CS100 Recitation 11

GKxx

May 2, 2022

Contents

What have we learnt?

1. Getting Started	
2. Variables and Basic Types	
3. Strings, Vectors, and Arrays	std::string, std::vector, iterators
4. Expressions	
5. Statements	Exception handling (try-catch, throw)
6. Functions	
7. Classes	
8. The IO Library	fstream and stringstream
9. Sequential Containers	
10. Generic Algorithms	
11. Associative Containers	
12. Dynamic Memory	allocator and smart pointers
13. Copy Control	
14. Overloaded Operations and Conversions	
15. Object-Oriented Programming	
16. Templates and Generic Programming	
17. Specialized Library Facilities	
18. Tools for Large Programs	
19. Specialized Tools and Techniques	

A Federation of 4 Languages

Effective C++ Item 1: View C++ as a federation of languages.

- ☑ C
- ☑ Object-Oriented C++
- ☐ Template C++
- ☐ The STL

Operator Overloading

- At least one class-type parameter.
- Cannot change the **precedence** or the **associativity**.

Operators that may be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

Operator Overloading

Operators that may not be overloaded:

```
::   .   .*   ?:
```

Overloaded operator is a function:

- A special name: the `operator` keyword followed by the symbol of the operator.
- Non-member function: Operands are the parameters from left to right.
- Member function: The leftmost operand is implicitly bound to `this`. Other operands are the parameters from left to right.

Operator Overloading

Operators that may not be overloaded:

```
::   .   .*   ?:
```

Overloaded operator is a function:

- A special name: the `operator` keyword followed by the symbol of the operator.
- Non-member function: Operands are the parameters from left to right.
- Member function: The leftmost operand is implicitly bound to `this`. Other operands are the parameters from left to right.

More Effective C++ Item 7 says that never overload operator `&&`, `||` and `,`. Why?

Operator Overloading

We have seen that

- The IO library overloads `operator<<` and `operator>>`.
- The string library overloads `operator+` and `operator[]`.
  - Why won't '`"ABC" + "DEF"`' compile?

Contents

`iostream`, `cin` and `cout`

- `std::cin`: object of type `std::istream`.
- `std::cout`: object of type `std::ostream`.
- `std::istream` and `std::ostream` are **uncopyable** types.

`iostream`, `cin` and `cout`

- `std::cin`: object of type `std::istream`.
- `std::cout`: object of type `std::ostream`.
- `std::istream` and `std::ostream` are **uncopyable** types.
- Outputs can be chained together as in '`cout << a << b`'. Why?

iostream, cin and cout

- `std::cin`: object of type `std::istream`.
- `std::cout`: object of type `std::ostream`.
- `std::istream` and `std::ostream` are **uncopyable** types.
- Outputs can be chained together as in '`cout << a << b`'. Why?

```
inline std::ostream &operator<<
        (std::ostream &os, const Point2d &p) {
  os << "(" << p.get_x() << ", " << p.get_y() << ")";
  return os;
}
```

Test the State of `iostream`

On input failure, no error would be thrown, but we can test this by using the stream object as a condition.

```
struct Vector2d {
  double x, y, norm_l2;
};
inline std::istream &operator>>
        (std::istream &is, Vector2d &v) {
  is >> v.x >> v.y;
  // On input failure, set the object to a valid state.
  if (is)
    v.norm_l2 = std::sqrt(v.x * v.x + v.y * v.y);
  else
    v = Vector2d{};
  return is;
}
```

Examples

Read an unknown number of integers?

```
std::vector<int> v;
int x;
while (std::cin >> x)
  v.push_back(x);
```

Examples

Read an unknown number of integers?

```
std::vector<int> v;
int x;
while (std::cin >> x)
  v.push_back(x);
```

Read a line as a string?

```
std::string line;
std::getline(std::cin, line);
```

Examples

Read an unknown number of integers?

```
std::vector<int> v;
int x;
while (std::cin >> x)
  v.push_back(x);
```

Read a line as a string?

```
std::string line;
std::getline(std::cin, line);
```

- `std::getline` reads until the first newline character (‘\n’), and throws away that newline character.
- What happens?

```
int n; std::cin >> n;
std::string line;
std::getline(std::cin, line);
```

Manipulators

`endl`, `flush` and the like are **manipulators**.

- `endl` outputs a newline character and flushes the buffer.
- `flush` only flushes the buffer.

More manipulators: (some defined in `<iomanip>`)

- `boolalpha`, `noboolalpha`
- `oct`, `hex`, `dec`, `showbase`, `noshowbase`, `setbase`
- `fixed`, `setprecision`, `scientific`
- ……

C++ Primer 17.5

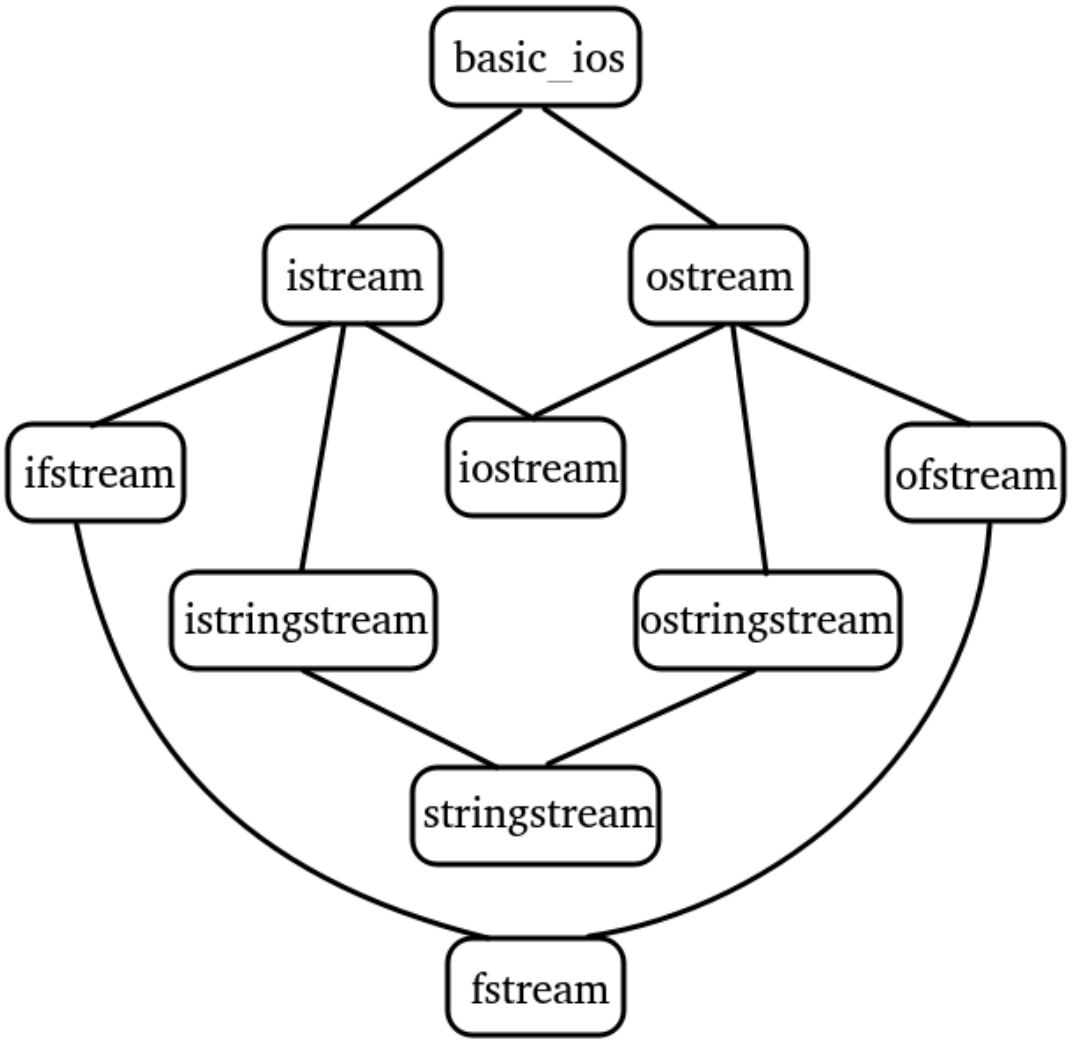
Contents

File Streams

Read an unknown number of integers from a file ‘student_score.txt’?

```
std::ifstream infile("student_score.txt");
// Equivalent way:
// std::ifstream infile;
// infile.open("student_score.txt");
std::vector<int> score;
int x;
while (infile >> x)
  score.push_back(x);
infile.close();
```

Inheritance



- Multiple inheritance
- Virtual inheritance
- What can we know from this?

Real World Example

Read a '.tex' file. Change math from '$...$' to '\(...\)'.

```
std::ifstream infile("hw3.tex");
std::ofstream result("result.tex");
bool in_math = false;
std::string line;
while (std::getline(infile, line)) {
  // process the line
}
infile.close();
result.close();
```

File Modes

Append something to a file, instead of overwriting it?

```
std::ofstream out_file("name.txt", std::ofstream::app);
```

File Modes

Append something to a file, instead of overwriting it?

```
std::ofstream out_file("name.txt", std::ofstream::app);
```

`in`	Open for input
`out`	Open for output
`app`	Seek to the end before every write
`ate`	Seek to the end immediately after the open
`trunc`	Truncate the file
`binary`	Do IO operations in binary mode

- *C++ Primer* 8.2.2

Contents

Stringstreams

Read data from a string, or generate a string by writing different kinds of data.

```
struct Person_info {
  std::string name;
  std::vector<std::string> phones;
};
std::string line;
std::vector<Person_info> people;
while (std::getline(std::cin, line)) {
  Person_info info;
  std::istringstream record(line);
  record >> info.name;
  std::string phone;
  while (record >> phone)
    info.phones.push_back(phone);
  people.push_back(info);
}
```

Stringstreams

Convert some `double` or `int` to a string?

```
inline std::string convert(double value) {
  std::ostringstream oss;
  oss << value;
  return oss.str();
}
```

Stringstreams

Convert some `double` or `int` to a string?

```
inline std::string convert(double value) {
  std::ostringstream oss;
  oss << value;
  return oss.str();
}
```

It works, but `std::to_string` is a better choice!

Contents

Construction

`string s1;`	Default initialization. `s1` contains `""`.
`string s2(s1);`	Copy initialization. `s2` is a copy of `s1`.
`string s2 = s1;`	Equivalent to `string s2(s1)`.
`string s3("hello");`	`s3` is a copy of the string literal.
`string s3 = "hello";`	Equivalent to `string s3("hello")`.
`string s4(n, c);`	Initialize `s4` with `n` copies of a `char c`.

- A string object is **NOT** null-terminated!

IO

`os << s`	Writes `s` onto output stream `os`.
`is >> s`	Reads a string from `is` into `s`.
`getline(is, s)`	Reads a line of input from `is` into `s`.

- `is >> s` starts reading **from the first non-whitespace character**, and reads until the next whitespace character. The whitespace in the end **is not read and still in the stream**.
- `getline(is, s)` starts reading **from the next character** and reads until the first newline character. The newline character is **read, but not stored into `s`, and thrown away**.

Operations

`s.empty()`	Returns `true` iff `s` is empty (`""`).
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a **reference** to the character indexed `n` in `s`.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Copy-assignment. Replaces the content in `s1` with a **copy** of `s2`.
`s1 += s2`	Equivalent (?) to `s1 = s1 + s2`.
`==,!=`	Equality and inequality.
`<,<=,>,>=`	Lexicographical-order comparison.

Operations

Concatenation: `s1 + s2`.

- `s1` and `s2` can be a C-style string (`const char *`) or a `char`.
- At least one of `s1` and `s2` should be `std::string`!
- `s + "a" + "b"` compiles, while `"a" + "b" + s` does not compile. Why?

Operations

`s1 += s2` and `s1 = s1 + s2` are **NOT** actually equivalent:

```
auto n = 1000000;
std::string s = "";
for (auto i = 0; i != n; ++i)
  s += "a";
s = "";
for (auto i = 0; i != n; ++i)
  s = s + "a";
```

The first loop takes $O(n)$ time, while the second takes $O\left(n^2\right)$!

- Always prefer compound assignment operators.

Operations

`s.size()` returns a value of the type `std::string::size_type`.

- An unsigned integer type.
- It is guaranteed to be able to store the length of any string.
- It is highly possible that it is `std::size_t`, but not guaranteed.

Operations

`s.size()` returns a value of the type `std::string::size_type`.

- An unsigned integer type.
- It is guaranteed to be able to store the length of any string.
- It is highly possible that it is `std::size_t`, but not guaranteed.

`s.length()` is equivalent to `s.size()`, but `s.size()` is preferred. (Why?)

Examples

Example: Convert a nonnegative integer to a string. Add leading zeros.

```
inline std::string convert(int x) {
  auto s = std::to_string(x);
  return std::string(9 - s.size(), '0') + s;
}
```

Examples

Example: Convert a nonnegative integer to a string. Add leading zeros.

```
inline std::string convert(int x) {
  auto s = std::to_string(x);
  return std::string(9 - s.size(), '0') + s;
}
```

Example: Count the number of upper-case letters, and convert them to lower-case.

```
int upper_cnt = 0;
for (decltype(s.size()) i = 0; i != s.size(); ++i)
  if (std::isupper(s[i])) {
    ++upper_cnt;
    s[i] = std::tolower(s[i]);
  }
```

Range-based `for` Loops

Count the number of upper-case letters:

```
int upper_cnt = 0;
for (char c : s)
  if (std::isupper(c))
    ++upper_cnt;
```

Range-based `for` Loops

Count the number of upper-case letters:

```
int upper_cnt = 0;
for (char c : s)
  if (std::isupper(c))
    ++upper_cnt;
```

Convert upper-case letters to lower:

```
for (char &c : s)        // '&' is necessary!
  c = std::tolower(c);
```

Range-based `for` Loops

Count the number of upper-case letters:

```
int upper_cnt = 0;
for (char c : s)
  if (std::isupper(c))
    ++upper_cnt;
```

Convert upper-case letters to lower:

```
for (char &c : s)        // '&' is necessary!
  c = std::tolower(c);
```

- It is common to use `auto` in range-`for`.
- Looks like `Python` `for` loops?

Example

Suppose `s` contains an English sentence. Convert every letter of the first word into upper-case.

Example

Suppose s contains an English sentence. Convert every letter of the first word into upper-case.

```
for (decltype(s.size()) i = 0;
     i != s.size() && !std::isspace(s[i]); ++i)
  s[i] = std::toupper(s[i]);
```

Example

Suppose s contains an English sentence. Convert every letter of the first word into upper-case.

```
for (decltype(s.size()) i = 0;
     i != s.size() && !std::isspace(s[i]); ++i)
  s[i] = std::toupper(s[i]);
```

Range-for:

```
for (auto &c : s) {
  if (std::isspace(c))
    break;
  c = std::toupper(c);
}
```

Contents

Templated Type

```
std::vector<int> vi;
std::vector<std::string> vs;
std::vector<Widget> vw;
std::vector<std::vector<double>> vvd;
```

- **Instantiation** of a class template
- Template arguments: Must be known at compile-time!
- C++ is statically-typed!

Construction

`vector<T> v1`	Default initialization. `v1` is empty.
`vector<T> v2(v1)`	Copy initialization. `v2` is a copy of `v1`.
`vector<T> v2 = v1`	Equivalent to `vector<T> v2(v1)`.
`vector<T> v3(n, val)`	`v3` contains `n` copies of `val`.
`vector<T> v4(n)`	`v4` contains `n` default-initialized elements.

Every STL container has **value semantics**. Copy of a container will copy every element.

Construction

Since C++11, one more way of construction:

```
vector<T> v5 = {a, b, c, d};
vector<T> v6{a, b, c, d};   // Equivalent way.
```

For example,

```
vector<int> v = {2, 3, 5, 7, 11};
```

Construction

Since C++11, one more way of construction:

```
vector<T> v5 = {a, b, c, d};
vector<T> v6{a, b, c, d};   // Equivalent way.
```

For example,

```
vector<int> v = {2, 3, 5, 7, 11};
```

However, this causes troubles to the widely-used **braced-initialization**:

```
Point2d p{3, 4}; // Call Point2d::Point2d(double, double)
vector<int> v(10, 20); // v has 10 elements, each 20.
vector<int> v{10, 20}; // v has 2 elements: 10, 20.
```

Construction

`string` also supports such initialization.

```
string s1 = {'a', 'b', 'c'}; // s1 is "abc"
string s2(48, 'c'); // s2 is "ccc......c"
string s3{48, 'c'}; // s3 is "0c"
```

Allowing initialization from a braced list is now seen as **an error in the design**. (*Effective Modern C++* Item 7)

- Be careful when using braced initialization for every STL container.
- Avoid such design in your own classes.

Construction

`string` also supports such initialization.

```
string s1 = {'a', 'b', 'c'}; // s1 is "abc"
string s2(48, 'c'); // s2 is "ccc......c"
string s3{48, 'c'}; // s3 is "0c"
```

Allowing initialization from a braced list is now seen as **an error in the design**. (*Effective Modern C++* Item 7)

- Be careful when using braced initialization for every STL container.
- Avoid such design in your own classes.

Empty braces is undoubtedly default initialization. It calls the default constructor.

```
vector<int> v{}; // Equivalent to vector<int> v;
                 // Calls the default constructor.
```

Operations

`v.empty()`	Returns `true` iff `v` is empty.
`v.size()`	Returns the number of elements in `v`.
`v.push_back(t)`	Adds an element with value `t` to end of `v`.
`v[n]`	Returns a **reference** to the elmeent indexed `n`.
`v = {a, b, c}`	Replaces the elements in `v` with a copies of `a, b, c`.
`==, !=`	Equaltiy and inequality.
`<, <=, >, >=`	Lexicographical-order comparison.

- `v.size()` returns a value of type `vector<T>::size_type`.
- Compare it with `string`'s operation table.
  - In fact, `string` also supports `push_back`.
- Why doesn't `vector` provide concatenation `operator+`?

Access by Subscript

For containers that supports `operator[]` (`string`, `vector`, ...):

- `operator[]` does not check boundaries.
- Every sequential container that provides `operator[]` also provides the `at` member function. `v.at(n)` will `throw` an exception if `n` is out of range.
- It is the programmer's responsibility to ensure every subscript is valid.

```
std::vector<int> v;
std::cout << v[0] << std::endl; // Error!
```

Range-based `for` Loops

Also works for `vector`:

```
std::vector<int> v = {2, 3, 5, 7, 11, 13};
for (auto &x : v)
  x = x * x;
for (auto x : v)
  std::cout << x << " ";
std::cout << std::endl;
```

Range-based `for` Loops

Never change the size of the container within the range-`for`!

```
for (decltype(v.size()) i = 0; i != v.size(); ++i)
  if (v[i] % 2 == 1)
    v.push_back(v[i]); // ok
for (auto x : v)
  if (x % 2 == 1)
    v.push_back(x); // Probably causes undefined behavior!
```

This rule also applies to `string`, since their way of growing is similar.

Contents

Idea

For an array:

```
for (int i = 0; i != n; ++i)
  do_something(a[i]);
```

For a `vector` or a `string`:

```
for (std::size_t i = 0; i != c.size(); ++i)
  do_something(c[i]);
```

For a linked-list?

Idea

For an array:

```
for (int i = 0; i != n; ++i)
  do_something(a[i]);
```

For a `vector` or a `string`:

```
for (std::size_t i = 0; i != c.size(); ++i)
  do_something(c[i]);
```

For a linked-list?

```
for (Node *p = l.head; p; p = p->next)
  do_something(p->value);
```

Idea


苹果用户桌面:
华为用户桌面:
小红书
小红书ID: 108767203


苹果用户出差:
华为用户出差:
小红书
小红书ID: 108767203

Using Iterators

```
std::vector<int> v = {2, 3, 5, 7, 11, 13};
for (auto it = v.begin(); it != v.end(); ++it)
  *it = *it * *it;
for (auto it = v.begin(); it != v.end(); ++it)
  std::cout << *it << " ";
std::cout << std::endl;
```

- The type of 'it' is `std::vector<int>::iterator`.
- `v.begin()` returns the iterator pointing to the first element.
- `v.end()` returns the **off-the-end** iterator, positioned "one past the end" of the container.
- If the container is empty, we have `v.begin() == v.end()`.

Iterator Operations

The iterator of every container supports these operations:

`*iter`	Returns reference to the element denoted by `iter`.
`iter->mem`	Equivalent to `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element in the container.
`iter++`	Postfix version. Returns the copy of the original iterator.
`==, !=`	Equal iff both iterators are pointing to the same position.

- Dereferencing or incrementing an off-the-end iterator is undefined behavior.

Examples

Count the number of upper-case letters and convert them to lower-case.

Examples

Count the number of upper-case letters and convert them to lower-case.

```
int upper_cnt = 0;
for (auto it = s.begin(); it != s.end(); ++it) {
  if (std::isupper(*it)) {
    ++upper_cnt;
    *it = std::tolower(*it);
  }
}
```

Examples

Count the number of upper-case letters and convert them to lower-case.

```
int upper_cnt = 0;
for (auto it = s.begin(); it != s.end(); ++it) {
  if (std::isupper(*it)) {
    ++upper_cnt;
    *it = std::tolower(*it);
  }
}
```

Convert the first word to upper-case.

Examples

Count the number of upper-case letters and convert them to lower-case.

```
int upper_cnt = 0;
for (auto it = s.begin(); it != s.end(); ++it) {
  if (std::isupper(*it)) {
    ++upper_cnt;
    *it = std::tolower(*it);
  }
}
```

Convert the first word to upper-case.

```
for (auto it = s.begin();
    it != s.end() && !std::isspace(*it); ++it)
  *it = std::toupper(*it);
```

Range-`for`

The range-based `for` loop is treated as traversing using iterators.

```
for (auto x : v)
  do_something(x);
// Equivalent:
for (auto it = v.begin(); it != v.end(); ++it) {
  auto x = *it;
  do_something(x);
}
```

Range-for

```
for (const auto &x : v)
  do_something(x);
// Equivalent:
for (auto it = v.begin(); it != v.end(); ++it) {
  const auto &x = *it;
  do_something(x);
}
```

Range-`for`

```
for (const auto &x : v)
  do_something(x);
// Equivalent:
for (auto it = v.begin(); it != v.end(); ++it) {
  const auto &x = *it;
  do_something(x);
}
```

- Any object that could be traversed using the range-`for` loop must provide `begin()` and `end()`, which return an iterator that supports
  - `operator*`: dereference
  - `operator++`: prefix increment
  - `operator!=`

const_iterator

On a const object, begin() and end() return a different iterator type:

```
inline void print_vector(const std::vector<int> &v) {
  for (auto it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
  std::cout << std::endl;
}
```

- it here is of type vector<int>::const_iterator.
- Dereferencing it returns a reference-to-const, which is not modifiable.
- Since C++11, we have explicit way to obtain const_iterators: The cbegin() and cend() member functions.

Other Iterator Operations

Operations Supported by `vector` and `string` iterators:

`iter + n`	Returns an iterator positioned `n` elements forward.
`iter - n`	Returns an iterator positioned `n` elements backward.
`n + iter`	Equivalent to `iter + n`.
`iter += n`	Equivalent to `iter = iter + n`.
`iter -= n`	Equivalent to `iter = iter - n`.
`iter1 - iter2`	Returns the distance between two iterators.
`<,<=,>,>=`	Relational operators.

- The return-type of `iter1 - iter2` is a **type-alias member** named `difference_type`. e.g. `string::difference_type`. It is a signed integer type.
- `iter1 - iter2` returns the number that when added to `iter2` yields `iter1`.

Relational Operators of Iterators

`<`, `<=`, `>`, `>=`: comparing the positions of two iterators in the container.

- If they are not positioned in the same container, the result is undefined.
- Why did I always use `!=` in the `for` loops?

Relational Operators of Iterators

`<`, `<=`, `>`, `>=`: comparing the positions of two iterators in the container.

- If they are not positioned in the same container, the result is undefined.
- Why did I always use `!=` in the `for` loops?
  - Not all kinds of iterators support `<`, `<=`, `>`, `>=`, but all of them support `==` and `!=`.