74fb9184 140722271129988

Addresses / Contents

74fb9184 74fb9188 74fb918c 74fb9190 74fb9194 74fb9198 74fb919c 74fb91a0 74fb91a4 74fb91a8 74fb91ac

351cdcdf d2209649 15cd5b70 4e61bc00 87d61200 40e21000 39300000 d2400000 7b000000 c0000000 10000000

Note: memory addresses are 6 bytes in size

\*\*\*\*\*\* Output – MS Visual Studio 2017 \*\*\*\*\*\*

Var Dec Value Hex Value Hex Address Dec Address

a = 1 0x1 001CFACC 1899212

b = 12 0xc 001CFAC0 1899200

c = 123 0x7b 001CFAB4 1899188

d = 1234 0x4d2 001CFAA8 1899176

e = 12345 0x3039 001CFA9C 1899164

f = 123456 0x1e240 001CFA90 1899152

g = 1234567 0x12d687 001CFA84 1899140

h = 12345678 0xbc614e 001CFA78 1899128

i = 123456789 0x75bcd15 001CFA6C 1899116

j = 1234567890 0x499602d2 001CFA60 1899104

k = -539222987 0xdfdc1c35 001CFA54 1899092

Addresses / Contents

001cfa54 001cfa58 001cfa5c 001cfa60 001cfa64 001cfa68 001cfa6c 001cfa70 001cfa74 001cfa78 001cfa7c 001cfa80 001cfa84 001cfa88 001cfa8c 001cfa90 001cfa94 001cfa98 001cfa9c 001cfaa0 001cfaa4 001cfaa8 001cfaac 001cfab0 001cfab4 001cfab8 001cfabc 001cfac0 001cfac4 001cfac8 001cfacc

351cdcdf cccccccc cccccccc d2209649 cccccccc cccccccc 15cd5b70 cccccccc cccccccc 4e61bc00 cccccccc cccccccc 87d61200 cccccccc cccccccc 40e21000 cccccccc cccccccc 39300000 cccccccc cccccccc d2400000 cccccccc cccccccc 7b000000 cccccccc cccccccc c0000000 cccccccc cccccccc 10000000

Note: memory addresses are 3 bytes in size. The memory address display is in uppercase with no base indicators. The storage locations use 12 bytes of memory (8 bytes of padding).

### Example 2 – Storage of negative ints

This example shows how negative int values are stored in memory.

1. #include <iostream>
2. #include <iomanip>
3. #include <string>
4. #include <cmath>
5. using namespace std;
6. void print(char ch, const int&);
7. string printIntInBinary(int arg);
8. int power(int pow);
9. int main()
10. {
11. int a = 1;
12. int b = -1;
13. int c = 255;
14. int d = -255;
15. int e = 256;
16. int f = -256;
17. int g = 0x7fffffff;
18. int h = -0x7fffffff;
19. int i = 0x1a2b3c4d;
20. int j = -0x1a2b3c4d;
21. int k = 0xffffffff;
22. int l = 0x00ff00ff;
23. int m = -0x00ff00ff;
24. cout << "Var Dec Value Hex Value Binary Value (4 bytes / 32 bits)" << endl;
25. print('a', a);
26. print('b', b);
27. print('c', c);
28. print('d', d);
29. print('e', e);
30. print('f', f);
31. print('g', g);
32. print('h', h);
33. print('i', i);
34. print('j', j);
35. print('k', k);
36. print('l', l);
37. print('m', m);
38. }
39. void print(char ch, const int& i)
40. {
41. cout << showbase;
42. cout << setfill(' ') << ch << " = " << setw(11) << i << ' '
43. << setw(10) << hex
44. << i << dec << " " << printIntInBinary(i)
45. << endl;
46. }
47. string printIntInBinary(int arg)
48. {
49. string value;
50. for (auto i = 31; i >= 0; --i)
51. {
52. if (arg & power(i))
53. value += '1';
54. else
55. value += '0';
56. if (i%8 == 0)
57. value += ' ';
58. }
59. return value;
60. }
61. int power(int pow)
62. {
63. int value = 1;
64. for (auto i = 0; i < pow; ++i)
65. value \*= 2;
66. return value;
67. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Var Dec Value Hex Value Binary Value (4 bytes / 32 bits)

a = 1 0x1 00000000 00000000 00000000 00000001

b = -1 0xffffffff 11111111 11111111 11111111 11111111

c = 255 0xff 00000000 00000000 00000000 11111111

d = -255 0xffffff01 11111111 11111111 11111111 00000001

e = 256 0x100 00000000 00000000 00000001 00000000

f = -256 0xffffff00 11111111 11111111 11111111 00000000

g = 2147483647 0x7fffffff 01111111 11111111 11111111 11111111

h = -2147483647 0x80000001 10000000 00000000 00000000 00000001

i = 439041101 0x1a2b3c4d 00011010 00101011 00111100 01001101

j = -439041101 0xe5d4c3b3 11100101 11010100 11000011 10110011

k = -1 0xffffffff 11111111 11111111 11111111 11111111

l = 16711935 0xff00ff 00000000 11111111 00000000 11111111

m = -16711935 0xff00ff01 11111111 00000000 11111111 00000001

To convert a positive int value to negative, “flip” the bits and add 1. This is the two’s complement method of storing negative int values. For negative int values, the high order (left-most) bit is 1. This is called the sign bit.

### Example 3 – Non-primitive data at the bit level

1. #include <iostream>
2. #include <iomanip>
3. #include <climits>
4. using namespace std;
5. long address2long(const void\* address);
6. unsigned powerOf2(int exp);
7. template <typename T> void printBits(T type);
8. struct Struct1
9. {
10. char c1;
11. char c2;
12. short s1;
13. int i;
14. };
15. ostream& operator<<(ostream& out, const Struct1& d)
16. {
17. out << "Address: " << address2long(&d) << " " << sizeof(d) << " bytes" << endl;
18. out << " &c1: " << address2long(&d.c1);
19. printBits(d.c1);
20. out << " &c2: " << address2long(&d.c2);
21. printBits(d.c2);
22. out << " &s1: " << address2long(&d.s1);
23. printBits(d.s1);
24. out << " &i: " << address2long(&d.i);
25. printBits(d.i);
26. return out;
27. }
28. struct Struct2
29. {
30. char c1;
31. int i;
32. char c2;
33. short s1;
34. };
35. ostream& operator<<(ostream& out, const Struct2& d)
36. {
37. out << "Address: " << address2long(&d) << " " << sizeof(d) << " bytes" << endl;
38. out << " &c1: " << address2long(&d.c1);
39. printBits(d.c1);
40. out << " &i: " << address2long(&d.i);
41. printBits(d.i);
42. out << " &c2: " << address2long(&d.c2);
43. printBits(d.c2);
44. out << " &s1: " << address2long(&d.s1);
45. printBits(d.s1);
46. return out;
47. }
48. int main()
49. {
50. Struct1 s1 = {'A','B',static\_cast<short>(13),55};
51. printBits(s1);
52. cout << endl;
53. Struct2 s2 = {'A',55,'B',static\_cast<short>(13)};
54. printBits(s2);
55. }
56. long address2long(const void\* address)
57. {
58. return reinterpret\_cast<long>(address);
59. }
60. template <typename T>
61. void printBits(T t)
62. {
63. cout << setw(6) << t << " ";
64. unsigned mask;
65. unsigned char\* ptr;
66. for (size\_t i = 0; i < sizeof(T); i++)
67. {
68. // Advance ptr each byte of the argument
69. ptr = reinterpret\_cast<unsigned char\*>(&t) + i;
70. // Print the contents of the byte
71. for (int i = 7; i >= 0; --i)
72. {
73. mask = powerOf2(i);
74. cout << (\*ptr & mask ? 1 : 0);
75. }
76. cout << " ";
77. }
78. cout << endl;
79. }
80. unsigned powerOf2(int exp)
81. {
82. unsigned value = 1;
83. for (int i = 0; i < exp; ++i)
84. {
85. value \*= 2;
86. }
87. return value;
88. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Address: 4294953904 8 bytes

&c1: 4294953904 A 01000001

&c2: 4294953905 B 01000010

&s1: 4294953906 13 00001101 00000000

&i: 4294953908 55 00110111 00000000 00000000 00000000

01000001 01000010 00001101 00000000 00110111 00000000 00000000 00000000

Address: 4294953936 12 bytes

&c1: 4294953936 A 01000001

&i: 4294953940 55 00110111 00000000 00000000 00000000

&c2: 4294953944 B 01000010

&s1: 4294953946 13 00001101 00000000

01000001 00000000 00000000 00000000 00110111 00000000 00000000 00000000 01000010 00000000 00001101 00000000

Note: The bit representation may vary between big endian and little endian platforms. The contents of “padded” bytes may also vary.

## Bitwise Operators

|  |  |
| --- | --- |
| **Operator** | **Symbol Name** |
| & | and |
| | | or |
| ^ | exclusive or |
| ~ | not (a unary operator) |
| << | left-shift |
| >> | right-shift |
| &= | and assignment |
| |= | or assignment |
| ^= | exclusive or assignment |
| <<= | left shift assignment |
| >>= | right shift assignment |

### & operator

The bitwise and operator returns a 1 only when both bits being compared are 1. For example:

10101110 & 00101010 🡺 00101010

### | operator

The bitwise or operator returns a 1 only when either bits being compared are 1. For example:

10101110 | 00101010 🡺 10101110

### ^ operator

The bitwise exclusive or operator returns a 1 only when either, but not both, bits being compared are 1. For example:

10101110 | 00101010 🡺 10000100

### ~ operator

The bitwise not, or complement operator is a unary bitwise operator. It returns a 1 when the bit is 0 and returns a 0 when the bit is 1. For example:

~10101110 🡺 01010001

### << operator

The bitwise left-shift operator shifts bits to left the number of positions as the right-hand operand. Bits on the right are filled with zeros. Bits on the left are lost. The left-shift operator may be used to perform multiplication by integer powers of two. For example,

10101110 << 2 🡺 …10 10111000

### >> operator

The bitwise right-shift operator shifts bits to right the number of positions as the right-hand operand. Bits on the left are filled with zeros. Bits on the right are lost. The left-shift operator may be used to perform division by integer powers of two. For example,

10101110 >> 2 🡺 00101011 10…

### The bitwise assignment operators

The bitwise assignment operators: &=, |=, ^=, <<=, and >>= perform the implied operation and assign the resultant value to the left-hand argument.

### Example 3 – Bitwise operators

1. #include <iostream>
2. #include <iomanip>
3. #include <climits>
4. using namespace std;
5. unsigned powerOf2(int exp);
6. template <typename T> void printBits(T type);
7. int main()
8. {
9. unsigned char a = 77;
10. unsigned char b = 20;
11. cout << " a =";printBits(a);
12. cout << " b =";printBits(b);
13. cout << "a&b =";printBits(a&b);
14. cout << "a|b =";printBits(a|b);
15. cout << "a^b =";printBits(a^b);
16. cout << " ~a =";printBits(~a);
17. cout << "a<<1=";printBits(a<<1);
18. cout << "a<<2=";printBits(a<<2);
19. cout << "a<<8=";printBits(a<<8);
20. cout << "a<<9=";printBits(a<<9);
21. cout << "a>>1=";printBits(a>>1);
22. cout << "a>>2=";printBits(a>>2);
23. cout << "a>>9=";printBits(a>>9);
24. }
25. template <typename T>
26. void printBits(T t)
27. {
28. unsigned mask;
29. unsigned char\* ptr;
30. cout << setw(5) << static\_cast<int>(t) << " ";
31. for (size\_t i = 0; i < sizeof(T); i++)
32. {
33. // Advance ptr each byte of the argument
34. ptr = reinterpret\_cast<unsigned char\*>(&t) + i;
35. // Print the contents of the byte
36. for (int i = 7; i >= 0; --i)
37. {
38. mask = powerOf2(i);
39. cout << (\*ptr & mask ? 1 : 0);
40. }
41. cout << " ";
42. }
43. cout << endl;
44. }
45. unsigned powerOf2(int exp)
46. {
47. unsigned value = 1;
48. for (int i = 0; i < exp; ++i)
49. {
50. value \*= 2;
51. }
52. return value;
53. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

a = 77 01001101

b = 20 00010100

a&b = 4 00000100 00000000 00000000 00000000

a|b = 93 01011101 00000000 00000000 00000000

a^b = 89 01011001 00000000 00000000 00000000

~a = -78 10110010 11111111 11111111 11111111

a<<1= 154 10011010 00000000 00000000 00000000

a<<2= 308 00110100 00000001 00000000 00000000

a<<8=19712 00000000 01001101 00000000 00000000

a<<9=39424 00000000 10011010 00000000 00000000

a>>1= 38 00100110 00000000 00000000 00000000

a>>2= 19 00010011 00000000 00000000 00000000

a>>9= 0 00000000 00000000 00000000 00000000

### Bitwise Techniques

#### Turn a bit on

Use the or assignment bitwise operator to turn a bit on. If the bit is already turned on, the operation has no effect.

Integer\_value |= bit

#### Turn a bit off

Use the and assignment with the not bitwise operators to turn a bit off. If the bit is already turned on, the operation has no effect.

Integer\_value &= ~bit

#### Toggle a bit

Use the exclusive or assignment operator to turn a bit off.

Integer\_value ^= bit

#### Test a bit

Use the and operator to see if a bit is turned on.

Integer\_value & bit

### Example 4 – Bitwise operator techniques

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. unsigned powerOf2(int exp);
5. template <typename T> void printBits(T type);
6. int main()
7. {
8. unsigned char a;
9. unsigned char b;
10. // turn a bit on
11. a = 34;
12. cout << " a =";printBits(a);
13. b= 4;
14. cout << " b =";printBits(b);
15. cout << "a|=b"; printBits(a|=b); cout << endl;
16. // turn a bit off
17. a = 34;
18. cout << " a =";printBits(a);
19. b= 2;
20. cout << " b =";printBits(b);
21. cout << "a&~b"; printBits(a&~b); cout << endl;
22. // toggle a bit
23. a = 34;
24. cout << " a =";printBits(a);
25. b= 66;
26. cout << " b =";printBits(b);
27. cout << "a^=b"; printBits(a^=b); cout << endl;
28. // test to see if a bit is turned on
29. a = 34;
30. cout << boolalpha;
31. cout << " a =";printBits(a);
32. cout << " 2 =";printBits(2);
33. cout << "a & 2 = " << static\_cast<bool>(a & 2) << endl;
34. cout << " 4 =";printBits(4);
35. cout << "a & 4 = " << static\_cast<bool>(a & 4) << endl;
36. }
37. template <typename T>
38. void printBits(T t)
39. {
40. unsigned mask;
41. unsigned char\* ptr;
42. cout << setw(5) << static\_cast<int>(t) << " ";
43. for (size\_t i = 0; i < sizeof(T); i++)
44. {
45. // Advance ptr each byte of the argument
46. ptr = reinterpret\_cast<unsigned char\*>(&t) + i;
47. // Print the contents of the byte
48. for (int i = 7; i >= 0; --i)
49. {
50. mask = powerOf2(i);
51. cout << (\*ptr & mask ? 1 : 0);
52. }
53. cout << " ";
54. }
55. cout << endl;
56. }
57. unsigned powerOf2(int exp)
58. {
59. unsigned value = 1;
60. for (int i = 0; i < exp; ++i)
61. {
62. value \*= 2;
63. }
64. return value;
65. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

a = 34 00100010

b = 4 00000100

a|=b 38 00100110

a = 34 00100010

b = 2 00000010

a&~b 32 00100000 00000000 00000000 00000000

a = 34 00100010

b = 66 01000010

a^=b 96 01100000

a = 34 00100010

2 = 2 00000010 00000000 00000000 00000000

a & 2 = true

4 = 4 00000100 00000000 00000000 00000000

a & 4 = false

## Practical Applications

The following examples illustrate working with binary data.

### Example 5 – Bitwise operator techniques

The following example shows how to extract each nibble (4 bits) from a byte.

1. #include <iostream>
2. #include <iomanip>
3. #include <cstdlib>
4. using namespace std;
5. string uchar2binary(unsigned char);
6. unsigned char powerOf2(unsigned char exp);
7. int main()
8. {
9. unsigned char x;
10. cout << showbase;
11. for (auto i = 0; i < 10; i++)
12. {
13. x = rand() % 255; // 0-255
14. cout << dec << setw(5) << static\_cast<int>(x) // decimal
15. << hex << setw(8) << static\_cast<int>(x) // hex
16. << setw(12) << uchar2binary(x) // binary
17. << setw(12) << uchar2binary(x >> 4) // first nibble
18. << setw(12) << uchar2binary(x & 0xf) // second nibble
19. << endl;
20. }
21. }
22. // returns unsigned char as a binary string
23. string uchar2binary(unsigned char arg)
24. {
25. string out;
26. unsigned char mask;
27. for (auto i = 7; i >= 0; --i)
28. {
29. mask = powerOf2(i);
30. out += (arg & mask ? '1' : '0');
31. }
32. return out;
33. }
34. // returns 2 raised to exp power
35. unsigned char powerOf2(unsigned char exp)
36. {
37. unsigned char value = 1u;
38. for (auto i = 0u; i < exp; ++i)
39. {
40. value \*= 2u;
41. }
42. return value;
43. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

41 0x29 00101001 00000010 00001001

107 0x6b 01101011 00000110 00001011

214 0xd6 11010110 00001101 00000110

235 0xeb 11101011 00001110 00001011

44 0x2c 00101100 00000010 00001100

169 0xa9 10101001 00001010 00001001

3 0x3 00000011 00000000 00000011

33 0x21 00100001 00000010 00000001

187 0xbb 10111011 00001011 00001011

239 0xef 11101111 00001110 00001111

##### Explanation

This example makes use of an unsigned char to limit the perspective to just one byte.

Line 19: The first nibble is extracted by shifting the 8 bits to the right by 4. The right shift bitwise operator returns an int (32 bits). That int result is then passed to the uchar2binary function which is converted to an unsigned char.

Line 20: The second nibble is extracted using a 0xf mask with the bitwise ***and*** operator. Keep in mind that mask is 00001111 in binary. With this mask the second nibble bits will be replicated.

### Example 6 – Extracting specified bits from a byte

The following example shows how to extract a specified number of bits from a byte. The user specifies the starting bit and the number of bits to extract. The default argument, numbits = 8, allows the user to specify only a starting bit. In that case the function will return all bits from the starting bit to the end of the byte. The problem is solved using the getBitsFromByte function. Note that a byte is returned, not just the specified number of bits. This is because there is no built-in type for less than 8 bits.

1. #include <iostream>
2. #include <iomanip>
3. #include <cstdlib>
4. using namespace std;
5. string uchar2binary(unsigned char);
6. unsigned char powerOf2(unsigned char exp);
7. unsigned char getBitsFromByte(unsigned char byte,
8. unsigned startingBit, unsigned numbits = 8u);
9. int main()
10. {
11. unsigned char x, sb, nb;
12. cout << showbase;
13. for (auto i = 0; i < 15; i++)
14. {
15. x = rand() % 255; // unsigned char 0-255
16. sb = rand() % 8; // starting bit 0-7
17. nb = rand() % (9-sb); // number of bits 0-8
18. cout << dec << setw(4) << static\_cast<int>(x) // decimal
19. << hex << setw(6) << static\_cast<int>(x) // hex
20. << setw(10) << uchar2binary(x); // binary
21. cout << dec;
22. if (nb)
23. {
24. cout << " sb=" << static\_cast<int>(sb) // start bit
25. << " nb=" << static\_cast<int>(nb) // num bits
26. << " => "
27. << uchar2binary(getBitsFromByte(x,sb,nb));
28. }
29. else
30. {
31. cout << " sb=" << static\_cast<int>(sb) // start bit
32. << " "
33. << " => "
34. << uchar2binary(getBitsFromByte(x,sb));
35. }
36. cout << endl;
37. }
38. }
39. // returns unsigned char as a binary string
40. string uchar2binary(unsigned char arg)
41. {
42. string out;
43. unsigned char mask;
44. for (auto i = 7; i >= 0; --i)
45. {
46. mask = powerOf2(static\_cast<unsigned char>(i));
47. out += (arg & mask ? '1' : '0');
48. }
49. return out;
50. }
51. unsigned char powerOf2(unsigned char exp)
52. {
53. unsigned char value = 1u;
54. for (auto i = 0u; i < exp; ++i)
55. value <<= 1;
56. return value;
57. }
58. // assume bits are numbered 0-7, left-to-right
59. unsigned char getBitsFromByte(unsigned char byte,
60. unsigned startingBit, unsigned numBits)
61. {
62. byte <<= startingBit; // shift bits left
63. byte >>= (8 - numBits); // shift bits right
64. return byte;
65. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

41 0x29 00101001 sb=3 nb=4 => 00000100

235 0xeb 11101011 sb=1 nb=4 => 00001101

3 0x3 00000011 sb=6 nb=1 => 00000001

239 0xef 11101111 sb=1 nb=1 => 00000001

76 0x4c 01001100 sb=3 nb=1 => 00000000

236 0xec 11101100 sb=3 nb=2 => 00000001

237 0xed 11101101 sb=4 nb=1 => 00000001

69 0x45 01000101 sb=6 => 01000000

37 0x25 00100101 sb=6 => 01000000

101 0x65 01100101 sb=6 nb=2 => 00000001

92 0x5c 01011100 sb=6 nb=2 => 00000000

63 0x3f 00111111 sb=5 => 11100000

167 0xa7 10100111 sb=3 nb=1 => 00000000

204 0xcc 11001100 sb=7 nb=1 => 00000000

212 0xd4 11010100 sb=5 nb=1 => 00000001

##### Explanation

As in the previous example, type unsigned char is used to represent the byte. The method for extraction in the getBitsFromByte function involves shifting the unwanted bits off the left side of the byte, then off the right side of the byte.

Line 68: Bits to the left of the starting bit are shifted off the left side. Notice the use of the <<= operator instead of the << operator. In both cases, an int (32 bits) is returned. With the << operator the unspecified bits to the left of the starting bit would be shift into the next byte. They would then reappear in a right shift. By using <<= the result of the left shift is stored into the unsigned char (one byte), so the is no problem in the subsequent right shift.

Line 69: Bits a shifted to the right so that exactly the number of bits desired are remaining, right justified in the byte.

# Multiple Inheritance

Multiple inheritance permits a class to be derived from two (or more) other classes. In this way the derived classes inherits the members and properties of both (or more) base classes.

### Example 1 – Multiple Inheritance

1. // Easy multiple inheritance example
2. #include <iostream>
3. using namespace std;
4. class one
5. {
6. protected:
7. int a,b;
8. public:
9. one(int z,int y) : a(z), b(y)
10. { }
11. void show() const
12. {
13. cout << a << ' ' << b << endl;
14. }
15. };
16. class two
17. {
18. protected:
19. int c,d;
20. public:
21. two(int z,int y) : c(z), d(y)
22. { }
23. void show() const
24. {
25. cout << c << ' ' << d << endl;
26. }
27. };
28. class three : public one, public two
29. {
30. private:
31. int e;
32. public:
33. three(int,int,int,int,int);
34. void show() const
35. {
36. cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' << e << endl;
37. }
38. };
39. three::three(int a1, int a2, int a3, int a4, int a5)
40. : one(a1,a2),two(a3,a4), e(a5)
41. { }
42. int main()
43. {
44. one abc(5,7);
45. abc.show(); // prints 5 7
46. two def(8,9);
47. def.show(); // prints 8 9
48. three ghi(2,4,6,8,10);
49. ghi.show(); // prints 2 4 6 8 10
50. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

5 7

8 9

2 4 6 8 10

## Multiple Inheritance with Virtual Base Classes

The next example illustrates a more complicated inheritance situation. It models the relationship between types of quadrilaterals. This relationship is shown in the following figure:

quadrilateral

trapezoid

parallelogram

isosceles\_trapezoid

rectangle

rhombus

square

Note that the parallelogram class will be derived from the quadrilateral class, both the rhombus and rectangle classes will be derived from the parallelogram class. And the square is derived from both the rhombus and the rectangle classes. It's the square class that makes this multiple inheritance.

### Example 2 - Multiple Inheritance with Virtual Base classes

1. #include <iostream>
2. #include <cmath>
3. using namespace std;
4. class quadrilateral
5. {
6. protected:
7. double a,b,c,d;
8. public:
9. quadrilateral(double s1,double s2,double s3,double s4)
10. : a(s1), b(s2), c(s3), d(s4) {}
11. quadrilateral() : a(0), b(0), c(0), d(0) {}
12. void show()
13. {
14. cout << "quadrilateral: " << this << " sides " <<
15. a << ' ' << b << ' ' << c << ' ' << d << endl;
16. }
17. };
18. class trapezoid : public quadrilateral
19. {
20. public:
21. trapezoid(double base1, double base2, double leg1, double leg2)
22. : quadrilateral(base1,leg1,base2,leg2) {}
23. };
24. class isosceles\_trapezoid : public trapezoid
25. {
26. public:
27. isosceles\_trapezoid(double base1, double base2, double leg)
28. : trapezoid(base1,leg,base2,leg) {}
29. };
30. class parallelogram : public quadrilateral
31. {
32. protected:
33. int angle;
34. public:
35. parallelogram(double s1,double s2, int ang)
36. : quadrilateral(s1,s2,s1,s2), angle(ang)
37. { }
38. parallelogram() : angle(0) { }
39. void show\_angles(void)
40. {
41. cout << "angles = " << angle << ' ' << (180-angle) << endl;
42. }
43. };
44. class rectangle : virtual public parallelogram
45. {
46. public:
47. rectangle(double base, double height)
48. : parallelogram(base,height,90) {}
49. rectangle() {}
50. };
51. class rhombus: virtual public parallelogram
52. {
53. public:
54. rhombus(double side,int ang) : parallelogram(side,side,ang) {}
55. rhombus() {}
56. };
57. class square : public rhombus,public rectangle
58. {
59. public:
60. square(double side) : parallelogram(side,side,90) {}
61. };
62. int main(void)
63. {
64. quadrilateral q1(1,2,3,4);
65. q1.show();
66. trapezoid q2(22,13,8,15);
67. q2.show();
68. isosceles\_trapezoid q3(18,8,13);
69. q3.show();
70. parallelogram q4(4,3,45);
71. q4.show();
72. q4.show\_angles();
73. rectangle q5(4,3);
74. q5.show();
75. q5.show\_angles();
76. rhombus q6(5,45);
77. q6.show();
78. q6.show\_angles();
79. cout << endl;
80. square q7(5);
81. q7.show();
82. q7.show\_angles();
83. }

# Exception Handling

Exception handling in C++ is methodology used to deal with error conditions that usually results in a program failure. These methods are implemented using:

* the try, throw, and catch keywords in C++
* exception class types
* functions, such as set\_terminate() and set\_unexpected() found in the header files, <stdexcept> and <exception>.

They allow the user to detect specific errors and control the program exit or recover and continue the program. Exception handling is used to handle exceptional situations, not to replace typical error messages.

Exception handling is a standard feature of the language.

Exception handling is designed to provide an alternate means of handling a code situation which would normally abend or abort a program. This mechanism allows transfer of control to another location where the error may be "handled". The transfer is specified by a throw expression. This expression allows the user to pass a value to the "handler". The "handler" catches the thrown expression by matching the type of the throw and deals with the problem as the author desires.

## When are Exception Handling Methods Appropriate?

As stated earlier, exception handling is for the exceptional situation, not the common. Consider the following application:

1. A training (relational) database, written in C++, is used to track student training, enrollments, class schedules, etc. How should the following situations be "handled"?
2. A student trying to enroll in a course, but doesn't have the prerequisites for it?
3. A student tries to enroll in a class that is full.
4. A student tries to enroll in a class that is identified as open, but is refused, because the class is really full.
5. A student tries to enroll in a class, but is already enrolled in another section of the same course.
6. A student tries to enroll in a course that is retired.
7. A student tries to enroll in a course in which there are no sections scheduled.
8. A student tries to enroll in a class section, but the schedule record containing the date and number of students is missing or defective.
9. A student tries to enroll in a course, but enters the incorrect course number.

## Previous Error Handling Methods

### The assert() Macro

A common way of dealing with error conditions is the use of the assert() macro. This macro is most often used in program development to insure that certain conditions are true during the execution of a program. If the assert condition is false, the program aborts displaying an assert diagnostic message. The assert() macro is declared in the <cassert> header file.

Note, the assert macro can be suppressed if the macro, NDEBUG is defined before the <cassert> header file is included, like this:

#define NDEBUG

#include <cassert>

The following example illustrates its use.

### Example 1 - assert

1. #include <iostream>
2. #include <cassert>
3. #include <cstdlib>
4. using namespace std;
5. class Fraction
6. {
7. int numer, denom;
8. public:
9. Fraction(int n = 0, int d = 1) : numer(n), denom(d)
10. {
11. assert(denom!=0); // make sure denom is not 0
12. }
13. friend ostream& operator<<(ostream& o, const Fraction& f)
14. {
15. return (o << f.numer << '/' << f.denom);
16. }
17. };
18. int main()
19. {
20. int i1, i2;
21. cout << "Enter two ints => ";
22. cin >> i1 >> i2;
23. if (cin.good())
24. {
25. Fraction f(i1,i2);
26. cout << f << endl;
27. }
28. else cerr << "Bad input\n";
29. cout << "\*\*\* End of Program \*\*\*\n";
30. }

\*\*\*\*\*\* Sample Run #1 \*\*\*\*\*\*

Enter two ints => **1 2**

1/2

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run #2 Code::Blocks \*\*\*\*\*\*

Enter two ints => 2 0

Assertion failed: denom!=0, file ex10-1.cpp, line 13

This application has requested the Runtime to terminate it in an unusual way.

Please contact the application's support team for more information.

\*\*\*\*\*\* Sample Run #2 Linux \*\*\*\*\*\*

Enter two ints => 2 0

assertion "denom!=0" failed: file "ex10-1.cpp", line 12, function: Fraction::Fraction(int, int)

Aborted (core dumped)

Note: this approach is used to catch a run‑time error. This is not a compile error. Of course, there are other ways of handling this problem. The programmer could put a check in main() to verify that the second int entered is non‑zero. Another approach is to put a check for a denom = 0 in the fraction constructor. The problem, of course, could be "handled" not by aborting the program, but maybe by asking the user for another denominator. This may not always be feasible, since the numerator may not always be supplied by the user. Maybe it's a problem that you want to recognize, but continue the program execution. This is known as *fault‑tolerant processing*.

### The longjmp() function

The longjmp() function is an ANSI C standard function that may be used the jump out of a function containing an error. longjmp() executes after a setjmp() function has be called to capture and store the task state of the program. longjmp() causes a "rollback" of the program state to a previous time. The advantage of this approach is that an error situation may be detected and corrected and the offending code may be rerun.

### Example 2 – longjump()

1. #include <iostream>
2. #include <cstdlib>
3. using namespace std;
4. #include <setjmp.h>
5. jmp\_buf jumper; // declare a jump buffer to save program state
6. class Fraction
7. {
8. int numer, denom;
9. public:
10. Fraction(int n = 0, int d = 1) : numer(n), denom(d)
11. {
12. cout << "Fraction " << this << " created" << endl;
13. if (d == 0)
14. longjmp(jumper,1); // make sure denom is not 0
15. }
16. ~Fraction()
17. {
18. cout << "~Fraction " << this << " destroyed" << endl;
19. }
20. friend ostream& operator<<(ostream& o, const Fraction& f)
21. {
22. return (o << f.numer << '/' << f.denom);
23. }
24. };
25. int main()
26. {
27. int i1, i2;
28. int state;
29. state = setjmp(jumper);
30. if (state != 0)
31. cout << "\*\* Go back in time with state " << state << endl;
32. cout << "Enter two ints => ";
33. cin >> i1 >> i2;
34. Fraction f(i1,i2);
35. cout << f << endl;
36. cout << "\*\*\* End of Program \*\*\*\n";
37. }

\*\*\*\*\*\* Sample Run 1 \*\*\*\*\*\*

Enter two ints => **2 3**

Fraction 0x6dfedc created

2/3

\*\*\* End of Program \*\*\*

~Fraction 0x6dfedc destroyed

\*\*\*\*\*\* Sample Run 2 \*\*\*\*\*\*

Enter two ints => **2 0**

Fraction 0x6dfedc created

\*\* Go back in time with state 1

Enter two ints => **2 3**

Fraction 0x6dfedc created

2/3

\*\*\* End of Program \*\*\*

~Fraction 0x6dfedc destroyed

What is wrong with this approach?

## Exception Handling Basics

### try, throw, and catch

Exception handling is, for the most part, accomplished using three keywords, try, throw, and catch. The try block contains code that may result in an error. The error is detected and you throw an exception‑expression. The handling is accomplished by a catch of the expression. The following example illustrates the technique.

### Example 3 – try, throw, catch

1. #include <iostream>
2. #include <cstdlib>
3. using namespace std;
4. class Fraction
5. {
6. int numer, denom;
7. public:
8. Fraction(int n = 0, int d = 1) : numer(n), denom(d)
9. {
10. cout << "Fraction " << this << " created" << endl;
11. if (d == 0)
12. throw("Error: denominator = 0");
13. }
14. ~Fraction()
15. {
16. cout << "~Fraction " << this << " destroyed" << endl;
17. }
18. friend ostream& operator<<(ostream& o, const Fraction& f)
19. {
20. return (o << f.numer << '/' << f.denom);
21. }
22. };
23. int main()
24. {
25. int i1, i2;
26. cout << "Enter two ints => ";
27. cin >> i1 >> i2;
28. try
29. {
30. Fraction f(i1,i2);
31. cout << f << endl;
32. }
33. catch (const string& errmsg)
34. {
35. cerr << errmsg <<endl;
36. }
37. cout << "\*\*\* End of Program \*\*\*\n";
38. }

\*\*\*\*\*\* Sample Run 1 \*\*\*\*\*\*

Enter two ints => **2 3**

Fraction 0x6dfedc created

2/3

~Fraction 0x6dfedc destroyed

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 2 on Code::Blocks \*\*\*\*\*\*

Enter two ints => **2 0**

Fraction 0x6dfedc created

terminate called after throwing an instance of 'char const\*'

This application has requested the Runtime to terminate it in an unusual way.

Please contact the application's support team for more information.

\*\*\*\*\*\* Sample Run 2 on Linux (voyager) \*\*\*\*\*\*

Enter two ints => **2 0**

Fraction 0x7fffc4477540 created

terminate called after throwing an instance of 'char const\*'

Aborted

* How is this program an improvement?
* Is there a problem?

### Example 4 – Handling a file open error

Here's an example of handling a file open error. The user is given the option to try again.

1. #include <fstream>
2. #include <iostream>
3. #include <string>
4. #include <cstdlib>
5. using namespace std;
6. int main()
7. {
8. ifstream fin;
9. string filename;
10. cout << "Enter filename => ";
11. cin >> filename;
12. try
13. {
14. fin.open(filename);
15. if (fin.is\_open())
16. {
17. cout << "file " << filename << " opened\n";
18. }
19. else
20. throw(string("Can't open file ") + filename);
21. }
22. catch (const string& errmsg)
23. {
24. cout << errmsg << "\nTry again? ";
25. char yn;
26. cin >> yn;
27. if (yn == 'y')
28. {
29. fin.clear();
30. cout << "Enter filename => ";
31. cin >> filename;
32. fin.open(filename);
33. if (!fin)
34. {
35. cout << "I quit! I can't find file " << filename << " either.\n";
36. }
37. else
38. {
39. cout << "file " << filename << " opened\n";
40. }
41. }
42. else
43. {
44. cout << "I didn't think you wanted to open a file anyway!\n";
45. }
46. }
47. cout << "\*\*\* End of Program \*\*\*\n";
48. }

\*\*\*\*\*\* Sample Run 1 \*\*\*\*\*\*

Enter filename => **ex10-4.cpp**

file ex10-4.cpp opened

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 2 \*\*\*\*\*\*

Enter filename => **ex10-4.ccp**

Can't open file ex10-4.ccp

Try again? **n**

I didn't think you wanted to open a file anyway!

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 3 \*\*\*\*\*\*

Enter filename => **ex10-4.ccp**

Can't open file ex10-4.ccp

Try again? **y**

Enter filename => **ex10-4.cpc**

I quit! I can't find file ex10-4.cpc either.

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 4 \*\*\*\*\*\*

Enter filename => **ex10-4.ccp**

Can't open file ex10-4.ccp

Try again? y

Enter filename => **ex10-4.cpp**

file ex10-4.cpp opened

\*\*\* End of Program \*\*\*

Later we'll look at a technique for "re-throwing" the same **throw**.

This next example shows two different styles for throwing exceptions.

The first five exceptions occur in and are handled in main(). The next five occur and are handled in another function called by main().

### Example 5 – Where to throw, where to catch

1. #include <iostream>
2. void funk(int it)
3. {
4. try
5. {
6. throw it;
7. }
8. catch(int whatever)
9. {
10. std::cout << "I caught a " << whatever << std::endl;
11. }
12. }
13. int main()
14. {
15. for (auto up = 1; up <= 5; up++)
16. {
17. try
18. {
19. throw up;
20. }
21. catch(int z)
22. {
23. std::cout << "You threw me a " << z << std::endl;
24. }
25. }
26. for (auto i = 16; i <= 20; i++)
27. funk(i);
28. std::cout << "End of program\n";
29. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

You threw me a 1

You threw me a 2

You threw me a 3

You threw me a 4

You threw me a 5

I caught a 16

I caught a 17

I caught a 18

I caught a 19

I caught a 20

End of program

### Example 6 - Throwing and catching more than one type

It is common to throw more than one type in a program. The following example illustrates shows how this is handled.

Note: When a user-defined type is thrown, the copy constructor is used to create the thrown object.

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. class Dog
5. {
6. string name;
7. string breed;
8. public:
9. Dog(const string& n = "Fido", const string& b = "mutt")
10. : name(n), breed (b) { }
11. friend ostream& operator<<(ostream& o,const Dog& dog)
12. {
13. return (o << dog.name << " is a " << dog.breed);
14. }
15. };
16. void funk(int i)
17. {
18. try
19. {
20. switch (i)
21. {
22. case 1:
23. throw("Have a nice day");
24. case 2:
25. throw(5);
26. case 3:
27. throw(3.14);
28. case 4:
29. throw(5L);
30. case 5:
31. throw(&i);
32. case 6:
33. throw(Dog());
34. }
35. }
36. catch(const char\* it)
37. {
38. cout << "You threw me a const char\*: " << it << endl;
39. }
40. catch (const string& it)
41. {
42. cout << "You threw me a const string&: " << it << endl;
43. }
44. catch(int it)
45. {
46. cout << "You threw me an int: " << it << endl;
47. }
48. catch(float it)
49. {
50. cout << "You threw me a float: " << it << endl;
51. }
52. catch(double it)
53. {
54. cout << "You threw me a double: " << it << endl;
55. }
56. catch(long it)
57. {
58. cout << "You threw me long: " << it << endl;
59. }
60. catch(int\* it)
61. {
62. cout << "You threw me an int address: " << it << endl;
63. }
64. catch(Dog it)
65. {
66. cout << "You threw me an Dog: " << it << endl;
67. }
68. }
69. int main()
70. {
71. funk(1);
72. funk(2);
73. funk(3);
74. funk(4);
75. funk(5);
76. funk(6);
77. cout << "End of program\n";
78. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

You threw me a const char\*: Have a nice day

You threw me an int: 5

You threw me a double: 3.14

You threw me long: 5

You threw me an int address: 0x6dff00

You threw me an Dog: Fido is a mutt

End of program

Which catch did not get used?

What if you throw a type that you haven't written a catch for?

### Example 7 - Unhandled Exceptions

This example shows what happens if you don't write a catch for the type that you throw. This is called an unhandled exception.

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. void funk(int i)
5. {
6. try
7. {
8. switch (i)
9. {
10. case 1:
11. throw(string("Have a nice day"));
12. case 2:
13. throw(5);
14. case 3:
15. throw(3.14);
16. }
17. }
18. catch(const string& it)
19. {
20. cerr << "You threw me a string: " << it << endl;
21. }
22. catch(double it)
23. {
24. cerr << "You threw me a double: " << it << endl;
25. }
26. }
27. int main()
28. {
29. funk(1);
30. funk(2);
31. funk(3);
32. cout << "End of program\n";
33. }

**\*\*\*\*\*\* Output \*\*\*\*\*\***

You threw me a const char\*: Have a nice day

Abnormal program termination

### Example 8 - How to catch anything

You may use **catch(...)** to catch a throw of a type for which you have not specified a catch.

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. void funk(int i)
5. {
6. try
7. {
8. switch (i)
9. {
10. case 0:
11. throw(0);
12. case 1:
13. throw(string("Have a nice day"));
14. case 2:
15. throw(5);
16. case 3:
17. throw(3.14);
18. }
19. }
20. catch (const string& it)
21. {
22. cout << "You threw me a string: " << it << endl;
23. }
24. catch(const char\* it)
25. {
26. cout << "You threw me a const char\*: " << it << endl;
27. }
28. catch(double it)
29. {
30. cout << "You threw me a double: " << it << endl;
31. }
32. catch(...)
33. {
34. cout << "You threw me something. I know not what!\n";
35. }
36. }
37. int main()
38. {
39. funk(1);
40. funk(2);
41. funk(3);
42. funk(0);
43. cout << "End of program\n";
44. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

You threw me a string: Have a nice day

You threw me something. I know not what!

You threw me a double: 3.14

You threw me something. I know not what!

End of program

### Example 9 - Exception Handling Classes

It might be a good idea to create a class to handle the exception.

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. class ZeroDenominator
5. {
6. public:
7. ZeroDenominator() {}
8. friend ostream& operator<<(ostream& out, const ZeroDenominator& error);
9. };
10. class Fraction
11. {
12. int numer, denom;
13. public:
14. Fraction(int n = 0, int d = 1) : numer(n), denom(d)
15. {
16. cout << "Fraction constructor called\n";
17. if (denom == 0) throw ZeroDenominator();
18. }
19. ~Fraction()
20. {
21. cout << "Fraction destructor called\n";
22. }
23. friend ostream& operator<<(ostream& o, const Fraction& f)
24. {
25. return (o << f.numer << '/' << f.denom);
26. }
27. };
28. class InputError
29. {
30. string stream;
31. public:
32. InputError(string name) : stream(name) {}
33. friend ostream& operator<<(ostream& out, const InputError& error);
34. };
35. ostream& operator<<(ostream& out, const InputError& error)
36. {
37. out << "Error in " << error.stream << endl;
38. return out;
39. }
40. ostream& operator<<(ostream& out, const ZeroDenominator& /\*error\*/)
41. {
42. out << "ZeroDenominator Error" << endl;
43. return out;
44. }
45. int main()
46. {
47. int i1, i2;
48. cout << "Enter two ints => ";
49. try
50. {
51. cin >> i1 >> i2;
52. if (cin.fail()) throw InputError("cin");
53. // You could also use (!cin) instead of (cin.fail())
54. // cin.bad() did not detect error in cin
55. Fraction f(i1,i2);
56. cout << f << endl; // Should this be in the try block?
57. }
58. catch (const InputError& error)
59. {
60. cerr << error << endl;
61. }
62. catch (const ZeroDenominator& errmsg)
63. {
64. cerr << errmsg << endl;
65. }
66. catch (...)
67. {
68. cerr << "help\n";
69. }
70. cout << "\*\*\* End of Program \*\*\*\n";
71. }

\*\*\*\*\*\* Sample Run 1 \*\*\*\*\*\*

Enter two ints => **2 3**

Fraction constructor called

2/3

Fraction destructor called

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 2 \*\*\*\*\*\*

Enter two ints => **2 three**

Error in cin

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 3 \*\*\*\*\*\*

Enter two ints **2 0**

Fraction constructor called

ZeroDenominator Error

\*\*\* End of Program \*\*\*

### Example 10 – Use a class to access different values that may be thrown

Another technique is to use a class to access different values that might be thrown.

1. #include <iostream>
2. #include <cctype>
3. #include <cfloat> // for FLT\_MAX
4. using namespace std;
5. class ErrorStuff
6. {
7. public:
8. static const int BadInt;
9. static const float BadFloat;
10. static const char BadChar;
11. ErrorStuff(int arg)
12. : x(arg), y(BadFloat), z(BadChar)
13. {
14. }
15. ErrorStuff(float arg)
16. : x(BadInt), y(arg), z(BadChar)
17. {
18. }
19. ErrorStuff(char arg)
20. : x(BadInt), y(BadFloat), z(arg)
21. {
22. }
23. int get\_x() const
24. {
25. return x;
26. }
27. float get\_y() const
28. {
29. return y;
30. }
31. char get\_z() const
32. {
33. return z;
34. }
35. private:
36. int x;
37. float y;
38. char z;
39. };
40. const int ErrorStuff::BadInt = 0xffffffff;
41. const float ErrorStuff::BadFloat = FLT\_MAX;
42. const char ErrorStuff::BadChar = 0;
43. int main()
44. {
45. int i;
46. float f;
47. char c;
48. try
49. {
50. cout << "Enter an even int, a positive float, and a alphabetic char => ";
51. cin >> i >> f >> c;
52. if (cin.fail())
53. throw string{"cin"};
54. if (i % 2)
55. throw ErrorStuff(i);
56. else if (f < 0)
57. throw ErrorStuff(f);
58. else if (!isalpha(c))
59. throw ErrorStuff(c);
60. else
61. cout << "Thanks\n";
62. }
63. catch (const string& what)
64. {
65. if (what == "cin")
66. {
67. cerr << "\*\*\* Can't you type?\n";
68. cin.clear();
69. }
70. else
71. {
72. cout << "whatever\n";
73. }
74. }
75. catch (const ErrorStuff& e)
76. {
77. cout << "Hey!!! ";
78. if (e.get\_x() != ErrorStuff::BadInt)
79. cerr << "You entered an invalid int: " << e.get\_x() << endl;
80. else if (e.get\_y() != ErrorStuff::BadFloat)
81. cerr << "You entered an invalid float: " << e.get\_y() << endl;
82. else
83. cerr << "You entered an invalid char: " << e.get\_z() << endl;
84. }
85. cout << "\*\*\* End of Program \*\*\*\n";
86. }

\*\*\*\*\*\* Sample Run 1 \*\*\*\*\*\*

Enter an even int, a positive float, and a alphabetic char => **2 2.2 A**

Thanks

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 2 \*\*\*\*\*\*

Enter an even int, a positive float, and a alphabetic char => **two 2.2 A**

\*\*\* Can't you type?

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 3 \*\*\*\*\*\*

Enter an even int, a positive float, and a alphabetic char => **3 2.2 A**

Hey!!! You entered an invalid int: 3

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 4 \*\*\*\*\*\*

Enter an even int, a positive float, and a alphabetic char => **2 -2.2 A**

Hey!!! You entered an invalid float: -2.2

\*\*\* End of Program \*\*\*

\*\*\*\*\*\* Sample Run 5 \*\*\*\*\*\*

Enter an even int, a positive float, and a alphabetic char => **2 2.2 2**

Hey!!! You entered an invalid char: 2

\*\*\* End of Program \*\*\*

### Catching Uncaught Exceptions with set\_terminate()

You can name a function to execute using set\_terminate() for any unhandled exceptions. The **set\_terminate()** function will execute, then the program will abort.

The terminate function has a void argument and void return. By default, an unhandled exception will cause a call to the **terminate()** function, which will, in turn call the **abort()** function. This causes the program to end with a "Abnormal program termination error". The use of **set\_terminate()** overrides this default behavior.

**set\_terminate**() returns the previous function assigned.

An uncaught exception will terminate the program. **set\_terminate**() cannot override this, so you should not attempt to continue processing by returning to the calling function or jumping to another location. This will result in undefined program behavior.

Further, the **set\_terminate**() function, itself, had better not throw an exception!

**Syntax**

typedef void (\*terminate\_function)();

terminate\_function **set\_terminate**(terminate\_function fn);

1

Both the **terminate**() and the **abort**() functions are C++ standard library functions.

### Example 11 – set\_terminate()

1. #include <iostream>
2. #include <exception> // for set\_terminate()
3. #include <string>
4. using namespace std;
5. void uncaught()
6. {
7. cerr << "I wasn't able to catch an exception\n";
8. }
9. void funk(int i)
10. {
11. try
12. {
13. switch (i)
14. {
15. case 1:
16. throw(string("have a nice day"));
17. case 2:
18. throw(5);
19. case 3:
20. throw(3.14);
21. }
22. }
23. catch(const string& it)
24. {
25. cout << "You threw me a string: " << it << endl;
26. }
27. catch(double it)
28. {
29. cout << "You threw me a double: " << it << endl;
30. }
31. }
32. int main()
33. {
34. set\_terminate(uncaught);
35. funk(1);
36. funk(2);
37. funk(3);
38. cout << "End of program\n";
39. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

You threw me a const char\*: Have a nice day

I wasn't able to catch an exception

Program Aborted

### Exception Specifications

Dynamic exception specifications **are no longer supported** since C++17.

**Examples**

void funk1() throw (sometype); // Error: not allowed in C++17

void funk2() throw (); // Error: not allowed in C++17

void funk2() noexcept; // OK

### set\_unexpected()

The set\_unepected() function was removed in C++17.

### Example 14 - Re-throwing a throw

Sometimes a catch block is not meant to handle the current error. If this is the case, one option is to re-throw the current throw, so that it is handled by a prior catch block. To do this, just place a **throw;** without an throw-expression in the current catch block. Control is transferred to a higher level catch block. This is illustrated in the following example.

1. #include <iostream>
2. #include <string>
3. void funky(void)
4. {
5. try
6. {
7. throw(std::string("This is a funky booboo"));
8. }
9. catch(...)
10. {
11. std::cout << "I don't know how to handle this\n";
12. throw;
13. }
14. }
15. int main()
16. {
17. try
18. {
19. funky();
20. }
21. catch(const std::string& x)
22. {
23. std::cout << "Somebody threw me: " << x << std::endl;
24. }
25. std::cout << "\*\*\* End of Program \*\*\*\n";
26. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

I don't know how to handle this

Somebody threw me: This is a funky booboo

\*\*\* End of Program \*\*\*

### Example 15 - Unwinding the stack

When an exception is thrown, destructors are automatically called for automatic objects that were constructed in the try-block. If the exception is thrown during the construction of an object, the destructor is not called for that object. For example, if an array of objects is being constructed when an exception is thrown, destructors will only be called for the array elements which were fully constructed. This process of calling of destructors for automatic objects after an exception is thrown is called **stack unwinding**.

1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. class Thing
5. {
6. char\* name;
7. public:
8. Thing(const char\* arg = nullptr);
9. Thing(const Thing& t); // copy ctor
10. ~Thing();
11. const char\* get\_name() const
12. {
13. return name;
14. }
15. };
16. Thing::Thing(const char\* arg)
17. : name(new char[strlen(arg)+1])
18. {
19. if (strcmp(arg,"Satan")==0)
20. throw (this);
21. else
22. strcpy(name,arg);
23. cout << ">>> " << name << " successfully constructed\n";
24. }
25. Thing::Thing(const Thing& arg) : name(new char[strlen(arg.name)+6])
26. {
27. strcpy(name,arg.name);
28. strcat(name, " Clone");
29. cout << ">>> " << name << " successfully copy constructed\n";
30. }
31. Thing::~Thing()
32. {
33. cout << "<<< destructor called for Thing " << name << endl;
34. if (name)
35. delete [] name;
36. name = nullptr;
37. }
38. int main()
39. {
40. Thing\* pThing;
41. try
42. {
43. Thing aFriend("Sam");
44. Thing aFriendClone(aFriend);
45. cout << endl;
46. pThing = new Thing("Sarah");
47. delete pThing;
48. pThing = nullptr;
49. cout << endl;
50. Thing satan("Satan");
51. Thing harry("Harry");
52. }
53. catch(const Thing\* ptr)
54. {
55. cerr << "I caught an evil Thing" << endl;
56. delete [] ptr->get\_name();
57. }
58. if (pThing) delete pThing;
59. cerr << "\*\*\* End of Program \*\*\*\n";
60. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

>>> Sam successfully constructed

>>> Sam Clone successfully copy constructed

>>> Sarah successfully constructed

<<< destructor called for Thing Sarah

<<< destructor called for Thing Sam Clone

<<< destructor called for Thing Sam

I caught an evil Thing

<<< destructor called for Thing \*\*\* End of Program \*\*\*

### Example 16 - Standard Exceptions

1. #include <iostream>
2. #include <string>
3. #include <exception>
4. #include <new> // for bad\_alloc
5. #include <typeinfo> // for bad\_cast
6. #include <stdexcept>
7. using namespace std;
8. class Base
9. {
10. public:
11. virtual void funk() {}
12. virtual ~Base() {}
13. };
14. class Derived : public Base
15. {
16. public:
17. void funk() {}
18. };
19. int main()
20. {
21. // test bad\_alloc
22. try
23. {
24. while (1)
25. {
26. cout << "Can I have some memory?\n";
27. new char[0x7fffffff];
28. }
29. }
30. catch(const bad\_alloc& error)
31. {
32. cerr << "\*\*\* I caught a " << error.what() << endl << endl;
33. }
34. // test bad\_cast
35. try
36. {
37. Base baseObject;
38. // try to cast a base object to a derived object
39. Derived& ref2Derived = dynamic\_cast<Derived&>(baseObject);
40. }
41. catch(const bad\_cast& error)
42. {
43. cerr << "!!! I caught a " << error.what() << endl << endl;
44. }
45. // test out\_of\_range error
46. try
47. {
48. string S = "Hey";
49. cout << "S.at(2)=" << S.at(2) << endl;
50. cout << "S.at(5)=" << S.at(5) << endl; // string throws an out\_of\_range error
51. }
52. catch (const out\_of\_range& error)
53. {
54. cout << "$$$ I caught a " << error.what() << endl << endl;
55. }
56. cout << "\*\*\* End of Program \*\*\*\n";
57. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Can I have some memory?

\*\*\* I caught a std::bad\_alloc

!!! I caught a std::bad\_cast

S.at(2)=y

$$$ I caught a basic\_string::at: \_\_n (which is 5) >= this->size() (which is 3)

\*\*\*\* End of Program \*\*\*

### Example 17 - Derive your own exceptions from standard exceptions

1. #include <exception>
2. #include <stdexcept>
3. #include <iostream>
4. #include <cmath> // for sqrt()
5. #include <cstring>
6. #include <cstdlib>
7. #include <sstream> // for istreamstream/ostringstream
8. #include <climits> // for SHRT\_MAX
9. #include <typeinfo> // for typeid operator
10. using namespace std;
11. ostream& operator<<(ostream& out, const exception& error)
12. {
13. out << "I caught an error of type: " << typeid(error).name()
14. << "\nMessage: " << error.what() << endl;
15. return out;
16. }
17. class my\_domain\_error : public domain\_error
18. {
19. public:
20. my\_domain\_error(const char\* message) : domain\_error(message)
21. {}
22. // override the virtual what() function
23. const char\* what() const noexcept override
24. {
25. static char temp[128];
26. strcpy(temp,"my\_domain\_error: ");
27. strcat(temp,domain\_error::what());
28. return temp;
29. }
30. };
31. double mysqrt1(double number) throw (domain\_error)
32. {
33. if (number < 0)
34. throw domain\_error("mysqrt1 error: negative argument");
35. return sqrt(number);
36. }
37. double mysqrt2(double number) throw (my\_domain\_error)
38. {
39. if (number < 0)
40. throw my\_domain\_error("mysqrt2 error: negative argument");
41. return sqrt(number);
42. }
43. // Derive the zero\_denominator class from invalid\_argument
44. class zero\_denominator : public invalid\_argument
45. {
46. public:
47. zero\_denominator()
48. : invalid\_argument("Error: zero denominator")
49. { }
50. };
51. class fraction
52. {
53. int numerator, denominator;
54. public:
55. fraction(int n = 0, int d = 1) : numerator(n), denominator(d)
56. {
57. if (d == 0 )
58. throw zero\_denominator();
59. }
60. };
61. // convert a hexadecimal string to unsigned int
62. unsigned
63. hex\_string\_to\_unsigned(const string& text) throw (invalid\_argument)
64. {
65. if (text.find\_first\_not\_of("0123456789abcdefABCDEF") != string::npos)
66. {
67. throw invalid\_argument(string("Invalid hexadecimal char in: " ) + text);
68. }
69. istringstream sin(text);
70. unsigned number;
71. sin >> hex >> number;
72. return number;
73. }
74. // returns sum of two shorts, make sure sum is valid short
75. short
76. add2shorts(short one, short two, bool check\_limit = false) throw (overflow\_error)
77. {
78. if (check\_limit)
79. {
80. if (static\_cast<int>(one) + two > SHRT\_MAX) // SHRT\_MAX = 32767
81. {
82. ostringstream sout;
83. sout << "add2shorts failed with arguments " << one << " and " << two;
84. throw overflow\_error(sout.str());
85. }
86. }
87. return one + two;
88. }
89. int main()
90. {
91. // test throw/catch of domain\_error
92. try
93. {
94. cout << "mysqrt1(2.0)=" << mysqrt1(2.0) << endl;
95. cout << "mysqrt1(-2.0)=" << mysqrt1(-2.0) << endl;
96. }
97. catch (const domain\_error& error)
98. {
99. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
100. }
101. // test throw/catch of logic\_error
102. try
103. {
104. cout << "mysqrt1(-2.0)=" << mysqrt1(-2.0) << endl;
105. }
106. catch (const logic\_error& error)
107. {
108. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
109. }
110. // test throw/catch of (base class) exception
111. try
112. {
113. cout << "mysqrt1(-2.0)=" << mysqrt1(-2.0) << endl;
114. }
115. catch (const exception& error)
116. {
117. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
118. }
119. // test throw/catch of my\_domain\_error
120. try
121. {
122. cout << "mysqrt2(-2.0)=" << mysqrt2(-2.0) << endl;
123. }
124. catch (const my\_domain\_error& error)
125. {
126. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
127. }
128. // test throw/catch of zero\_denominator
129. try
130. {
131. fraction F(2,0);
132. }
133. catch (const zero\_denominator& error)
134. {
135. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
136. }
137. // test throw/catch of invalid\_argument
138. try
139. {
140. cout << "hex abc=" << hex\_string\_to\_unsigned(string("abc")) << endl;
141. cout << "hex abz=" << hex\_string\_to\_unsigned(string("abz")) << endl;
142. }
143. catch (const invalid\_argument& error)
144. {
145. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
146. }
147. // test throw/catch of overflow\_error
148. try
149. {
150. cout << "short 31000+32000=" << add2shorts(31000,32000) << endl;
151. cout << "short 31000+32000=" << add2shorts(31000,32000,true) << endl;
152. }
153. catch (const overflow\_error& error)
154. {
155. cerr << "Line " << \_\_LINE\_\_ << ": " << error << endl;
156. }
157. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

mysqrt1(2.0)=1.41421

Line 111: I caught an error of type: St12domain\_error

Message: mysqrt1 error: negative argument

Line 121: I caught an error of type: St12domain\_error

Message: mysqrt1 error: negative argument

Line 131: I caught an error of type: St12domain\_error

Message: mysqrt1 error: negative argument

Line 141: I caught an error of type: 15my\_domain\_error

Message: my\_domain\_error: mysqrt2 error: negative argument

Line 151: I caught an error of type: 16zero\_denominator

Message: Error: zero denominator

hex abc=2748

Line 162: I caught an error of type: St16invalid\_argument

Message: Invalid hexadecimal char in: abz

short 31000+32000=-2536

Line 173: I caught an error of type: St14overflow\_error

Message: add2shorts failed with arguments 31000 and 32000

# Namespaces

A namespace is a group of types, variables, or objects. This grouping may be used to avoid name clashes. In other words, by using namespaces, an application may reuse a type name or variable name without an ambiguity conflict.

The keyword, namespace, is used to create a namespace and to reference an existing namespace name.

Namespace usage make use of the using directive and the using declaration. A using directive,

is used to qualify all unqualified symbol names of a namespace, such as

using namespace std;

allows you to write

cout << whatever << endl;

instead of

std::cout << whatever << std::endl;

A using declaration allows you to refer to a symbol name without qualifying the entire namespace. For example:

using std::cout;

…

cout << whatever << std::end;

### Example 1 – Create a namespace

1. #include <iostream>
2. #include <cmath>
3. #include <cstring>
4. #include <cstdlib>
5. #include <cctype>
6. using namespace std;
7. // Create a namespace
8. namespace mystuff
9. {
10. int cout = 5;
11. double sqrt(double x)
12. {
13. return x / 2.0;
14. }
15. }
16. int main()
17. {
18. char cout[32] = "This is a bad idea";
19. char temp[80];
20. std::cout << "hey\n";
21. std::cout << "the square root of 2 is " << sqrt(2.) << endl;
22. strcpy(temp,"hello");
23. strcat(temp," there");
24. std::cout << strlen(temp) << temp << endl;
25. std::cout << atoi("4") << endl;
26. std::cout << toupper('a') << endl;
27. std::cout << static\_cast<char>(toupper('a')) << endl;
28. std::cout << mystuff::cout << ' ' << cout << endl;
29. std::cout << sqrt(5.75) << ' ' << mystuff::sqrt(5.75) << endl;
30. }

\*\*\*\*\*\* Program Output \*\*\*\*\*\*

hey

the square root of 2 is 1.41421

11hello there

4

65

A

5 This is a bad idea

2.39792 2.875

### Example 2 – namespace scope

Note that symbols default to their local definitions first, then to std definitions.

1. #include <iostream>
2. namespace test
3. {
4. int I = 9;
5. }
6. void funk1();
7. void funk2();
8. void funk3();
9. int main()
10. {
11. funk1();
12. funk2();
13. funk3();
14. }
15. void funk1()
16. {
17. std::cout << test::I << std::endl; // This is OK
18. // std::cout << I << std::endl; // Compile error
19. using namespace test;
20. std::cout << I << std::endl; // OK, now
21. }
22. void funk2()
23. {
24. std::cout << test::I << std::endl; // This is
25. // std::cout << I << std::endl; // Compile error
26. }
27. using namespace test;
28. void funk3()
29. {
30. std::cout << I << std::endl; // OK, now
31. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

9

9

9

9

### Example 3 - namespaces and multiple files

This example illustrates the use of namespace in multiple files.

1. // File: node.h
2. #ifndef NODE\_H
3. #define NODE\_H
4. #include <iostream>
5. **namespace joelinkedlist**
6. **{**
7. class Node
8. {
9. int data;
10. Node\* next;
11. public:
12. Node(int d,Node\* n);
13. int get\_data() const;
14. Node\* get\_next() const;
15. void set\_next(Node\* ptr);
16. };
17. std::ostream& operator<<(std::ostream&, const Node&);
18. **}**
19. #endif
20. // File: node.cpp
21. #include "node.h"
22. #include <iostream>
23. using namespace std;
24. **joelinkedlist::**Node::Node(int d, Node\* n)
25. : data(d), next(n)
26. {
27. }
28. int **joelinkedlist::**Node::get\_data() const
29. {
30. return data;
31. }
32. **using namespace joelinkedlist;**
33. Node\* Node::get\_next() const
34. {
35. return next;
36. }
37. void Node::set\_next(Node\* ptr)
38. {
39. next = ptr;
40. }
41. **namespace joelinkedlist**
42. **{**
43. ostream& operator<<(ostream& out, const Node& obj)
44. {
45. out << obj.get\_data();
46. return out;
47. }
48. **}**
49. // File: list.h
50. #ifndef LIST\_H
51. #define LIST\_H
52. #include "node.h"
53. #include <iostream>
54. **namespace joelinkedlist**
55. **{**
56. class List
57. {
58. Node\* top;
59. public:
60. List();
61. ~List();
62. void push(int item);
63. int pop();
64. Node\* get\_top() const;
65. bool remove(int item);
66. Node\* find(int item) const;
67. bool remove\_last();
68. };
69. std::ostream& operator<<(std::ostream&, const List&);
70. **}**
71. #endif
72. // File: list.cpp
73. #include <iostream>
74. #include <cstdlib>
75. using namespace std;
76. #include "list.h"
77. **using joelinkedlist::List;**
78. **using joelinkedlist::Node;**
79. List::List() : top(0)
80. { }
81. List::~List()
82. {
83. Node\* temp = top;
84. while (temp != nullptr) {
85. top = top -> get\_next();
86. delete temp;
87. temp = top;
88. }
89. }
90. void List::push(int item)
91. {
92. Node\* temp = new Node(item, top);
93. top = temp;
94. }
95. int List::pop()
96. {
97. Node\* temp = top;
98. top = top->get\_next();
99. int value = temp->get\_data();
100. delete temp;
101. return value;
102. }
103. Node\* List::get\_top() const
104. {
105. return top;
106. }
107. Node\* List::find(int item) const
108. {
109. Node\* temp = top;
110. while (temp != 0) {
111. if (temp->get\_data() == item) return temp;
112. temp = temp -> get\_next();
113. }
114. return 0;
115. }
116. bool List::remove(int item)
117. {
118. if (!find(item)) {
119. cerr << item << " is not in the List\n";
120. return false;
121. }
122. Node\* temp1 = top;
123. Node\* temp2;
124. if (top->get\_data() == item) {
125. top = top -> get\_next();
126. delete temp1;
127. return true;
128. }
129. while (temp1->get\_next()->get\_data() != item) {
130. temp1 = temp1 -> get\_next();
131. }
132. temp2 = temp1 -> get\_next();
133. temp1->set\_next(temp2->get\_next());
134. delete temp2;
135. return true;
136. }
137. **namespace joelinkedlist**
138. **{**
139. ostream& operator<<(ostream& out, const List& object)
140. {
141. Node\* temp = object.get\_top();
142. while (temp != 0) {
143. out << \*temp << ' ';
144. temp = temp -> get\_next();
145. }
146. return out;
147. }
148. **}**
149. // File: main.cpp
150. #include <iostream>
151. using namespace std;
152. #include "list.h"
153. **using joelinkedlist::List;**
154. int main()
155. {
156. List L;
157. L.push(2);
158. L.push(4);
159. L.push(6);
160. L.push(8);
161. L.push(10);
162. cout << L << endl;
163. cout << "top value is " << L.get\_top()->get\_data() << endl;
164. if (L.find(2)) cout << 2 << " is in the list\n";
165. if (L.find(5)) cout << 5 << " is in the list\n";
166. if (L.find(6)) cout << 6 << " is in the list\n";
167. if (L.find(10)) cout << 10 << " is in the list\n";
168. cout << L.pop() << " removed from the list\n";
169. cout << L << endl;
170. L.remove(3);
171. L.remove(6);
172. cout << L << endl;
173. L.remove(2);
174. L.remove(8);
175. cout << L << endl;
176. }

# Libraries

Libraries are used to isolate common code that may be used by different applications. By designing and using a library, you do not have to “reinvent the wheel”. You simply “invent the wheel” one time and then you “link it in” to your current application whenever you need it. As part of this process, you also have to tell your current application what the wheel “looks like”. This is typically accomplished by including a heading file.

The use of libraries mandates that the associated libraries files be logically organized in directories that are easily identified and accessed.

## Creating a Library

* The library files will usually consist of one or more source files and one or more header files.
* The source files and header files may be located in separate directories. The source file(s) may contain one or (usually) more functions.
* There is no main() function that is usually present in any C++ application.
* Each library source code file is compiled into its own object file.
* The object file(s) are combined together into a library file, sometimes called an archive.
* A library typically contains functions, variables, constants, and types.
* In general, a libraries source file will contain definitions (function definitions and variable definitions). A libraries header file will contain declarations (function prototypes, class declarations, and declarations of other types).

## Using a Library

* An application that uses a library must include the libraries header file(s) in order to “see” the libraries declarations. That is required for compilation of the application. When the application file is compiled, it must identify to the compiler the location of the included header file.
* Then the application must “link in” the library. In the “link” step of the application, the location of the library file (or archive) must be identified to the “linker”.

## Types of Linking

There are two basic types of linking performed by an application – static and dynamic linking. With static linking the necessary (or referenced) code is inserted into the final executable and becomes part of that binary file. With dynamic linking, the referenced code is not directly inserted into the final executable. The dynamic library “sits out on disk” and the necessary parts are included or accessed as needed during run-time. Applications that use dynamic linking are usually smaller than those that use static linking. Dynamically linking applications will usually run slower than the equivalent statically linked applications, since the dynamically linked library must be loaded into memory at run-time.

## Examples

### Example 1 – a factorial library

The following example demonstrates a library that is used to calculate factorial. This example makes use of 3 files:

1. A library header file that contains a function prototype
2. A library source file containing the factorial function definition. This file will be compiled and the resulting function will be placed in a library.
3. A test source file containing calls to the factorial function.

Library header file

1. // File: factorial.h
2. #ifndef FACTORIAL\_H
3. #define FACTORIAL\_H
4. long factorial(long arg);
5. #endif

Library source file

1. // File: factorial.cpp
2. long factorial(long arg)
3. {
4. long total = 1;
5. for (long num = 2; num <= arg; num++)
6. total \*= num;
7. return total;
8. }

Test source file

1. // File: factorial\_test.cpp
2. #include <iostream>
3. using namespace std;
4. #include "factorial.h"
5. int main()
6. {
7. cout << factorial(2) << endl;
8. cout << factorial(4) << endl;
9. cout << factorial(6) << endl;
10. cout << factorial(8) << endl;
11. cout << factorial(10) << endl;
12. }

#### \*\*\*\*\*\* Output \*\*\*\*\*\*

2

24

720

40320

3628800

#### The Process

1. The header file and library source files are first created and compiled as a library (static or dynamic). It is important to give the resulting library an appropriate name and place it in a logical location, probably with other libraries.
2. The test source file must include the library header file for compilation. This means that you must tell the compiler where to find that header file.
3. To link the test application you must “link in” the library. That means telling the compiler where to find the library and what its name is.

### Example 2 – a fraction library

This example illustration implementation of a fraction library.

fraction library header file

1. // File: fraction.h
2. #ifndef FRACTION\_H
3. #define FRACTION\_H
4. class fraction
5. {
6. int numer, denom;
7. public:
8. fraction(int = 0, int = 1);
9. void operator!(void) const; // print the fraction
10. fraction& operator~(void); // reduce the fraction
11. fraction operator-(void) const; // negative of fraction
12. fraction operator\*(void) const; // reciprocal of fraction
13. fraction& operator+=(const fraction&);
14. fraction& operator-=(const fraction&);
15. fraction& operator\*=(const fraction&);
16. fraction& operator/=(const fraction&);
17. fraction operator+(int) const;
18. fraction operator-(int) const;
19. fraction operator\*(int) const;
20. fraction operator/(int) const;
21. int operator>(const fraction&) const;
22. int operator<(const fraction&) const;
23. int operator>=(const fraction&) const;
24. int operator<=(const fraction&) const;
25. int operator==(const fraction&) const;
26. int operator!=(const fraction&) const;
27. fraction operator+(const fraction&) const;
28. fraction operator-(const fraction&) const;
29. fraction operator\*(const fraction&) const;
30. fraction operator/(const fraction&) const;
31. fraction& operator++(); // prefix operator returns by ref
32. fraction operator++(int); // postfix operator returns by value
33. };
34. #endif

fraction library source file

1. // File: fraction.cpp
2. #include "fraction.h"
3. #include <iostream>
4. using namespace std;
5. // member function definitions
6. fraction::fraction(int n, int d)
7. {
8. // assert(d != 0);
9. numer = n;
10. denom = d;
11. }
12. void fraction::operator!(void) const
13. {
14. cout << numer << '/' << denom << endl;
15. }
16. fraction& fraction::operator~(void)
17. {
18. int min;
19. // find the minimum of the denom and numer
20. min = denom < numer ? denom : numer;
21. for (int i = 2; i <= min; i++)
22. {
23. while ((numer % i == 0) && (denom % i == 0))
24. {
25. numer /= i;
26. denom /= i;
27. }
28. }
29. return \*this;
30. }
31. fraction fraction::operator-(void) const
32. {
33. return fraction(-numer,denom);
34. }
35. fraction fraction::operator\*(void) const
36. {
37. return fraction(denom,numer);
38. }
39. fraction& fraction::operator+=(const fraction& f)
40. {
41. numer = numer\*f.denom+denom\*f.numer;
42. denom = denom\*f.denom;
43. return \*this;
44. }
45. fraction& fraction::operator-=(const fraction& f)
46. {
47. \*this += (-f);
48. return \*this;
49. }
50. fraction& fraction::operator\*=(const fraction& f)
51. {
52. numer = numer\*f.numer;
53. denom = denom\*f.denom;
54. return \*this;
55. }
56. fraction& fraction::operator/=(const fraction& f)
57. {
58. \*this \*= (\*f);
59. return \*this;
60. }
61. int fraction::operator>(const fraction& f) const
62. {
63. return (float) numer/denom > (float) f.numer/f.denom;
64. }
65. int fraction::operator<(const fraction& f) const
66. {
67. return f>\*this;
68. }
69. int fraction::operator==(const fraction& f) const
70. {
71. return numer\*f.denom == denom\*f.numer;
72. }
73. int fraction::operator!=(const fraction& f) const
74. {
75. return !(\*this == f);
76. }
77. int fraction::operator<=(const fraction& f) const
78. {
79. return !(\*this > f);
80. }
81. int fraction::operator>=(const fraction& f) const
82. {
83. return !(\*this<f);
84. }
85. fraction fraction::operator+(const fraction& f) const
86. {
87. return fraction(numer\*f.denom+denom\*f.numer,denom\*f.denom);
88. }
89. fraction fraction::operator-(const fraction& f) const
90. {
91. return fraction(numer\*f.denom-denom\*f.numer,denom\*f.denom);
92. }
93. fraction fraction::operator\*(const fraction& f) const
94. {
95. return fraction(numer\*f.numer,denom\*f.denom);
96. }
97. fraction fraction::operator/(const fraction& f) const
98. {
99. return (\*this) \* (\*f);
100. }
101. fraction fraction::operator+(int i) const
102. {
103. return fraction(numer+i\*denom,denom);
104. }
105. fraction fraction::operator-(int i) const
106. {
107. return (\*this) + -i;
108. }
109. fraction fraction::operator\*(int i) const
110. {
111. return fraction(numer\*i,denom);
112. }
113. fraction fraction::operator/(int i) const
114. {
115. return fraction(numer,i\*denom);
116. }
117. // prefix increment operator
118. fraction& fraction::operator++()
119. {
120. numer += denom;
121. return \*this;
122. }
123. // postfix increment operator
124. fraction fraction::operator++(int) // Note dummy int argument
125. {
126. fraction temp(\*this);
127. ++\*this; // call the prefix operator
128. return temp;
129. }

fraction library test

1. // File: fraction\_main.cpp
2. #include "fraction.h"
3. #include <iostream>
4. using namespace std;
5. int main(void)
6. {
7. fraction f(3,4); // initialize fraction f & g
8. fraction g(1,2);
9. cout << "!f ";
10. !f;
11. cout << "!g ";
12. !g;
13. cout << endl;
14. cout << "-g ";
15. !-g;
16. cout << "\*g ";
17. !\*g;
18. fraction h = g + f;
19. cout << endl;
20. cout << "h=g+f " << " !h ";
21. !h;
22. cout << "!~h ";
23. !~h;
24. cout << endl;
25. cout << "f+g ";
26. ! (f + g);
27. cout << "f-g ";
28. ! (f - g);
29. cout << "f\*g ";
30. ! (f \* g);
31. cout << "f/g ";
32. ! (f / g);
33. cout << endl;
34. cout << "f+=g ";
35. !~(f+=g);
36. cout << "f-=g ";
37. !~(f-=g);
38. cout << "f\*=g ";
39. !~(f\*=g);
40. cout << "f/=g ";
41. !~(f/=g);
42. cout << endl;
43. cout << "f<g " << (f<g) << endl;
44. cout << "f>g " << (f>g) << endl;
45. cout << "f==g " << (f==g) << endl;
46. cout << "f!=g " << (f!=g) << endl;
47. cout << "f<=g " << (f<=g) << endl;
48. cout << "f>=g " << (f>=g) << endl;
49. cout << endl;
50. cout << "f+5 ";
51. !(f+5);
52. cout << "f-5 ";
53. !(f-5);
54. cout << "f\*5 ";
55. !(f\*5);
56. cout << "f/5 ";
57. !(f/5);
58. cout << endl;
59. cout << "f+=5 ";
60. f+=5;
61. cout << "!~f ";
62. !~f; // How does this work?
63. cout << "++f ";
64. !++f;
65. cout << "f=";
66. !f;
67. cout << "f++ ";
68. !f++;
69. cout << "f=";
70. !f;
71. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

!f 3/4

!g 1/2

-g -1/2

\*g 2/1

h=g+f !h 10/8

!~h 5/4

f+g 10/8

f-g 2/8

f\*g 3/8

f/g 6/4

f+=g 5/4

f-=g 3/4

f\*=g 3/8

f/=g 3/4

f<g 0

f>g 1

f==g 0

f!=g 1

f<=g 0

f>=g 1

f+5 23/4

f-5 -17/4

f\*5 15/4

f/5 3/20

f+=5 !~f 23/4

++f 27/4

f=27/4

f++ 27/4

f=31/4

#### Linux compilation

These Linux commands are meant to demonstrate the compilation process.

1. g++ -Wall -c fraction.cpp
2. ar r libfraction.a fraction.o
3. g++ -Wall fraction\_main.cpp -L. -lfraction -o fraction\_test
4. ls

\*\*\*\*\*\* Output \*\*\*\*\*\*

fraction.cpp fraction.o fraction\_test

fraction.h fraction\_main.cpp libfraction.a

##### Explanation

Assumption: all files are located in the same directory for this example.

1. The fraction.cpp source file is compiled. The result is an object file, fraction.o. Note, the compiler finds the fraction.h header file in the same directory as the fraction.cpp file.
2. The fraction.o object file is placed in (archived) the library file, libfraction.a.
3. The fraction\_main.cpp test file is compiled. The include directory is assumed to be the current directory. The library directory is also the current directory (that’s the -L.). The library to *link in* is libfraction.a (that’s the -lfraction). The output binary is fraction\_test.
4. The ls command lists the 6 files related to this example.

fraction.h – fraction header

fraction.cpp – fraction source

fraction.o – fraction object

libfraction.a – fraction library

fraction\_main.cpp – fraction test source

fraction\_test – fraction test binary

### Example 3 – a linked list library

This example illustration implementation of a linked list library.

Node class header file

1. // File: node.h
2. #ifndef NODE\_H
3. #define NODE\_H
4. #include <iostream>
5. class Node
6. {
7. int data;
8. Node\* next;
9. public:
10. Node(int d,Node\* n);
11. int get\_data() const;
12. Node\* get\_next() const;
13. void set\_next(Node\* ptr);
14. };
15. std::ostream& operator<<(std::ostream&, const Node&);
16. #endif

Node class source file

1. // File: node.cpp
2. #include "node.h"
3. #include <iostream>
4. using namespace std;
5. Node::Node(int d,Node\* n)
6. : data(d), next(n)
7. { }
8. int Node::get\_data() const
9. {
10. return data;
11. }
12. Node\* Node::get\_next() const
13. {
14. return next;
15. }
16. void Node::set\_next(Node\* ptr)
17. {
18. next = ptr;
19. }
20. ostream& operator<<(ostream& out, const Node& obj)
21. {
22. out << obj.get\_data();
23. return out;
24. }

List class header file

1. // File: list.h
2. #ifndef LIST\_H
3. #define LIST\_H
4. #include "node.h"
5. #include <iostream>
6. class List
7. {
8. Node\* top;
9. public:
10. List();
11. ~List();
12. void push(int item);
13. int pop();
14. Node\* get\_top() const;
15. bool remove(int item);
16. Node\* find(int item) const;
17. bool remove\_last();
18. };
19. std::ostream& operator<<(std::ostream&, const List&);
20. #endif

List class source file

1. // File: list.cpp
2. #include <iostream>
3. #include <cstdlib>
4. using namespace std;
5. #include "list.h"
6. List::List() : top(0)
7. { }
8. List::~List()
9. {
10. Node\* temp = top;
11. while (temp != nullptr)
12. {
13. top = top -> get\_next();
14. delete temp;
15. temp = top;
16. }
17. }
18. void List::push(int item)
19. {
20. Node\* temp = new Node(item,top);
21. top = temp;
22. }
23. int List::pop()
24. {
25. Node\* temp = top;
26. top = top->get\_next();
27. int value = temp->get\_data();
28. delete temp;
29. return value;
30. }
31. Node\* List::get\_top() const
32. {
33. return top;
34. }
35. Node\* List::find(int item) const
36. {
37. Node\* temp = top;
38. while (temp != 0)
39. {
40. if (temp->get\_data() == item) return temp;
41. temp = temp -> get\_next();
42. }
43. return 0;
44. }
45. bool List::remove(int item)
46. {
47. if (!find(item))
48. {
49. cerr << item << " is not in the List\n";
50. return false;
51. }
52. Node\* temp1 = top;
53. Node\* temp2;
54. if (top->get\_data() == item)
55. {
56. top = top -> get\_next();
57. delete temp1;
58. return true;
59. }
60. while (temp1->get\_next()->get\_data() != item)
61. {
62. temp1 = temp1 -> get\_next();
63. }
64. temp2 = temp1 -> get\_next();
65. temp1->set\_next(temp2->get\_next());
66. delete temp2;
67. return true;
68. }
69. ostream& operator<<(ostream& out, const List& object)
70. {
71. Node\* temp = object.get\_top();
72. while (temp != 0)
73. {
74. out << \*temp << ' ';
75. temp = temp -> get\_next();
76. }
77. return out;
78. }

Library test file

1. File: main.cpp
2. #include <iostream>
3. using namespace std;
4. #include "list.h"
5. int main (void)
6. {
7. List L;
8. L.push(2);
9. L.push(4);
10. L.push(6);
11. L.push(8);
12. L.push(10);
13. cout << L << endl;
14. cout << "top value is " << L.get\_top()->get\_data() << endl;
15. if (L.find(2)) cout << 2 << " is in the list\n";
16. if (L.find(5)) cout << 5 << " is in the list\n";
17. if (L.find(6)) cout << 6 << " is in the list\n";
18. if (L.find(10)) cout << 10 << " is in the list\n";
19. cout << L.pop() << " removed from the list\n";
20. cout << L << endl;
21. L.remove(3);
22. L.remove(6);
23. cout << L << endl;
24. L.remove(2);
25. L.remove(8);
26. cout << L << endl;
27. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

10 8 6 4 2

top value is 10

2 is in the list

6 is in the list

10 is in the list

10 removed from the list

8 6 4 2

3 is not in the List

8 4 2

4

#### Linux compilation

These Linux commands are meant to demonstrate the compilation process.

1. g++ \*.cpp -Wall -c -I.
2. ar r liblinked\_list.a \*.o
3. g++ main.cpp -Wall -I. -L. -llinked\_list -o linked\_list\_test
4. ls

\*\*\*\*\*\* Output \*\*\*\*\*\*

liblinked\_list.a list.cpp list.o main.o node.h

linked\_list\_test list.h main.cpp node.cpp node.o

##### Explanation

Assumption: all files are located in the same directory for this example.

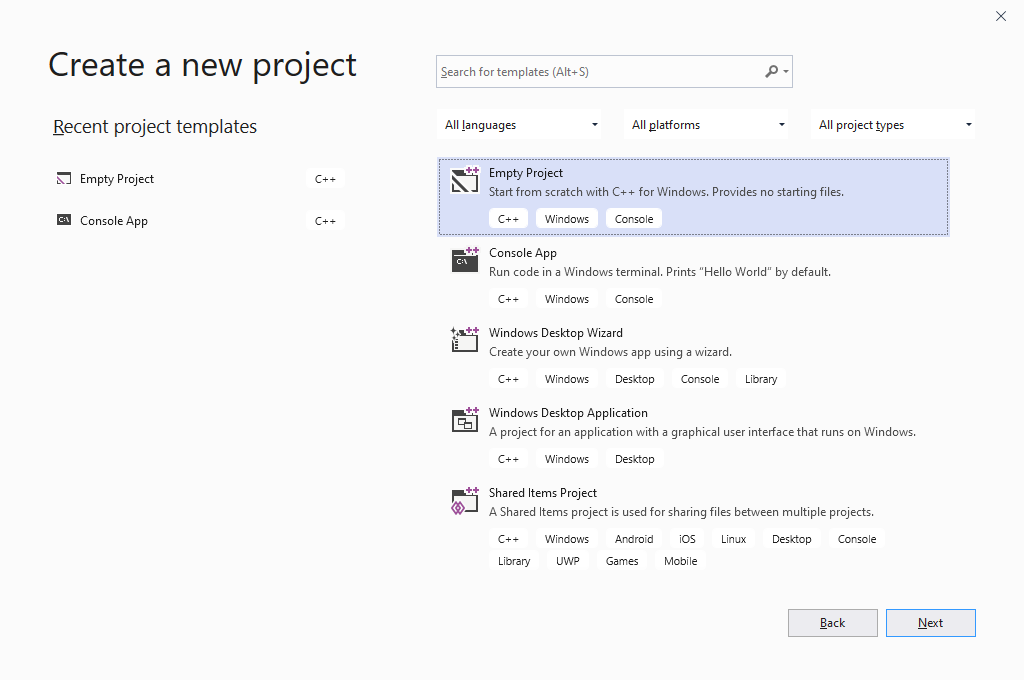
1. The two source files (node.cpp and list.cpp) are compiled. The result is two object files (node.o and list.o). The -c option means to compile only, not produce an executable file. The -I. option means to look in the current directory for include files.
2. Archive all object files into the library file, liblinked\_list.a.
3. Compile the test file, main.cpp. Identify the current directory as an include directory. Identify the current directory as a link directory. Link in the library, liblinked\_list.a. Name the output file, linked\_list\_test.
4. The ls command lists the 10 files related to this example.

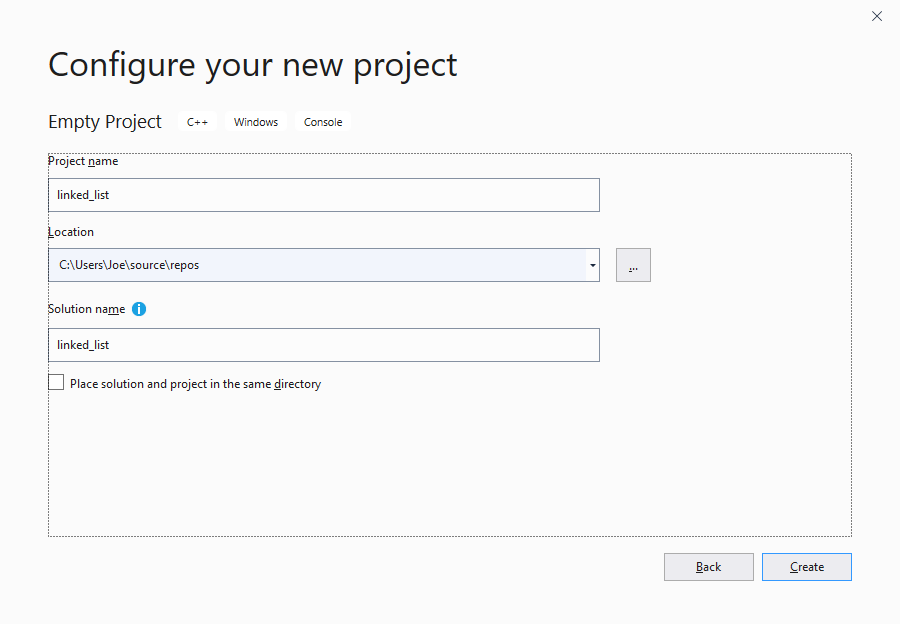
### Example 4 - Create a Static Library Using MS Visual Studio 2019

The following example demonstrates building and using a library with Microsoft Visual Studio 2019. In this example, the same files will be used to create the linked list library and to use it. For simplicity, the same directory is used for the source files, header files, the library file, and the application binary.

**Create a new project.**

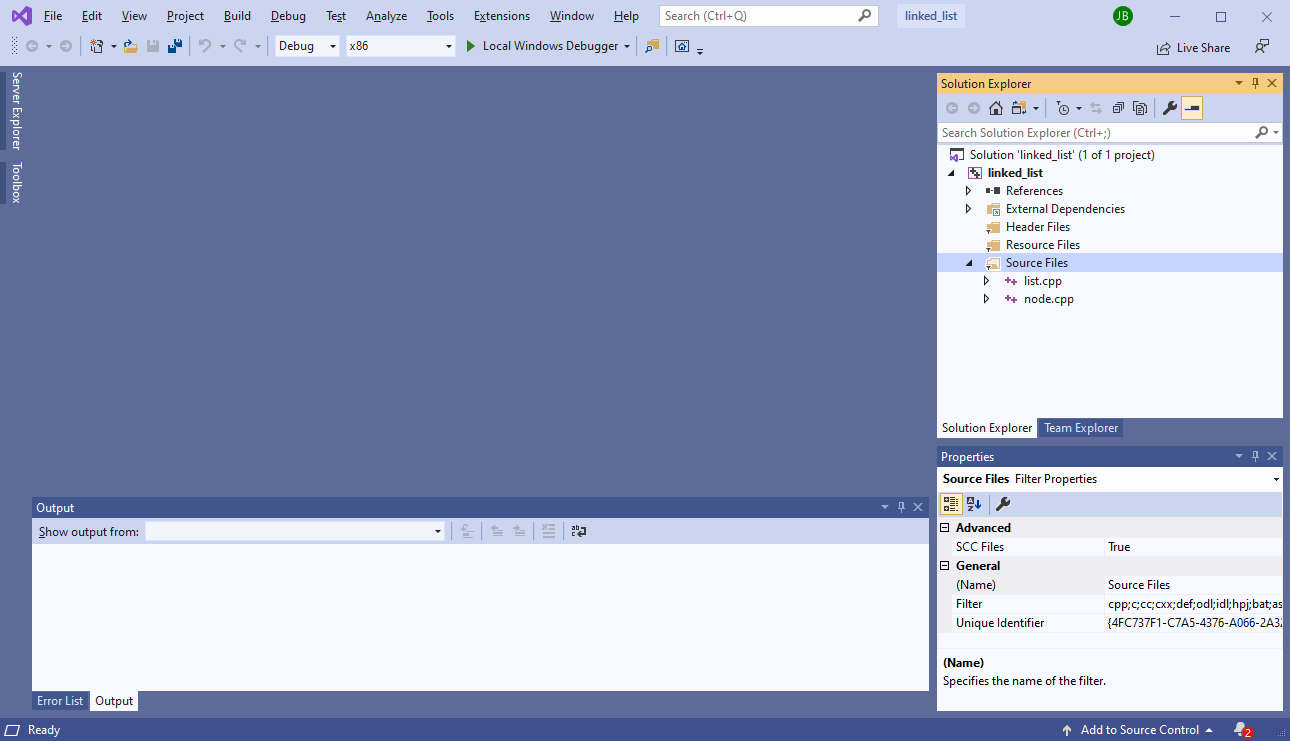
Choose **Empty Project**.





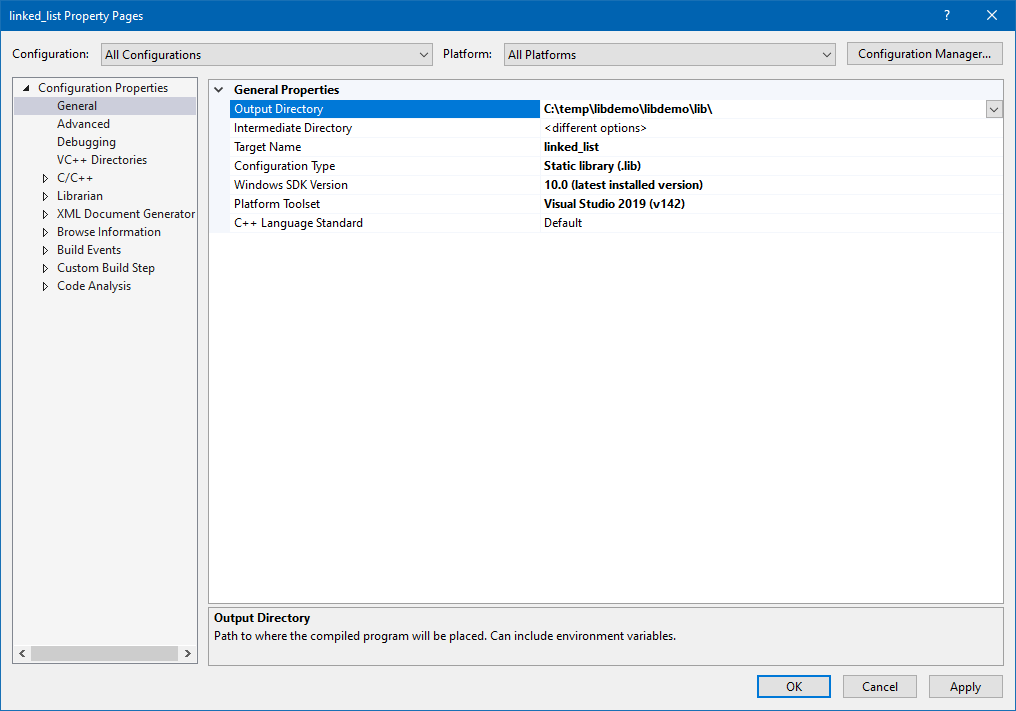
**Add the source files for the library**

Use a right-mouse click under Source Files in the Solution Explorer.



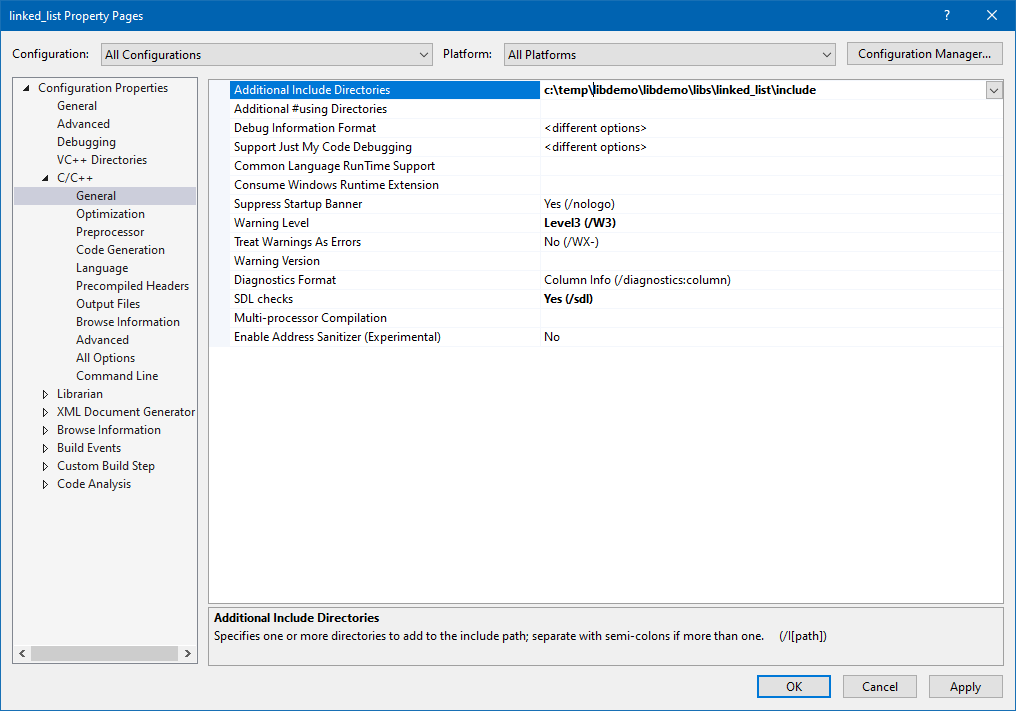
**Set the project configuration properties**

* Right-mouse click on the project name (linked\_list) and select Properties.
* In the Property Pages
  + Enter the name of the Output Directory. End directory path with a \
  + Enter the Target Name (it will default to the project name)
  + Change the Configuration Type to Static library (.lib)



**Add the include directories**

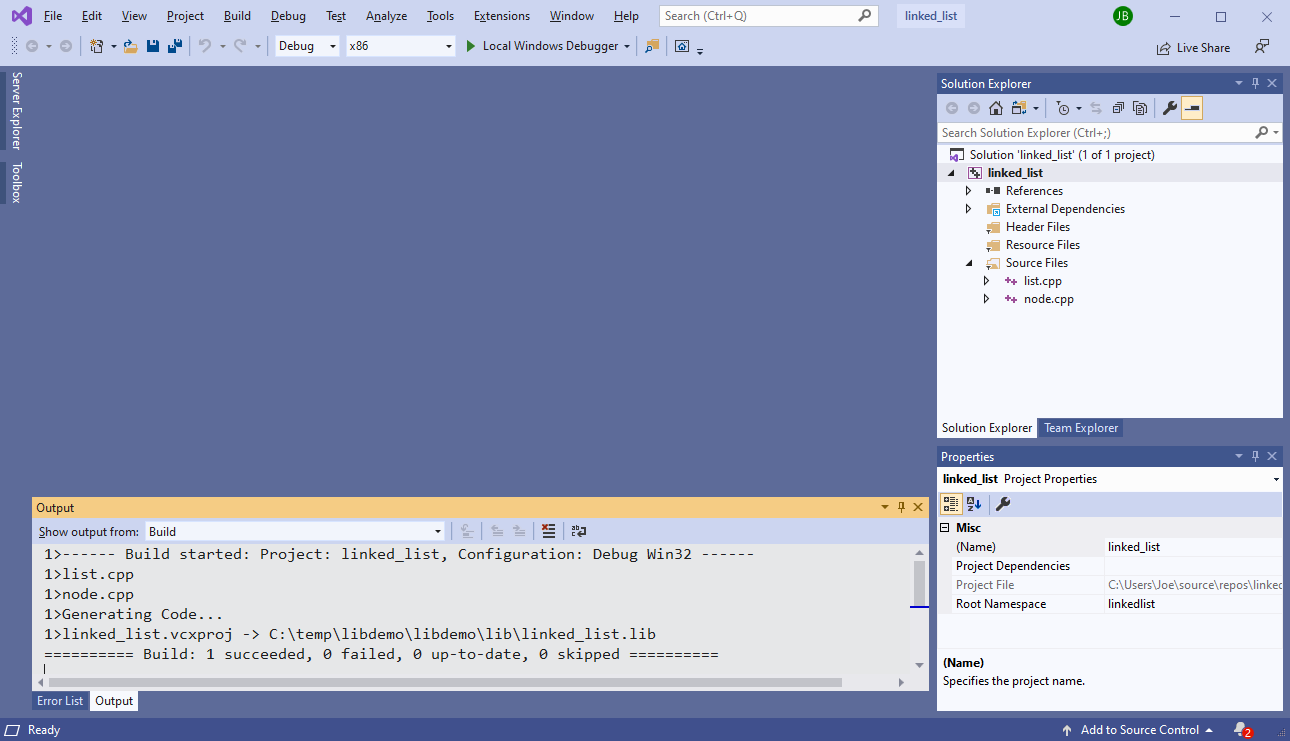
* Right-mouse click on the project name (linked\_list) and select Properties.
* In the Property Pages
  + Under Configuration Properties, expand C/C++ and select the General property
  + Click in the input area to the right of Additional Include Directories
  + Enter the directory path to the header files



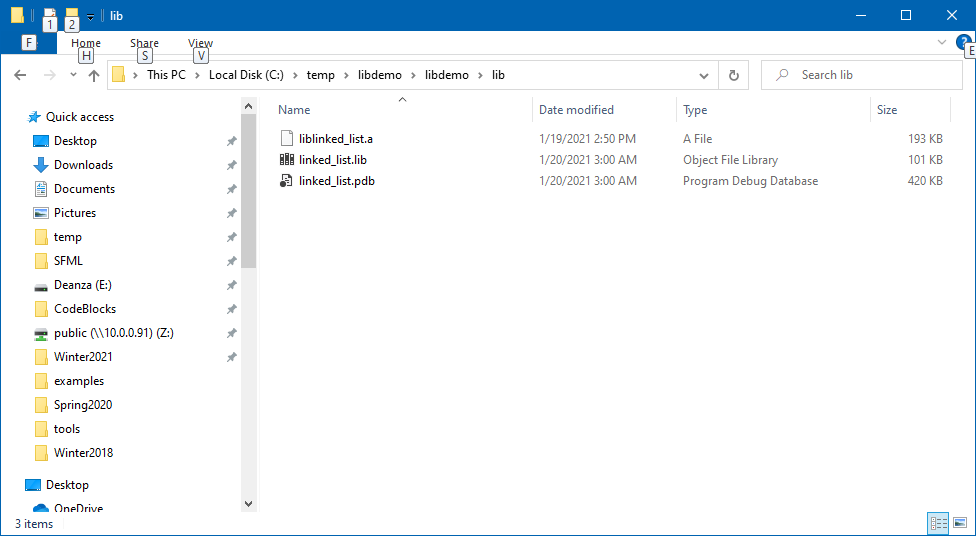
**Build the library**

Choose Build in the menu, then Build Solution.

You should see messages in the output window indicating success.

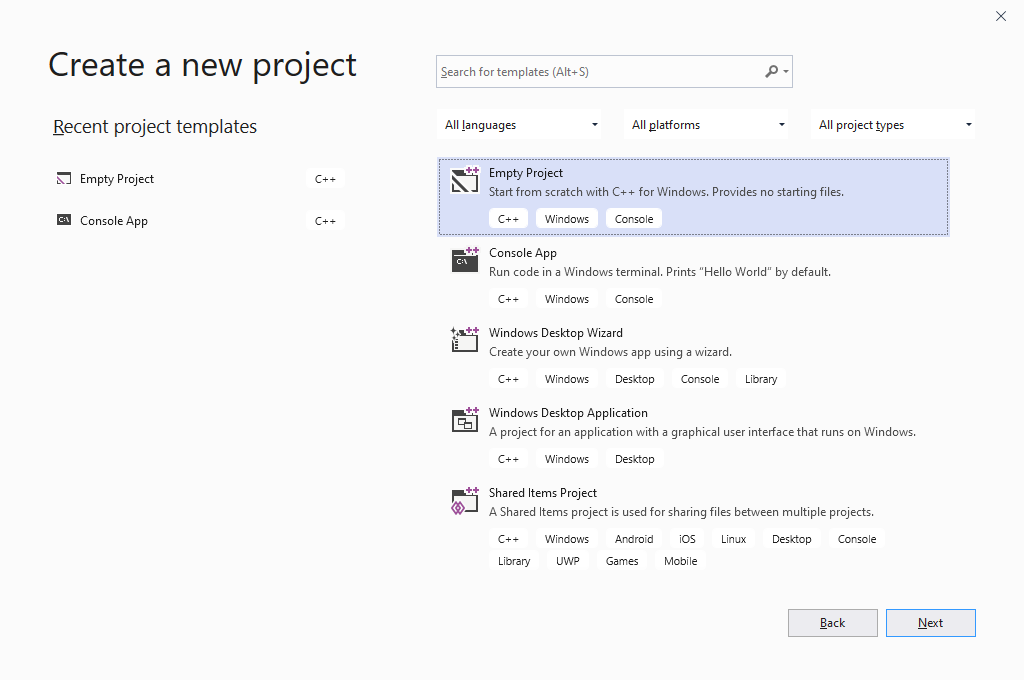


You should see the library now in your Output directory.

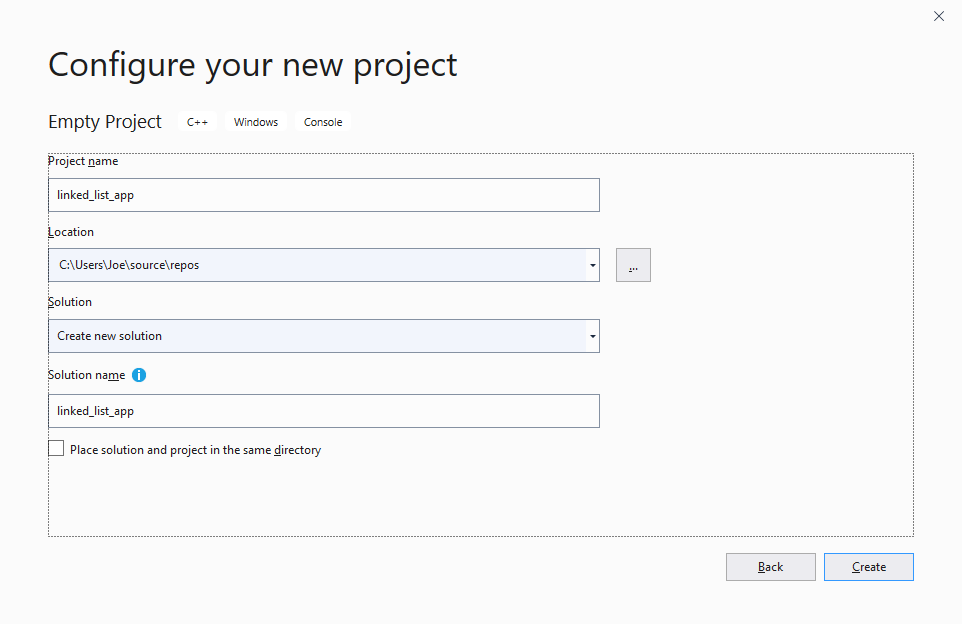


**Create the application program project.**

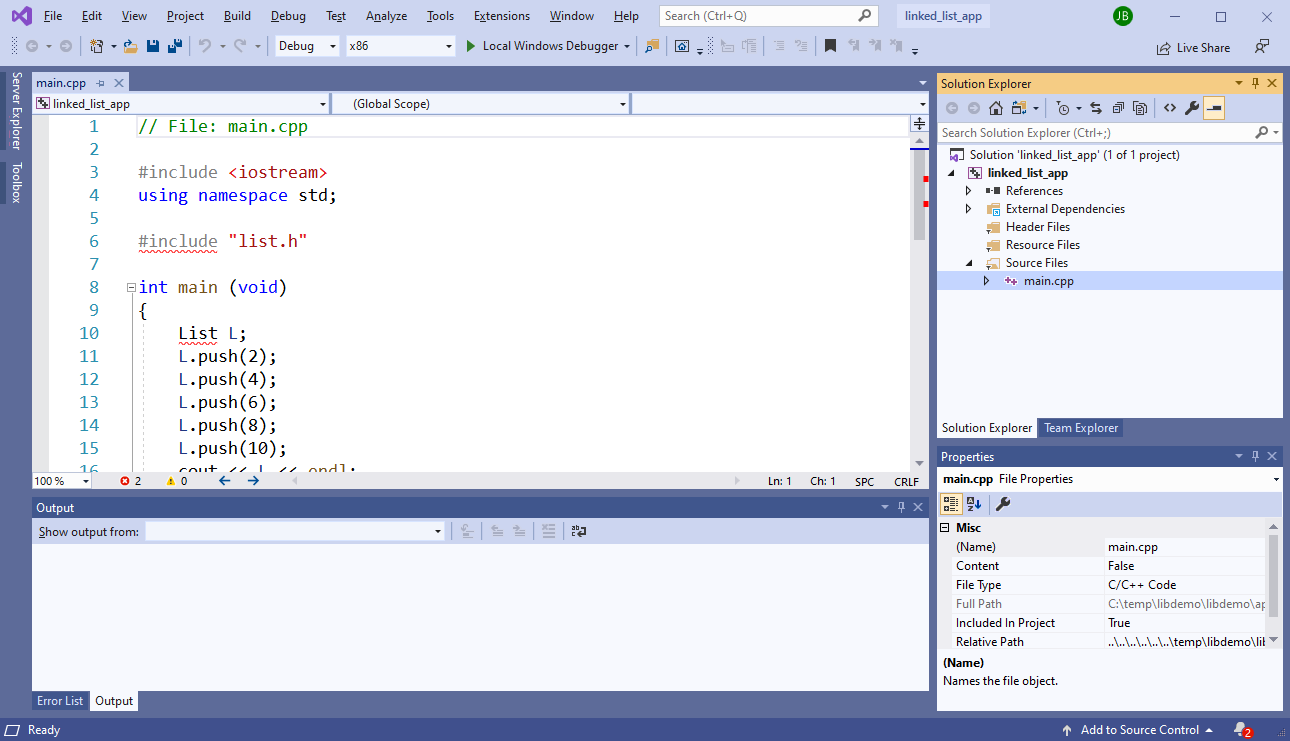
Create a new project just like you did to create the static library.



**Name the project**



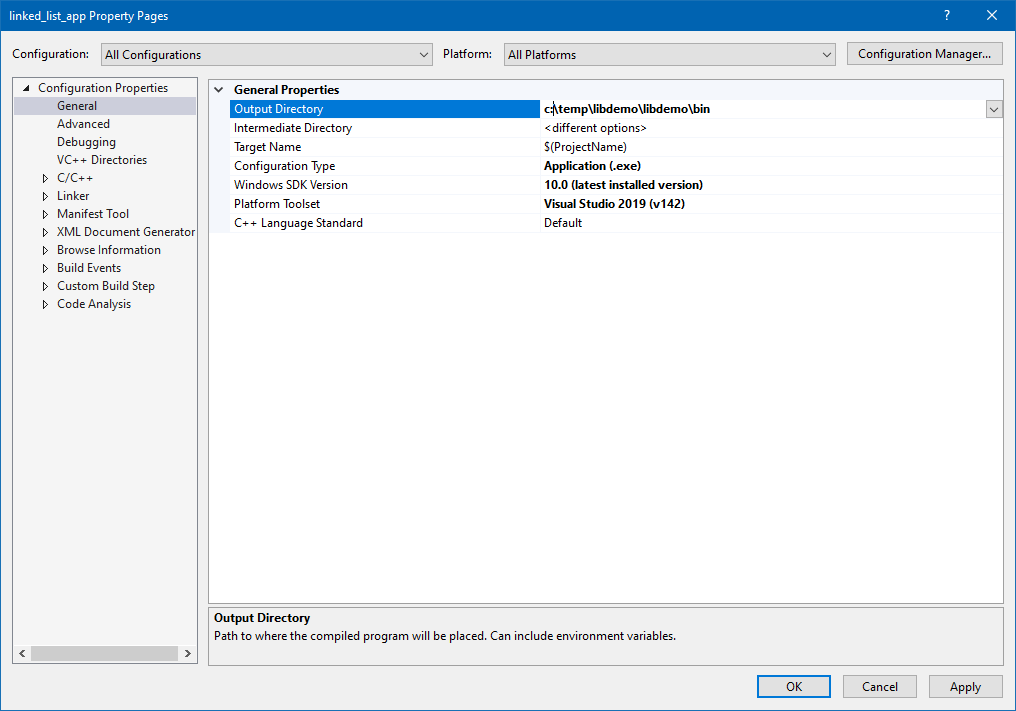
Add the source file(s) as you did earlier



**Set the Output Directory**

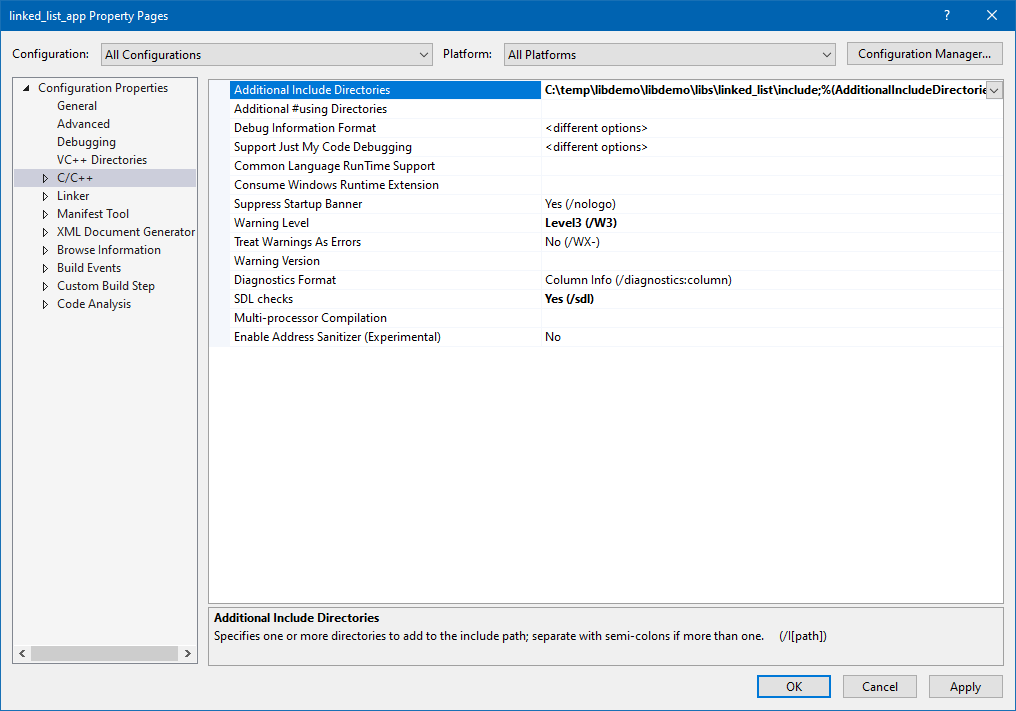
Add the Project Properties (right-mouse click) on the project name and select Properties.

In the Property Pages pop-up window, under General Configuration Properties, change the Output Directory.



**Add Include Directories**

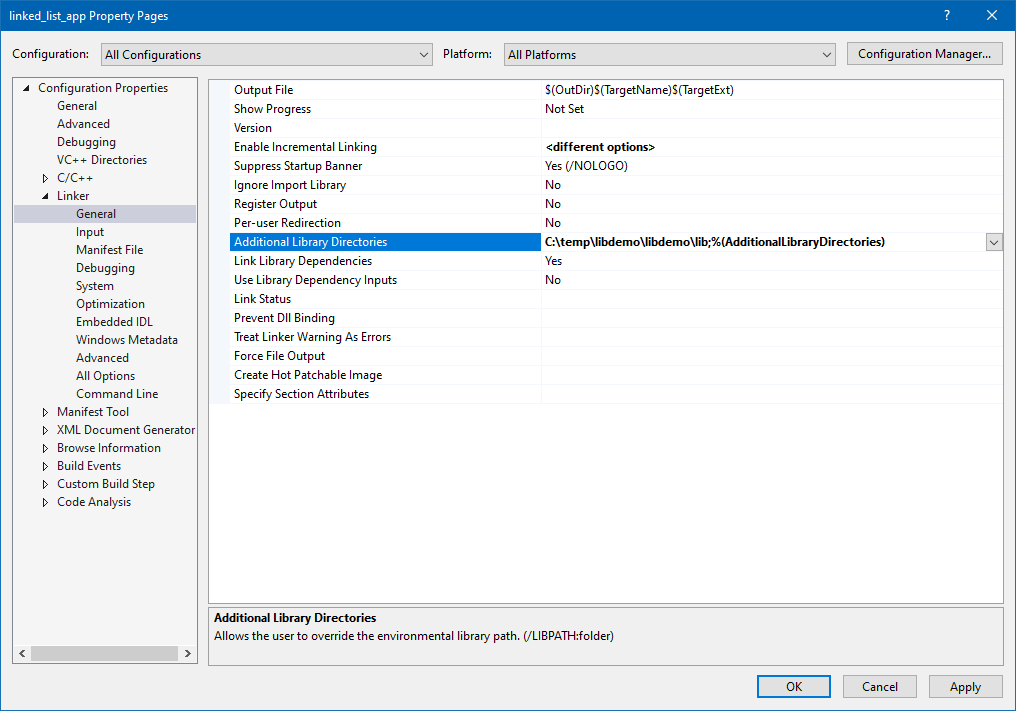
Under C/C++, add the Additional Include Directories.



**Add the Library Directory to “link in”**

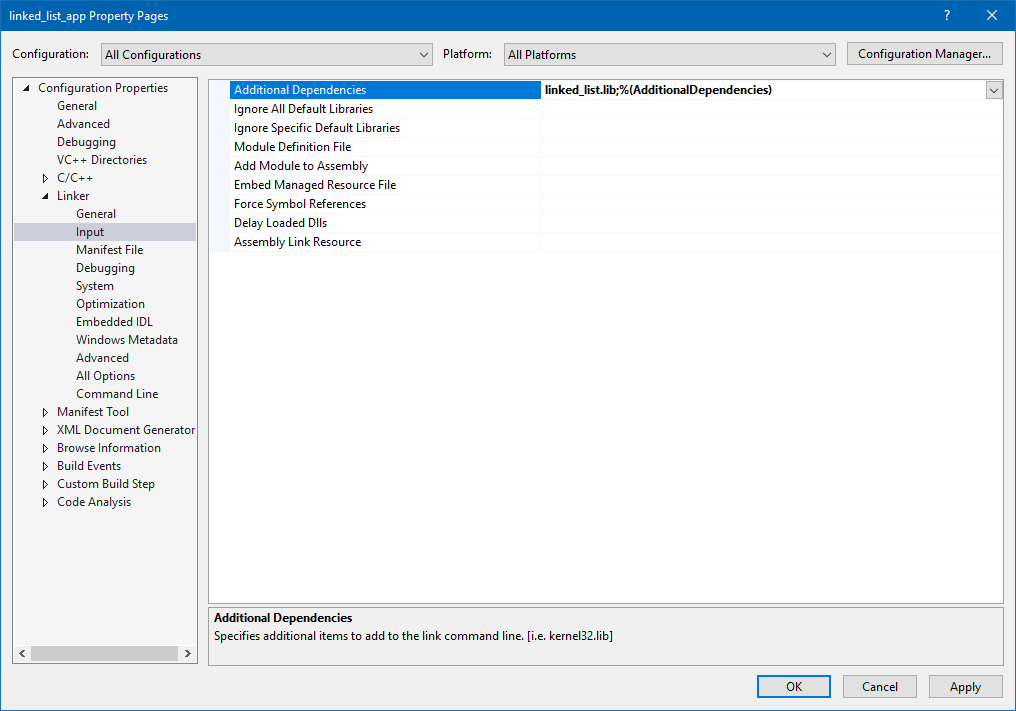
Expand the Linker Configuration Properties and select the General page

Under Additional Library Directories, add the path to the libraries to be “linked in”.



**Add the Libraries to be “linked in”**

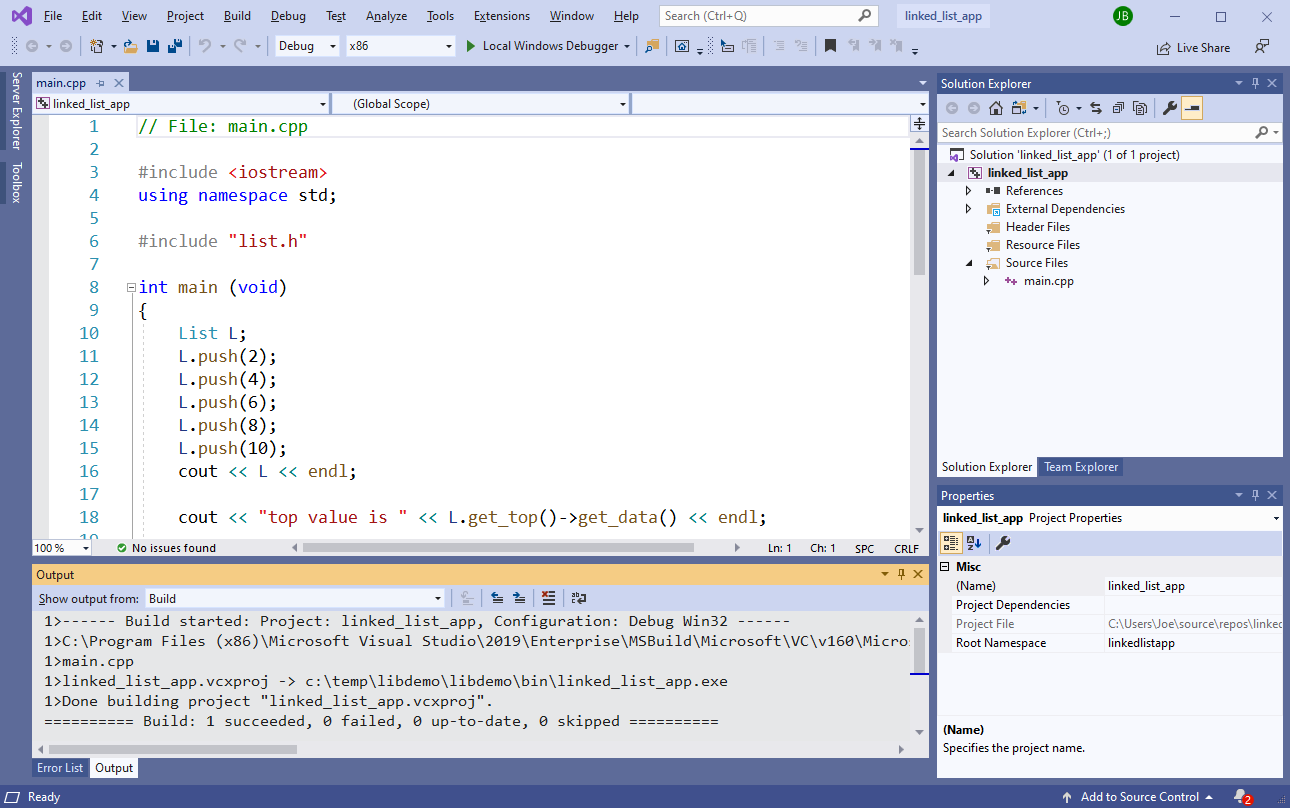
Under the Linker, Input Configuration Properties, enter the Additional Dependencies (library filenames).



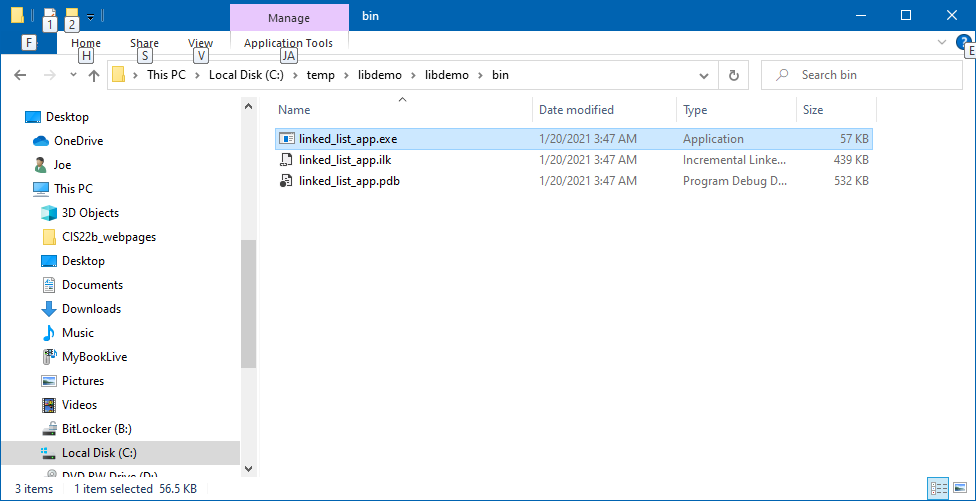
**Build and run the application**

Choose Build in the menu, then Build Solution.

You should see messages in the output window indicating success.

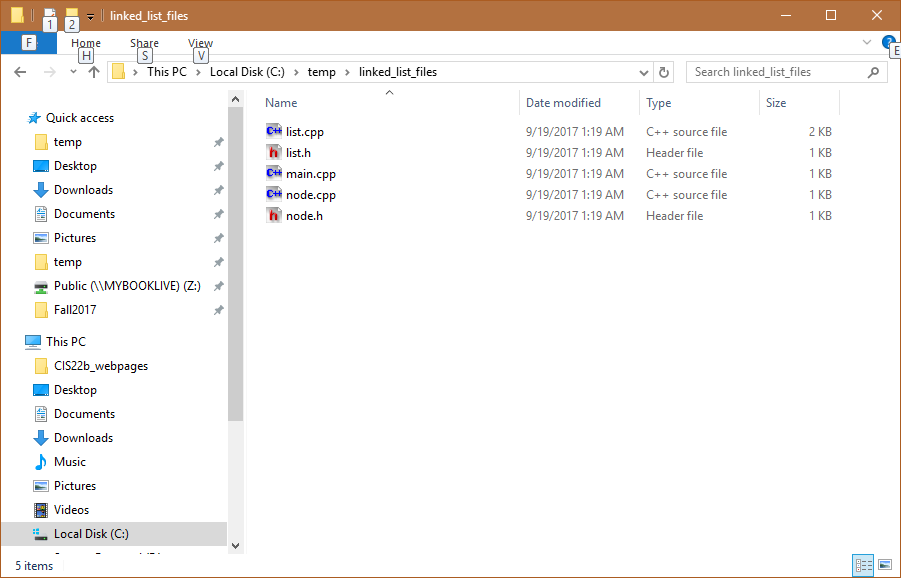


You should see the application executable file in the assigned directory location.

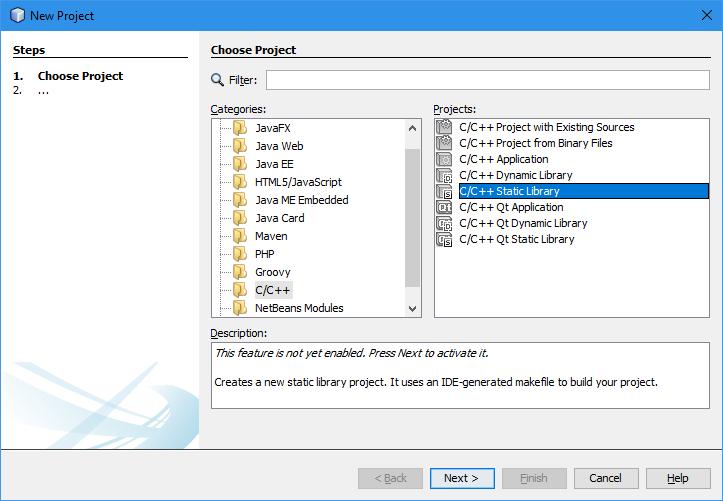


### Example 5 - Create Static Library Using NetBeans 8.2

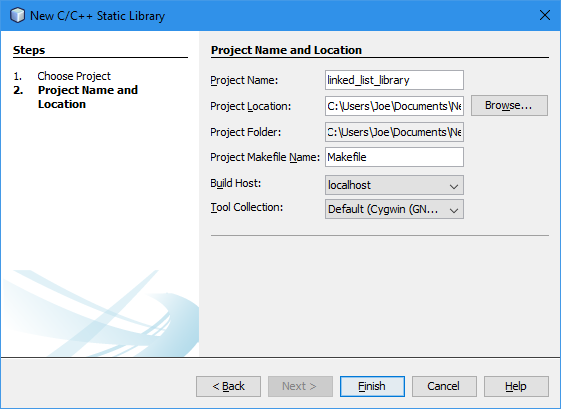
Starting File List



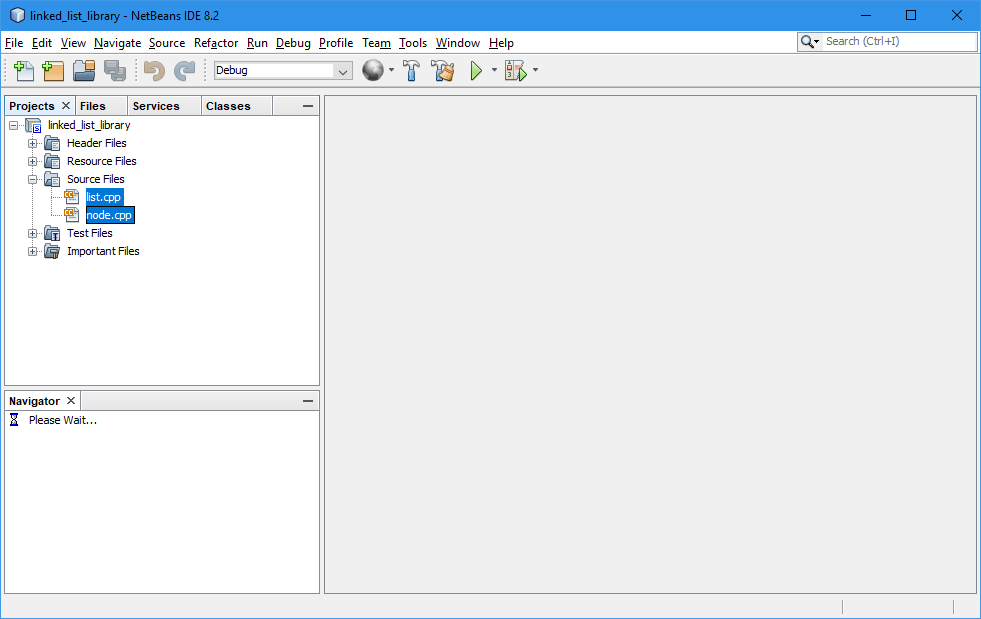
Create a new project. Select File -> New Project … -> C/C++ -> C/C++ Static Library



On the next pop-up, provide a Project Name (recommended). In this example, we will use linked\_list\_library.

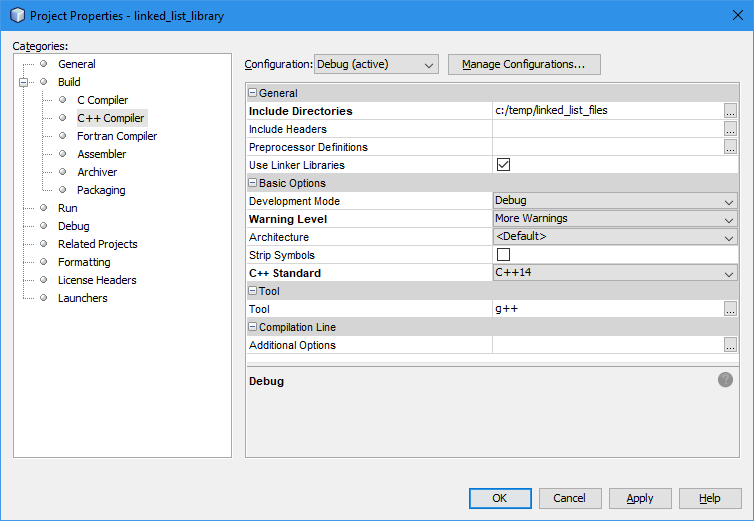


Add the source files for the library. You can use right-mouse click under Source Files in the Project Window.

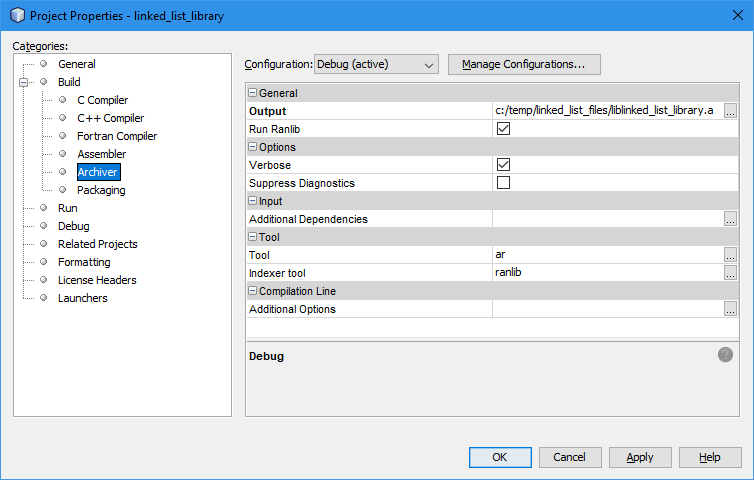


Change the project properties.

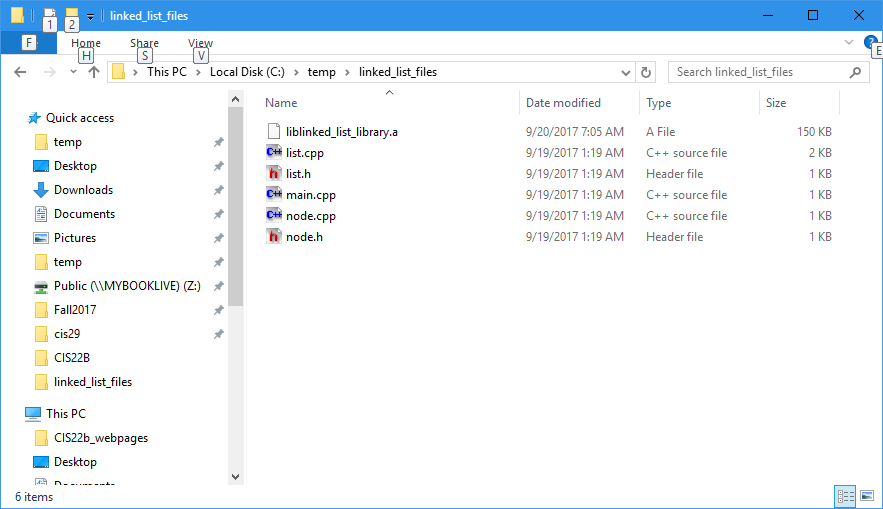
* Right-mouse click on the library name (linked\_list\_library) and select Properties.
* In the Project Properties pop-up, under Build, C++ Compiler, add the Include Directories.



* And under Achiver, change the Output directory.

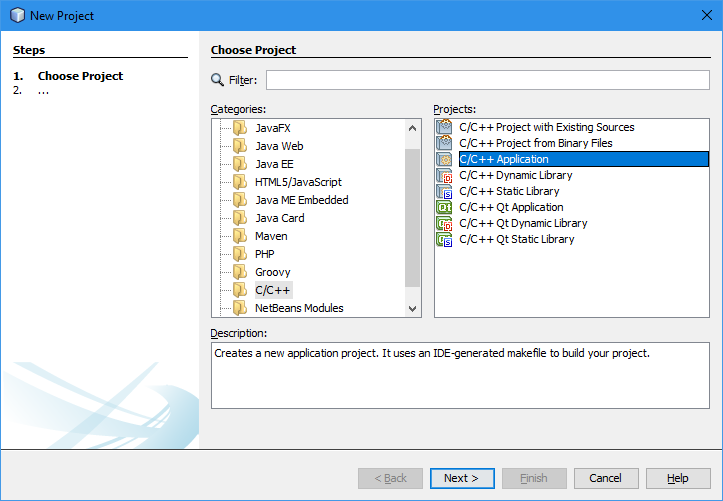


Now, you can build the library. You should see the library now in your Output directory.

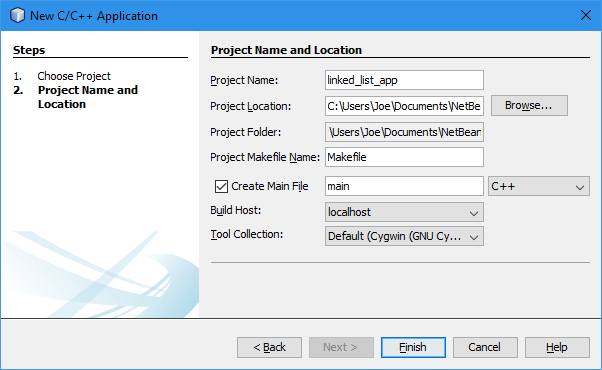


Create the application program project:

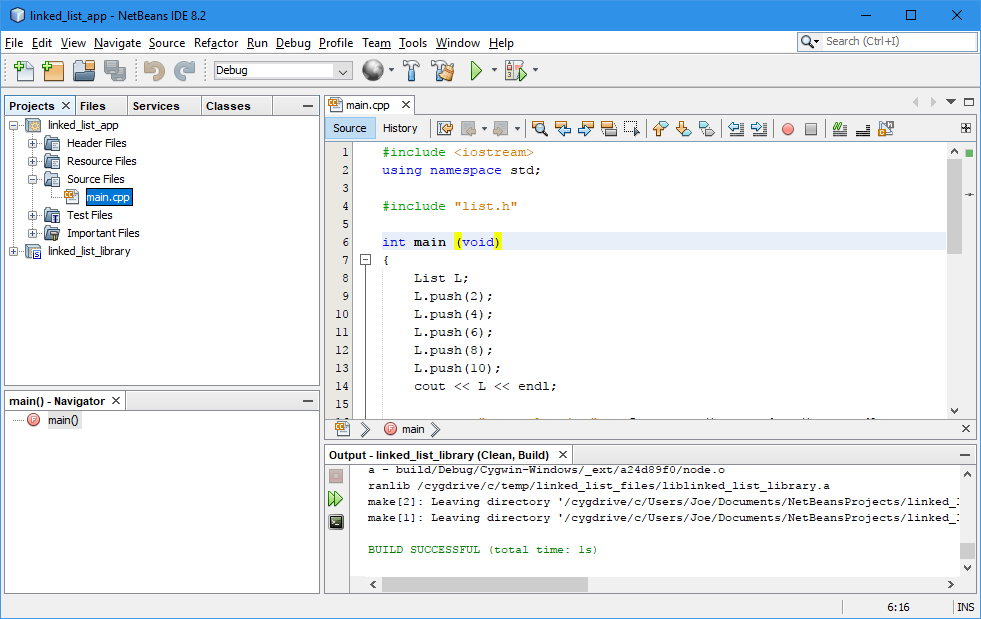
File -> New Project … -> C/C++ -> C/C++ Application



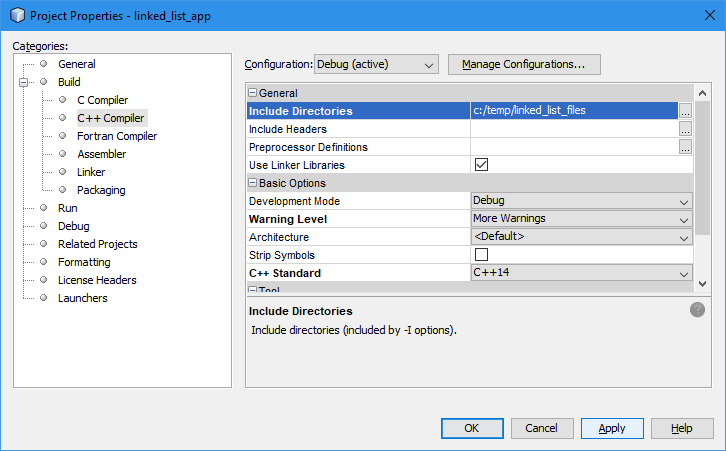
Name the project



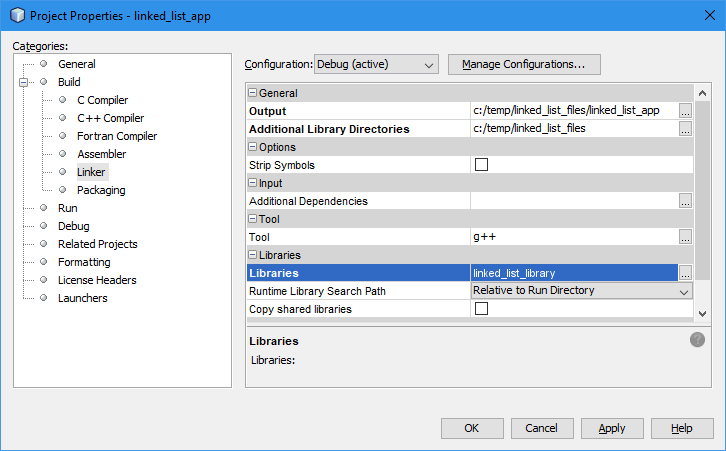
Add the source file(s)



Under Project Properties, add the Include Directories

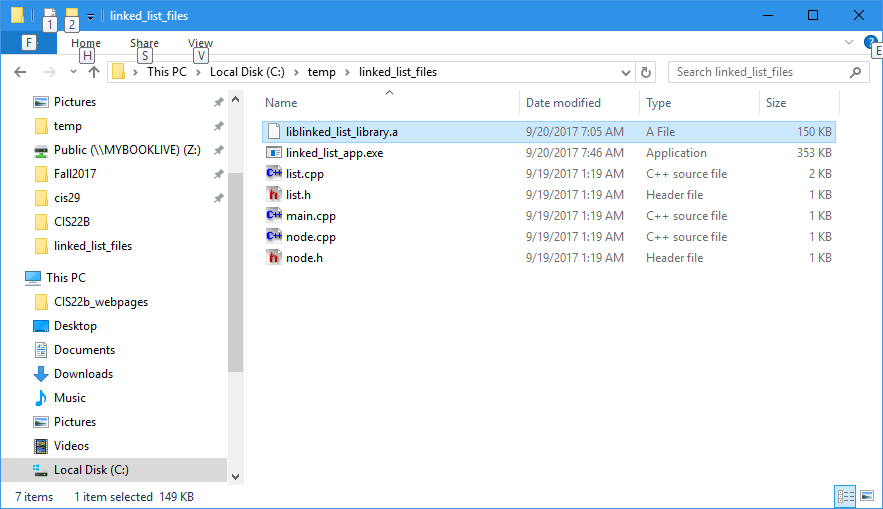


Under Linker add Output, Additional Library Directories and Libraries.



You should now be able to build and run the application.

Your file list directory should now contain the linked list executable.



### What is a shared library?

A shared library is a library file that is linked in at run time, like a dll file. Shared libraries are used on Linux and Unix. Dynamically linked libraries may not have to be present at compile time, and does not have to be present at application startup. Shared libraries must be present at both times.

#### Library extensions

|  |  |
| --- | --- |
| **Library Type** | **Extension** |
| Static | .a |
| Dynamically linked | .dll |
| Shared | .so |

### Example 6 - Create a shared library under Linux

1. $ **ls**

list.cpp list.h main.cpp node.cpp node.h

1. $ g++ -I. -shared -fPIC list.cpp node.cpp -o liblinked\_list.so
2. $ ls

liblinked\_list.so list.cpp list.h main.cpp node.cpp node.h

1. $ g++ -L. -llinked\_list main.cpp -o linked\_list\_app
2. $ ls

liblinked\_list.so linked\_list\_app list.cpp list.h main.cpp node.cpp node.h

1. $ linked\_list\_app

linked\_list\_app: error while loading shared libraries: liblinked\_list.so: cannot open shared object file: No such file or directory

1. [added current directory to LD\_LIBRARY\_PATH environment variable]
2. $ linked\_list\_app

10 8 6 4 2

top value is 10

2 is in the list

6 is in the list

10 is in the list

10 removed from the list

8 6 4 2

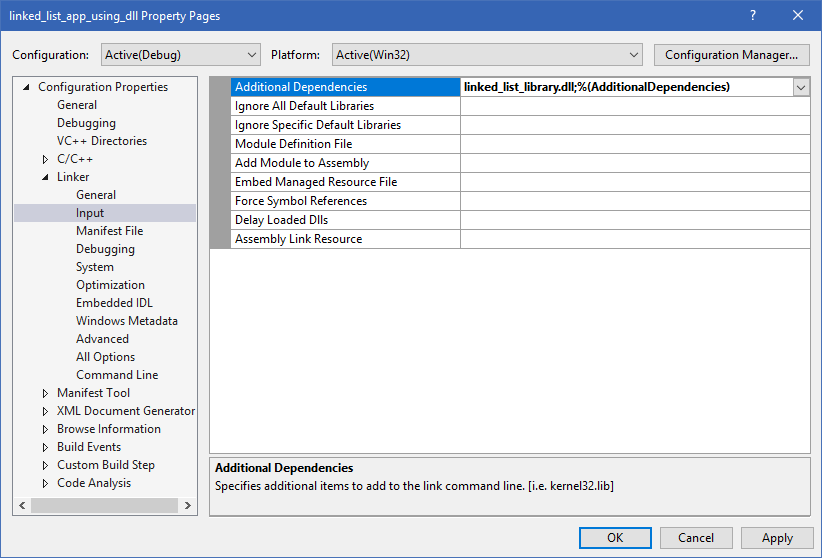
3 is not in the List

8 4 2

4

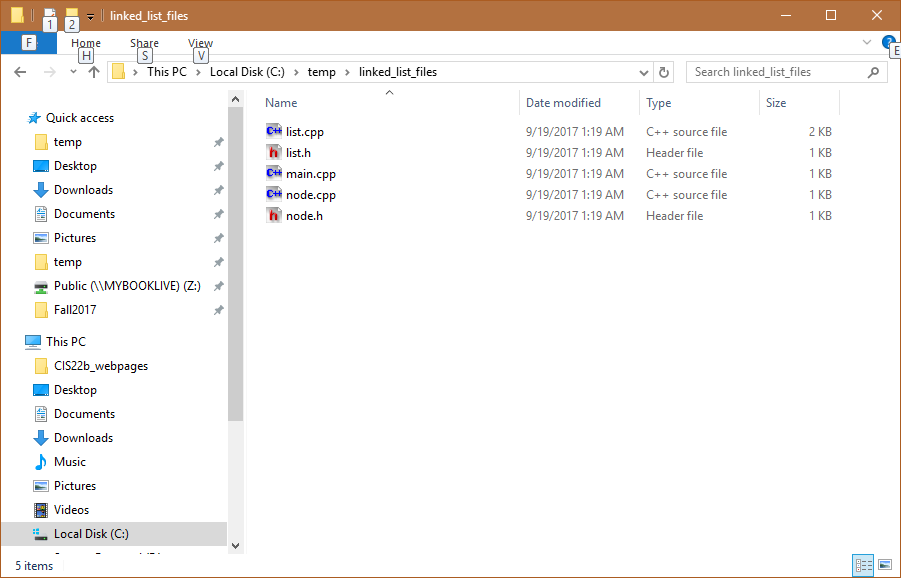
#### Explanation

1. List the files in the current directory.
2. Compile list.cpp and node.cpp into a shared library, named liblinked\_list.so. -I. means to include the current directory for compilation. The -fPIC option tells the compiler to generate position-independent code (i.e. code that can be loaded at any particular virtual memory address at runtime).
3. List the files in the current directory.
4. Compile main.cpp to the executable name linked\_list\_app. Link in the library called liblinked\_list that is located in the current directory.
5. List the files in the current directory.
6. Attempt to run the linked\_list\_app executable. The run fails because the shared library is not found.
7. The environment variable, LD\_LIBRARY\_PATH must be modified so that the current directory is also searched for the shared library.
8. The application now runs.

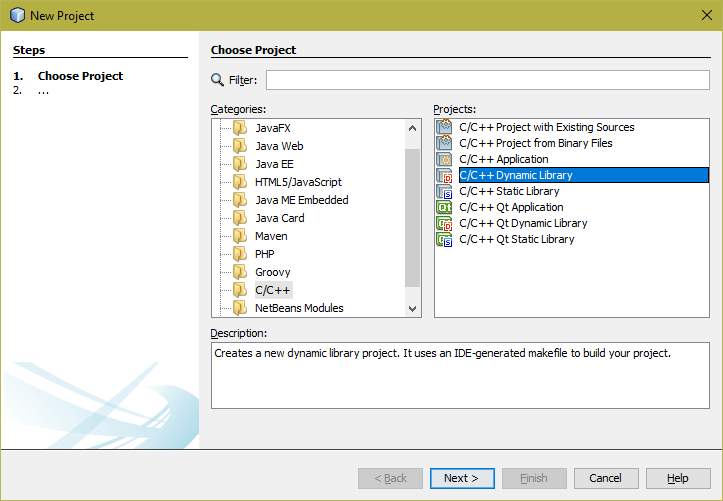


### Example 7 - Create Dynamic Library Using NetBeans 8.2 On Windows

Starting File List



Create a new project. Select File -> New Project … -> C/C++ -> C/C++ Dynamic Library

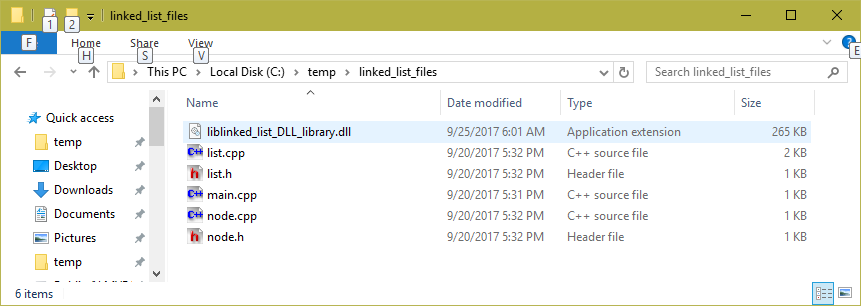


Follow the same steps that was demonstrated in the Static Library Using NetBeans.

Change the project properties.

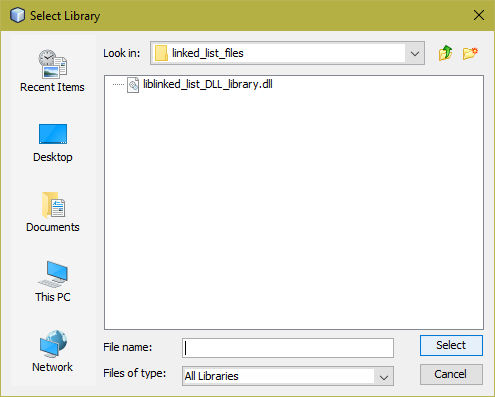
* Right-mouse click on the library name and select Properties.
* In the Project Properties pop-up, under Build, C++ Compiler, add the Include Directories.
* And under ***Linker***, change the Output directory.

Now, you can build the library. You should see the library now in your Output directory.



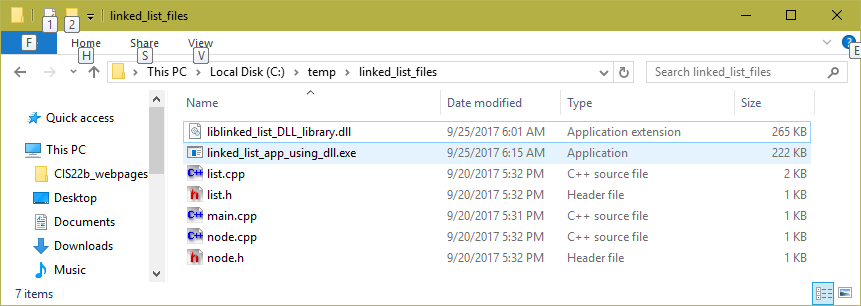
To build an application that uses the DLL library, follow the same steps that you did for an application that uses a static library.

When you select the dynamically linked library from the library directory, you should see it in the list, like this:



When you build the application, NetBeans will automatically link in the DLL library.

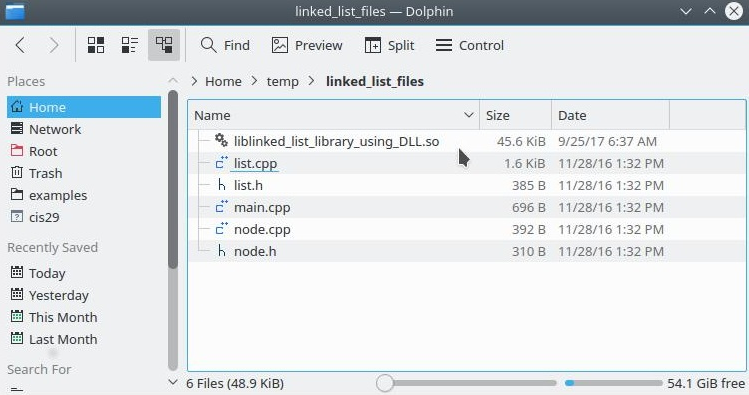
The resultant files are these:



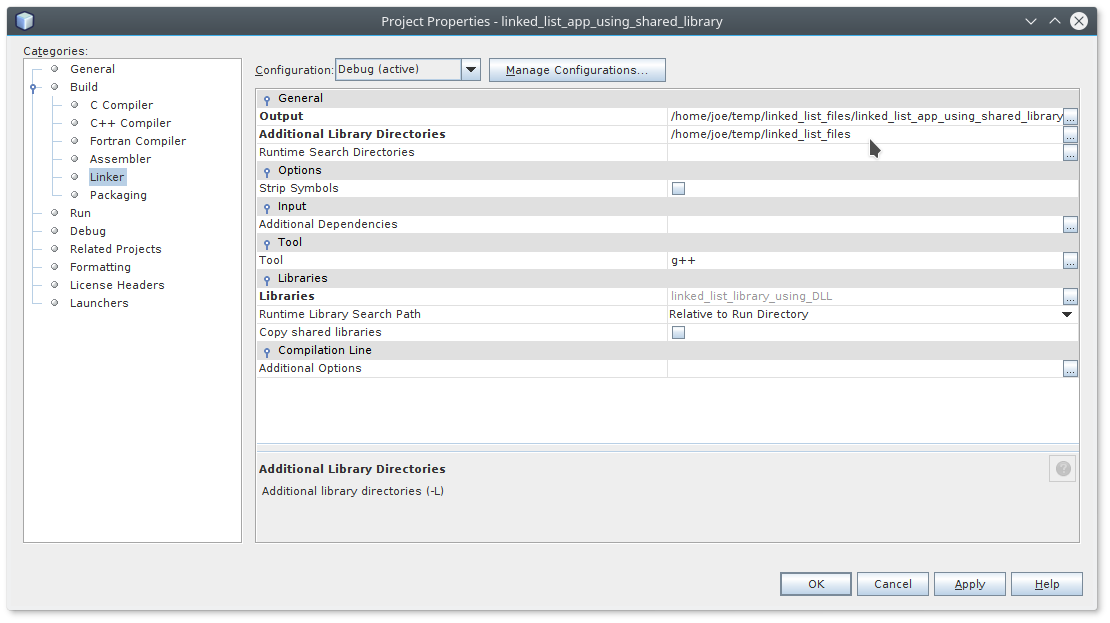
Compare the sizes of the executables of the application using static linking and dynamic linking.

### Example 8 - Create Dynamic Library Using NetBeans 8.2 On Linux

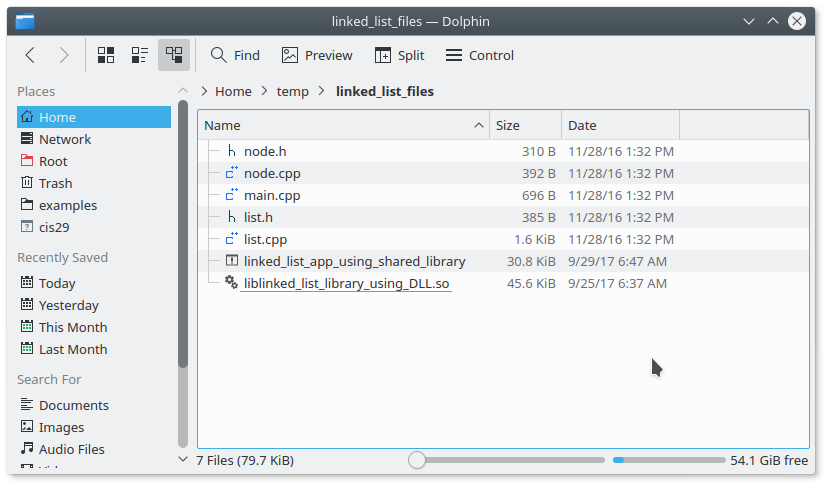
The technique for building a shared library using NetBeans on Linux is the same as building a DLL (dynamically linked library) using NetBeans on Windows. The result is a shared library as shown below.



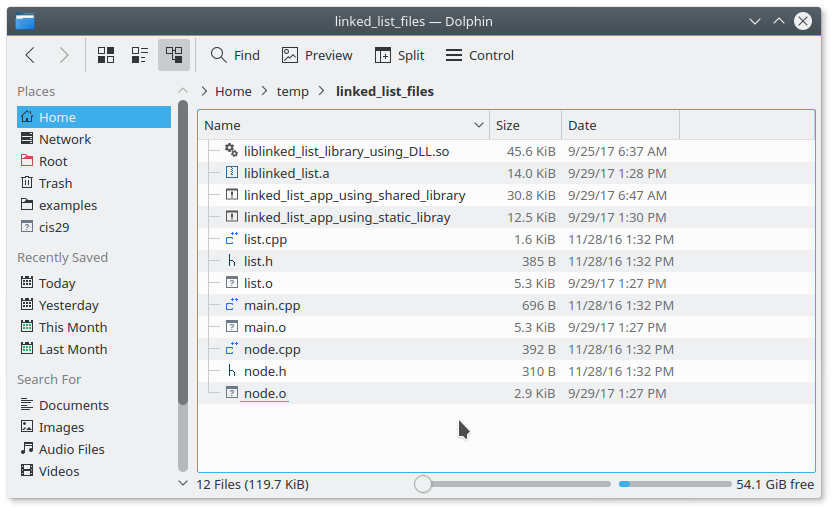
To build the application you have to “link in” the shared library as shown below. Note, the library prefix and extension are not needed here.



The following shows the application after the build using the shared library. Notice the file sizes of the shared library and the executable.



The following shows the same application built using a static library. Again, notice the file sizes.



## Using the Curl Library

The curl (or cURL) library is an open source C library that is used for downloading (and uploading internet files). This library may be used to easily retrieve files of almost any type. The library was developed as Linux/Unix as a gnu compatible library. The library is available for Linux/Unix, PC compilers that use a gnu port (Code::Blocks, NetBeans, Eclipse) and Mac IOS. The library may have to be downloaded and installed on your computer.

### Example 9 – Using cURL

1. // Command syntax: curl\_example [input file] [output file]
2. #include <iostream>
3. #include <fstream>
4. #include <cstdlib>
5. #include <string>
6. #include <cstring>
7. #include <curl/curl.h>
8. using namespace std;
9. ofstream OutFile;
10. size\_t TotalBytesDownloaded = 0;
11. size\_t writeBufferToFile(char \*buffer, size\_t dummy, size\_t numBytes, const char\* filename);
12. void getInternetFile(const char\* inputfile, const char\* outputfile);
13. int main(int argc, char\* argv[])
14. {
15. char inputFileName[256];
16. char outputFileName[256];
17. if (argc > 2) // 2 filenames given as arguments
18. {
19. strcpy(inputFileName, argv[1]);
20. strcpy(outputFileName, argv[2]);
21. }
22. else if (argc > 1) // 1 filename given as an argument
23. {
24. strcpy(inputFileName, argv[1]);
25. cout << "Enter output file => ";
26. cin >> outputFileName;
27. }
28. else
29. {
30. cout << "Enter input file => ";
31. cin >> inputFileName;
32. cout << "Enter output file => ";
33. cin >> outputFileName;
34. }
35. OutFile.open(outputFileName);
36. if (!OutFile)
37. {
38. cerr << "Unable to open output file " << outputFileName << endl;
39. exit(EXIT\_FAILURE);
40. }
41. getInternetFile(inputFileName, outputFileName);
42. cout << "Total bytes downloaded: " << TotalBytesDownloaded << endl;
43. OutFile.close();
44. }
45. size\_t writeBufferToFile(char \*buffer, size\_t dummy, size\_t numBytes, const char\* filename)
46. {
47. cout << "Writing " << numBytes << " bytes to " << filename << endl;
48. OutFile.write(buffer, numBytes);
49. TotalBytesDownloaded += numBytes;
50. return numBytes;
51. }
52. void getInternetFile(const char\* inputfile, const char\* outputfile)
53. {
54. CURL \*curl;
55. CURLcode res;
56. curl\_global\_init(CURL\_GLOBAL\_DEFAULT);
57. curl = curl\_easy\_init();
58. if (curl)
59. {
60. curl\_easy\_setopt(curl, CURLOPT\_URL, inputfile);
61. /\* Define our **callback** to get called when there's data to be written \*/
62. curl\_easy\_setopt(curl, CURLOPT\_WRITEFUNCTION, writeBufferToFile);
63. /\* Set a pointer to our struct to pass to the callback \*/
64. curl\_easy\_setopt(curl, CURLOPT\_WRITEDATA, outputfile);
65. res = curl\_easy\_perform(curl);
66. /\* always cleanup \*/
67. curl\_easy\_cleanup(curl);
68. if (CURLE\_OK != res)
69. {
70. /\* we failed \*/
71. cerr << "curl told us " << res << endl;
72. }
73. }
74. curl\_global\_cleanup();
75. }

The following execution was performed on Linux (Voyager).

Note, there is a curl include directory under /usr/include. This directory contains the header files for the curl library. If you did your own curl library install, the header files may be found in /usr/local/include.

[bentley@voyager cis29\_test]$ **ls /usr/include/curl**

curl.h curlver.h easy.h mprintf.h multi.h stdcheaders.h types.h

The curl libraries are in the directory, /usr/lib. If you did your own curl library install, the library files may be found in /usr/local/lib.

[bentley@voyager cis29\_test]$ **ls /usr/lib/\*curl\***

/usr/lib/libcurl.a /usr/lib/libcurl.so /usr/lib/libcurl.so.3 /usr/lib/libcurl.so.3.0.0

Here is the compile command

[bentley@voyager cis29\_test]$ **g++ curl\_example.cpp -Wall -o curl\_example**

Notice the link errors

/tmp/ccpFuDRi.o: In function `getInternetFile(char const\*, char const\*)':

curl\_example.cpp:(.text+0xb9): undefined reference to `curl\_global\_init'

curl\_example.cpp:(.text+0xbe): undefined reference to `curl\_easy\_init'

curl\_example.cpp:(.text+0xe4): undefined reference to `curl\_easy\_setopt'

curl\_example.cpp:(.text+0xfc): undefined reference to `curl\_easy\_setopt'

curl\_example.cpp:(.text+0x113): undefined reference to `curl\_easy\_setopt'

curl\_example.cpp:(.text+0x11c): undefined reference to `curl\_easy\_perform'

curl\_example.cpp:(.text+0x128): undefined reference to `curl\_easy\_cleanup'

curl\_example.cpp:(.text+0x15c): undefined reference to `curl\_global\_cleanup'

collect2: ld returned 1 exit status

The problem is that the linker doesn’t’ know what library to link in.

[bentley@voyager cis29\_test]$ **g++ curl\_example.cpp -Wall -o curl\_example -lcurl**

Notice that the compiler knew where to find the include files and the library files. That can be facilitated by including the appropriate directories in the $PATH and $LD\_LIBRARY\_PATH environment variables.

This execution makes use of command-line arguments.

[bentley@voyager cis29\_test]$ **curl\_example http://www.stroustrup.com/glossary.html stroupstrup\_glossary.html**

Writing 1127 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

…

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1368 bytes to stroupstrup\_glossary.html

Writing 1635 bytes to stroupstrup\_glossary.html

Total bytes downloaded: 168290

Here is the transferred file in the current directory.

[bentley@voyager cis29\_test]$ **ll stroupstrup\_glossary.html**

-rw-r--r-- 1 bentley cisStaff 168290 Dec 16 16:23 stroupstrup\_glossary.html

# Templates

## Function Templates

A function template is a feature in the language that allows the user to define a pattern for a function. Function templates are also called generic functions. The primary reason from writing function templates is to avoid having to write several overloaded versions of a function which performs the same logic on different types. For example, if you needed a function, max to return the maximum value of two numbers, you would have to write a version for int, one for floats, doubles, etc. Not to mention overloaded versions for your own class types. You will end up with:

int max(int n1,int n2);

float max(float n1,float n2);

double max(double n1 ,double n2);

long max(long n1,long n2);

char max(char n1,char n2);

my\_type max(my\_type n1,my\_type n2);

The logic for each function would be the same:

{

return a > b ? a : b ;

}

### Example 1 – Function Templates

1. #include <iostream>
2. #include <string>
3. #include <cstring>
4. using namespace std;
5. template <typename T> T Max(T a, T b)
6. {
7. return (a > b ? a : b);
8. }
9. int main(void)
10. {
11. // Testing primitive types
12. cout << Max(3,4) << endl;
13. cout << Max(4.55,1.23) << endl;
14. cout << Max('a','d') << endl;
15. cout << Max('N',Max('H','U')) << endl;
16. cout << Max('N',Max('H','U')) << endl;
17. // cout << Max(static\_cast<short>(2),3) << endl; // ERROR
18. cout << Max(static\_cast<short>(2), static\_cast<short>(3))
19. << endl << endl;
20. // Testing strings
21. string s1("Dog");
22. string s2("Cat");
23. string s3("Horse");
24. cout << Max(s1,s2) << endl;
25. cout << Max(s2,s3) << endl << endl;
26. // Testing char arrays
27. char array1[16], array2[16], array3[16];
28. strcpy(array1,"dog");
29. strcpy(array2,"cat");
30. strcpy(array3,"horse");
31. cout << Max(array1,array2) << endl;
32. cout << Max(array2,array3) << endl;
33. cout << reinterpret\_cast<long>(array1) << endl;
34. cout << reinterpret\_cast<long>(array2) << endl;
35. cout << reinterpret\_cast<long>(array3) << endl;
36. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

4

4.55

d

U

U

3

Dog

Horse

dog

cat

7012024

7012008

7011992

Comments

A function template

* begins with the keyword, template.
* This is followed by angle brackets that represent the different types used in the template. The types are identified with the keyword, typename. In the old days, the keyword class was used for this.
* Next comes a *normal-looking* function heading. In place of function argument types and return types, the typename(s) is/are used.
* The rest of the function looks *normal*.

When the function template is called, the compiler instantiates a unique version of the function using the argument types. This instantiation is called a template function.

### Example 2 – Function Templates with an overloaded function

1. #include <iostream>
2. #include <string>
3. #include <cstring>
4. using namespace std;
5. template <typename T> T Max(T a, T b)
6. {
7. return (a > b ? a : b);
8. }
9. char\* Max(char\* a, char\* b)
10. {
11. return ((strcmp(a,b) > 0) ? a : b);
12. }
13. int main(void)
14. {
15. // Testing primitive types
16. cout << Max(3,4) << endl;
17. cout << Max(4.55,1.23) << endl;
18. cout << Max('a','d') << endl;
19. cout << Max('N',Max('H','U')) << endl;
20. cout << Max('N',Max('H','U')) << endl;
21. // cout << Max(static\_cast<short>(2),3) << endl; // ERROR
22. cout << Max(static\_cast<short>(2), static\_cast<short>(3)
23. << endl << endl;
24. // Testing strings
25. string s1("Dog");
26. string s2("Cat");
27. string s3("Horse");
28. cout << Max(s1,s2) << endl;
29. cout << Max(s2,s3) << endl << endl;
30. // Testing char arrays
31. char array1[16], array2[16], array3[16];
32. strcpy(array1,"dog");
33. strcpy(array2,"cat");
34. strcpy(array3,"horse");
35. cout << Max(array1,array2) << endl;
36. cout << Max(array2,array3) << endl;
37. cout << reinterpret\_cast<long>(array1) << endl;
38. cout << reinterpret\_cast<long>(array2) << endl;
39. cout << reinterpret\_cast<long>(array3) << endl;
40. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

4

4.55

d

U

U

3

Dog

Horse

dog

horse

7012024

7012008

7011992

### Example 3 – A Function Template that always returns a double

1. #include <iostream>
2. using namespace std;
3. template <typename Z> double half(Z n)
4. {
5. return static\_cast<double>(n/2.);
6. }
7. int main(void)
8. {
9. cout << half(3) << endl;
10. cout << half(4.55) << endl;
11. cout << half(static\_cast<short>(2)) << endl;
12. cout << half(static\_cast<long>(19)) << endl;
13. cout << half(1/2) << endl;
14. cout << half('x') << endl;
15. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

1.5

2.275

1

9.5

0

60

### Example 4 – A Function Template with an array argument

#include <iostream>

#include <cstring>

using namespace std;

template <typename T> double average(T\* n,int size)

{

double sum = 0;

for (int i = 0; i < size; i++) sum += \*(n+i);

return sum/size;

}

int main()

{

int x[5] = {2,4,7,8,9};

double y[3] = {7.8,9.1,0.9};

unsigned short z[4] = {2,4,6,8};

const char cstring[] = "ABCD";

cout << average(x,5) << endl;

cout << average(y,3) << endl;

cout << average(z,4) << endl;

cout << average(cstring,strlen(cstring));

}

\*\*\*\*\*\* Output \*\*\*\*\*\*

6

5.93333

5

66.5

### Example 5 – A Function Template using two types

1. #include <iostream>
2. using namespace std;
3. template <typename X, typename Y> void print\_em(X a, Y b)
4. {
5. cout.setf(ios::right,ios::adjustfield);
6. cout.width(10);
7. cout << static\_cast<long>(a);
8. cout.precision(2);
9. cout.setf(ios::showpoint);
10. cout.width(10);
11. cout << static\_cast<double>(b) << endl;
12. }
13. int main(void)
14. {
15. print\_em(3,4);
16. print\_em(3,5.7);
17. print\_em(5.11,9);
18. print\_em(static\_cast<short>(3),7.777);
19. print\_em(5,static\_cast<float>(3.456));
20. print\_em('A',5);
21. print\_em(5,'A');
22. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

3 4.0

3 5.7

5 9.0

3 7.8

5 3.5

65 5.0

5 65.

### Example 6 – A Function Template with a user defined type

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. class Card
5. {
6. private:
7. int pips;
8. int suit;
9. public:
10. Card(int n = 0) : pips(n % 13), suit(n / 13)
11. { }
12. bool operator>(const Card& c) const
13. {
14. return pips > c.pips;
15. }
16. static const string pips\_name[13];
17. static const string suit\_name[4];
18. friend ostream& operator<<(ostream&, const Card&);
19. };
20. const string Card::pips\_name[13] =
21. {"two","three","four","five","six","seven",
22. "eight","nine","ten","jack","queen","king","ace"};
23. const string Card::suit\_name[4] =
24. {"clubs","diamonds","hearts","spades"};
25. ostream& operator<<(ostream& out, const Card& card)
26. {
27. out << Card::pips\_name[card.pips] << " of " <<
28. Card::suit\_name[card.suit];
29. return out;
30. }
31. template <typename T> const T& Max(const T& a, const T& b)
32. {
33. return (a > b) ? a : b;
34. }
35. int main(void)
36. {
37. cout << Max(3,4) << endl;
38. Card c1(23), c2(9);
39. cout << c1 << endl;
40. cout << c2 << endl;
41. cout << Max(c1,c2) << endl;
42. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

4

queen of diamonds

jack of clubs

queen of diamonds

### Example 7 – A Function Template in header files

1. #ifndef FT7\_H
2. #define FT7\_H
3. #include <iostream>
4. template <typename U> void swap(U& a,U& b)
5. {
6. U temp;
7. temp = a;
8. a = b;
9. b = temp;
10. }
11. template <typename T> void sort(T\* a,int size)
12. {
13. int i,j;
14. for (i = 1; i < size; i++)
15. for (j = 0; j < i; j++)
16. if ( a[i] < a[j] ) swap(a[i],a[j]);
17. }
18. template <typename V> void arrayPrint(const V\* a,int size)
19. {
20. int i;
21. for (i = 0; i < size; i++) std::cout << a[i] << std::endl;
22. std::cout << std::endl;
23. }
24. #endif
25. #include "ft7.h"
26. #include <iostream>
27. using namespace std;
28. class fraction
29. {
30. private:
31. int numer,denom;
32. public:
33. fraction(int n = 0, int d = 1) : numer(n), denom(d) {}
34. void assign(int n, int d)
35. {
36. numer = n;
37. denom = d;
38. }
39. int operator<(fraction& f);
40. friend ostream& operator<<(ostream& s, const fraction& f);
41. };
42. int fraction::operator<(fraction& f)
43. {
44. return (static\_cast<float>(numer)/denom <
45. static\_cast<float>(f.numer)/f.denom);
46. }
47. ostream& operator<<(ostream& s,const fraction& f)
48. {
49. s << f.numer << '/' << f.denom;
50. return s;
51. }
52. class Card
53. {
54. protected:
55. int pips;
56. int suit;
57. public:
58. Card(int n = 0) : pips(n % 13), suit(n / 13)
59. { }
60. bool operator<(const Card& c) const
61. {
62. return pips < c.pips;
63. }
64. static const string pips\_name[13];
65. static const string suit\_name[4];
66. friend ostream& operator<<(ostream&, const Card&);
67. };
68. const string Card::pips\_name[13] = {"two","three","four",”five”,
69. "six","seven","eight","nine","ten","jack","queen","king","ace"};
70. const string Card::suit\_name[4] =
71. {"clubs","diamonds","hearts","spades"};
72. ostream& operator<<(ostream& out, const Card& card)
73. {
74. out << Card::pips\_name[card.pips] << " of " <<
75. Card::suit\_name[card.suit];
76. return out;
77. }
78. class PinocleCard : public Card
79. {
80. public:
81. PinocleCard(int n = 0) : Card(n)
82. {
83. pips = n % 6 + 7;
84. suit = n / 2 % 4;
85. }
86. int operator<(PinocleCard&);
87. };
88. int PinocleCard::operator<(PinocleCard& c)
89. {
90. if (pips != 8 && c.pips != 8) return (pips < c.pips);
91. else if (pips == 8 && c.pips != 12) return 0;
92. else if (c.pips == 8 && pips != 12) return 1;
93. else return 0;
94. }
95. int main()
96. {
97. // array of int
98. int a1[5] = { 3, 5, 1, 9, 94};
99. arrayPrint(a1,5);
100. sort(a1,5);
101. arrayPrint(a1,5);
102. // array of double
103. double a2[4] = { 3.7, 1.5, -1.1,.9};
104. arrayPrint(a2,4);
105. sort(a2,4);
106. arrayPrint(a2,4);
107. // array of char
108. char a3[4] = {"hey"};
109. arrayPrint(a3,3);
110. sort(a3,3);
111. arrayPrint(a3,3);
112. // array of fractions
113. fraction a4[4] {{2,3},{1,2},{3,4},{5,9}};
114. arrayPrint(a4,4);
115. sort(a4,4);
116. arrayPrint(a4,4);
117. // array of cards
118. Card a5[4] = {47,23,43,1};
119. arrayPrint(a5,4);
120. sort(a5,4);
121. arrayPrint(a5,4);
122. // array of PinocleCards
123. PinocleCard a6[6] = {32,18,41,10,13,27};
124. arrayPrint(a6,6);
125. sort(a6,6);
126. arrayPrint(a6,6);
127. }

**\*\*\*\*\*\* Output \*\*\*\*\*\***

3

5

1

9

94

1

3

5

9

94

3.7

1.5

-1.1

0.9

-1.1

0.9

1.5

3.7

h

e

y

e

h

y

2/3

1/2

3/4

5/9

1/2

5/9

2/3

3/4

ten of spades

queen of diamonds

six of spades

three of clubs

three of clubs

six of spades

ten of spades

queen of diamonds

jack of clubs

nine of diamonds

ace of clubs

king of diamonds

ten of hearts

queen of diamonds

nine of diamonds

jack of clubs

queen of diamonds

king of diamonds

ten of hearts

ace of clubs

## Class Templates

A class template is a class definition that contains a generic type, and one or more function templates. Just like function templates, instantiations of a class template are called template classes. Class templates are commonly used with container classes.

### Example 8 – class template

1. #include <iostream>
2. #include <string>
3. #include <typeinfo>
4. using namespace std;
5. template <typename T>
6. class Thing
7. {
8. private:
9. T x;
10. public:
11. Thing();
12. Thing(T);
13. Thing(const Thing<T>&);
14. T get() const;
15. operator T() const;
16. };
17. template <typename T>
18. Thing<T>::Thing() : x(0) {}
19. template <typename T>
20. Thing<T>::Thing(T n) : x(n) {}
21. template <typename T>
22. Thing<T>::Thing(const Thing<T>& t) : x(t.x) {}
23. template <typename T>
24. T Thing<T>::get() const
25. {
26. return x;
27. }
28. template <typename T>
29. Thing<T>::operator T() const
30. {
31. return x;
32. }
33. template <typename T>
34. ostream& operator<<(ostream& s, const Thing<T>& t)
35. {
36. return s << t.get();
37. }
38. int main(void)
39. {
40. Thing<int> t1;
41. cout << "t1=" << t1 << endl;
42. Thing<int> t2(18);
43. cout << "t2=" << t2 << endl;
44. Thing<double> t3(1.28);
45. cout << "t3=" << t3 << endl;
46. Thing<double> t4(t3);
47. cout << "t4=" << t4 << endl;
48. cout << "(t2.get() + t3.get()) = " << (t2.get() + t3.get()) <<
49. endl;
50. cout << "t2 + t3 = " << t2 + t3 << endl;
51. Thing<char> t5('z');
52. cout << "t5=" << t5 << endl;
53. Thing<string> t6("howdy");
54. cout << "t6=" << t6 << endl;
55. cout << t6.get()[2] << endl;
56. }

**\*\*\*\*\*\* Output \*\*\*\*\*\***

t1=0

t2=18

t3=1.28

t4=1.28

(t2.get() + t3.get()) = 19.28

t2 + t3 = 19.28

t5=z

t6=howdy

w

### Example 9 – class template: a generic array

1. #include <iostream>
2. #include <cstdlib>
3. using namespace std;
4. template <typename T>
5. class Array
6. {
7. private:
8. T\* ptrT;
9. int size;
10. public:
11. Array(): ptrT(0), size(0) {}
12. Array(int);
13. T& operator[](int);
14. };
15. template <typename T>
16. Array<T>::Array(int n) : ptrT(new T[n]), size(n)
17. {
18. for (int i = 0; i < size; i++) ptrT[i] = 0;
19. }
20. template <typename T>
21. T& Array<T>::operator[](int index)
22. {
23. if (index < 0 || index >= size)
24. {
25. cerr << "invalid Array index\n";
26. return \*ptrT;
27. }
28. else return ptrT[index];
29. }
30. class Fraction
31. {
32. private:
33. int numer, denom;
34. public:
35. Fraction(int z = 0) : numer(z), denom(0) {}
36. Fraction(int n, int d) : numer(n), denom(d) {}
37. friend ostream& operator<<(ostream&, const Fraction&);
38. };
39. ostream& operator<<(ostream& s, const Fraction& f)
40. {
41. return s << f.numer << '/' << f.denom;
42. }
43. int main(void)
44. {
45. int i;
46. Array<int> a1(3);
47. for (i = 0; i < 3; i++) a1[i] = (2 \* i);
48. for (i = 0; i < 3; i++) cout << a1[i] << endl;
49. Array<float> a2(3);
50. for (i = 0; i < 3; i++) a2[i] = (2.7 \* i);
51. for (i = 0; i < 3; i++) cout << a2[i] << endl;
52. Array<char> a3(6);
53. for (i = 0; i < 3; i++) a3[i] = 65+3\*i;
54. for (i = 0; i < 3; i++) cout << a3[i] << endl;
55. Array<Fraction> a4(3);
56. a4[0] = Fraction(3,4);
57. a4[1] = Fraction(1,2);
58. a4[2] = Fraction(5,8);
59. for (i = 0; i < 3; i++) cout << a4[i] << endl;
60. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

0

2

4

0

2.7

5.4

A

D

G

3/4

1/2

5/8

### Example 10 – a container and iterator class template

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. template <typename T, const int size = 7> class Iterator; // Forward declaration
5. template <typename T, const int size = 7>
6. class Container
7. {
8. T array[size];
9. public:
10. friend class Iterator<T, size>;
11. };
12. template <typename T, const int size>
13. class Iterator
14. {
15. Container<T,size>& ref;
16. int index;
17. public:
18. Iterator(Container<T,size>& cr)
19. : ref(cr), index(0)
20. {}
21. void reset()
22. {
23. index = 0;
24. }
25. // prefix increment operator
26. Iterator<T,size>& operator++()
27. {
28. if(index < size - 1)
29. index++;
30. else
31. index = size;
32. return \*this; // indicates end of list
33. }
34. // dereferencing operator
35. T& operator\*()
36. {
37. return ref.array[index];
38. }
39. // conversion operator
40. operator bool() const
41. {
42. return index < size;
43. }
44. };
45. class X
46. {
47. int i;
48. public:
49. X(int I = 0) : i(I) {}
50. X& operator=(const int& I)
51. {
52. i = I;
53. return \*this;
54. }
55. friend ostream& operator<<(ostream& out, const X& object)
56. {
57. out << object.i;
58. return out;
59. }
60. };
61. class Fraction
62. {
63. int numer, denom;
64. public:
65. Fraction(int n = 0, int d = 1) : numer(n),denom(d) {}
66. Fraction& operator=(const Fraction& f)
67. {
68. numer = f.numer;
69. denom = f.denom;
70. return \*this;
71. }
72. friend ostream& operator<<(ostream& out, const Fraction& object)
73. {
74. out << object.numer << '/' << object.denom;
75. return out;
76. }
77. };
78. class Card
79. {
80. private:
81. int pips, suit;
82. static const string SuitName[4];
83. static const string PipsName[13];
84. public:
85. Card(int n = 0) : pips(n%13), suit(n/13) {}
86. Card& operator=(const Card& c)
87. {
88. pips = c.pips;
89. suit = c.suit;
90. return \*this;
91. }
92. friend ostream& operator<<(ostream& out, const Card& object)
93. {
94. out <<PipsName[object.pips] << " of " << SuitName[object.suit];
95. return out;
96. }
97. };
98. const string Card::SuitName[4] =
99. {"clubs","diamonds","hearts","spades"};
100. const string Card::PipsName[13] =
101. "two","three","four","five","six","seven",
102. "eight","nine","ten","jack","queen","king","ace"};
103. int main()
104. {
105. Container<X> xC;
106. Iterator<X> iX(xC);
107. for(auto i = 0; i < 7; i++)
108. {
109. \*iX = i;
110. ++iX;
111. }
112. iX.reset();
113. do cout << \*iX << endl;
114. while(++iX);
115. Container<Fraction,3> fractionContainer;
116. Iterator<Fraction,3> fractionIterator(fractionContainer);
117. for(auto i = 0; i < 3; i++)
118. {
119. \*fractionIterator = Fraction(i+1,i+2);
120. ++fractionIterator;
121. }
122. fractionIterator.reset();
123. do cout << \*fractionIterator << endl;
124. while(++fractionIterator);
125. Container<Card,5> CardC;
126. Iterator<Card,5> itCard(CardC);
127. for(auto i = 0; i < 5; i++)
128. {
129. \*itCard = Card(3\*i+5);
130. ++itCard;
131. }
132. itCard.reset();
133. do cout << \*itCard << endl;
134. while(++itCard);
135. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

0

1

2

3

4

5

6

1/2

2/3

3/4

seven of clubs

ten of clubs

king of clubs

three of diamonds

six of diamonds

### Example 11 – a generic file I/O class

1. #include <fstream>
2. #include <iostream>
3. #include <string>
4. using namespace std;
5. template <class T>
6. class IO
7. {
8. private:
9. fstream file;
10. int eof()
11. {
12. return file.eof();
13. }
14. public:
15. IO(const string& filename = "temp.bin")
16. {
17. file.open(filename,ios\_base::in | ios\_base::out |
18. ios\_base::trunc | ios\_base::binary);
19. }
20. void rewind()
21. {
22. file.seekg(0L);
23. file.seekp(0L);
24. file.clear();
25. }
26. IO& operator>>(T& t);
27. IO& operator<<(const T& t);
28. operator bool()
29. {
30. if (!file) return false;
31. else return true;
32. }
33. };
34. template <class T>
35. IO<T>& IO<T>::operator<<(const T& t)
36. {
37. file.write((char\*) &t,sizeof(T));
38. return \*this;
39. }
40. template <class T>
41. IO<T>& IO<T>::operator>>(T& t)
42. {
43. file.read((char\*)&t,sizeof(T));
44. return \*this;
45. }
46. class A
47. {
48. int a;
49. public:
50. friend istream& operator>>(istream& in, A& AA);
51. friend ostream& operator<<(ostream& out, A& AA);
52. };
53. istream& operator>>(istream& in, A& AA)
54. {
55. cout << "Enter an int for an A object => ";
56. return in >> AA.a;
57. }
58. ostream& operator<<(ostream& out, A& AA)
59. {
60. return out << AA.a;
61. }
62. class B
63. {
64. protected:
65. double bl;
66. char b2[16] ;
67. long b3;
68. public:
69. friend istream& operator>>(istream& in, B& BB);
70. friend ostream& operator<<(ostream& out, B& BB);
71. };
72. istream& operator>>(istream& in, B& BB)
73. {
74. cout << "Enter double, char\* and long for a B object => ";
75. return in >> BB.bl >> BB.b2 >> BB.b3;
76. }
77. ostream& operator<<(ostream& out, B& BB)
78. {
79. return out << BB.bl << ' ' << BB.b2 << ' ' << BB.b3;
80. }
81. int main(void)
82. {
83. A apple;
84. IO<A> appleIO("apple.bin");
85. cin >> apple;
86. appleIO << apple;
87. cin >> apple;
88. appleIO << apple;
89. B banana;
90. IO<B> bananaIO("banana.bin");
91. cin >> banana;
92. bananaIO << banana;
93. cin >> banana;
94. bananaIO << banana;
95. cin >> banana;
96. bananaIO << banana;
97. int temp;
98. IO<int> intIO;
99. intIO << rand() % 100;
100. intIO << rand() % 100;
101. intIO << rand() % 100;
102. intIO << rand() % 100;
103. intIO << rand() % 100;
104. appleIO.rewind();
105. while (appleIO >> apple) cout << apple << endl;
106. bananaIO.rewind();
107. while (bananaIO >> banana) cout << banana << endl;
108. intIO.rewind();
109. while (intIO >> temp) cout << temp << endl;
110. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Enter an int for an A object =>**123**

Enter an int for an A object =>**456**

Enter double, char\* and long for a B object =>**1.1 Hey 98765**

Enter double, char\* and long for a B object =>**2.2 you 87654**

Enter double, char\* and long for a B object =>**3.3 guys 76543**

123

456

1.1 Hey 98765

2.2 you 87654

3.3 guys 76543

41

67

34

0

69

### Example 12 – a generic Linked List

1. #include <iostream>
2. #include <string>
3. #include <cstdlib>
4. using namespace std;
5. template<typename T>
6. class Node
7. {
8. T data\_;
9. Node\* next\_;
10. Node(const Node&) = delete; // disable copy ctor
11. Node& operator=(const Node&) = delete; // disable ass operator
12. public:
13. Node();
14. Node(T d, Node\* n);
15. const T& data() const;
16. T& data();
17. Node\* next() const;
18. Node\*& next();
19. };
20. template<typename T> Node<T>::Node()
21. : data\_(), next\_(0)
22. {}
23. template<typename T> Node<T>::Node(T d, Node\* n)
24. : data\_(d), next\_(n)
25. {}
26. template<typename T> const T& Node<T>::data() const
27. {
28. return data\_;
29. }
30. template<typename T> T& Node<T>::data()
31. {
32. return data\_;
33. }
34. template<typename T> Node<T>\* Node<T>::next() const
35. {
36. return next\_;
37. }
38. template<typename T> Node<T>\*& Node<T>::next()
39. {
40. return next\_;
41. }
42. template<typename T> ostream& operator<<(ostream& out, const Node<T>& N)
43. {
44. out << N.data();
45. return out;
46. }
47. template<typename T> class List
48. {
49. Node<T>\* top\_;
50. List(const List&) = delete; // disable copy ctor
51. List& operator=(const List&) = delete; // disable ass operator
52. public:
53. List();
54. ~List();
55. void push(T object);
56. T pop();
57. const Node<T>\* top() const;
58. bool remove(T object);
59. const Node<T>\* find(T object) const;
60. };
61. template<typename T>
62. ostream& operator<<(ostream& out, const List<T>& L)
63. {
64. const Node<T>\* ptr = L.top();
65. while (ptr)
66. {
67. out << (\*ptr) << '\t';
68. ptr = ptr -> next();
69. }
70. return out;
71. }
72. template<typename T> List<T>::List()
73. : top\_(0)
74. {}
75. template<typename T> List<T>::~List()
76. {
77. Node<T>\* ptr = top\_;
78. while (ptr)
79. {
80. top\_ = top\_->next();
81. delete ptr;
82. ptr = top\_;
83. }
84. }
85. template<typename T> void List<T>::push(T object)
86. {
87. Node<T>\* ptr = new Node<T>(object, top\_);
88. top\_ = ptr;
89. }
90. template<typename T> const Node<T>\* List<T>::top() const
91. {
92. return top\_;
93. }
94. template<typename T> T List<T>::pop()
95. {
96. Node<T>\* ptr = top\_;
97. top\_ = top\_ -> next();
98. T data = ptr->data();
99. delete ptr;
100. return data;
101. }
102. template<typename T> const Node<T>\* List<T>::find(T object) const
103. {
104. const Node<T>\* ptr = top();
105. while (ptr)
106. {
107. if (ptr->data() == object)
108. {
109. return ptr;
110. }
111. ptr = ptr->next();
112. }
113. return 0;
114. }
115. template<typename T> bool List<T>::remove(T object)
116. {
117. if (!find(object))
118. {
119. cerr << object << " not found\n";
120. return false;
121. }
122. Node<T>\* ptr2current = top\_;
123. Node<T>\* ptr2previous = top\_;
124. if (top\_->data() == object)
125. {
126. top\_ = top\_ -> next();
127. delete ptr2current;
128. return true;
129. }
130. while (ptr2current)
131. {
132. ptr2current = ptr2current->next();
133. if (ptr2current->data() == object)
134. {
135. ptr2previous->next() = ptr2current->next();
136. delete ptr2current;
137. return true;
138. }
139. ptr2previous = ptr2current;
140. }
141. return false;
142. }
143. class Card
144. {
145. private:
146. int pips, suit;
147. static const string SuitName[4];
148. static const string PipsName[13];
149. public:
150. Card() : pips(rand()%13), suit(rand()%4) {}
151. Card(int n) : pips(n%13), suit(n%4) {}
152. friend ostream& operator<<(ostream& out, const Card& object)
153. {
154. out << PipsName[object.pips] << " of "
155. << SuitName[object.suit];
156. return out;
157. }
158. };
159. const string Card::SuitName[4] =
160. {"clubs","diamonds","hearts","spades"};
161. const string Card::PipsName[13] =
162. {"two","three","four","five","six","seven",
163. "eight","nine","ten","jack","queen","king","ace"};
164. int main()
165. {
166. List<int> Lint;
167. Lint.push(2);
168. Lint.push(4);
169. Lint.push(6);
170. Lint.push(8);
171. Lint.push(10);
172. cout << Lint << endl;
173. Lint.pop();
174. cout << Lint << endl;
175. Card C1;
176. Card C2;
177. Card C3(25);
178. Card C4;
179. Card C5;
180. List<Card> LCard;
181. LCard.push(C1);
182. LCard.push(C2);
183. LCard.push(C3);
184. LCard.push(C4);
185. LCard.push(C5);
186. cout << LCard << endl;
187. List<string> Lstring;
188. Lstring.push("day");
189. Lstring.push("nice");
190. Lstring.push("very");
191. Lstring.push("a");
192. Lstring.push("Have");
193. cout << Lstring << endl;
194. Lstring.remove("very");
195. cout << Lstring << endl;
196. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

10 8 6 4 2

8 6 4 2

ace of hearts nine of clubs ace of diamonds five of clubs four of spades

Have a very nice day

Have a nice day

# Hash Tables

A hash table is an abstract data type that uses an array for storage. It makes use of a mapped key as an index. A hash table uses a hash function to translate a value into an index that can you used with an array. The location in the array where the data is stored is referred to as a bucket or slot.

### Example 1 – First hash table example

This example demonstrates an array of strings stored in a *hash table*. The *hash table*, itself, is an array of string pointers. The *hash function*, hash, converts each string into an unsigned int value. The unsigned int return value is then used as an index in the array of string pointers. Notice, that some of the string arguments with produce the same return value. This situation is referred to as a *collision*. In this example when a *collision* occurs, the target string is not able to be stored in the *hash table*.

1. #include <iostream>
2. #include <string>
3. #include <cctype>
4. using namespace std;
5. unsigned hash(const string&);
6. const unsigned NumberOfBuckets = 10;
7. int main()
8. {
9. string animals[NumberOfBuckets] =
10. {"monkey","dog","cat","horse","pig","goat","hippo",
11. "dinosaur","walrus","manatee"};
12. string\* ptr2strings[NumberOfBuckets] = {nullptr};
13. for (auto i = 0u; i < NumberOfBuckets; i++)
14. {
15. auto index = ::hash(animals[i]);
16. // if the index is unused, use it
17. if (ptr2strings[index] == nullptr)
18. {
19. ptr2strings[index] = new string(animals[i]);
20. }
21. else
22. {
23. cout << "Can't store " << animals[i] << ". Bucket "
24. << index << " is already taken\n";
25. }
26. }
27. for (auto i = 0u; i < NumberOfBuckets; i++)
28. {
29. cout << i << ' '
30. << (ptr2strings[i] ? \*ptr2strings[i] : "" )<< endl;
31. }
32. }
33. unsigned hash(const string& str)
34. {
35. static string alphabet = "abcdefghijklmnopqrstuvwxyz";
36. size\_t pos;
37. unsigned sum = 0;
38. for (auto i = 0u; i < str.size(); i++)
39. {
40. pos = alphabet.find(tolower(str[i]));
41. sum += pos;
42. }
43. return sum % NumberOfBuckets;
44. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Can't store goat. Bucket 9 is already taken

Can't store hippo. Bucket 9 is already taken

Can't store dinosaur. Bucket 3 is already taken

0 horse

1 cat

2 manatee

3 dog

4

5

6

7 monkey

8 walrus

1. pig

### Example 2 – Use a hash table to store a dictionary

This example simulates an “Unscramble” game in which scrambled words are unscrambled by using a hash table to find the word with the same hashed value. Note, in this solution, *collisions* are also not handled.

1. #include <iostream>
2. #include <string>
3. #include <cctype>
4. #include <fstream>
5. #include <cstdlib>
6. #include <stdexcept>
7. using namespace std;
8. unsigned hash(const string&);
9. class Dictionary
10. {
11. string\*\* ptrWords;
12. public:
13. Dictionary(const string& wordfile);
14. ~Dictionary();
15. string findScrambledWord(const string& word);
16. static const unsigned NumberOfBuckets;
17. };
18. const unsigned Dictionary::NumberOfBuckets = 100000;
19. Dictionary::Dictionary(const string& wordfile)
20. : ptrWords(new string\*[NumberOfBuckets])
21. {
22. ifstream fin(wordfile.c\_str());
23. if (!fin)
24. {
25. throw (invalid\_argument(string("Can't find file ") +
26. wordfile));
27. }
28. string word;
29. unsigned numberOfBucketsUsed = 0;
30. unsigned numberOfWordsNotStored = 0;
31. unsigned numberOfWords = 0;
32. for (auto i = 0u; i < NumberOfBuckets; i++)
33. {
34. ptrWords[i] = nullptr;
35. }
36. // create hash table
37. while (fin >> word)
38. {
39. ++numberOfWords;
40. auto index = ::hash(word);
41. if (ptrWords[index])
42. {
43. // bucket already taken
44. ++numberOfWordsNotStored;
45. }
46. else
47. {
48. ptrWords[index] = new string(word);
49. numberOfBucketsUsed++;
50. }
51. }
52. cout << "number of buckets used = " << numberOfBucketsUsed
53. << endl;
54. cout << "number of words not stored = "
55. << numberOfWordsNotStored << endl;
56. cout << "number of words = " << numberOfWords << endl;
57. }
58. Dictionary::~Dictionary()
59. {
60. for (auto i = 0u; i < NumberOfBuckets; i++)
61. {
62. if (ptrWords[i])
63. {
64. delete ptrWords[i];
65. }
66. }
67. delete [] ptrWords;
68. ptrWords = nullptr;
69. }
70. string Dictionary::findScrambledWord(const string& word)
71. {
72. auto index = ::hash(word);
73. if (ptrWords[index])
74. return \*(ptrWords[index]);
75. else
76. return string("");
77. }
78. int main()
79. {
80. string scrambledWord;
81. try
82. {
83. Dictionary Words("c:/temp/words");
84. while (1)
85. {
86. cout << "Enter a scrambled word (\"quit\" to exit)=> ";
87. cin >> scrambledWord;
88. if (scrambledWord == "quit")
89. return 0;
90. else
91. cout << "unscramble = "
92. << Words.findScrambledWord(scrambledWord) << endl;
93. }
94. }
95. catch (const invalid\_argument& error)
96. {
97. cout << error.what() << endl;
98. exit(-1);
99. }
100. }
101. unsigned hash(const string& str)
102. {
103. static unsigned primes[26] = {2, 3, 5, 7, 11, 13, 17, 19, 23,
104. 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
105. 73, 79, 83, 89, 97, 101};
106. unsigned product = 1;
107. for (auto i = 0u; i < str.size(); i++)
108. {
109. product \*= primes[tolower(str[i])-'a'];
110. }
111. return product % Dictionary::NumberOfBuckets;
112. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

number of buckets used = 19735

number of words not stored = 4320

number of words = 24055

Enter a scrambled word ("quit" to exit) => **ksa**

unscramble = ask

Enter a scrambled word ("quit" to exit) => **bilrray**

unscramble = library

Enter a scrambled word ("quit" to exit) => **hsear**

unscramble = Asher

Enter a scrambled word ("quit" to exit) => **fntcunoi**

unscramble = function

Enter a scrambled word ("quit" to exit) => **asked**

unscramble =

Enter a scrambled word ("quit" to exit) => **yranoitcid**

unscramble = combatted

Enter a scrambled word ("quit" to exit) => **belramcs**

unscramble = scramble

Enter a scrambled word ("quit" to exit) => **quit**

Notes

***hsear*** *was supposed to be share*

***yranoitcid*** *was supposed to be dictionary*

***belramcs*** *was supposed to be scramble (but was not found)*

# Standard Template Library

The STL consists of

* containers (in the form of class templates),
* iterators - to be used "like" pointers in a container
* function objects (or functors) - A class object that can act like a function.
* algorithms - functions applied to containers.

## Containers

### Types of containers

#### Sequential

A sequential container is one in which elements are accessed sequentially. That access is usually performed using an iterator.

#### Sorted Associative

An associative container is one in which elements are accessed using a key.

#### Adaptors

Adaptors are adaptations of specific sequential containers for specific purposes.

#### Unsorted Associative

Unsorted associative containers are implemented using hashing algorithms.

|  |  |  |
| --- | --- | --- |
| **Container** | **Type** | **Purpose** |
| array | sequential | A C-style fixed size replacement |
| vector | sequential | All-purpose, variable size |
| list | sequential | Linked-list, double ended |
| forward\_list | sequential | Linked-list, single ended |
| deque | sequential | Like a vectors with access at ends |
| queue | Adapter | Implements FIFO |
| priority\_queue | Adapter | Implements FIFO with priority |
| stack | Adapter | Implements LIFO |
| set | Sorted associative | Similar to mathematical set |
| multi\_set | Sorted associative | A set with duplicate values |
| map | Sorted associative | Key-value pairs |
| multimap | Sorted associative | Key-value pairs with duplicate keys |
| unordered\_set | Unsorted associative | set implemented as hash table |
| unordered\_multiset | Unsorted associative | Multiset implemented as hash table |
| unordered\_map | Unsorted associative | map implemented as hash table |
| unordered\_multimap | Unsorted associative | multimap implemented as hash table |
| bitset | N/A | Bit manipulators replacement |

## array

The array container is a replacement for the fixed size C array. This sequence container surfaced in C++ 11. The array container exhibits the indexing behaviors of a C array. To declare an array class object, class template syntax is used and only the default constructor is available. The array container requires the <array> header file.

Examples

array<int,10> object; // instantiates an array of 10 int

array<dog,5> hounds; // instantiates an array of 10 dogs

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the array

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the array

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last element in the array

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the *non-existing* element in front of the first element of the array

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a const iterator pointing to the first element of the array

const\_iterator begin() const noexcept;

#### cend

Returns a const iterator pointing to the *non-existing* element beyond the end of the array

const\_iterator end() const noexcept;

#### crbegin

Returns a const reverse iterator pointing to the last element of the array

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a const reverse iterator pointing to the non-existing element in front of the first element of the array

const\_reverse\_iterator rend() const noexcept;

### Capacity Functions

#### size

Returns the number of elements in the array

constexpr size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements in an array. This is the same as the size.

constexpr size\_t max\_size() const noexcept;

#### empty

Returns whether the array is empty – has size 0.

constexpr bool empty() const noexcept;

### Access Functions

#### at

Returns element at position

value\_type& at (size\_t position);

const value\_type& at (size\_t position) const;

#### back

Returns a reference to the last element in the array

value\_type& back();

const value\_type& back() const;

#### front

Returns a reference to the first element in the array

value\_type& front();

const value\_type& front() const;

#### data

Returns a pointer to the memory location where a array’s first element is stored. Note, array elements are stored in contiguous memory.

value\_type\* data() noexcept;

const value\_type\* data() const noexcept;

### Modifier Functions

#### fill

assigns a value to all elements of an array

void fill(const value\_type& value);

**swap**

Swaps the contents of two arrays. The arrays must be of the same type and contain the same number of elements.

void swap (array& vec);

### operator[]

Index operator: returns the element at the specified location

value\_type& operator[] (size\_t location);

const value\_type& operator[] (size\_t location) const;

### Example 1 – The array container

1. #include <array>
2. #include <iostream>
3. #include <cstring> // for memcpy
4. using namespace std;
5. void print\_array(const array<int,5>&);
6. void print\_array(const array<char,3>&);
7. // function template prototype
8. template <typename T, unsigned long size>
9. ostream& operator<<(ostream&, const array<T,size>&);
10. int main()
11. {
12. array<int,5> a1 = {2,3,5,7,11};
13. cout << "a1="; print\_array(a1);
15. array<char,3> a2 = {'h','e','y'};
16. cout << "a2="; print\_array(a2);
18. memcpy(a2.data(),"Wow",a2.size());
19. cout << "a2="; print\_array(a2);
21. array<char,3> a3;
22. a3.fill('$');
23. a3.swap(a2);
24. cout << "a2="; print\_array(a2);
26. cout << "a1=" << a1 << endl;
27. }
28. void print\_array(const array<int,5>& arr)
29. {
30. // iterator for loop
31. for (auto arrIt = arr.cbegin(); arrIt != arr.cend(); ++arrIt)
32. cout << \*arrIt << ' ';
33. cout << endl;
34. }
35. void print\_array(const array<char,3>& arr)
36. {
37. // index for loop
38. for (auto i = 0u; i < arr.size(); ++i)
39. cout << arr[i];
40. cout << endl;
41. }
42. template <typename T, unsigned long size>
43. ostream& operator<<(ostream& out, const array<T, size>& object)
44. {
45. // range-based for loop
46. for (const auto& element : object)
47. out << element << ' ';
48. return out;
49. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

a1=2 3 5 7 11

a2=hey

a2=Wow

a2=$$$

a1=2 3 5 7 11

## vector

The vector container is a replacement for an array. Unlike an array it has a variable size and can grow and shrink as needed. Further, you may insert new elements into the vector at the beginning or end of the vector . and even in the middle. Vectors may be indexed just like an array. Instead of using pointers to access array elements, iterators are used. The vector container requires the <vector> header file.

### Constructors

Default constructor

vector();

Fill constructors

explicit vector(size\_type n, const allocator\_type& alloc =

allocator\_type());

vector(size\_type n, const value\_type& val,

const allocator\_type& alloc = allocator\_type());

Range constructor

template <class InputIterator>

vector(InputIterator first, InputIterator last,

const allocator\_type& alloc = allocator\_type());

Copy constructor

vector(const vector& x);

Move constructor

vector(vector&& x);

Initializer list constructor

vector(initializer\_list<value\_type> lst,

const allocator\_type& alloc = allocator\_type());

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the vector

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the vector

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last element in the vector

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the *non-existing* element in front of the first element of the vector

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a const iterator pointing to the first element of the vector

const\_iterator begin() const noexcept;

#### cend

Returns a const iterator pointing to the *non-existing* element beyond the end of the vector

const\_iterator end() const noexcept;

#### crbegin

Returns a const reverse iterator pointing to the last element of the vector

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a const reverse iterator pointing to the non-existing element in front of the first element of the vector

const\_reverse\_iterator rend() const noexcept;

### Capacity Functions

#### size

Returns the number of elements in the vector

size\_t size() const noexcept;

#### capacity

Returns the size allocated for the vector

size\_t capacity() const noexcept;

#### max\_size

Returns the maximum number of elements that a vector can hold

size\_t max\_size() const noexcept;

#### reserve

Change the vector’s capacity

void reserve(size\_t n);

#### resize

Resizes a vector to n elements

void resize (size\_t n);

void resize (size\_t n, const value\_type& value);

#### empty

Returns whether the vector is empty

bool empty() const noexcept;

#### shrink\_to\_fit

Changes the capacity to the size of the vector

void shrink\_to\_fit();

### Access Functions

#### at

Returns element at position

value\_type& at (size\_t position);

const value\_type& at (size\_t position) const;

#### back

Returns a reference to the last element in the vector

value\_type& back();

const value\_type& back() const;

#### front

Returns a reference to the first element in the vector

value\_type& front();

const value\_type& front() const;

#### data

Returns a pointer to the memory location where a vector’s first element is stored. Note, vector elements are stored in contiguous memory.

value\_type\* data() noexcept;

const value\_type\* data() const noexcept;

### Modifier Functions

#### assign

Assigns new contents to a vector

template <class InputIterator>

void assign(InputIterator beg, InputIterator \_end);

void assign(size\_type n, const value\_type& value);

void assign(initializer\_list<value\_type> list);

#### clear

Erases a vector. Size becomes 0

void clear() noexcept;

#### erase

Erases part of a vector

iterator erase(const\_iterator p);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into a vector at a specified location

iterator insert(const\_iterator loc, const value\_type& value);

iterator insert(const\_iterator loc, size\_type n, const value\_type& value);

template <class InputIterator>

iterator insert(const\_iterator loc, InputIterator first, InputIterator last);

iterator insert(const\_iterator loc, value\_type&& value);

iterator insert(const\_iterator loc, initializer\_list<value\_type> list);

#### push\_back

Adds an element to the end of a vector

void push\_back(const value\_type& value);

void push\_back(value\_type&& value);

#### pop\_back

Deletes the last element of a vector

void pop\_back();

**swap**

Swaps two vectors

void swap(vector& vec);

### Non-member Functions

#### swap

Swaps two vector

void swap(vector& x, vector& y);

### Member Operators

#### operator=

The assignment operator: assigns new contents to a vector.

vector& operator=(const vector& x);

vector& operator=(vector&& x);

vector& operator=(initializer\_list<value\_type> list);

#### operator[]

Index operator: returns the element at the specified location

value\_type& operator[](size\_t location);

const value\_type& operator[](size\_t location) const;

#### Relational operators

== > < >= <= !=

Used to compare the contents of two vectors.

Two vectors are equal (==) if their sizes match and each of the corresponding elements match.

A less than (<) comparison is made between two vectors by comparing successive elements in order.

Note: these operators, > < >= <= != will be removed in C++20. The <=> operator will be added. More to say about that later.

### Example 2 – The vector container

1. #include <vector>
2. #include <iostream>
3. using namespace std;
4. ostream& operator<<(ostream& out, const vector<int>& v);
5. int main()
6. {
7. // Constructors
8. vector<int> v1;
9. vector<int> v2(5);
10. vector<int> v3(5,19);
11. vector<int> v4{2,3,5,7,11,13,17};
13. cout << "v2=" << v2 << endl;
14. cout << "v3=" << v3 << endl;
15. cout << "v4=" << v4 << endl << endl;
17. vector<int> v5(v4.begin(),v4.begin()+3);
18. vector<int> v6(v4);
19. vector<int> v7(move(v4));
21. cout << "v4=" << v4 << endl;
22. cout << "v5=" << v5 << endl;
23. cout << "v6=" << v6 << endl;
24. cout << "v7=" << v7 << endl << endl;
26. // Capacity functions
27. cout << "v7.size()=" << v7.size() << endl;
28. cout << "v7.capacity()=" << v7.capacity() << endl;
29. cout << "v7.max\_size()=" << v7.max\_size() << endl;
30. v7.reserve(16);
31. v7.resize(v7.size()\*2);
32. cout << "v7.size()=" << v7.size() << endl;
33. cout << "v7.capacity()=" << v7.capacity() << endl;
34. cout << "v7=" << v7 << endl;
35. v7.shrink\_to\_fit();
36. cout << "v7.size()=" << v7.size() << endl;
37. cout << "v7.capacity()=" << v7.capacity() << endl << endl;
39. // Access functions
40. cout << "v6.front()=" << v6.front() << endl;
41. cout << "v6.back()=" << v6.back() << endl;
42. cout << "v6.at(3)=" << v6.at(3) << endl;
43. int\* ptr = v6.data();
44. cout << \*ptr << ' ' << \*(ptr+2) << endl;
45. for (auto\* p = v6.data(); p < v6.data()+v6.size(); ++p)
46. \*p \*= 2;
47. cout << "v6=" << v6 << endl << endl;
49. // Modifier functions
50. v1.assign({7,6,5,4,3,2,1});
51. cout << "v1=" << v1 << endl;
52. v2.assign(v1.crbegin(),v1.crend());
53. cout << "v2=" << v2 << endl;
54. v2.erase(v2.begin()+3);
55. cout << "v2=" << v2 << endl;
56. v2.insert(v2.begin()+3,15);
57. v2.pop\_back();
58. v2.push\_back(30);
59. cout << "v2=" << v2 << endl;
60. v1.swap(v2);
61. cout << "v1=" << v1 << endl;
62. cout << "v2=" << v2 << endl << endl;
64. // Member operators
65. v1[2] = v2[3]\*2;
66. cout << "v1=" << v1 << endl;
67. v1.assign(v2.begin(),v2.begin()+5);
68. v1.push\_back(13);
69. cout << "v1=" << v1 << endl;
70. cout << "v2=" << v2 << endl << endl;
71. v3 = v1;
72. v3.resize(10);
73. cout << "v3=" << v3 << endl;
74. cout << boolalpha;
75. cout << "v1 == v3: " << (v1 == v3) << endl;
76. cout << "v1 < v2: " << (v1 < v2) << endl;
77. cout << "v1 < v3: " << (v1 < v3) << endl;
78. cout << "v2 < v3: " << (v2 < v3) << endl;
79. }
80. ostream& operator<<(ostream& out, const vector<int>& v)
81. {
82. for (auto element : v)
83. out << element << ' ';
84. return out;
85. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

v2=0 0 0 0 0

v3=19 19 19 19 19

v4=2 3 5 7 11 13 17

v4=

v5=2 3 5

v6=2 3 5 7 11 13 17

v7=2 3 5 7 11 13 17

v7.size()=7

v7.capacity()=7

v7.max\_size()=2305843009213693951

v7.size()=14

v7.capacity()=16

v7=2 3 5 7 11 13 17 0 0 0 0 0 0 0

v7.size()=14

v7.capacity()=14

v6.front()=2

v6.back()=17

v6.at(3)=7

2 5

v6=4 6 10 14 22 26 34

v1=7 6 5 4 3 2 1

v2=1 2 3 4 5 6 7

v2=1 2 3 5 6 7

v2=1 2 3 15 5 6 30

v1=1 2 3 15 5 6 30

v2=7 6 5 4 3 2 1

v1=1 2 8 15 5 6 30

v1=7 6 5 4 3 13

v2=7 6 5 4 3 2 1

v3=7 6 5 4 3 13 0 0 0 0

v1 == v3: false

v1 < v2: false

v1 < v3: true

v2 < v3: true

## list

The list container is implemented as a double-ended linked list. It has the advantage of efficient insert and delete operations. The list container requires the <list> header file.

### Constructors

Default constructor

list();

Fill constructors

explicit list(size\_type n, const allocator\_type& alloc =

allocator\_type());

list(size\_type n, const value\_type& val,

const allocator\_type& alloc = allocator\_type());

Range constructor

template <class InputIterator>

list(InputIterator first, InputIterator last,

const allocator\_type& alloc = allocator\_type());

Copy constructor

list(const list& x);

Move constructor

list(list&& x);

Initializer list constructor

list(initializer\_list<value\_type> lst,

const allocator\_type& alloc = allocator\_type());

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the list

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the list

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last element in the list

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the *non-existing* element in front of the first element of the list

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a const iterator pointing to the first element of the list

const\_iterator begin() const noexcept;

#### cend

Returns a const iterator pointing to the *non-existing* element beyond the end of the list

const\_iterator end() const noexcept;

#### crbegin

Returns a const reverse iterator pointing to the last element of the list

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a const reverse iterator pointing to the non-existing element in front of the first element of the list

const\_reverse\_iterator rend() const noexcept;

### Capacity Functions

#### size

Returns the number of elements in the list

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a list can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the list is empty

bool empty() const noexcept;

### Access Functions

#### back

Returns a reference to the last element in the list

value\_type& back();

const value\_type& back() const;

#### front

Returns a reference to the first element in the list

value\_type& front();

const value\_type& front() const;

### Modifier Functions

#### assign

Assigns new contents to a list

template <class InputIterator>

void assign(InputIterator beg, InputIterator \_end);

void assign(size\_type n, const value\_type& value);

void assign(initializer\_list<value\_type> lst);

#### clear

Erases a list. Size becomes 0

void clear() noexcept;

#### erase

Erases part of a list

iterator erase(const\_iterator p);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into a list at a specified location

iterator insert(const\_iterator loc, const value\_type& value);

iterator insert(const\_iterator loc, size\_type n, const value\_type& value);

template <class InputIterator>

iterator insert(const\_iterator loc, InputIterator first, InputIterator last);

iterator insert(const\_iterator loc, value\_type&& value);

iterator insert(const\_iterator loc, initializer\_list<value\_type> lst);

#### emplace

Constructs and inserts a new element at a specified location in the list

template <class Type> void emplace(const iterator loc, Type&&... args);

#### push\_back

Adds an element to the end of a list

void push\_back(const value\_type& value);

void push\_back(value\_type&& value);

#### push\_front

Adds an element to the beginning of a list

void push\_front(const value\_type& value);

void push\_front(value\_type&& value);

#### pop\_back

Deletes the last element of a list

void pop\_back();

#### pop\_front

Deletes the first element of a list

void pop\_front();

**swap**

Swaps two lists

void swap(list& lst);

**resize**

Changes the size of a list. If the size is smaller, elements are removed. If the size is larger, elements are added to the list.

void resize(size\_type n);

void resize(size\_type n, const value& val);

### Example 3 – The list container

1. #include <list>
2. #include <iostream>
3. using namespace std;
4. ostream& operator<<(ostream& out, const list<int>& li);
5. int main()
6. {
7. // Constructors
8. list<int> li1;
9. list<int> li2(5);
10. list<int> li3(5,19);
11. list<int> li4{2,3,5,7,11,13,17};
13. cout << "li2=" << li2 << endl;
14. cout << "li3=" << li3 << endl;
15. cout << "li4=" << li4 << endl << endl;
17. // list<int> li5(li4.begin(),li4.begin()+3); ERROR
18. list<int> li5(li4.begin(),++++++li4.begin()); // ???
19. list<int> li6(li4);
20. list<int> li7(move(li4));
22. cout << "li4=" << li4 << endl;
23. cout << "li5=" << li5 << endl;
24. cout << "li6=" << li6 << endl;
25. cout << "li7=" << li7 << endl << endl;
27. cout << "capacity functions" << endl;
28. cout << li1.size() << ' ' << boolalpha << li1.empty() << endl;
30. cout << endl << "access functions" << endl;
31. cout << "li6.front()=" << li6.front() << endl;
32. cout << "li6.back()=" << li6.back() << endl;
34. cout << endl << "iterator functions" << endl;
35. cout << "\*li6.begin()=" << \*li6.begin() << endl;
36. cout << "\*++li6.begin()=" << \*++li6.begin() << endl;
37. cout << "\*--li6.end()=" << \*--li6.end() << endl;
38. cout << "\*li6.rbegin()=" << \*li6.rbegin() << endl;
39. cout << "\*++li6.rbegin()=" << \*++li6.rbegin() << endl;
40. cout << "\*--li6.rend()=" << \*--li6.rend() << endl;
42. cout << endl << "assign" << endl;
43. li1.assign({7,6,5,4,3,2,1});
44. cout << "li1=" << li1 << endl;
45. li2.assign(++li1.crbegin(),--li1.crend());
46. cout << "li2=" << li2 << endl;
47. li3.assign(5,7);
48. cout << "li3=" << li3 << endl << endl;
50. cout << "erase" << endl;
51. li2.erase(++li2.begin());
52. cout << "li2=" << li2 << endl;
53. li1.erase(++li1.begin(),--li1.end());
54. cout << "li1=" << li1 << endl << endl;
56. cout << "insert" << endl;
57. li2.insert(++li2.begin(),3);
58. cout << "li2=" << li2 << endl;
59. li2.insert(++li2.begin(),li3.begin(),li3.end());
60. cout << "li2=" << li2 << endl << endl;
62. cout << "push\_front / pop\_back" << endl;
63. li1.push\_front(1);
64. li1.pop\_back();
65. cout << "li1=" << li1 << endl << endl;
67. cout << "swap" << endl;
68. li1.swap(li2);
69. cout << "li1=" << li1 << endl << endl;
71. cout << "resize" << endl;
72. li1.resize(5);
73. cout << "li1=" << li1 << endl;
74. li1.resize(10);
75. cout << "li1=" << li1 << endl;
76. }
77. ostream& operator<<(ostream& out, const list<int>& li)
78. {
79. for (auto element : li)
80. out << element << ' ';
81. return out;
82. }

\*\*\*\*\* OUTPUT \*\*\*\*\*\*

li2=0 0 0 0 0

li3=19 19 19 19 19

li4=2 3 5 7 11 13 17

li4=

li5=2 3 5

li6=2 3 5 7 11 13 17

li7=2 3 5 7 11 13 17

capacity functions

0 true

access functions

li6.front()=2

li6.back()=17

iterator functions

\*li6.begin()=2

\*++li6.begin()=3

\*--li6.end()=17

\*li6.rbegin()=17

\*++li6.rbegin()=13

\*--li6.rend()=2

assign

li1=7 6 5 4 3 2 1

li2=2 3 4 5 6

li3=7 7 7 7 7

erase

li2=2 4 5 6

li1=7 1

insert

li2=2 3 4 5 6

li2=2 7 7 7 7 7 3 4 5 6

push\_front / pop\_back

li1=1 7

swap

li1=2 7 7 7 7 7 3 4 5 6

resize

li1=2 7 7 7 7

li1=2 7 7 7 7 0 0 0 0 0

## forward\_list

The forward\_list container is implemented as a single-ended linked list. Because it only uses a forward pointer, it is usually considered more efficient that a list container. The forward\_list container requires the <forward\_list> header file. The forward\_list container was introduced in C++11.

### Constructors

Default constructor

forward\_list();

Fill constructors

explicit forward\_list (size\_type n, const allocator\_type& alloc =

allocator\_type());

forward\_list (size\_type n, const value\_type& val,

const allocator\_type& alloc = allocator\_type());

Range constructor

template <class InputIterator>

forward\_list (InputIterator first, InputIterator last,

const allocator\_type& alloc = allocator\_type());

Copy constructor

forward\_list (const vector& x);

Move constructor

forward\_list (vector&& x);

Initializer list constructor

forward\_list (initializer\_list<value\_type> lst,

const allocator\_type& alloc = allocator\_type());

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the forward\_list

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### before\_begin

Returns an iterator pointing to the location before first element of the forward\_list

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the forward\_list

iterator end() noexcept;

const\_iterator end() const noexcept;

#### cbegin

Returns a const iterator pointing to the first element of the forward\_list

const\_iterator begin() const noexcept;

#### cbefore\_begin

Returns a const iterator pointing to the location before first element of the forward\_list

const\_iterator begin() const noexcept;

#### cend

Returns a const iterator pointing to the *non-existing* element beyond the end of the forward\_list

const\_iterator end() const noexcept;

### Capacity Functions

#### max\_size

Returns the maximum number of elements that a forward\_list can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the forward\_list is empty

bool empty() const noexcept;

### front

Returns a reference to the first element in the forward\_list

value\_type& front();

const value\_type& front() const;

### Modifier Functions

#### assign

Assigns new contents to a forward\_list

template <class InputIterator>

void assign(InputIterator beg, InputIterator \_end);

void assign(size\_type n, const value\_type& value);

void assign(initializer\_list<value\_type> lst);

#### clear

Erases a forward\_list. Size becomes 0

void clear() noexcept;

#### erase\_after

Erases part of a list

iterator erase\_after(const\_iterator p);

iterator erase\_after(const\_iterator first, const\_iterator last);

#### insert\_after

Inserts elements into a forward\_list at a specified location

iterator insert\_after(const\_iterator loc, const value\_type& value);

iterator insert\_after(const\_iterator loc, size\_type n, const value\_type& va);

template <class InputIterator>

iterator insert\_after(const\_iterator loc, InputIterator f, InputIterator ls);

iterator insert\_after(const\_iterator loc, value\_type&& value);

iterator insert\_after(const\_iterator loc, initializer\_list<value\_type> lst);

#### push\_front

Adds an element to the beginning of a forward\_list

void push\_front(const value\_type& value);

void push\_front(value\_type&& value);

#### pop\_front

Deletes the first element of a forward\_list

void pop\_front();

#### emplace\_front

Constructs and inserts a new element in the beginning of the forward list

template <class Type> void emplace\_front(Type&&... args);

#### emplace\_after

Constructors and inserts a new element in a location in the forward list

template <class Type> void emplace\_after(const iterator loc, Type&&... args);

#### swap

Swaps two forward\_lists

void swap(forward\_list& lst);

#### resize

Changes the size of a forward\_list. If the size is smaller, elements are removed. If the size is larger, elements are added to the list.

void resize(size\_type n);

void resize(size\_type n, const value& val);

### Operation Functions

#### merge

Merge two forward\_lists. The merge function assumes both forward\_lists are sorted.

void merge(forward\_list& fwdlst);

void merge(forward\_list&& fwdlst);

template <class Compare> void merge(forward\_list& fwdlst, Compare comp);

template <class Compare> void merge(forward\_list&& fwdlst, Compare comp);

#### remove

Removes all elements with a specified value from the forward\_list

void remove(const value\_type& value);

#### remove\_if

Removes elements that meet a specified condition

template <class Predicate> void remove\_if(Predicate pred);

#### reverse

Reverses the order of elements in a forward\_list

void reverse() noexcept;

#### sort

Sorts elements in a forward\_list

void sort();

template <class Compare> void sort(Compare comp);

#### splice\_after

Inserts part of another forward\_list into a forward\_list

void splice\_after(const\_iterator position, forward\_list& fwdlst);

void splice\_after(const\_iterator position, forward\_list&& fwdlst);

void splice\_after(const\_iterator position, forward\_list& fwdlst,

const\_iterator i);

void splice\_after(const\_iterator position, forward\_list&& fwdlst,

const\_iterator i);

void splice\_after(const\_iterator position, forward\_list& fwdlst,

const\_iterator first, const\_iterator last);

void splice\_after(const\_iterator position, forward\_list&& fwdlst,

const\_iterator first, const\_iterator last);

#### unique

Removes duplicate values from a forward\_list

void unique();

template <class BinaryPredicate> void unique(BinaryPredicate binary\_pred);

### Example 4 – The forward\_list container

1. #include <forward\_list>
2. #include <iostream>
3. using namespace std;
4. ostream& operator<<(ostream& out, const forward\_list<int>& obj);
5. int main()
6. {
7. // Constructors
8. forward\_list<int> f1;
9. forward\_list<int> f2(5);
10. forward\_list<int> f3(5,19);
11. forward\_list<int> f4{2,3,5,7,11,13,17};
12. cout << "f2 = "<< f2 << endl;
13. cout << "f3 = "<< f3 << endl;
14. cout << "f4 = "<< f4 << endl;
15. cout << endl;
16. forward\_list<int> f5(f4);
17. forward\_list<int> f6(move(f4));
18. cout << "f4 = "<< f4 << endl;
19. cout << "f5 = "<< f5 << endl;
20. cout << "f6 = "<< f6 << endl;
21. cout << endl;
22. // Capacity functions
23. cout << "f1.max\_size() = " << f1.max\_size() << ' '
24. << boolalpha << " f1.empty() = " << f1.empty() << endl << endl;
25. // Access and Iterator functions
26. cout << "f5.front() = " << f5.front() << endl;
27. cout << "\*f5.begin() = " << \*f5.begin() << endl;
28. cout << "\*++f5.before\_begin() = " << \*++f5.before\_begin() << endl << endl;
29. // Modifier functions
30. cout << "assign" << endl;
31. f1.assign(5,7);
32. cout << "f1 = " << f1 << endl;
33. f1.assign({7,6,5,4,3,2,1});
34. cout << "f1 = " << f1 << endl;
35. cout << endl;
36. cout << "erase\_after" << endl;
37. f1.erase\_after(f1.begin());
38. cout << "f1 = " << f1 << endl << endl;
39. cout << "insert\_after" << endl;
40. f1.insert\_after(f1.before\_begin(),3);
41. cout << "f1 = " << f1 << endl;
42. f1.insert\_after(f1.begin(),f3.begin(),f3.end());
43. cout << "f1 = " << f1 << endl << endl;
44. cout << "emplace" << endl;
45. f1.emplace\_front(1);
46. cout << "f1 = " << f1 << endl;
47. f1.emplace\_after(f1.begin(),2);
48. cout << "f1 = " << f1 << endl << endl;
49. cout << "push\_front" << endl;
50. f1.push\_front(1);
51. cout << "f1 = " << f1 << endl << endl;
52. cout << "swap" << endl;
53. f1.swap(f6);
54. cout << "f1 = " << f1 << endl;
55. f1.resize(5);
56. cout << "f1 = " << f1 << endl << endl;
57. cout << "reverse" << endl;
58. f1.reverse();
59. cout << "f1 = "<< f1 << endl << endl;
60. f1.assign({2,4,7,4,5,9,5});
61. f2.assign({1,5,7,3,6,2,5});
62. // forward\_lists are supposed to be sorted before merge
63. cout << "sort" << endl;
64. cout << "before sort" << endl;
65. cout << "f1 = " << f1 << endl;
66. cout << "f2 = " << f2 << endl;
67. f1.sort();
68. f2.sort();
69. cout << "after sort" << endl;
70. cout << "f1 = " << f1 << endl;
71. cout << "f2 = " << f2 << endl << endl;
72. cout << "merge" << endl;
73. cout << "f1.merge(f2);" << endl;
74. f1.merge(f2);
75. cout << "f1 = " << f1 << endl;
76. cout << "f2 = " << f2 << endl << endl;
77. cout << "f1.unique();" << endl;
78. f1.unique();
79. cout << "f1 = " << f1 << endl << endl;
80. cout << "splice\_after" << endl;
81. cout << "f3 = " << f3 << endl;
82. f1.splice\_after(++f1.begin(),f3);
83. cout << "f1 = " << f1 << endl;
84. }
85. ostream& operator<<(ostream& out, const forward\_list<int>& obj)
86. {
87. for (auto forward\_listIt = obj.cbegin(); forward\_listIt != obj.cend(); ++forward\_listIt)
88. out << \*forward\_listIt << ' ';
89. return out;
90. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

f2 = 0 0 0 0 0

f3 = 19 19 19 19 19

f4 = 2 3 5 7 11 13 17

f4 =

f5 = 2 3 5 7 11 13 17

f6 = 2 3 5 7 11 13 17

f1.max\_size() = 1152921504606846975 f1.empty() = true

f5.front() = 2

\*f5.begin() = 2

\*++f5.before\_begin() = 2

assign

f1 = 7 7 7 7 7

f1 = 7 6 5 4 3 2 1

erase\_after

f1 = 7 5 4 3 2 1

insert\_after

f1 = 3 7 5 4 3 2 1

f1 = 3 19 19 19 19 19 7 5 4 3 2 1

emplace

f1 = 1 3 19 19 19 19 19 7 5 4 3 2 1

f1 = 1 2 3 19 19 19 19 19 7 5 4 3 2 1

push\_front

f1 = 1 1 2 3 19 19 19 19 19 7 5 4 3 2 1

swap

f1 = 2 3 5 7 11 13 17

f1 = 2 3 5 7 11

reverse

f1 = 11 7 5 3 2

sort

before sort

f1 = 2 4 7 4 5 9 5

f2 = 1 5 7 3 6 2 5

after sort

f1 = 2 4 4 5 5 7 9

f2 = 1 2 3 5 5 6 7

merge

f1.merge(f2);

f1 = 1 2 2 3 4 4 5 5 5 5 6 7 7 9

f2 =

f1.unique();

f1 = 1 2 3 4 5 6 7 9

splice\_after

f3 = 19 19 19 19 19

f1 = 1 2 19 19 19 19 19 3 4 5 6 7 9

## deque

The deque container is similar to vectors and lists. The deque container provides direct access to elements, like a vector and efficient insertion and deletion at both ends, like a list. Unlike a vector, a deque elements are not stored in contiguous memory. The deque container requires the <deque> header file.

### Constructors

Default constructor

deque();

Fill constructors

explicit deque(size\_type n, const allocator\_type& alloc =

allocator\_type());

deque(size\_type n, const value\_type& val,

const allocator\_type& alloc = allocator\_type());

Range constructor

template <class InputIterator>

deque(InputIterator first, InputIterator last,

const allocator\_type& alloc = allocator\_type());

Copy constructor

deque(const deque& x);

Move constructor

deque(deque&& x);

Initializer list constructor

deque(initializer\_list<value\_type> lst,

const allocator\_type& alloc = allocator\_type());

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the deque

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the deque

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last element in the deque

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the *non-existing* element in front of the first element of the deque

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a const iterator pointing to the first element of the deque

const\_iterator begin() const noexcept;

#### cend

Returns a const iterator pointing to the *non-existing* element beyond the end of the deque

const\_iterator end() const noexcept;

#### crbegin

Returns a const reverse iterator pointing to the last element of the deque

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a const reverse iterator pointing to the non-existing element in front of the first element of the deque

const\_reverse\_iterator rend() const noexcept;

### Capacity Functions

#### size

Returns the number of elements in the deque

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a deque can hold

size\_t max\_size() const noexcept;

#### resize

Resizes a deque to n elements

void resize (size\_t n);

void resize (size\_t n, const value\_type& value);

#### empty

Returns whether the deque is empty

bool empty() const noexcept;

#### shrink\_to\_fit

Changes the capacity to the size of the deque

void shrink\_to\_fit();

### Access Functions

#### at

Returns element at position

value\_type& at(size\_t position);

const value\_type& at(size\_t position) const;

#### back

Returns a reference to the last element in the deque

value\_type& back();

const value\_type& back() const;

#### front

Returns a reference to the first element in the deque

value\_type& front();

const value\_type& front() const;

### Modifier Functions

#### assign

Assigns new contents to a deque

template <class InputIterator>

void assign(InputIterator beg, InputIterator \_end);

void assign(size\_type n, const value\_type& value);

void assign(initializer\_list<value\_type> list);

#### clear

Erases a deque. Size becomes 0

void clear() noexcept;

#### erase

Erases part of a deque

iterator erase(const\_iterator p);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into a deque at a specified location

iterator insert(const\_iterator loc, const value\_type& value);

iterator insert(const\_iterator loc, size\_type n, const value\_type& value);

template <class InputIterator>

iterator insert(const\_iterator loc, InputIterator first, InputIterator last);

iterator insert(const\_iterator loc, value\_type&& value);

iterator insert(const\_iterator loc, initializer\_list<value\_type> list);

#### push\_back

Adds an element to the end of a deque

void push\_back(const value\_type& value);

void push\_back(value\_type&& value);

#### pop\_back

Deletes the last element of a deque

void pop\_back();

#### push\_front

Adds an element to the beginning of a deque

void push\_front(const value\_type& value);

void push\_front(value\_type&& value);

#### pop\_front

Deletes the first element of a deque

void pop\_front();

**swap**

Swaps two deques

void swap(deque& vec);

#### emplace

Constructs and inserts a new element at a specified location in the deque

template <class Type> void emplace(const iterator loc, Type&&... args);

#### emplace\_front

Constructs and inserts a new element in the beginning of a deque

template <class Type> void emplace\_front(Type&&... args);

#### emplace\_back

Constructs and inserts a new element at the end of the deque

template <class Type> void emplace\_back(Type&&... args);

### Member Operators

#### operator=

The assignment operator: assigns new contents to a deque.

deque& operator=(const deque& x);

deque& operator=(deque&& x);

deque& operator=(initializer\_list<value\_type> lst);

#### operator[]

Index operator: returns the element at the specified location

value\_type& operator[](size\_t location);

const value\_type& operator[](size\_t location) const;

#### Relational operators

== > < >= <= !=

Used to compare the contents of two deques.

Two deques are equal (==) if their sizes match and each of the corresponding elements match.

A less than (<) comparison is made between two deques by comparing successive elements in order.

### Example 5 – The deque container

1. #include <forward\_list>
2. #include <iostream>
3. using namespace std;
4. ostream& operator<<(ostream& out, const forward\_list<int>& obj);
5. int main()
6. {
7. // Constructors
8. forward\_list<int> f1;
9. forward\_list<int> f2(5);
10. forward\_list<int> f3(5,19);
11. forward\_list<int> f4{2,3,5,7,11,13,17};
12. cout << "f2 = "<< f2 << endl;
13. cout << "f3 = "<< f3 << endl;
14. cout << "f4 = "<< f4 << endl;
15. cout << endl;
16. forward\_list<int> f5(f4);
17. forward\_list<int> f6(move(f4));
18. cout << "f4 = "<< f4 << endl;
19. cout << "f5 = "<< f5 << endl;
20. cout << "f6 = "<< f6 << endl;
21. cout << endl;
22. // Capacity functions
23. cout << "f1.max\_size() = " << f1.max\_size() << ' ' << boolalpha
24. << " f1.empty() = " << f1.empty() << endl << endl;
25. // Access and Iterator functions
26. cout << "f5.front() = " << f5.front() << endl;
27. cout << "\*f5.begin() = " << \*f5.begin() << endl;
28. cout << "\*++f5.before\_begin() = " << \*++f5.before\_begin()
29. << endl << endl;
30. // Modifier functions
31. cout << "assign" << endl;
32. f1.assign(5,7);
33. cout << "f1 = " << f1 << endl;
34. f1.assign({7,6,5,4,3,2,1});
35. cout << "f1 = " << f1 << endl;
36. cout << endl;
37. cout << "erase\_after" << endl;
38. f1.erase\_after(f1.begin());
39. cout << "f1 = " << f1 << endl << endl;
40. cout << "insert\_after" << endl;
41. f1.insert\_after(f1.before\_begin(),3);
42. cout << "f1 = " << f1 << endl;
43. f1.insert\_after(f1.begin(),f3.begin(),f3.end());
44. cout << "f1 = " << f1 << endl << endl;
45. cout << "emplace" << endl;
46. f1.emplace\_front(1);
47. cout << "f1 = " << f1 << endl;
48. f1.emplace\_after(f1.begin(),2);
49. cout << "f1 = " << f1 << endl << endl;
50. cout << "push\_front" << endl;
51. f1.push\_front(1);
52. cout << "f1 = " << f1 << endl << endl;
53. cout << "swap" << endl;
54. f1.swap(f6);
55. cout << "f1 = " << f1 << endl;
56. f1.resize(5);
57. cout << "f1 = " << f1 << endl << endl;
58. cout << "reverse" << endl;
59. f1.reverse();
60. cout << "f1 = "<< f1 << endl << endl;
61. cout << "merge" << endl;
62. f1.assign({2,4,7,4,5,9,5});
63. f2.assign({1,5,7,3,6,2,5});
64. cout << "before merge: f1 = " << f1 << endl;
65. cout << "before merge: f2 = " << f2 << endl;
66. cout << "f1.merge(f2);" << endl;
67. f1.merge(f2);
68. cout << "after merge: f1 = " << f1 << endl;
69. cout << "after merge: f2 = " << f2 << endl << endl;
70. // forward\_lists are supposed to be sorted before merge
71. f1.assign({2,4,7,4,5,9,5});
72. f2.assign({1,5,7,3,6,2,5});
73. cout << "sort" << endl;
74. cout << "before sort" << endl;
75. cout << "f1 = " << f1 << endl;
76. cout << "f2 = " << f2 << endl;
77. f1.sort();
78. f2.sort();
79. cout << "after sort" << endl;
80. cout << "f1 = " << f1 << endl;
81. cout << "f2 = " << f2 << endl << endl;
82. cout << "f1.merge(f2);" << endl;
83. f1.merge(f2);
84. cout << "f1 = " << f1 << endl;
85. cout << "f2 = " << f2 << endl << endl;
86. cout << "f1.unique();" << endl;
87. f1.unique();
88. cout << "f1 = " << f1 << endl << endl;
89. cout << "splice\_after" << endl;
90. cout << "f3 = " << f3 << endl;
91. f1.splice\_after(++f1.begin(),f3);
92. cout << "f1 = " << f1 << endl;
93. }
94. ostream& operator<<(ostream& out, const forward\_list<int>& obj)
95. {
96. for (auto forward\_listIt = obj.cbegin(); forward\_listIt != obj.cend(); ++forward\_listIt)
97. out << \*forward\_listIt << ' ';
98. return out;
99. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

f2 = 0 0 0 0 0

f3 = 19 19 19 19 19

f4 = 2 3 5 7 11 13 17

f4 =

f5 = 2 3 5 7 11 13 17

f6 = 2 3 5 7 11 13 17

f1.max\_size() = 1152921504606846975 f1.empty() = true

f5.front() = 2

\*f5.begin() = 2

\*++f5.before\_begin() = 2

assign

f1 = 7 7 7 7 7

f1 = 7 6 5 4 3 2 1

erase\_after

f1 = 7 5 4 3 2 1

insert\_after

f1 = 3 7 5 4 3 2 1

f1 = 3 19 19 19 19 19 7 5 4 3 2 1

emplace

f1 = 1 3 19 19 19 19 19 7 5 4 3 2 1

f1 = 1 2 3 19 19 19 19 19 7 5 4 3 2 1

push\_front

f1 = 1 1 2 3 19 19 19 19 19 7 5 4 3 2 1

swap

f1 = 2 3 5 7 11 13 17

f1 = 2 3 5 7 11

reverse

f1 = 11 7 5 3 2

merge

before merge: f1 = 2 4 7 4 5 9 5

before merge: f2 = 1 5 7 3 6 2 5

f1.merge(f2);

after merge: f1 = 1 2 4 5 7 4 5 7 3 6 2 5 9 5

after merge: f2 =

sort

before sort

f1 = 2 4 7 4 5 9 5

f2 = 1 5 7 3 6 2 5

after sort

f1 = 2 4 4 5 5 7 9

f2 = 1 2 3 5 5 6 7

f1.merge(f2);

f1 = 1 2 2 3 4 4 5 5 5 5 6 7 7 9

f2 =

f1.unique();

f1 = 1 2 3 4 5 6 7 9

splice\_after

f3 = 19 19 19 19 19

f1 = 1 2 19 19 19 19 19 3 4 5 6 7 9

## queue

The queue container *adaptor* implements a FIFO (first in, first out) container. The queue is an *adaptor*. This means that its data is a container itself. The queue adapter is simply an interface to the underlying container. Elements of a queue are pushed on to the back of the queue and popped off the front of the queue. The queue container requires the <queue> header file.

### Constructors

Initialize constructor

explicit queue(const container\_type& ctnr);

Move initialize constructor

explicit queue(container\_type&& ctnr = container\_type());

Where is the copy constructor?

### Member Functions

#### size

Returns the number of elements in the queue

size\_type size() const;

#### empty

Returns whether the queue is empty

bool empty() const;

#### back

Returns a reference to the last element added to the queue.

value\_type& back();

const value\_type& back() const;

#### front

Returns a reference to the first element in the queue. This is the next element that will be *popped off*.

value\_type& front();

const value\_type& front() const;

#### push

Adds an element to the end of a queue.

void push(const value\_type& value);

void push(value\_type&& value);

#### pop

Removes the first element in the queue. That is, the *oldest* element in the queue.

void pop();

#### emplace

Constructs and add a new element to the back of the queue.

template <class Type> void emplace(Type&&... args);

#### swap

Swaps the contents of two queues. The types of the queues must match.

void swap(queue& another\_queue) noexcept;

#### Relational operators

== > < >= <= !=

Used to compare the contents of two queues.

Two deques are equal (==) if their sizes match and each of the corresponding elements match.

A less than (<) comparison is made between two queues by comparing successive elements in order.

### Example 6 – The queue adaptor

1. #include <list>
2. #include <vector>
3. #include <queue>
4. #include <iostream>
5. using namespace std;
6. int main()
7. {
8. // Constructors
9. queue<int> q1;
10. q1.push(10);
11. q1.push(20);
12. q1.push(30);
13. cout << "q1.size() = " << q1.size() << endl;
14. cout << "q1.front() = " << q1.front() << endl;
15. cout << "q1.back() = " << q1.back() << endl << endl;
16. cout << "\"process q1\"" << endl;
17. while (!q1.empty())
18. {
19. cout << q1.front() << ' ';
20. q1.pop();
21. }
22. cout << endl << endl;
23. cout << "Create a queue using an underlying list" << endl;
24. list<int> l1{2,3,5,7};
25. queue<int, list<int>> q2(l1);
26. cout << "q2.size() = " << q2.size() << endl;
27. cout << "q2.front() = " << q2.front() << endl;
28. cout << "q2.back() = " << q2.back() << endl << endl;
29. cout << "\"process q2\"" << endl;
30. while (!q2.empty())
31. {
32. cout << q2.front() << ' ';
33. q2.pop();
34. }
35. cout << endl << endl;
36. cout << "emplace" << endl;
37. q2.emplace(17);
38. q2.emplace(18);
39. cout << "q2.front() = " << q2.front() << endl;
40. cout << "q2.back() = " << q2.back() << endl;
41. cout << endl;
42. cout << "Create a queue by moving a vector" << endl;
43. vector<double> v1{1.2,3.4,5.6,7.8};
44. queue<double, vector<double>> q4(move(v1));
45. cout << "q4.size() = " << q4.size() << endl;
46. cout << "v1.size() = " << v1.size() << endl;
47. cout << endl;
48. queue<double> q5;
49. // q5.swap(q4); ERROR
50. v1 = {1.1,2.2,3.3}; // reassign vector v1
51. cout << "create a queue using an underlying vector of doubles" << endl;
52. queue<double, vector<double>> q6(v1);
53. cout << "swap two queues" << endl;
54. q6.swap(q4);
55. cout << "q6.size() = " << q6.size() << endl;
56. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

q1.size() = 3

q1.front() = 10

q1.back() = 30

"process q1"

10 20 30

Create a queue using an underlying list

q2.size() = 4

q2.front() = 2

q2.back() = 7

"process q2"

2 3 5 7

emplace

q2.front() = 17

q2.back() = 18

Create a queue by moving a vector

q4.size() = 4

v1.size() = 0

create a queue using an underlying vector of doubles

swap two queues

q6.size() = 4

## priority\_queue

The priority\_queue *adaptor* implements a container in which the first element is always the one that is considered the maximum value. Hence, the maximum value will always be *popped off* first. The determination of the maximum value requires a *binary predicate*[[5]](#footnote-5) to make comparison of the priority\_queue values. The priority\_queue container requires the **<queue>** header file.

### Constructors

Initialize constructor

priority\_queue (const Compare& comp, const Container& ctnr);

Move initialize constructor

explicit priority\_queue (const Compare& comp = Compare(),

Container&& ctnr = Container());

Range constructor

template <class InputIterator>

priority\_queue (InputIterator first, InputIterator last,

const Compare& comp, const Container& ctnr);

Move range constructor

template <class InputIterator>

priority\_queue (InputIterator first, InputIterator last,

const Compare& comp, Container&& ctnr = Container());

### Member Functions

#### size

Returns the number of elements in the priority\_queue

size\_type size() const;

#### empty

Returns whether the priority\_queue is empty

bool empty() const;

#### top

Returns a reference to the top (first to be *popped*) element in the queue.

const value\_type& top() const;

#### push

Inserts a new element into the priority\_queue.

void push(const value\_type& value);

void push(value\_type&& value);

#### pop

Removes the top element in the priority\_queue. This is the element with the maximum *value*.

void pop();

#### emplace

Constructs and inserts a new element into the priority\_queue.

template <class Type> void emplace(Type&&... args);

#### swap

Swaps the contents of two priority\_queues. Both the value types and the comparison functions of the two priority\_queues must match.

void swap(priority\_queue& another\_pq) noexcept;

### Example 7 – The priority\_queue adaptor

1. #include <iostream>
2. #include <queue>
3. #include <vector>
4. #include <functional> // for greater<int>
5. #include <string>
6. using namespace std;
7. // "Non-destructive" print function?
8. template<typename T> void print\_queue(T q)
9. {
10. while(!q.empty())
11. {
12. std::cout << q.top() << " ";
13. q.pop();
14. }
15. std::cout << '\n';
16. }
17. // binary predicate (function object/functor) for comparing strings
18. // returns true if first string is shorter than second string
19. struct longer
20. {
21. bool operator()(const string& a, const string& b)
22. {
23. return a.size() < b.size();
24. }
25. };
26. int main ()
27. {
28. int myints[]= {10,60,50,20};
29. vector<int> v1{10,20,30,40};
30. vector<string> v2{"Have","a","really","very","nice","day","."};
31. // pq1, pq2, pq3 uses default < comparison for type int
32. priority\_queue<int> pq1;
33. priority\_queue<int> pq2 (v1.begin(), v1.end());
34. priority\_queue<int> pq3 (myints,myints+4);
35. // pq4 uses default > comparison for type int for priority
36. priority\_queue<int, vector<int>, std::greater<int> > pq4 (myints,myints+4);
37. // pq5 uses default < comparison for type string
38. priority\_queue<string> pq5 (v2.begin(),v2.end());
39. // pq6 uses longer binary predicate comparison for type string
40. priority\_queue<string, vector<string>, longer> pq6 (v2.begin(),v2.end());
41. cout << "pq2 = "; print\_queue(pq2);
42. cout << "pq3 = "; print\_queue(pq3);
43. cout << "pq4 = "; print\_queue(pq4);
44. cout << "pq5 = "; print\_queue(pq5);
45. cout << "pq6 = "; print\_queue(pq6);
46. cout << "pq3.size()=" << pq3.size() << endl;
47. cout << "pq4.size()=" << pq4.size() << endl << endl;
48. cout << "pq2 and pq3 swapped" << endl;
49. pq2.swap(pq3);
50. // pq3.swap(pq4); ERROR - why?
51. cout << "pq2 = "; print\_queue(pq2);
52. pq2.push(95);
53. pq2.push(5);
54. pq2.push(25);
55. pq2.emplace(35);
56. cout << "pq2 = "; print\_queue(pq2);
57. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

pq2 = 40 30 20 10

pq3 = 60 50 20 10

pq4 = 10 20 50 60

pq5 = very really nice day a Have .

pq6 = really Have nice very day . a

pq3.size()=4

pq4.size()=4

pq2 and pq3 swapped

pq2 = 60 50 20 10

pq2 = 95 60 50 35 25 20 10 5

## stack

The stack container *adaptor* implements a LIFO (last in, first out) container. The stack, like a queue and a priority\_queue is an *adaptor*, meaning that its data is a container itself. The stack uses a deque, by default as its underlying container. Elements of a stack are pushed on to the top of the stack and popped off the top of the stack. The queue container requires the <stack> header file.

### Constructors

Initialize constructor

explicit stack(const container\_type& ctnr);

Move initialize constructor

explicit stack(container\_type&& ctnr = container\_type());

### Member Functions

#### size

Returns the number of elements in the stack.

size\_type size() const;

#### empty

Returns whether the stack is empty

bool empty() const;

#### top

Returns a reference to the last element added to the stack.

value\_type& top();

const value\_type& top() const;

#### push

Adds an element to the top of the stack.

void push(const value\_type& value);

void push(value\_type&& value);

#### pop

Removes the element on the top of the stack. That is, the *last* element pushed on the stack.

void pop();

#### emplace

Constructs and add a new element to the top of the stack.

template <class Type> void emplace(Type&&... args);

#### swap

Swaps the contents of two stacks. The types of the stacks must match. Note, swap swaps the two underlying containers.

void swap(stack& another\_stack) noexcept;

#### Relational operators

== > < >= <= !=

Used to compare the contents of two stacks.

Two deques are equal (==) if their sizes match and each of the corresponding elements match.

A less than (<) comparison is made between two deques by comparing successive elements in order.

### Example 8 – The stack adaptor

1. #include <list>
2. #include <vector>
3. #include <stack>
4. #include <iostream>
5. using namespace std;
6. // Why is this a template?
7. template<typename T> void print\_stack(T q)
8. {
9. while(!q.empty())
10. {
11. cout << q.top() << " ";
12. q.pop();
13. }
14. cout << endl;
15. }
16. int main()
17. {
18. // Constructors
19. stack<int> stk1;
20. stk1.push(10);
21. stk1.push(20);
22. stk1.push(30);
23. cout << "stk1 = "; print\_stack(stk1);
24. cout << endl;
25. list<int> l1{2,3,5,7};
26. stack<int, list<int>> stk2(l1);
27. cout << "stk2 = "; print\_stack(stk2);
28. cout << endl;
29. stk2.emplace(17);
30. stk2.emplace(18);
31. cout << "stk2 = "; print\_stack(stk2);
32. cout << endl;
33. vector<double> v1{1.2,3.4,5.6,7.8};
34. stack<double, vector<double>> stk3(move(v1));
35. cout << stk3.size() << endl;
36. cout << v1.size() << endl;
37. cout << "stk3 = "; print\_stack(stk3);
38. cout << endl;
39. stack<double> stk4;
40. // stk4.swap(stk3); ERROR - why?
41. v1 = {1.3,2.2,3.3};
42. stack<double, vector<double>> stk5(v1);
43. stk5.swap(stk3);
44. cout << "stk3 = "; print\_stack(stk3);
45. cout << "stk5 = "; print\_stack(stk5);
46. stk5.push(3.2);
47. cout << "stk5 = "; print\_stack(stk5);
48. cout << "stk3 > stk5: " << boolalpha << (stk3 > stk5) << endl;
49. cout << endl;
50. stk3.push(stk3.top());
51. stk3.push(stk3.top());
52. cout << "stk3 = "; print\_stack(stk3);
53. cout << "stk5 = "; print\_stack(stk5);
54. cout << boolalpha << endl;
55. cout << "stk3 > stk5: " << (stk3 > stk5) << endl;
56. cout << "stk3 < stk5: " << (stk3 < stk5) << endl;
57. cout << "stk3 == stk5: " << (stk3 == stk5) << endl;
58. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

stk1 = 30 20 10

stk2 = 7 5 3 2

stk2 = 18 17 7 5 3 2

4

0

stk3 = 7.8 5.6 3.4 1.2

stk3 = 3.3 2.2 1.3

stk5 = 7.8 5.6 3.4 1.2

stk5 = 3.2 7.8 5.6 3.4 1.2

stk3 > stk5: true

stk3 = 3.3 3.3 3.3 2.2 1.3

stk5 = 3.2 7.8 5.6 3.4 1.2

stk3 > stk5: true

stk3 < stk5: false

stk3 == stk5: false

## set

The set container is an associative container in which elements are unique and stored in a sorted order. The set container requires the <set> header file.

### Constructors

Default constructor

set();

empty constructor

explicit set (const key\_compare& comp, const allocator\_type& alloc = allocator\_type());

range constructor

template <class InputIterator>

set(InputIterator first, InputIterator last,

const key\_compare& comp = key\_compare(),

const allocator\_type& = allocator\_type());

template <class InputIterator>

set(InputIterator first, InputIterator last,

const allocator\_type& = allocator\_type());

copy constructor

set(const set& x);

move constructor

set(set&& x);

initializer list constructor

set(initializer\_list<value\_type> lst,

const key\_compare& comp = key\_compare(),

const allocator\_type& alloc = allocator\_type());

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the set

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the set

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last element in the set

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the *non-existing* element in front of the first element of the set

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a ***const*** iterator pointing to the first element of the set

const\_iterator begin() const noexcept;

#### cend

Returns a ***const*** iterator pointing to the *non-existing* element beyond the end of the set

const\_iterator end() const noexcept;

#### crbegin

Returns a ***const*** reverse iterator pointing to the last element of the set

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a ***const*** reverse iterator pointing to the non-existing element in front of the first element of the set

const\_reverse\_iterator rend() const noexcept;

### Capacity Functions

#### size

Returns the number of elements in the set

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a set can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the set is empty

bool empty() const noexcept;

### Modifier Functions

#### clear

Erases all elements of a set. Size becomes 0

void clear() noexcept;

#### erase

Erases elements in a set

iterator erase(const\_iterator p);

size\_t erase(const value\_type& value);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into a set at a specified location. Elements must be unique, so duplicate values may not be inserted.

pair<iterator,bool> insert(const value\_type& value);

pair<iterator,bool> insert(value\_type&& value);

iterator insert(const\_iterator position, const value\_type& value);

iterator insert(const\_iterator position, value\_type&& value);

template <class InputIterator>

void insert(InputIterator first, InputIterator last);

void insert(initializer\_list<value\_type> lst);

#### swap

Swaps two sets

void swap(set& another\_set);

### Operation Functions

#### count

Returns the number of elements that are equal to a value in the set. Because the elements in a set must be unique, count can only return 1 or 0.

size\_type count(const value\_type& value) const;

#### find

Searches the set for a value. Returns an iterator to the found element, otherwise it returns set::end().

const\_iterator find(const value\_type& value) const;

iterator find(const value\_type& value);

#### lower\_bound

Returns an iterator pointing to the first element in the set that is not less than a value. If there are no elements less than the value, then then function returns set::end().

iterator lower\_bound (const value\_type& value);

const\_iterator lower\_bound (const value\_type& value) const;

#### upper\_bound

Returns an iterator pointing to the first element in the set that is greater than a value. If there are no elements greater than the value, then then function returns set::end().

iterator upper\_bound (const value\_type& value);

const\_iterator upper\_bound (const value\_type& value) const;

### Example 9 – The set container

1. #include <iostream>
2. #include <set>
3. using namespace std;
4. class Student
5. {
6. unsigned id;
7. string name;
8. public:
9. Student() = delete;
10. Student(unsigned arg1, string arg2 = "") : id(arg1), name(arg2) {}
11. Student(const Student&) = default;
12. bool operator<(const Student& obj) const
13. {
14. return id < obj.id;
15. }
16. bool operator==(const Student& obj) const
17. {
18. return id == obj.id;
19. }
20. friend ostream& operator<<(ostream& out, const Student& obj)
21. {
22. out << obj.id << " " << obj.name;
23. return out;
24. }
25. };
26. ostream& operator<<(ostream& out, const set<Student>& stu)
27. {
28. for (auto it = stu.cbegin(); it != stu.cend(); ++it)
29. {
30. out << \*it << endl;
31. }
32. return out;
33. }
34. int main()
35. {
36. set<Student> Students;
37. Students.insert({117,"John"});
38. Students.insert({124,"Paul"});
39. Students.insert({102,"George"});
40. Students.insert({106,"Ringo"});
41. Students.insert({223,"Peter"});
42. Students.insert({203,"Paul"});
43. Students.insert({243,"Mary"});
44. cout << "Students.size() = " << Students.size() << endl;
45. cout << Students << endl;
46. bool insertSuccess;
47. cout << boolalpha;
48. insertSuccess = Students.insert({309,"Mick"}).second;
49. cout << "insert 309: " << insertSuccess << endl;
50. insertSuccess = Students.insert({117,"Nobody"}).second;
51. cout << "insert 117: " << insertSuccess << endl << endl;
52. cout << "find 106: " << \*(Students.find(106)) << endl; // How does this work?
53. // cout << \*(Students.find(107)) << endl; // ERROR
54. unsigned id;
55. set<Student>::const\_iterator it;
56. cout << "find 203: " << (Students.find(203) != Students.end()) << endl;
57. cout << "find 107: " << (Students.find(107) != Students.end()) << endl << endl;
58. cout << "Before erase: Students.size() = " << Students.size() << endl;
59. id = 203;
60. Students.erase(Students.find(id)); // Did this work?
61. cout << "After erase of 203: Students.size() = " << Students.size() << endl;
62. cout << "Students.erase(102) = " << Students.erase(102) << endl;
63. cout << "Students.erase(103) = " << Students.erase(103) << endl;
64. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Students.size() = 7

102 George

106 Ringo

117 John

124 Paul

203 Paul

223 Peter

243 Mary

insert 309: true

insert 117: false

find 106: 106 Ringo

find 203: true

find 107: false

Before erase: Students.size() = 8

After erase of 203: Students.size() = 7

Students.erase(102) = 1

Students.erase(103) = 0

## multiset

The multiset container is an associative container in which elements stored in a sorted order, but element values are not unique. The multiset container requires the <set> header file.

### Member Functions

The multiset constructors and member functions are essentially the same as the set container. The following illustrates some of the differences.

#### erase

Erases elements in a multiset

iterator erase(const\_iterator p);

Only a single element of the multiset is erased.

size\_t erase(const value\_type& value);

Erases all elements in the multiset with a key equal to the specified value. The function returns the number of elements erased.

#### insert

iterator insert(const value\_type& val);

iterator insert(value\_type&& val);

This version of the insert function returns only an iterator to the element that was inserted. Unlike the set::insert, there is no bool indication of success or failure.

As of C++11, when duplicate values are inserted into the multiset, newly inserted elements are inserted after those with the same value.

#### count

Like the set::count the function returns the number of elements that are equal to a value in the set. Since the elements in a multiset are not necessarily unique, the count may be greater than 1.

size\_type count(const value\_type& value) const;

#### equal\_range

Returns a pair of iterators pointer to the first and last element that is equal to a value in the multiset. If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that is greater than the value.

pair<const\_iterator,const\_iterator> equal\_range(const value\_type& value) const;

pair<iterator,iterator> equal\_range(const value\_type& value);

### Non-member Functions

Note: these operators, > < >= <= != will be removed in C++20. The <=> operator will be added. More to say about that later.

### Example 10 – The multiset container

1. #include <iostream>
2. #include <set>
3. using namespace std;
4. class Student
5. {
6. unsigned id;
7. string name;
8. public:
9. Student() = delete;
10. Student(unsigned arg1, string arg2 = "") : id(arg1), name(arg2) {}
11. Student(const Student&) = default;
12. bool operator<(const Student& obj) const
13. {
14. return id < obj.id;
15. }
16. bool operator==(const Student& obj) const
17. {
18. return id == obj.id;
19. }
20. friend ostream& operator<<(ostream& out, const Student& obj)
21. {
22. out << obj.id << " " << obj.name;
23. return out;
24. }
25. };
26. ostream& operator<<(ostream& out, const multiset<Student>& stu)
27. {
28. for (auto it = stu.cbegin(); it != stu.cend(); ++it)
29. {
30. out << \*it << endl;
31. }
32. return out;
33. }
34. int main()
35. {
36. **multiset**<Student> Students;
37. Students.insert({117,"John"});
38. Students.insert({124,"Paul"});
39. Students.insert({102,"George"});
40. Students.insert({106,"Ringo"});
41. Students.insert({223,"Peter"});
42. Students.insert({203,"Paul"});
43. Students.insert({243,"Mary"});
44. cout << "Students.size() = " << Students.size() << endl;
45. cout << Students << endl;
46. multiset<Student>::iterator msIt;
47. msIt = Students.insert({309,"Mick"});
48. cout << "New student: " << \*msIt << endl;
49. msIt = Students.insert({117,"Elvis"});
50. cout << "Another new student: " << \*msIt << endl << endl;
51. cout << Students << endl;
52. // Check count
53. cout << "count of 117 = " << Students.count(117) << endl;
54. // cout << "# of Paul = " << Students.count("Paul") << endl; // ERROR
55. cout << endl;
56. // check find
57. multiset<Student>::const\_iterator cMsIt;
58. cMsIt = Students.find(124);
59. cout << "find 124: " << \*cMsIt << endl;
60. // cout << \*(Students.find(107)) << endl; // ERROR
61. ++cMsIt;
62. cout << \*cMsIt << endl;
63. ++cMsIt;
64. cout << \*cMsIt << endl;
65. int id = 125;
66. cMsIt = Students.find(id);
67. // cout << \*cMsIt << endl; // CRASH
68. if (cMsIt == Students.end())
69. cout << "Can't find " << id << endl << endl;
70. // equal\_range
71. cout << "equal\_range 117" << endl;
72. auto twoIterators = Students.equal\_range(117);
73. cout << \*twoIterators.first << endl << \*twoIterators.second << endl << endl;
74. cout << "equal\_range 203" << endl;
75. twoIterators = Students.equal\_range(203);
76. cout << \*twoIterators.first << endl << \*twoIterators.second << endl << endl;
77. cout << "equal\_range 204" << endl;
78. twoIterators = Students.equal\_range(204);
79. cout << \*twoIterators.first << endl << \*twoIterators.second << endl << endl;
80. if (twoIterators.first == twoIterators.second) cout << "204 not found" << endl << endl;
81. // erase
82. cout << "Erase 117: " << Students.erase(117) << endl;
83. cout << "Erase 118: " << Students.erase(118) << endl << endl;
84. cout << Students << endl;
85. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Students.size() = 7

102 George

106 Ringo

117 John

124 Paul

203 Paul

223 Peter

243 Mary

New student: 309 Mick

Another new student: 117 Elvis

102 George

106 Ringo

117 John

117 Elvis

124 Paul

203 Paul

223 Peter

243 Mary

309 Mick

count of 117 = 2

find 124: 124 Paul

203 Paul

223 Peter

Can't find 125

equal\_range 117

117 John

124 Paul

equal\_range 203

203 Paul

223 Peter

equal\_range 204

223 Peter

223 Peter

204 not found

Erase 117: 2

Erase 118: 0

102 George

106 Ringo

124 Paul

203 Paul

223 Peter

243 Mary

309 Mick

## map

The map container is an associative container in which elements, consisting of a key-mapped value ***pair*** stored in a sorted order by the key. The key value must be unique in the map. The map container requires the <map> header file.

### Constructors

Default constructor

map();

empty constructor

explicit map(const key\_compare& comp, const allocator\_type& alloc = allocator\_type());

range constructor

template <class InputIterator>

map(InputIterator first, InputIterator last,

const key\_compare& comp = key\_compare(),

const allocator\_type& = allocator\_type());

template <class InputIterator>

map(InputIterator first, InputIterator last,

const allocator\_type& = allocator\_type());

copy constructor

map(const map& x);

move constructor

map(map&& x);

initializer list constructor

map(initializer\_list<value\_type> lst,

const key\_compare& comp = key\_compare(),

const allocator\_type& alloc = allocator\_type());

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the map

iterator begin() noexcept;

const\_iterator begin() const noexcept;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the map

iterator end() noexcept;

const\_iterator end() const noexcept;

#### rbegin

Returns a reverse iterator pointing to the last element in the map

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

#### rend

Returns a reverse iterator pointing to the *non-existing* element in front of the first element of the map

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

#### cbegin

Returns a ***const*** iterator pointing to the first element of the map

const\_iterator begin() const noexcept;

#### cend

Returns a ***const*** iterator pointing to the *non-existing* element beyond the end of the map

const\_iterator end() const noexcept;

#### crbegin

Returns a ***const*** reverse iterator pointing to the last element of the map

const\_reverse\_iterator rbegin() const noexcept;

#### crend

Returns a ***const*** reverse iterator pointing to the non-existing element in front of the first element of the map

const\_reverse\_iterator rend() const noexcept;

### Capacity Functions

#### size

Returns the number of elements in the map

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a map can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the map is empty

bool empty() const noexcept;

### Modifier Functions

#### clear

Erases all elements of a map. Size becomes 0

void clear() noexcept;

#### erase

Erases elements in a map

iterator erase(const\_iterator p);

size\_t erase(const *key\_type*& value);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into a map at a specified location

Note, the value\_type is a *key, mapped-value pair*, in which the *key* must be unique.

pair<iterator,bool> insert(const value\_type& value);

pair<iterator,bool> insert(value\_type&& value);

iterator insert(const\_iterator position, const value\_type& value);

iterator insert(const\_iterator position, value\_type&& value);

template <class InputIterator>

void insert(InputIterator first, InputIterator last);

void insert(initializer\_list<value\_type> lst);

#### swap

Swaps two maps

void swap(map & another\_ map);

### Operation Functions

#### count

Returns the number of elements that are equal to a key in the map. Because the elements in a map must be unique, count can only return 1 or 0.

size\_type count(const *key\_type*& value) const;

#### find

Searches the map for a key. Returns an iterator to the found element, otherwise it returns map::end().

const\_iterator find(const *key\_type*& key) const;

iterator find(const *key\_type*& key);

#### lower\_bound

Returns an iterator pointing to the first element in the map that is not less than a key\_value. If there are no elements less than the key\_value, then then function returns map::end().

iterator lower\_bound (const key\_type& key);

const\_iterator lower\_bound (const key\_type& key) const;

#### upper\_bound

Returns an iterator pointing to the first element in the map that is greater than a key\_value. If there are no elements greater than the key\_value, then then function returns map::end().

iterator upper\_bound (const key\_type& key);

const\_iterator upper\_bound (const key\_type& key) const;

### Accessor function/operator

#### operator[]

Returns the mapped-value for a given key-value. If the key-value is not contained in the map, then the operator inserts a new element into the map, with a *default-constructed mapped-value*.

mapped\_type& operator[] (const key\_type& key);

mapped\_type& operator[] (key\_type&& key);

#### at

Returns the mapped-value for a given key-value. If the key-value is not contained in the map, the function throws an *out\_of\_range exception*.

mapped\_type& at(const key\_type& key);

const mapped\_type& at(const key\_type& key) const;

### Example 11 – The map container

1. #include <iostream>
2. #include <iomanip>
3. #include <map>
4. #include <string>
5. #include <cstdlib>
6. using std::cout;
7. using std::endl;
8. using std::string;
9. // Alias declarations
10. using StudentId = unsigned;
11. using Name = string;
12. using Students = std::map<StudentId,Name>;
13. // function prototypes
14. unsigned rand100u();
15. Students::const\_iterator
16. getInteratorForName(Students&, const Name& name);
17. std::ostream& operator<<(std::ostream&, const Students&);
18. int main()
19. {
20. Students students;
21. // insert 4 Students into the map
22. students[rand100u()] = "John Lennon";
23. students.insert(std::pair<StudentId,Name>(rand100u(),"Paul McCartney"));
24. using Student = std::pair<StudentId,Name>;
25. Student george{rand100u(),"George Harrison"};
26. students.insert(george);
27. StudentId ringoId = rand100u();
28. Student ringo{ringoId,"Ringo Star"};
29. students.insert(std::move(ringo));
30. cout << students << endl;
31. // What does this mean?
32. students[50];
33. cout << students << endl;
34. // Correct the spelling of Ringo's name
35. students[ringoId] = "Ringo Starr";
36. cout << students << endl;
37. // Remove Student 50
38. students.erase(students.find(50));
39. cout << students << endl;
40. // What is John's number?
41. cout << "John's number is "
42. << getInteratorForName(students,"John Lennon")->first
43. << endl << endl;
44. auto it = getInteratorForName(students,"Mick Jagger");
45. if (it == students.end())
46. cout << "Mick Jagger ain't there" << endl << endl;
47. // count
48. cout << "number of elements with key " << ringoId << " = "
49. << students.count(ringoId) << endl;
50. cout << "number of elements with key " << ringoId+1 << " = "
51. << students.count(ringoId+1) << endl;
52. }
53. unsigned rand100u()
54. {
55. return rand() % 100 + 1;
56. }
57. std::ostream& operator<<(std::ostream& out, const Students& studs)
58. {
59. out << std::left;
60. for (auto it = studs.begin(); it != studs.end(); ++it)
61. {
62. out << std::setw(5) << it->first << std::setw(10)
63. << it->second << endl;
64. }
65. return out;
66. }
67. Students::const\_iterator
68. getInteratorForName(Students& Students, const string& name)
69. {
70. for (auto it = Students.cbegin(); it != Students.cend(); ++it)
71. {
72. if (it->second == name) return it;
73. }
74. return Students.end();
75. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

30 Ringo Star

34 John Lennon

44 Paul McCartney

63 George Harrison

30 Ringo Star

34 John Lennon

44 Paul McCartney

50

63 George Harrison

30 Ringo Starr

34 John Lennon

44 Paul McCartney

50

63 George Harrison

30 Ringo Starr

34 John Lennon

44 Paul McCartney

63 George Harrison

John's number is 34

Mick Jagger ain't there

number of elements with key 30 = 1

number of elements with key 31 = 0

## multimap

The multimap container is an associative container in which elements stored in a sorted order. Element values in a multimap are pairs of key and mapped values. Unlike the map container, element key values are not unique. The multimap container requires the <map> header file.

### Member Functions

The multimap constructors and member functions are essentially the same as the map container. The following illustrates some of the differences.

#### erase

Erases elements in a multimap

iterator erase(const\_iterator p);

Only a single element of the multimap is erased.

size\_t erase(const value\_type& value);

Erases all elements in the multimap with a key equal to the specified value. The function returns the number of elements erased.

#### insert

iterator insert(const value\_type& val);

iterator insert(value\_type&& val);

This version of the insert function returns only an iterator to the element that was inserted. Unlike the map::insert, there is no bool indication of success or failure. The multimap::insert does not fail like the map::insert when duplicate key values are inserted.

As of C++11, when duplicate values of the key are inserted into the multimap, newly inserted elements are inserted after those with the same key.

#### count

Like the map::count the function returns the number of elements that are equal to a value in the set. Since the elements in a multimap are not unique, the count may be greater than 1.

size\_type count(const value\_type& value) const;

#### equal\_range

Returns a pair of iterators pointer to the first and last element that has a key value equal to the argument value in the multimap. If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that is greater than the value.

pair<const\_iterator,const\_iterator> equal\_range(const value\_type& value) const;

pair<iterator,iterator> equal\_range(const value\_type& value);

### Example 12 – The multimap container

1. #include <iostream>
2. #include <iomanip>
3. #include <map>
4. #include <string>
5. #include <cstdlib>
6. using namespace std;
7. using fraction = pair<int,int>;
8. ostream& operator<<(ostream&, const fraction&);
9. ostream& operator<<(ostream&, const pair<double,fraction>&);
10. ostream& operator<<(ostream&, const multimap<double,fraction>&);
11. int main()
12. {
13. multimap<double,fraction> fractions;
14. // insert 7 elements into the multimap
15. fractions.insert(pair<double,fraction>(.75,fraction(3,4)));
16. fractions.insert(pair<double,fraction>(.75,fraction{6,8}));
17. fraction neg\_3\_4{-3,-4};
18. fractions.insert(pair<double,fraction>(.75,neg\_3\_4));
19. fraction temp\_fraction{1,2};
20. pair<double,fraction> temp\_double\_fraction;
21. temp\_double\_fraction = {.5,temp\_fraction};
22. fractions.insert(temp\_double\_fraction);
23. fractions.insert({.5,{2,4}});
24. fractions.insert({.333,{1,3}});
25. fractions.insert({.25,{1,4}});
26. fractions.insert({.5,{1,2}});
27. cout << fractions << endl << endl;
28. // fractions[.4] = fraction(2,5); // Error: no index operator
29. multimap<double,fraction>::const\_iterator cIt;
30. cIt = fractions.find(.333);
31. cout << "fractions.find(.333): " << \*cIt << endl;
32. cout << "fractions.find(.75): " <<\*fractions.find(.75) << endl;
33. cIt = fractions.find(.55);
34. cout << "fractions.find(.55): " <<\*cIt << endl;
35. if (cIt == fractions.end())
36. cout << "Can't find .55" << endl << endl;
37. cout << "fractions.count(.5)=" << fractions.count(.5) << endl;
38. cout << "fractions.count(.6)=" << fractions.count(.6) << endl << endl;
39. cout << "Elements with key = .5" << endl;
40. for (cIt = fractions.lower\_bound(.5); cIt != fractions.upper\_bound(.5); ++cIt)
41. cout << \*cIt << endl;
42. }
43. ostream& operator<<(ostream& out, const fraction& obj)
44. {
45. out << obj.first << '/' << obj.second;
46. return out;
47. }
48. ostream& operator<<(ostream& out, const pair<double,fraction>& obj)
49. {
50. out << "first: " << obj.first << " second: " << obj.second;
51. return out;
52. }
53. ostream& operator<<(ostream& out, const multimap<double,fraction>& obj)
54. {
55. for (auto it = obj.cbegin(); it != obj.cend(); ++it)
56. out << "key: " << it->first << " value: " << it->second << endl;
57. return out;
58. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

key: 0.25 value: 1/4

key: 0.333 value: 1/3

key: 0.5 value: 1/2

key: 0.5 value: 2/4

key: 0.5 value: 1/2

key: 0.75 value: 3/4

key: 0.75 value: 6/8

key: 0.75 value: -3/-4

fractions.find(.333): first: 0.333 second: 1/3

fractions.find(.75): first: 0.75 second: 3/4

fractions.find(.55): first: 3.95253e-323 second: 0/1072168960

Can't find .55

fractions.count(.5)=3

fractions.count(.6)=0

Elements with key = .5

first: 0.5 second: 1/2

first: 0.5 second: 2/4

first: 0.5 second: 1/2

## unordered\_set

The unordered\_set container stores unique values using a hash algorithm. This allows for fast retrieval of the elements using the key value. This container was introduced in C++ 11. Elements are stored in buckets using the hash value of the elements. Elements in an unordered\_set are not stored in any particular order.

### Constructors

default constructor

unordered\_set();

empty constructor

explicit unordered\_set(size\_type minimum\_number\_of\_buckets,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

range constructor

template <class InputIterator>

unordered\_set(InputIterator first, InputIterator last,

size\_type n = /\* see below \*/,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

copy constructor

unordered\_set(const unordered\_set& ust);

move constructor

unordered\_set(const unordered\_set&& ust);

initializer list constructor

unordered\_set(initializer\_list<value\_type> il,

size\_type n = automatically\_determined,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

### Capacity Functions

#### size

Returns the number of elements in the unordered\_set

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a unordered\_set can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the unordered\_set is empty

bool empty() const noexcept;

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the unordered\_set

iterator begin() noexcept;

const\_iterator begin() const noexcept;

**bucket iterator**[[6]](#footnote-6)

local\_iterator begin(size\_type n);

const\_local\_iterator begin(size\_type n) const;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the unordered\_set

iterator begin() noexcept;

const\_iterator begin() const noexcept;

**bucket iterator**

local\_iterator end(size\_type n);

const\_local\_iterator end(size\_type n) const;

#### cbegin

Returns a ***const*** iterator pointing to the first element of the unordered\_set

const\_iterator cbegin() const noexcept;

const\_local\_iterator cbegin(size\_type n) const;

#### cend

Returns a ***const*** iterator pointing to the *non-existing* element beyond the end of the unordered\_set

const\_iterator cend() const noexcept;

const\_local\_iterator cend(size\_type n) const;

### Lookup Functions

#### count

Returns the number of elements that are equal to a value in the unordered\_set. Because the elements in an unordered\_set must be unique, count can only return 1 or 0.

size\_type count(const key\_type& value) const;

#### find

Searches the unordered\_set for a key value. Returns an iterator to the found element, otherwise it returns unordered\_set::end().

const\_iterator find(const key\_type& value) const;

iterator find(const key\_type& value);

### Modifier Functions

#### clear

Erases the contents of the unordered\_set. Destructors are called for each object in the unordered\_set.

void clear() noexcept;

#### erase

Removes elements from an unordered\_set. Destructors are called for each object removed from the unordered\_set.

iterator erase(const\_iterator pos);

size\_type erase(const key\_type& key);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into an unordered\_set. unordered\_set elements must be unique, so duplicate values may not be inserted.

pair<iterator,bool> insert(const value\_type& value);

pair<iterator,bool> insert(value\_type&& value);

void insert(initializer\_list<value\_type> lst);

### Bucket Functions

#### bucket

Returns a bucket number for a given key value.

size\_type bucket (const key\_type& k) const;

#### bucket\_count

Returns the number of buckets in a unordered\_set.

size\_type bucket\_count() const noexcept;

#### bucket\_size

Returns the number of elements in a given bucket.

size\_type bucket\_size(size\_type n) const;

### Example 13 – The unordered\_set container

1. #include <iostream>
2. #include <unordered\_set>
3. using namespace std;
4. template<typename T>
5. ostream& operator<<(ostream& out, const unordered\_set<T>& obj);
6. int main()
7. {
8. unordered\_set<float> floats
9. {
10. 2.3, 6.2, 3.4, 5.6, .78, 5.5, 3.2, 0, 1.7,
11. 2, 4, 4.7, 6.6, 4, 7.3, 5.6, 2.1, 4.4, 5.5
12. };
13. cout << "floats.size() = " << floats.size() << endl;
14. for (auto it = floats.cbegin(); it != floats.cend(); ++it)
15. {
16. cout << \*it << " ";
17. }
18. cout << endl;
19. float temp = 2.4;
20. cout << temp << " is " << (floats.find(temp) == floats.end() ? "not " : "") << "present\n";
21. temp = 3.4;
22. cout << temp << " is " << (floats.find(temp) == floats.end() ? "not " : "") << "present\n\n";
23. floats.erase(3.4);
24. floats.insert(.5);
25. cout << floats << endl;
26. unordered\_set<int> ints;
27. for (int i = 0; i < 100; i++)
28. ints.insert(rand()%1000+1);
29. cout << ints << endl;
30. }
31. template<typename T>
32. ostream& operator<<(ostream& out, const unordered\_set<T>& obj)
33. {
34. out << "size = " << obj.size() << endl;
35. out << "number of buckets = " << obj.bucket\_count() << endl;
36. for (size\_t i = 0; i < obj.bucket\_count(); ++i)
37. {
38. if (obj.bucket\_size(i))
39. {
40. out << "bucket #" << i << ": ";
41. for (auto buckIt = obj.cbegin(i); buckIt != obj.cend(i); ++buckIt)
42. out << \*buckIt << " ";
43. out << endl;
44. }
45. }
46. return out;
47. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

floats.size() = 16

2.1 6.6 4.7 4 1.7 0 3.2 2 5.5 0.78 5.6 3.4 6.2 4.4 7.3 2.3

2.4 is not present

3.4 is present

size = 16

number of buckets = 19

bucket #0: 0

bucket #2: 5.6

bucket #3: 0.5 4.7

bucket #7: 0.78

bucket #8: 2.1

bucket #9: 2 5.5

bucket #11: 6.2

bucket #12: 4

bucket #14: 4.4 7.3 2.3

bucket #15: 6.6

bucket #17: 1.7

bucket #18: 3.2

size = 96

number of buckets = 97

bucket #2: 293

bucket #3: 779

bucket #4: 392

bucket #5: 102

bucket #6: 394

bucket #7: 7 492

bucket #9: 300

bucket #10: 107

bucket #16: 501

bucket #18: 309 891

bucket #22: 119 895

…

bucket #85: 85

bucket #86: 377 668

bucket #88: 282

bucket #89: 962

bucket #90: 963

bucket #91: 479

bucket #92: 674 383

bucket #93: 869 772

bucket #94: 967 191 870

bucket #95: 289

## unordered\_multiset

The unordered\_multiset container stores values using a hash algorithm. Element values are not necessarily unique as in an unordered\_set. This allow for very fast retrieval of the elements using the key value. This container was introduced in C++ 11. Elements are stored in buckets using the hash value of the elements. Elements in an unordered\_multiset are not stored in any particular order.

### Constructors

default constructor

unordered\_multiset();

empty constructor

explicit unordered\_multiset(size\_type minimum\_number\_of\_buckets,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

range constructor

template <class InputIterator>

unordered\_multiset(InputIterator first, InputIterator last,

size\_type n = /\* see below \*/,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

copy constructor

unordered\_multiset(const unordered\_multiset& ust);

move constructor

unordered\_multiset(const unordered\_multiset&& ust);

initializer list constructor

unordered\_multiset(initializer\_list<value\_type> il,

size\_type n = automatically\_determined,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

### Capacity Functions

#### size

Returns the number of elements in the unordered\_multiset

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a unordered\_multiset can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the unordered\_multiset is empty

bool empty() const noexcept;

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the unordered\_multiset

iterator begin() noexcept;

const\_iterator begin() const noexcept;

**bucket iterator**[[7]](#footnote-7)

local\_iterator begin(size\_type n);

const\_local\_iterator begin(size\_type n) const;

#### end

Returns an iterator pointing to the *non-existing* element beyond the end of the unordered\_multiset

iterator begin() noexcept;

const\_iterator begin() const noexcept;

**bucket iterator**

local\_iterator end(size\_type n);

const\_local\_iterator end(size\_type n) const;

#### cbegin

Returns a ***const*** iterator pointing to the first element of the unordered\_multiset

const\_iterator cbegin() const noexcept;

const\_local\_iterator cbegin(size\_type n) const;

#### cend

Returns a ***const*** iterator pointing to the *non-existing* element beyond the end of the unordered\_multiset

const\_iterator cend() const noexcept;

const\_local\_iterator cend(size\_type n) const;

### Lookup Functions

#### count

Returns the number of elements that are equal to a value in the unordered\_multiset

size\_type count(const key\_type& value) const;

#### find

Searches the unordered\_multiset for a key value. Returns an iterator to the found element, otherwise it returns unordered\_multiset::end().

const\_iterator find(const key\_type& value) const;

iterator find(const key\_type& value);

#### equal\_range

Returns a range (iterators) of elements for a key value. If the key value is not in the unordered\_multiset, a pair of unordered\_multiset::end() iterators is returned.

pair<iterator,iterator> equal\_range(const key\_type& value);

pair<const\_iterator,const\_iterator> equal\_range(const key\_type& value) const;

### Modifier Functions

#### clear

Erases the contents of the unordered\_multiset. Destructors are called for each object in the unordered\_multiset.

void clear() noexcept;

#### erase

Removes elements from an unordered\_multiset. Destructors are called for each object removed from the unordered\_multiset. For the erase function with a key argument, all elements in the unordered\_multiset with that key are removed.

iterator erase(const\_iterator pos);

size\_type erase(const key\_type& key);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into an unordered\_multiset. Duplicate values may be inserted, and hence, will be placed in the same bucket.

iterator insert(const value\_type& value);

iterator insert(value\_type&& value);

void insert(initializer\_list<value\_type> lst);

### Bucket Functions

#### bucket

Returns a bucket number for a given key value. Buckets are numbered from 0 to bucket\_count-1.

size\_type bucket(const key\_type& k) const;

#### bucket\_count

Returns the number of buckets in a unordered\_multiset.

size\_type bucket\_count() const noexcept;

#### bucket\_size

Returns the number of elements in a given bucket.

size\_type bucket\_size(size\_type n) const;

### Example 14 – The unordered\_multiset container

1. #include <iostream>
2. #include <iostream>
3. #include <unordered\_set>
4. using namespace std;
5. template<typename T>
6. ostream& operator<<(ostream& out, const unordered\_multiset<T>& obj);
7. int main()
8. {
9. unordered\_multiset<int> ints;
10. for (int i = 0; i < 50; i++)
11. ints.insert(rand()%10+1);
12. cout << ints << endl;
13. cout << "ints.erase(3) = " << ints.erase(3) << endl;
14. cout << "ints.erase(11) = " << ints.erase(11) << endl;
15. ints.insert(5);
16. cout << "ints.count(7) = " << ints.count(7) << endl;
17. cout << ints << endl;
18. }
19. template<typename T>
20. ostream& operator<<(ostream& out, const unordered\_multiset<T>& obj)
21. {
22. out << "size = " << obj.size() << endl;
23. out << "number of buckets = " << obj.bucket\_count() << endl;
24. for (size\_t i = 0; i < obj.bucket\_count(); ++i)
25. {
26. if (obj.bucket\_size(i))
27. {
28. out << "bucket #" << i << ": ";
29. for (auto buckIt = obj.cbegin(i); buckIt != obj.cend(i); ++buckIt)
30. out << \*buckIt << " ";
31. out << endl;
32. }
33. }
34. return out;
35. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

size = 50

number of buckets = 97

bucket #1: 1 1 1 1 1 1 1

bucket #2: 2 2 2 2 2 2

bucket #3: 3 3 3 3 3 3

bucket #4: 4 4 4 4 4 4

bucket #5: 5 5 5 5

bucket #6: 6 6 6

bucket #7: 7 7 7 7 7

bucket #8: 8 8 8 8

bucket #9: 9 9 9

bucket #10: 10 10 10 10 10 10

ints.erase(3) = 6

ints.erase(11) = 0

ints.count(7) = 5

size = 45

number of buckets = 97

bucket #1: 1 1 1 1 1 1 1

bucket #2: 2 2 2 2 2 2

bucket #4: 4 4 4 4 4 4

bucket #5: 5 5 5 5 5

bucket #6: 6 6 6

bucket #7: 7 7 7 7 7

bucket #8: 8 8 8 8

bucket #9: 9 9 9

bucket #10: 10 10 10 10 10 10

## unordered\_map

The unordered\_map container implements a map using a hash algorithm. This allows fast retrieval of the elements using the key value. Like the map container, the unordered\_map stores data in a key-value pair, with the key being the *look-up*. This container was introduced in C++ 11. Elements are stored in buckets using the hash value of the key. Elements in an unordered\_map are not stored in any particular order.

### Constructors

default constructor

unordered\_map(); // C++14

empty constructor

explicit unordered\_ map(size\_type minimum\_number\_of\_buckets,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

range constructor

template <class InputIterator>

unordered\_ map(InputIterator first, InputIterator last,

size\_type n = /\* see below \*/,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type() );

copy constructor

unordered\_ map(const unordered\_map& obj);

move constructor

unordered\_ map(const unordered\_map&& obj);

initializer list constructor

unordered\_map(initializer\_list<value\_type> il,

size\_type n = automatically\_determined,

const hasher& hf = hasher(),

const key\_equal& eql = key\_equal(),

const allocator\_type& alloc = allocator\_type());

### Capacity Functions

#### size

Returns the number of elements in the unordered\_map

size\_t size() const noexcept;

#### max\_size

Returns the maximum number of elements that a unordered\_map can hold

size\_t max\_size() const noexcept;

#### empty

Returns whether the unordered\_map is empty

bool empty() const noexcept;

### Iterator Functions

#### begin

Returns an iterator pointing to the first element of the unordered\_set

iterator begin() noexcept;

const\_iterator begin() const noexcept;

**bucket iterator**[[8]](#footnote-8)

local\_iterator begin(size\_type n);

const\_local\_iterator begin(size\_type n) const;

#### end

Returns an iterator pointing to the *non-existing* element beyond the last element of the unordered\_map

iterator begin() noexcept;

const\_iterator begin() const noexcept;

**bucket iterator**

local\_iterator end(size\_type n);

const\_local\_iterator end(size\_type n) const;

#### cbegin

Returns a ***const*** iterator pointing to the first element of the unordered\_map

const\_iterator cbegin() const noexcept;

const\_local\_iterator cbegin(size\_type n) const;

#### cend

Returns a ***const*** iterator pointing to the *non-existing* element beyond the last element of the unordered\_map

const\_iterator cend() const noexcept;

const\_local\_iterator cend(size\_type n) const;

### Lookup Functions

#### count

Returns the number of elements that are equal to a value in the unordered\_map. Because the elements in an unordered\_map must be unique, count can only return 1 or 0.

size\_type count(const key\_type& value) const;

#### find

Searches the unordered\_map for a key value. Returns an iterator to the found element, otherwise it returns unordered\_map::end().

const\_iterator find(const key\_type& value) const;

iterator find(const key\_type& value);

### Accessor function/operator

#### operator[]

Returns the mapped-value for a given key-value. If the key-value is not contained in the unordered\_map, then the operator inserts a new element into the map, with a *default-constructed mapped-value*.

mapped\_type& operator[] (const key\_type& key);

mapped\_type& operator[] (key\_type&& key);

#### at

Returns the mapped-value for a given key-value. If the key-value is not contained in the unordered\_map, the function throws an *out\_of\_range exception*.

mapped\_type& at(const key\_type& key);

const mapped\_type& at(const key\_type& key) const;

### Modifier Functions

#### clear

Erases the contents of the unordered\_map. Destructors are called for each object in the unordered\_map.

void clear() noexcept;

#### erase

Removes elements from an unordered\_map. Destructors are called for each object removed from the unordered\_map.

iterator erase(const\_iterator pos);

size\_type erase(const key\_type& key);

iterator erase(const\_iterator first, const\_iterator last);

#### insert

Inserts elements into an unordered\_map. unordered\_map elements must be unique, so duplicate values may not be inserted.

pair<iterator,bool> insert(const value\_type& value);

pair<iterator,bool> insert(value\_type&& value);

void insert(initializer\_list<value\_type> lst);

### Bucket Functions

#### bucket

Returns a bucket number for a given key value.

size\_type bucket (const key\_type& k) const;

#### bucket\_count

Returns the number of buckets in a unordered\_map

size\_type bucket\_count() const noexcept;

#### bucket\_size

Returns the number of elements in a given bucket.

size\_type bucket\_size(size\_type n) const;

### Example 15 – The unordered\_map container

1. #include <iostream>
2. #include <iomanip>
3. #include <unordered\_map>
4. #include <string>
5. #include <cstdlib>
6. using namespace std;
7. using hashUS = unordered\_map<unsigned,string>;
8. // prototypes
9. hashUS::iterator getInteratorForName(hashUS&, const string& name);
10. ostream& operator<<(ostream&, const hashUS&);
11. unsigned rand100();
12. int main()
13. {
14. hashUS students;
15. using US = pair<unsigned,string>;
16. students[rand100()] = "John";
17. students.insert(US(rand100(),"Paul"));
18. US george{rand100(),"George"};
19. students.insert(george);
20. auto ringo\_num = rand100();
21. US ringo{ringo\_num,"Ringo"};
22. students.insert(move(ringo));
23. cout << students << endl;
24. // What does this mean?
25. students[50];
26. cout << students << endl;
27. // Try to insert a new element using Ringo's number
28. students[ringo\_num] = "Ringo Clone";
29. cout << students << endl;
30. // What is John's number?
31. cout << "John's number is " <<
32. getInteratorForName(students,"John")->first << endl;
33. auto it = getInteratorForName(students,"maybe");
34. if (it == students.end())
35. cout << "maybe ain't there" << endl;
36. cout << "number of elements with key " << ringo\_num << " = "
37. << students.count(ringo\_num) << endl;
38. cout << "number of elements with key " << ringo\_num+1 << " = "
39. << students.count(ringo\_num+1) << endl << endl;
40. cout << "students.bucket\_count()=" << students.bucket\_count() << endl;
41. }
42. unsigned rand100()
43. {
44. return rand() % 100 + 1;
45. }
46. ostream& operator<<(ostream& out, const hashUS& obj)
47. {
48. out << left;
49. for (auto it = obj.begin(); it != obj.end(); ++it)
50. {
51. out << setw(5) << it->first << setw(10) << it->second << endl;
52. }
53. return out;
54. }
55. hashUS::iterator
56. getInteratorForName(hashUS& hash\_us, const string& name)
57. {
58. for (auto it = hash\_us.begin(); it != hash\_us.end(); ++it)
59. {
60. if (it->second == name)
61. return it;
62. }
63. return hash\_us.end();
64. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

30 Ringo

63 George

34 John

44 Paul

50

30 Ringo

63 George

34 John

44 Paul

50

30 Ringo Clone

63 George

34 John

44 Paul

John's number is 34

maybe ain't there

number of elements with key 30 = 1

number of elements with key 31 = 0

## unordered\_multimap

The unordered\_map container implements a multimap using a hash algorithm. This allows fast retrieval of the elements using the key value. Element values in a unordered\_multimap are pairs of key and mapped values. Unlike the unordered\_map container, element key values are not unique. This container was introduced in C++ 11. The unordered\_multimap container requires the <unordered\_map> header file.

### Member Functions

The unordered\_multimap constructors and member functions are essentially the same as the unordered\_map container. The following illustrates some of the differences.

#### erase

Erases elements in an unordered\_multimap

iterator erase(const\_iterator p);

Only a single element of the multimap is erased.

size\_t erase(const value\_type& value);

Erases all elements in the unordered\_multimap with a key equal to the specified value. The function returns the number of elements erased.

#### insert

iterator insert(const value\_type& val);

iterator insert(value\_type&& val);

This version of the insert function returns only an iterator to the element that was inserted. Unlike the unordered\_map::insert, there is no bool indication of success or failure. The unordered\_multimap::insert does not fail like the map::insert when duplicate key values are inserted.

#### count

Like the unordered\_map::count the function returns the number of elements that are equal to a value in the set. Since the elements in an unordered\_multimap are not unique, the count may be greater than 1.

size\_type count(const value\_type& value) const;

#### equal\_range

Returns a pair of iterators pointer to the first and last element that has a key value equal to the argument value in the unordered\_multimap. If no matches are found, the range returned has a length of zero, with both iterators pointing to the end of the unordered\_multimap.

pair<const\_iterator,const\_iterator> equal\_range(const value\_type& val) const;

pair<iterator,iterator> equal\_range(const value\_type& value);

### Example 16 – The unordered\_multimap container

1. #include <iostream>
2. #include <iomanip>
3. #include <unordered\_map>
4. #include <string>
5. #include <cstdlib>
6. using namespace std;
7. using Fraction = pair<int,int>;
8. ostream& operator<<(ostream& out, const Fraction& f)
9. {
10. out << f.first << '/' << f.second;
11. return out;
12. }
13. //function templates
14. template <typename F, typename S>
15. ostream& operator<<(ostream& out, const pair<F,S>& p)
16. {
17. out << "first: " << p.first << " second: " << p.second;
18. return out;
19. }
20. template <typename K, typename V>
21. ostream& operator<<(ostream& out, const unordered\_multimap<K,V>& m)
22. {
23. for (auto element : m) out << element << endl;
24. return out;
25. }
26. int main()
27. {
28. unordered\_multimap<double,Fraction> fractions;
29. fractions.insert(pair<double,Fraction>(.75,Fraction(3,4)));
30. fractions.insert(pair<double,Fraction>(.75,Fraction{6,8}));
31. Fraction neg\_3\_4{-3,-4};
32. fractions.insert(pair<double,Fraction>(.75,neg\_3\_4));
33. Fraction temp\_fraction;
34. pair<double,Fraction> temp\_doub\_fraction;
35. temp\_fraction = {1,2};
36. temp\_doub\_fraction = {.5,temp\_fraction};
37. fractions.insert(temp\_doub\_fraction);
38. fractions.insert({.5,{2,4}});
39. fractions.insert({.33,{1,3}});
40. fractions.insert({.25,{1,4}});
41. fractions.insert({.5,{1,2}});
42. cout << fractions << endl;
43. // fractions[.4] = fraction(2,5); // Error: no index operator
44. // find
45. unordered\_multimap<double,Fraction>::const\_iterator cIt;
46. cout << "fractions.find(.33): ";
47. cIt = fractions.find(.33);
48. cout << \*cIt << endl;
49. cout << "fractions.find(.75): " << \*fractions.find(.75) << endl;
50. cout << "fractions.find(.55): ";
51. cIt = fractions.find(.55);
52. // check to make sure find is OK
53. if (cIt == fractions.end())
54. cout << "Can't find .55" << endl << endl;
55. // count
56. cout << "fractions.count(.5)=" << fractions.count(.5) << endl;
57. cout << "fractions.count(.6)=" << fractions.count(.6) << endl
58. << endl;
59. // equal\_range
60. cout << "equal range(.5): " << endl;
61. auto iters = fractions.equal\_range(.5);
62. cout << \*(iters.first) << " / " << \*(iters.second) << endl;
63. for (auto iter = iters.first; iter != iters.second; ++iter)
64. cout << \*iter << endl;
65. cout << endl;
66. // erase
67. cout << "fractions.erase(.33) = " << fractions.erase(.33)<<endl;
68. cout << "fractions.erase(.5) = " << fractions.erase(.5) << endl;
69. cout << "fractions.erase(.55) = " << fractions.erase(.55)< endl
70. << endl;
71. cout << fractions << endl;
72. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

first: 0.25 second: 1/4

first: 0.33 second: 1/3

first: 0.5 second: 1/2

first: 0.5 second: 2/4

first: 0.5 second: 1/2

first: 0.75 second: -3/-4

first: 0.75 second: 6/8

first: 0.75 second: 3/4

fractions.find(.33): first: 0.33 second: 1/3

fractions.find(.75): first: 0.75 second: -3/-4

fractions.find(.55): Can't find .55

fractions.count(.5)=3

fractions.count(.6)=0

equal range(.5):

first: 0.5 second: 1/2 / first: 0.75 second: -3/-4

first: 0.5 second: 1/2

first: 0.5 second: 2/4

first: 0.5 second: 1/2

fractions.erase(.33) = 1

fractions.erase(.5) = 3

fractions.erase(.55) = 0

first: 0.25 second: 1/4

first: 0.75 second: -3/-4

first: 0.75 second: 6/8

first: 0.75 second: 3/4

## bitset

A bitset is a class that is used to store bits (binary digits). It is a templatized class in which the template parameter is the size of the sequence or array of bits. bitset is not a true STL container, since it is not templatized on a type, but it is part of the STL. Unlike the STL containers, it does not support iteration. Use of bitset requires the <bitset> header file.

### Constructors

default constructor

constexpr bitset() noexcept;

integer constructor

constexpr bitset (unsigned long long val) noexcept;

string constructor

explicit bitset(const string& str);[[9]](#footnote-9)

c-string constructor

explicit bitset(const char\* str);[[10]](#footnote-10)

### Bit Operation Functions

#### set

Sets bits to 1

bitset& set() noexcept;

sets all bits to 1

bitset& set(size\_t pos, bool val = true);

sets a single bit to 1 or 0

#### flip

flips bits

bitset& flip() noexcept;

flips all bits

bitset& flip(size\_t pos);

flips a single bit

#### reset

resets bits to 0

bitset& reset() noexcept;

resets all bits

bitset& reset(size\_t pos);

resets a single bit

### Bit Access Functions

#### all

Test all bits are set (equal to 1)

bool all() const noexcept;

#### any

Test to see if any bits are set

bool any() const noexcept;

#### none

Test to see if no bits are set

bool none() const noexcept;

#### count

Returns the number of bits that are set

size\_t count() const noexcept;

#### size

Returns the number of bits in the bitset

constexpr size\_t size() noexcept;

#### test

Tests to see if a bit is set

bool test (size\_t pos) const;

### Conversion Functions

#### to\_string

Returns the bitset as a string

string to\_string() const;[[11]](#footnote-11)

#### to\_ulong

Returns the bitset as an unsigned long

unsigned long to\_ulong() const;

#### to\_ullong

Returns the bitset as an unsigned long long

unsigned long long to\_ullong() const;

### Bitset operators

#### Member Functions

#### operator[] index operator

returns the bit value at a position in the bitset

bool operator[](size\_t pos) const;

reference operator[](size\_t pos);

##### Bitwise Operators

bitset& operator&=(const bitset& rhs) noexcept;

bitset& operator|=(const bitset& rhs) noexcept;

bitset& operator^=(const bitset& rhs) noexcept;

bitset& operator<<=(size\_t pos) noexcept;

bitset& operator>>=(size\_t pos) noexcept;

bitset operator~() const noexcept;

bitset operator<<(size\_t pos) const noexcept;

bitset operator>>(size\_t pos) const noexcept;

bool operator== (const bitset& rhs) const noexcept;

bool operator!= (const bitset& rhs) const noexcept;

#### Non-Member Functions

template<size\_t N>

bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

template<size\_t N>

bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

template<size\_t N>

bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

template<class charT, class traits, size\_t N>

istream& operator>>(istream& is, bitset<N>& rhs);

template<class charT, class traits, size\_t N>

ostream& operator<<(ostream& os, const bitset<N>& rhs);

### Example 17 – bitset

1. #include <iostream>
2. #include <bitset>
3. using namespace std;
4. int main()
5. {
6. // Constructor
7. bitset<8> b1;
8. bitset<16> b2(1234);
9. bitset<8> b3("1010");
10. string tenten("1010");
11. bitset<8> b4(tenten);
12. cout << "b1 = " << b1 << endl;
13. cout << "b2 = " << b2 << endl;
14. cout << "b3 = " << b3 << endl;
15. cout << "b4 = " << b4 << endl << endl;
16. // set
17. b1.set();
18. b2.set(15);
19. cout << "b1 = " << b1 << endl;
20. cout << "b2 = " << b2 << endl << endl;
21. // reset, flip
22. b1.reset();
23. b2.flip();
24. b3.flip(0);
25. cout << "b1 = " << b1 << endl;
26. cout << "b2 = " << b2 << endl;
27. cout << "b3 = " << b3 << endl << endl;
28. // all, any, none, count, size, test
29. cout << "b2.all() = " << b2.all() << endl;
30. cout << "b2.any() = " << b2.any() << endl;
31. cout << "b2.none() = " << b2.none() << endl;
32. cout << "b2.count() = " << b2.count() << endl;
33. cout << "b2.size() = " << b2.size() << endl;
34. cout << "b2.test(5) = " << b2.test(5) << endl << endl;
35. // to\_string, to ulong
36. cout << "b3.to\_string() = " << b3.to\_string() << endl;
37. cout << "b3.to\_ulong() = " << b3.to\_ulong() << endl << endl;
38. // index operator
39. b1[7] = 1;
40. cout << b1[6] << ' ' << b1 << ' ' << b1.to\_ulong() << endl
41. << endl;
42. cout << "b1 = " << b1 << endl;
43. cout << "b3 = " << b3 << endl;
44. cout << "b4 = " << b4 << endl << endl;
45. // bitwise operators
46. cout << "b1 | b3 = " << (b1 | b3) << endl;
47. cout << "b3 & b4 = " << (b3 & b4) << endl;
48. cout << "b3 ^ b4 = " << (b3 ^ b4) << endl;
49. cout << "b3 << 2 = " << (b3 << 2) << endl;
50. cout << "~b3 = " << (~b3) << endl;
51. cout << "b1 |= b3 = " << (b1 |= b3) << endl;
52. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

b1 = 00000000

b2 = 0000010011010010

b3 = 00001010

b4 = 00001010

b1 = 11111111

b2 = 1000010011010010

b1 = 00000000

b2 = 0111101100101101

b3 = 00001011

b2.all() = 0

b2.any() = 1

b2.none() = 0

b2.count() = 10

b2.size() = 16

b2.test(5) = 1

b3.to\_string() = 00001011

b3.to\_ulong() = 11

0 10000000 128

b1 = 10000000

b3 = 00001011

b4 = 00001010

b1 | b3 = 10001011

b3 & b4 = 00001010

b3 ^ b4 = 00000001

b3 << 2 = 00101100

~b3 = 11110100

b1 |= b3 = 10001011

## STL Algorithms

The STL algorithms are function templates that can be applied to STL containers.

This section needs more description and a list of the algorithms.

### Example 18 – The algorithm example

1. // algorithm example
2. #include <iostream>
3. #include <algorithm>
4. #include <vector>
5. #include <list>
6. #include <deque>
7. #include <iterator>
8. using namespace std;
9. // function generator - void argument function returns container type
10. int RandomNumber ()
11. {
12. return (rand()%100);
13. }
14. // binary function that returns a bool
15. bool funnyLessThan(const int& a, const int& b)
16. {
17. return a % 10 < b % 10;
18. }
19. bool lessthan10(int x)
20. {
21. return x < 10;
22. }
23. int main ()
24. {
25. vector<int> vec(20);
26. list<int> lst(20);
27. deque<int> deq(20);
28. // generate
29. generate(vec.begin(), vec.end(), RandomNumber);
30. // copy
31. copy(vec.begin(), vec.end(),lst.begin());
32. copy(vec.begin(), vec.end(),deq.begin());
33. cout << "The initial vector of random numbers\n";
34. copy(vec.begin(), vec.end(), ostream\_iterator<int>(cout," "));
35. cout << endl << endl;
36. // sort
37. sort(vec.begin(), vec.end());
38. sort(deq.begin(), deq.end());
39. // sort(lst.begin(), lst.end()); // Why doesn't this work?
40. cout << "The vector of random numbers after the first sort\n";
41. copy(vec.begin(), vec.end(), ostream\_iterator<int>(cout," "));
42. cout << endl << endl;
43. cout << "The deque of random numbers after the sort\n";
44. copy(deq.begin(), deq.end(), ostream\_iterator<int>(cout," "));
45. cout << endl << endl;
46. sort(vec.begin(), vec.end(),funnyLessThan);
47. cout << "The vector of random numbers after the second sort\n";
48. copy(vec.begin(), vec.end(), ostream\_iterator<int>(cout," "));
49. cout << endl << endl;
50. // count
51. cout << "count(vec.begin(), vec.end(),8) = " << count(vec.begin(), vec.end(),8) << endl;
52. cout << "count\_if(vec.begin(), vec.end(),lessthan10) = " << count\_if(vec.begin(), vec.end(),lessthan10) << endl << endl;
53. // the remove algorithm
54. string hand{"Have a nice day"};
55. remove(hand.begin(),hand.end(),'a');
56. cout << hand << endl;
57. hand = "Have a nice day";
58. string::iterator endit = remove(hand.begin(),hand.end(),'a');
59. hand.erase(endit,hand.end());
60. cout << hand << endl << endl;
61. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

The initial vector of random numbers

41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36

The vector of random numbers after the first sort

0 5 24 27 27 34 36 41 42 45 58 61 62 64 67 69 78 81 91 95

The deque of random numbers after the sort

0 5 24 27 27 34 36 41 42 45 58 61 62 64 67 69 78 81 91 95

The vector of random numbers after the second sort

0 91 81 41 61 42 62 24 34 64 5 95 45 36 67 27 27 58 78 69

Hve nice dyday

Hve nice dy

### Example 19 – The sort algorithm using compare function pointers, function objects and standard function objects

1. #include <iostream>
2. #include <iterator>
3. #include <algorithm>
4. #include <vector>
5. using namespace std;
6. ostream& operator<<(ostream& out, const vector<int>& v)
7. {
8. copy(v.cbegin(),v.cend(),ostream\_iterator<int>(out," "));
9. out << endl;
10. return out;
11. }
12. bool abs\_lt (int i,int j)
13. {
14. return abs(i) < abs(j);
15. }
16. class MyLessThan
17. {
18. public:
19. bool operator() (int i,int j)
20. {
21. return i < j;
22. }
23. };
24. int main()
25. {
26. int myints[] = {32,-71,12,45,-26,80,-53,33};
27. vector<int> myvector (myints, myints+8);
28. cout << "1) " << myvector << endl;
29. // using default comparison (operator <):
30. sort (myvector.begin(), myvector.begin()+4);
31. cout << "2) " << myvector << endl;
32. // using function as std compare function object
33. sort (myvector.begin(), myvector.end(), greater<int>());
34. cout << "3) " << myvector << endl;
35. // using function
36. sort (myvector.begin(), myvector.end(), abs\_lt);
37. cout << "4) " << myvector << endl;
38. // using function object (functor)
39. MyLessThan object;
40. sort (myvector.begin(), myvector.end(), object);
41. cout << "5) " << myvector << endl;
42. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

1) 32 -71 12 45 -26 80 -53 33

2) -71 12 32 45 -26 80 -53 33

3) 80 45 33 32 12 -26 -53 -71

4) 12 -26 32 33 45 -53 -71 80

5) -71 -53 -26 12 32 33 45 80

### Example 20 – The transform algorithm

1. #include <iostream>
2. #include <iterator>
3. #include <algorithm>
4. #include <string>
5. #include <vector>
6. #include <bitset>
7. using namespace std;
8. ostream& operator<<(ostream& out, const vector<char>& v)
9. {
10. copy(v.cbegin(),v.cend(),ostream\_iterator<char>(out," "));
11. out << endl;
12. return out;
13. }
14. char encode(char c)
15. {
16. bitset<8> ch(c);
17. ch.flip();
18. return static\_cast<char>(ch.to\_ulong());
19. }
20. int main()
21. {
22. string str("HAVE A NICE DAY");
23. vector<char> vc(str.size());
24. vector<char> vc2(str.size());
25. copy(str.cbegin(),str.cend(),vc.begin());
26. cout << vc << endl;
27. transform(vc.begin(),vc.end(),vc2.begin(),encode);
28. cout << vc2 << endl;
29. copy(vc2.begin(),vc2.end(),str.begin());
30. cout << str << endl;
31. transform(vc2.begin(),vc2.end(),vc.begin(),encode);
32. copy(vc.begin(),vc.end(),str.begin());
33. cout << str << endl;
34. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

H A V E A N I C E D A Y

╖ ╛ ⌐ ║ ▀ ╛ ▀ ▒ ╢ ╝ ║ ▀ ╗ ╛ ª

╖╛⌐║▀╛▀▒╢╝║▀╗╛ª

HAVE A NICE DAY

# Lambda Expressions / Functions

A lambda expression allows you to write an anonymous function. This function is used like an inline function. Here’s an easy example to get you started.

### Lambda Basics

### Example 1 – Easy Lambda example

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. auto hand = [](){cout << "Have a nice day\n";};
6. hand();
7. }

#### Explanation

[](){cout << "Have a nice day\n";} is the lambda expression. This expression returns a function. In the example the returned function is assigned to a variable, hand. The hand variable is declared as type auto. Type auto makes is easy so that you don’t have to determine the type of hand. In this case, the type is void (\*)(). So, you could replace line 6 with

void (\*hand)() = [](){cout << "Have a nice day\n";};

In this example the lambda expression consists of 3 parts

1. The capture list, []. In this case, nothing is captured. More about that later.
2. The lambda arguments, (). In this case, there are no arguments. More about that later.
3. The body of the lambda, between the { }. This is what the lambda does.

And, here, the lambda returns void.

So, hand is a function pointer, and it is called by adding the ().

### Example 2 – lambda capture and lambda arguments

1. #include <iostream>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6. string whatever = "kinda nice";
7. // capture variables (by value) in the same scope
8. auto havd = [=]()
9. {
10. cout << "Have a " << whatever <<" day\n";
11. };
12. havd();
13. // capture variables (by reference) in the same scope
14. auto hard = [&]()
15. {
16. whatever = "really nice";
17. cout << "Have a " << whatever <<" day\n";
18. };
19. hard();
20. cout << whatever << endl;
21. // pass a value to the lambda expression
22. auto argue = [](string arg)
23. {
24. cout << "Have a " << arg << " day\n";
25. };
26. argue(whatever);
27. argue("fun");
28. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Have a kinda nice day

Have a really nice day

really nice

Have a really nice day

Have a fun day

#### Explanation

The capture in line 10 is identified as [=]. This means that any variables in the same scope as the lambda expression are available in the lambda. In this case it is as if the variable whatever is passed by value.

The capture in line 17 is identified as [&]. This means that any variables in the same scope as the lambda expression are available in the lambda. In this case it is as if the variable whatever is passed by reference. Notice that whatever is changed in the lambda body.

Line 27 shows a lambda with an argument. This, like any other function argument, makes the argument available in the body of the lambda.

So, in the three cases in this example, the lambda expression creates a function pointer. This pointer is then assigned to an auto variable, and then with parentheses, the function may be called. In the third example, the function call had to provide an argument.

### Example 3 – captures, arguments, and returns

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int x = 8;
6. auto somefunk = [=](int arg)->int { return x + arg; };
7. cout << somefunk(7) << endl;
8. auto obviousreturntype = [](int arg1, int arg2)
9. {
10. return arg1 + arg2;
11. };
12. cout << obviousreturntype(13,4) << endl;
13. float f = 3.25;
14. double d = 2.0;
15. auto anotherfunk = [f,d]()
16. {
17. // f = 3.25; // Error, f is read-only
18. return f + d;
19. };
20. auto ret1 = anotherfunk();
21. cout << ret1 << ' ' << sizeof(ret1) << endl;
22. auto stillanotherfunk = [f,d]() -> float
23. {
24. // f = 3.25; // Error, f is read-only
25. return f + d;
26. };
27. auto ret2 = stillanotherfunk();
28. cout << ret2 << ' ' << sizeof(ret2) << endl;
29. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

15

17

5.25 8

5.25 4

#### Explanation

The lambda expression, on line 7, [=](int arg)->int { return x + arg; } captures in scope variables with [=], has an int argument and specifies an int return with ->int. The int return is optional, since the lambda expression would return an int anyway.

The second lambda, lines 10-13, returns a function pointer that requires two int arguments and assigns it to the auto variable obvious return type. The function pointer is then exercised on line 14.

The third lambda, lines 19-23, captures two local variables, f and d, by value. Note that line 21 is commented out, an error. This illustrates how capture values are different than lambda arguments. A lambda argument, passed by value, is a local copy of some other value and hence, modifiable, locally within the lambda body, and obviously not affecting the source. A capture value is not the same as a lambda argument. The capture, as specified by [=], or in this case [f,d] specifies that variables in the same scope are read only. The exception to this is when the capture is specified as [&], or [&f,&d]. In this case, the capture is by reference and those values are modifiable. This third lambda is used on line 25 and the return from the lambda inspired function is assigned to the auto variable ret1. This ret1 variable is demonstrated using sizeof to be type double.

The fourth lambda, lines 28-32, is the same as the third lambda, except that the return type is specified as float. Hence, the double result for f + d in line 31 is then converted to float. To match the lambda returned specification.

### Lambda and the STL

The return power of lambda expressions comes from their use with STL algorithms.

### Example 4 – lambda and STL algorithms

1. #include <vector>
2. #include <algorithm>
3. #include <iostream>
4. #include <cstdlib>
5. #include <climits> // for INT\_MIN
6. using namespace std;
7. int main()
8. {
9. vector<int> vec = {1,4,5,8,9,2,6,4,32,7,19};
10. // print the vector
11. auto printv = [](int i)
12. {
13. cout << i << " ";
14. };
15. for\_each(vec.begin(),vec.end(), printv);
16. cout << endl;
17. // find the maximum value in the vector
18. int max = INT\_MIN;
19. for\_each(vec.begin(),vec.end(),
20. [&max](int i)
21. {
22. if (i > max) max = i;
23. });
24. cout << "The maximum value is " << max << endl;
25. // sort the vector
26. sort(vec.begin(),vec.end(),
27. [](const int& i, const int& j)
28. {
29. return i < j;
30. });
31. for\_each(vec.begin(),vec.end(), printv);
32. cout << endl;
33. // how many vector values are greater than 10
34. cout << "The are " <<
35. count\_if(vec.begin(), vec.end(),[](int i)
36. {
37. return i > 10;
38. })
39. << " values greater than 10" << endl;
40. generate(vec.begin(),vec.end(),[] { return rand() % 100;});
41. for\_each(vec.begin(),vec.end(), printv);
42. cout << endl;
43. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

1 4 5 8 9 2 6 4 32 7 19

The maximum value is 32

1 2 4 4 5 6 7 8 9 19 32

The are 2 values greater than 10

1. 67 34 0 69 24 78 58 62 64 5

#### Explanation

The first lambda expression, lines 12 -15, is used to display an int. This expression is assigned to the function pointer, printv. That function pointer is then used as the third argument of the for\_each algorithm on line 16.

The second lambda expression, lines 22-25, is similarly used as the third argument of the for\_each algorithm. In this case, the lambda expression is placed directly *inline* as the third argument.

The third lambda expression, lines 30-33, is the third argument of the sort algorithm.

The fourth lambda expression, on line 45, returns a function pointer of a function that returns a random int.

### Example 5 – lambda and function templates

1. #include <vector>
2. #include <algorithm>
3. #include <iostream>
4. #include <iomanip>
5. using namespace std;
6. template<typename T>
7. void printvector(vector<T>& v)
8. {
9. for\_each(v.begin(),v.end(), [](T element)
10. {
11. cout << element << " ";
12. });
13. cout << endl;
14. }
15. // Generic overloaded insertion operator for a vector
16. template<typename T>
17. ostream& operator<<(ostream& out, const vector<T>& v)
18. {
19. for\_each(v.begin(),v.end(), [&out](T element)
20. {
21. out << element << " ";
22. });
23. out << endl;
24. return out;
25. }
26. class Money
27. {
28. unsigned dollars, cents;
29. public:
30. Money(unsigned d, unsigned c)
31. : dollars(d + c/100), cents(c%100) {}
32. friend ostream& operator<<(ostream& out, const Money& m)
33. {
34. out << setfill('0');
35. out << '$' << m.dollars << '.' << setw(2) << m.cents;
36. out << setfill(' ');
37. return out;
38. }
39. };
40. int main()
41. {
42. vector<int> vec1 = {1,4,5,8,9,2,6,4,32,7,19};
43. vector<double> vec2 = {1.4,5.8,9.2,6.4,32.7,19};
44. vector<Money> vec3 = {{12,34},{56,78},{910,1112}};
45. printvector(vec1);
46. printvector(vec2);
47. printvector(vec3);
48. cout << endl;
49. cout << vec1;
50. cout << vec2;
51. cout << vec3;
52. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

1 4 5 8 9 2 6 4 32 7 19

1.4 5.8 9.2 6.4 32.7 19

$12.34 $56.78 $921.12

1 4 5 8 9 2 6 4 32 7 19

1.4 5.8 9.2 6.4 32.7 19

$12.34 $56.78 $921.12

# Smart Pointers

Smart pointers are used to manage dynamically allocated memory. Their use will help to avoid memory leaks, calling delete on the same pointer address twice, and assist in avoiding segmentation faults in dereferencing a null pointer. You can think of a smart pointer as a wrapper for a pointer. It is an object stored in stack memory that *owns* a pointer. The obvious advantage is that when the stack memory object goes out of scope its destructor executes and automatically releases dynamically stored memory. There are two primary template classes used for this purpose, unique\_ptr and shared\_ptr. Both of these were introduced in C++11. Prior to that the auto\_ptr template was used for this. The auto\_ptr template was deprecated in C++11.

## unique\_ptr

A unique\_ptr is a smart pointer in which a pointer is uniquely owned by one unique\_pointer. The unique\_ptr template requires the <memory> header file.

### Example 1 – unique\_ptr example

1. #include <iostream>
2. #include <memory>
3. #include <vector>
4. #include <deque>
5. #include <iterator>
6. using namespace std;
7. class SomeClass
8. {
9. int data\_;
10. public:
11. SomeClass(int arg = 0) : data\_(arg)
12. {
13. cout << "SomeClass ctor called: address=" << this << endl;
14. }
15. ~SomeClass()
16. {
17. cout << "SomeClass dtor called address=" << this << endl;
18. }
19. int data() const
20. {
21. return data\_;
22. }
23. int& data()
24. {
25. return data\_;
26. }
27. };
28. int main ()
29. {
30. unique\_ptr<int> up1(new int(6));
31. cout << "\*up1=" << \*up1 << endl << endl;
32. // unique\_ptr<int> up2 = new int(7); // Error
33. unique\_ptr<int> up2;
34. // up2 = new int; // Error assignment operator does not take pointer argument, except ..
35. up2 = nullptr;
36. up2 = make\_unique<int>(5); // requires C++14
37. cout << "\*up2=" << \*up2 << endl;
38. cout << "up2.get()=" << up2.get() << endl;
39. cout << "\*up2.get()=" << \*up2.get() << endl << endl;
40. // If you don't have C++14
41. unique\_ptr<int> up3 = unique\_ptr<int>(new int(4));
42. cout << "\*up3=" << \*up3 << endl << endl;
43. // unique\_ptrs with class
44. auto upS1 = make\_unique<SomeClass>(7);
45. cout << "upS1->data()=" << upS1->data() << endl;
46. upS1->data() \*= 3;
47. cout << "upS1->data()=" << upS1->data() << endl << endl;
48. // unique\_ptr with STL container
49. auto upV = make\_unique<vector<int>>(); // parentheses required
50. upV -> push\_back(1);
51. upV -> push\_back(2);
52. upV -> push\_back(3);
53. copy(upV->begin(), upV->end(),ostream\_iterator<int>(cout," "));
54. cout << endl << endl;
55. deque<int> di={3,4,5,6,7};
56. auto upDi = make\_unique<deque<int>>(di);
57. (\*upDi)[2] = 77;
58. for (auto value : \*upDi) cout << value << ' ';
59. cout << endl << endl;
60. // release
61. cout << "up1.get()=" << up1.get() << endl;
62. auto ptr4up1 = up1.get();
63. cout << "ptr4up1=" << ptr4up1 << endl;
64. up1.release(); // Watch out for the leak!
65. cout << "up1.get()=" << up1.get() << endl;
66. cout << "\*ptr4up1=" << \*ptr4up1 << endl;
67. delete ptr4up1;
68. ptr4up1 = nullptr;
69. cout << endl;
70. // reset
71. unique\_ptr<int> up4(new int(4));
72. cout << "up4.get()=" << up4.get() << endl;
73. up4.reset();
74. cout << "up4.get()=" << up4.get() << endl;
75. up4 = make\_unique<int>(44);
76. cout << "up4.get()=" << up4.get() << endl;
77. cout << "\*up4=" << \*up4 << endl;
78. up4.reset(new int(444));
79. cout << "up4.get()=" << up4.get() << endl;
80. cout << "\*up4=" << \*up4 << endl << endl;
81. auto upS2 = make\_unique<SomeClass>(77);
82. cout << "upS2->data()=" << upS2->data() << endl;
83. upS2.reset();
84. cout << endl;
85. cout << "That's all folks!!!" << endl;
86. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

\*up1=6

\*up2=5

up2.get()=0x8000128d0

\*up2.get()=5

\*up3=4

SomeClass ctor called: address=0x800012910

upS1->data()=7

upS1->data()=21

1 2 3

3 4 77 6 7

up1.get()=0x800000400

ptr4up1=0x800000400

up1.get()=0

\*ptr4up1=6

up4.get()=0x800000400

up4.get()=0

up4.get()=0x800000400

\*up4=44

up4.get()=0x800012970

\*up4=444

SomeClass ctor called: address=0x800000400

upS2->data()=77

SomeClass dtor called address=0x800000400

That's all folks!!!

SomeClass dtor called address=0x800012910

## shared\_ptr

A shared\_ptr is a smart pointer that is used to manage multiple pointer to the same memory location. The shared\_ptr interface is similar to the unique\_ptr. It is commonly used in reference counting application.

### Example 2 – shared\_ptr example

1. #include <iostream>
2. #include <iomanip>
3. #include <string>
4. #include <memory>
5. #include <vector>
6. using namespace std;
7. class Demo
8. {
9. public:
10. Demo()
11. {
12. cout << "default Demo ctor: " << this << endl;
13. }
14. Demo(const Demo&)
15. {
16. cout << "copy Demo ctor: " << this << endl;
17. }
18. ~Demo()
19. {
20. cout << "Demo dtor: " << this << endl;
21. }
22. };
23. ostream& operator<<(ostream& out, const Demo&)
24. {
25. out << "Demo object";
26. return out;
27. }
28. template <typename T>
29. ostream& operator<<(ostream& out, const shared\_ptr<T>& obj);
30. int main()
31. {
32. shared\_ptr<string> sp1;
33. shared\_ptr<string> sp2(nullptr);
34. shared\_ptr<string> sp3(new string("carrot"));
35. shared\_ptr<string> sp4(make\_shared<string>("potato"));
36. shared\_ptr<string> sp5(sp3);
37. cout << "sp1: " << sp1 << endl;
38. cout << "sp2: " << sp2 << endl;
39. cout << "sp3: " << sp3 << endl;
40. cout << "sp4: " << sp4 << endl;
41. cout << "sp5: " << sp5 << endl << endl;
42. cout << "sp1 = sp4;" << endl;
43. sp1 = sp4;
44. cout << "sp1: " << sp1 << endl;
45. cout << "sp4: " << sp4 << endl << endl;
46. cout << "sp2 = sp3;" << endl;
47. sp2 = sp3;
48. cout << "sp2: " << sp2 << endl;
49. cout << "sp3: " << sp3 << endl << endl;
50. cout << "sp1.reset();" << endl;
51. sp1.reset();
52. cout << "sp1: " << sp1 << endl << endl;
53. shared\_ptr<Demo> sp6(nullptr); // create "empty" shared pointer
54. shared\_ptr<Demo> sp7(new Demo); // calls Demo default ctor
55. shared\_ptr<Demo> sp8(new Demo(\*sp7)); // calls Demo copy ctor
56. shared\_ptr<Demo> sp9(make\_shared<Demo>()); // Demo default ctor
57. shared\_ptr<Demo> sp10(sp7); // calls shared\_ptr copy ctor
58. cout << "sp6: " << sp6 << endl;
59. cout << "sp7: " << sp7 << endl;
60. cout << "sp8: " << sp8 << endl;
61. cout << "sp9: " << sp9 << endl;
62. cout << "sp10:" << sp10 << endl << endl;
63. cout << "sp6 = move(sp7);" << endl;
64. sp6 = move(sp7);
65. cout << "sp6: " << sp6 << endl;
66. cout << "sp7: " << sp7 << endl << endl;
67. cout << "sp6.reset();" << endl;
68. sp6.reset();
69. cout << "sp6: " << sp6 << endl;
70. cout << "sp10: " << sp10 << endl << endl;
71. cout << "sp10.reset();" << endl;
72. sp10.reset();
73. cout << "sp6: " << sp6 << endl;
74. cout << "sp7: " << sp7 << endl;
75. cout << "sp8: " << sp8 << endl;
76. cout << "sp9: " << sp9 << endl;
77. cout << "sp10:" << sp10 << endl << endl;
78. cout << "That's all folks" << endl;
79. }
80. template <typename T>
81. ostream& operator<<(ostream& out, const shared\_ptr<T>& obj)
82. {
83. if (obj.get())
84. out << setw(10) << obj.get() << " " << setw(8) << \*obj
85. << " " << obj.use\_count();
86. else
87. out << setw(10) << obj.get();
88. return out;
89. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

sp1: 0

sp2: 0

sp3: 0x800000400 carrot 2

sp4: 0x8000128e0 potato 1

sp5: 0x800000400 carrot 2

sp1 = sp4;

sp1: 0x8000128e0 potato 2

sp4: 0x8000128e0 potato 2

sp2 = sp3;

sp2: 0x800000400 carrot 3

sp3: 0x800000400 carrot 3

sp1.reset();

sp1: 0

default Demo ctor: 0x800012970

copy Demo ctor: 0x8000129b0

default Demo ctor: 0x800012a00

sp6: 0

sp7: 0x800012970 Demo object 2

sp8: 0x8000129b0 Demo object 1

sp9: 0x800012a00 Demo object 1

sp10:0x800012970 Demo object 2

sp6 = move(sp7);

sp6: 0x800012970 Demo object 2

sp7: 0

sp6.reset();

sp6: 0

sp10: 0x800012970 Demo object 1

sp10.reset();

Demo dtor: 0x800012970

sp6: 0

sp7: 0

sp8: 0x8000129b0 Demo object 1

sp9: 0x800012a00 Demo object 1

sp10: 0

That's all folks

Demo dtor: 0x800012a00

Demo dtor: 0x8000129b0

### Example 3 – shared\_ptr solution for CIS22B/Assignment 9

The following example demonstrates a solution for a CIS22B assignment. This is the description of the assignment:

## Assignment 9 - Reference Counting and a Linked List

The assignment will give you practice writing constructors and destructors, overloaded operator functions, and implementing a linked list.  You will also employ a technique called reference counting.

### The Plan

The goal of the assignment is to track a list of various (fruit) "items".  You will read and process a transaction file (partially displayed below).  The transaction file contains 5 types of transactions.  You are to store a count of the items in a sorted linked list.

### Details

The transaction file contains slightly over 100 random transaction entries.  The five transaction type entries are:

1. ***add*** <item> - add the item to the inventory, or increase the count for that item
2. ***remove*** <item> - remove the item from the inventory, or decrease the count for that item.  If the item does not exist, print error message.
3. ***print inventory*** - print the contents of the linked list (in sorted order) as shown below
4. ***misspelled transactions*** (add, remove, or print may be misspelled) - print an error message, including the line number in the file
5. ***blank lines***- skip over these (but count the lines)

### Program Requirements

1. You must write your own linked list.  You may not use any STL containers.
2. The linked list**must be maintained in sorted (alphabetical) order** by the item.
3. The linked list node must contain the item name (fruit name) and a count of the number of that item that are added to the list..
4. You must print out the contents of the linked list when a "print list" transaction record appears.  See sample output below.
5. You must write at least 2 classes, a "node" class and a "linked list" class.  Both classes must contain constructors and the "linked list" class must have a destructor.
6. You must include at least two overloaded operators as member functions.
7. The print function of your "linked list" class must be implemented as an overloaded insertion operator function.

### Input File

This is the first 32 records of the input file.

|  |
| --- |
| add banana add pear add orange  add orange add apple  add peach add plum ad plum  remove apple add watermelon add pear add plum reomve banana remove pear add apple remove orange remove plum add watermelon …  remove potato  add banana add papaya remove watermelon print list remove banana remove watermelon ... |

### Partial Program Output

|  |
| --- |
| Bad transaction: ad in line #10 Bad transaction: reomve in line #16 Unable to remove potato in line #26  Item       Quantity apple          1 banana         2 orange         1 papaya         3 peach          1 watermelon     1  Bad transaction: prlnt in line #50  Item       Quantity apple          2 apricot        2 banana         7 orange         1 papaya         4 ... |

This output shows the contents of the linked list after the first ***print list***transaction (plus a few more lines).

The solution below uses a forward\_list (container) of shared pointers. The solution produces the same output that is required in the CIS22B assignment. The assignment description and input file can be found here => http://voyager.deanza.edu/~bentley/cis22b/ass9.html

1. #include <forward\_list>
2. #include <cstdlib>
3. #include <fstream>
4. #include <iostream>
5. #include <iomanip>
6. #include <algorithm>
7. #include <memory>
8. using namespace std;
9. void processTransactions(const string& filename,
10. forward\_list<shared\_ptr<string>>&fwdlist);
11. shared\_ptr<string> find(forward\_list<shared\_ptr<string>>&fwdlist,
12. const string& str);
13. bool remove(forward\_list<shared\_ptr<string>>&fwdlist,
14. const string& str);
15. ostream& operator<<(ostream& out,
16. const forward\_list<shared\_ptr<string>>&lst);
17. ostream& operator<<(ostream& out, const shared\_ptr<string>& obj);
18. int main()
19. {
20. forward\_list<shared\_ptr < string>> fruit;
21. processTransactions("c:/temp/ass9data.txt", fruit);
22. }
23. void processTransactions(const string& filename,
24. forward\_list<shared\_ptr<string>>&fwdlist)
25. {
26. ifstream fin(filename);
27. if (!fin)
28. {
29. cerr << "Unable to open file " << filename << endl;
30. exit(1);
31. }
32. string buffer, transaction, dummy, numberString;
33. string item;
34. int lineNumber = 0;
35. size\_t pos;
36. while (!fin.eof())
37. {
38. lineNumber++;
39. getline(fin, buffer);
40. if (fin.eof())
41. break; // EOF check
42. // A gnu/Mac compiler may store \r in the last byte.
43. pos = buffer.find('\r');
44. if (pos != string::npos)
45. buffer.erase(pos);
46. if (buffer.size() < 1)
47. continue; // skip over blank line
48. // get the first word of the line
49. pos = buffer.find(' ');
50. transaction = buffer.substr(0, pos);
51. // for add or remove, get item
52. if (transaction == "add" or transaction == "remove")
53. item = buffer.substr(pos + 1);
54. if (transaction == "add")
55. {
56. // Create a shared ptr for the item
57. auto sharedPtr = find(fwdlist, item);
58. if (!sharedPtr)
59. sharedPtr = make\_shared<string>(item);
60. // Case 1: fwdlist is empty?
61. if (fwdlist.empty())
62. {
63. fwdlist.push\_front(sharedPtr);
64. }
65. // Case 2: item inserted at beginning of fwdlist?
66. else if (item <= \*(fwdlist.front()))
67. {
68. fwdlist.push\_front(sharedPtr);
69. }
70. // Case 3: item inserted in fwdlist containing one item
71. else if (++(fwdlist.begin()) == fwdlist.end())
72. {
73. fwdlist.insert\_after(fwdlist.begin(), sharedPtr);
74. }
75. // Case 4: fwdlist containing more than one item
76. else
77. {
78. // find the location to insert the new node
79. auto it = fwdlist.begin();
80. auto prev = fwdlist.before\_begin();
81. while (it != fwdlist.end() && \*\*it < item)
82. {
83. prev = it;
84. ++it;
85. }
86. fwdlist.insert\_after(prev, sharedPtr);
87. }
88. }
89. else if (transaction == "remove")
90. {
91. if (!remove(fwdlist, item))
92. cerr << "Unable to remove " << item
93. << " in line #" << lineNumber << endl;
94. }
95. else if (transaction == "print")
96. {
97. cout << fwdlist << endl;
98. }
99. else
100. {
101. cout << "Bad transaction: " << transaction
102. << " in line #" << lineNumber << endl;
103. }
104. }
105. fin.close();
106. }
107. shared\_ptr<string>
108. find(forward\_list<shared\_ptr<string>>&fwdlist, const string& str)
109. {
110. for (auto it = fwdlist.cbegin(); it != fwdlist.cend(); ++it)
111. {
112. if (\*\*it == str)
113. return \*it;
114. }
115. return nullptr;
116. }
117. bool remove(forward\_list<shared\_ptr<string>>&fwdlist,
118. const string& str)
119. {
120. for (auto it = fwdlist.begin(); it != fwdlist.end(); ++it)
121. {
122. if (\*\*it == str)
123. {
124. it->reset();
125. // if shared pointer count is 0, remove node
126. if (it->use\_count() == 0)
127. fwdlist.remove(\*it);
128. return true;
129. }
130. }
131. return false;
132. }
133. ostream& operator<<(ostream& out,
134. const forward\_list<shared\_ptr <string>>&fwdlist)
135. {
136. out << endl << "Item Quantity" << endl;
137. out << left;
138. shared\_ptr<string> prev\_shared\_ptr = nullptr;
139. for (auto it = fwdlist.cbegin(); it != fwdlist.cend(); ++it)
140. {
141. if (\*it && prev\_shared\_ptr != \*it)
142. out << \*it << endl;
143. prev\_shared\_ptr = \*it;
144. }
145. return out;
146. }
147. ostream& operator<<(ostream& out, const shared\_ptr<string>& obj)
148. {
149. out << left << setw(12) << \*obj;
150. out << right << setw(4) << obj.use\_count();
151. return out;
152. }

\*\*\*\*\*\* Output \*\*\*\*\*\*

Bad transaction: ad in line #10

Bad transaction: reomve in line #16

Unable to remove potato in line #26

Item Quantity

apple 1

banana 2

orange 1

papaya 3

peach 1

watermelon 1

Bad transaction: prlnt in line #50

Item Quantity

apple 2

apricot 2

banana 7

orange 1

papaya 4

peach 2

plum 1

tangarine 1

Bad transaction: aad in line #62

Unable to remove cabbage in line #81

Item Quantity

apple 2

apricot 2

banana 7

orange 4

papaya 5

peach 5

…

# Programming Style

1. streamsize is used to represent size and character counts. It is a signed integer type. [↑](#footnote-ref-1)
2. streampos is used to represent position in a stream. This type is an integer construction or conversion. [↑](#footnote-ref-2)
3. streamoff is used to represents an offset of a position in a stream. [↑](#footnote-ref-3)
4. The noexcept specification means the function will not throw any exceptions. [↑](#footnote-ref-4)
5. A binary predicate is a function object that requires two arguments and returns a bool. [↑](#footnote-ref-5)
6. A bucket iterator allows you to iterate through buckets instead of individual elements [↑](#footnote-ref-6)
7. A bucket iterator allows you to iterate through buckets instead of individual elements [↑](#footnote-ref-7)
8. A bucket iterator allows you to iterate through buckets instead of individual elements [↑](#footnote-ref-8)
9. This constructor syntax is an abstraction [↑](#footnote-ref-9)
10. This constructor syntax is an abstraction [↑](#footnote-ref-10)
11. This prototype is an abstraction [↑](#footnote-ref-11)