CISC 3142
Programming Paradigms in C++

Part IV – Selected Topics

The Standard Library (Ch30-32, 36, 38)

(*Stroustrup – The C++ Programming Language, 4th Ed*)

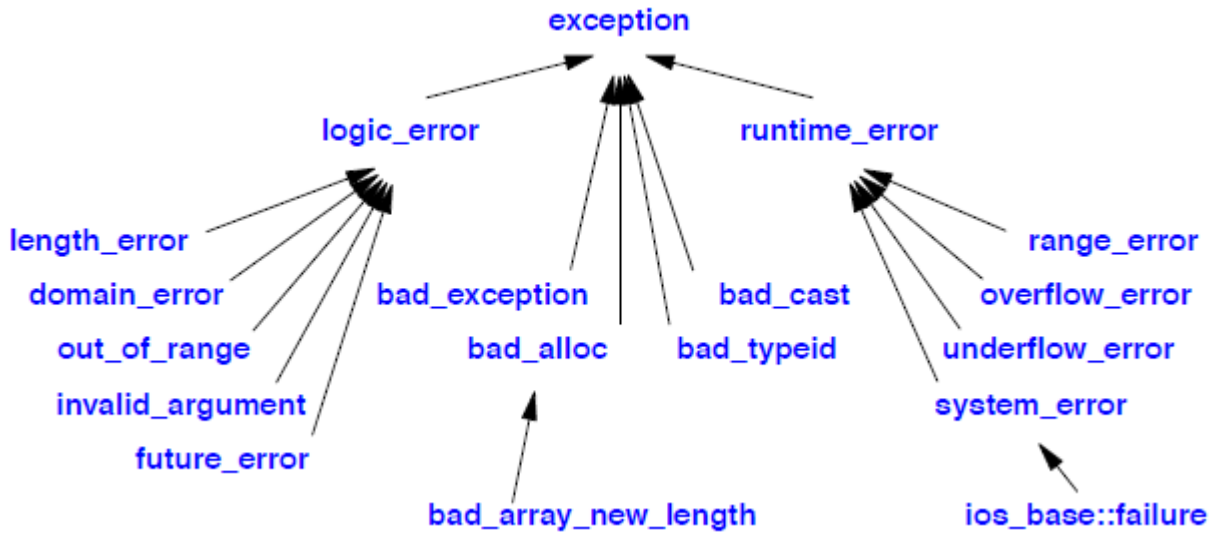
Standard-Library Overview

- For portability and long-term maintainability, using standard library whenever possible is strongly recommended
  - Don’t reinvent the wheel
- The specification of the standard library is many times more than that of the C++ language itself
- The standard library aims to be the common foundation for other libraries, the foundation includes 3 aspects:
  - Portability, performance, communication (common containers used by other libraries)
- Headers – the facilities of the Standard-Library are defined in the std namespace and presented as a set of headers
  - A standard header with a name starting with c is equivalent to a header in C standard library (<X.h> in C becomes <cX> in C++ within the std namespace)

The Standard Exception Hierarchy

- The hierarchy is rooted in class exception:

```
class exception {
public:
  exception();
  exception(const exception&);
  exception& operator=(const exception&);
  virtual ~exception();
  virtual const char* what() const;
};
```



- To derive own exception

```
struct My_error : runtime_error {
   My_error(int x) :runtime_error{"My_error"}, interesting_value{x} { }
   int interesting_value;
};
```

STL Containers

- Containers are categorized as:
  - *Sequence containers*: provide access to sequences of elements
    - vector, list, forward_list, deque
  - *Associative containers*: provide associative lookup based on key
    - Ordered: map, multimap, set, multiset
    - Unordered: unordered_map, unordered_multimap, unordered_set, unordered_multiset
  - *Container adaptors*: provide specialized access to underlying containers
    - stack, queue, priority_queue
  - *Almost containers*: sequences of elements that provide most, but not all of the facilities of a container
    - T[N] (no size()), array, string, bitset, etc

Container Oper. Complexity

- "**Front**" refers to insertion/deletion before the first element
- "**Back**" refers to that after the last element
- "**List**" refers to insertion/deletion not necessarily at the ends
- In the Iterators column, "**Ran**" means "random-access iterator", "**For**"- "forward iterator", and "**Bi**" – "bidirectional iterator

Standard Container Operation Complexity					
	[] §31.2.2	List §31.3.7	Front §31.4.2	Back §31.3.6	Iterators §33.1.2
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
built-in array	const				Ran
valarray	const				Ran
bitset	const				

Uses of Constructors, Destructor, and Assignments

```
void use()
{
    vector<int> vi {1,3,5,7,9}; // vector initialized by five ints
    vector<string> vs(7); // vector initialized by seven empty strings
    vector<int> vi2;
    vi2 = {2,4,6,8}; // assign sequence of four ints to vi2
    vi2.assign(&vi[1],&vi[4]); // assign the sequence 3,5,7 to vi2
    vector<string> vs2;
    vs2 = {"The Eagle", "The Bird and Baby"}; // assign two strings to vs2
    vs2.assign("The Bear", "The Bull and Vet"); // run-time error
}
```

Constructors, Destructor, and Assignment (continues)

C is a container; by default, a C uses the default allocator C::allocator_type{}

C c {};	Default constructor: c is an empty container
C c {a};	Default construct c; use allocator a
C c(n);	c initialized with n elements with the value value_type{}; not for associative containers
C c(n,x);	Initialize c with n copies of x; not for associative containers
C c(n,x,a);	Initialize c with n copies of x; use allocator a; not for associative containers

Constructors, Destructor, and Assignment (continued)

C is a container; by default, a C uses the default allocator C::allocator_type{}

C c {elem};	Initialize c from elem; if C has an initializer-list constructor, prefer that; otherwise, use another constructor
C c {c2};	Copy constructor: copy c2's elements and allocator into c
C c {move(c2)};	Move constructor: move c2's elements and allocator into c
C c {{elem},a};	Initialize c from the initializer_list {elem}; use allocator a
C c {b,e};	Initialize c with elements from [b:e)
C c {b,e,a};	Initialize c with elements from [b:e); use allocator a
c.~C()	Destructor: destroy c's elements and release all resources
c2=c	Copy assignment: copy c's elements into c2
c2=move(c)	Move assignment: move c's elements into c2
c={elem}	Assign to c from initializer_list {elem}
c.assign(n,x)	Assign n copies of x; not for associative containers
c.assign(b,e)	Assign to c from [b:e)
c.assign({elem})	Assign to c from initializer_list {elem}

Container Element Access

- While some implementations (usually debug versions) always do range checking, you cannot portably rely on that for correctness, or the absence of checking for performance

Element Access	
c.front()	Reference to first element of c; not for associative containers
c.back()	Reference to last element of c; not for forward_list or associative containers
c[i]	Reference to the ith element of c; unchecked access; not for lists or associative containers
c.at(i)	Reference to the ith element of c; throw an out_of_range if i is out of range; not for lists or associative containers
c[k]	Reference to the element with key k of c; insert (k,mapped_type{}) if not found; for map and unordered_map only
c.at(k)	Reference to the ith element of c; throw an out_of_range if k is not found; for map and unordered_map only

STL Algorithms

- Summary
  - Algorithms operate on sequences defined by
    - A pair of iterators for inputs, [b : e), and if necessary
    - A single iterator for outputs, b2, which is assumed to have range: [b2 : b2+(e-b))
  - Some algorithms, such as sort(), requires random-access iterators
  - Others, such as find(), can make do with a forward iterator
  - Many algorithms follow the convention of returning the end of a sequence to represent “not found”
- Main takeaway: use well-defined algorithms over “random code” for
  - Correctness, maintainability, and performance

Nonmodifying Sequence Algorithms

- for_each(), the simplest algorithm, and the least specific
  - f=for_each(b,e,f) // do f(x) for each x in [b:e); return f
  - It's still possible for f to modify elements:

```
void increment_all(vector<int>& v)   { // increment each element of v
    for_each(v.begin(),v.end(), [](int& x) {++x;});   }
```

- Sequence predicates
  - all_of(b,e,f), any_of(b,e,f), none_of(b,e,f)
- count(b,e,v) and count_if(b,e,f)
- find(b,e,v) and find_if(b,e,f), and many more variations
- equal(b,e,b2), and pair(p1, p2) = mismatch(b,e,b2) – sequence against sequence, equal (returns bool), and first mismatch !(*p1 == *p2)
- p=search(b,e,b2,e2), and p=search_n(b,e,n,v), finding one sequence [b2, e2) as a subsequence in another [b, e); or n consecutive values of v in [b, e)

Modifying Sequence Algorithms

- p=transform(b,e,out,f) – apply *q=f(*p1), i.e. unary op, to every *p1 in [b:e), writing *q to out, returned p points to one after last element in out
  - p=transform(b,e,b2,out,f) – apply *q=f(*p1, *p2), i.e. binary op, to every *p1 in [b:e) and *p2 in [b2:b2+(e-b)), writing to out
- copy(b,e,out), copy_if(b,e,out,f), copy_n(b,n,out)
- p=unique(b,e) – unique elements moved to front, p points to element right-after
  - uniq_copy(b,e,out) – eliminates adjacent duplicates, results copied to out
- p=remove(b,e,v) – moves all v's toward the back, headed by p
  - p=remove_copy(b,e,out,v) – copies all elements not equal to v's to out
- replace(b,e,v,v2) – replaces all v's with v2
- rotate(), random_shuffle(), partition(), permutations, fill(), swap(), and more
- There are also a whole suite of sorting and searching, set, heaps, min/max algorithms

Strings

- At character level, from <cctype>
  - **isspace(c)** Is **c** whitespace (space ' ', horizontal tab **'\t'**, newline **'\n'**, vertical tab **'\v'**, form feed **'\f'**, carriage return **'\r'**)?
  - **isalpha(c)** Is **c** a letter (**'a'..'z'**, **'A'..'Z'**)? note: not underscore '_'
  - **isdigit(c)** Is **c** a decimal digit (**'0'..'9'**)?
  - **isxdigit(c)** Is **c** hexadecimal digit (decimal digit or **'a'..'f'** or **'A'..'F'**)?
  - **isupper(c)** Is **c** an uppercase letter?
  - **islower(c)** Is **c** a lowercase letter?
  - **isalnum(c)** **isalpha(c)** or **isdigit(c)**
  - **iscntrl(c)** Is **c** a control character (ASCII 0..31 and 127)?
  - **ispunct(c)** Is **c** not a letter, digit, whitespace, or invisible control character?
  - **isprint(c)** Is **c** printable (ASCII ' '..'~')?
  - **isgraph(c)** **isalpha(c)** or **isdigit(c)** or **ispunct(c)**? note: not space
  - **toupper(c)** **c** or **c**'s uppercase equivalent
  - **tolower(c)** **c** or **c**'s lowercase equivalent

Strings – Fundamental Operations

Access →

s[i]	Subscripting: s[i] is a reference to the ith element of s; no range check
s.at(i)	Subscripting: s.at(i) is a reference to the ith element of s; throw range_error if s.size()<=i
s.front()	s[0]
s.back()	s[s.size()–1]
s.push_back(c)	Append the character c
s.pop_back()	Remove the last character from s: s.erase(s.size()–1)
s+=x	Append x at the end of s; x can be a character, a string, a C-style string, or an initializer_list<char_type>
s=s1+s2	Concatenation: optimized version of s=s1; s+=s2;
n2=s.copy(s2,n,pos)	s gets the characters from s2[pos:n2) where n2 is min(n,s.size()–pos); throw out_of_range if s.size()<pos
n2=s.copy(s2,n)	s gets all the characters from s2; n=s.copy(s2,n,0)
p=s.c_str()	p is a C-style string version (zero-terminated) of the characters in s; a const C*
p=s.data()	p=s.c_str()
s.swap(s2)	Exchange the values of s and s2; noexcept
swap(s,s2)	s.swap(s2)

Seems to be backwards
s -> s2

↓ Size and capacity

n=s.size()	n is the number of characters in s
n=s.length()	n=s.size()
n=s.max_size()	n is the largest possible value of s.size()
s.resize(n,c)	Make s.size()==n; added elements get the value c
s.resize(n)	s.resize(n,C{})
s.reserve(n)	Ensure that s can hold n characters without further allocation
s.reserve()	No effect: s.reserve(0)
n=s.capacity()	s can hold n characters without further allocation
s.shrink_to_fit()	Make s.capacity==s.size()
s.clear()	Make s empty
s.empty()	Is s empty?
a=s.get_allocator()	a is s's allocator

Strings – Fundamental Operations - 2

String I/O →

in>>s	Read a whitespace-separated word into s from in
out<<s	Write s to out
getline(in,s,d)	Read characters from in into s until the character d is encountered; d is removed from in but not appended to s
getline(in,s)	getline(in,s,'\n') where '\n' is widened to match the string's character type

Numeric Conversions →

x=stoi(s,p,b)	String to int; x is an integer; read starting with s[0] if p!=nullptr, *p is set to the number of characters used for x; b is the base of the number (between 2 and 36, inclusive)
x=stoi(s,p)	x=stoi(s,p,10); decimal numbers
x=stoi(s)	x=stoi(s,nullptr,10); decimal numbers; don't report the character count
x=stol(s,p,b)	String to long
x=stoul(s,p,b)	String to unsigned long
x=stoll(s,p,b)	String to long long
x=stoull(s,p,b)	String to unsigned long long
x=stof(s,p)	String to float
x=stod(s,p)	String to double
x=stold(s,p)	String to long double
s=to_string(x)	s is a string representation of x; x must be an integer or floating-point value
ws=to_wstring(x)	s is a wstring representation of x; x must be an integer or floating-point value

Strings – Fundamental Operations - 3

Finding Element →

x can be a char, string, or C-style string

string = basic_string<char>

pos=s.find(x)	Find x in s; pos is the index of the first character found or string::npos
pos=s.find(x,pos2)	pos=find(basic_string(s,pos2) starting from pos2 of s
pos=s.find(p,pos2,n)	pos=s.find(basic_string{p,n},pos2) only search the first n chars of p
pos=s.rfind(x,pos2)	Find x in s[0:pos2); pos is the position of the first character of the x closest to the end of s or string::npos
pos=s.rfind(x)	pos=s.rfind(p,string::npos)
pos=s.rfind(p,pos2,n)	pos=s.rfind(basic_string{p,n},pos2)

Finding Element from a Set →

Variations include

find_last_of, find_first_not_of
find_last_not_of

pos2=s.find_first_of(x,pos)	Find a character from x in s[pos:s.size()); pos2 is the position of the first character from x in s[pos:s.size()) or string::npos
pos=s.find_first_of(x)	pos=s.find_first_of(s2,0)
pos2=s.find_first_of(p,pos,n)	pos2=s.find_first_of(pos,basic_string{p,n})

Substrings →

s2=s.substr(pos,n)	s2=basic_string(&s[pos],m) where m=min(s.size()–n,n)
s2=s.substr(pos)	s2=s.substr(pos,string::npos)
s2=s.substr()	s2=s.substr(0,string::npos)

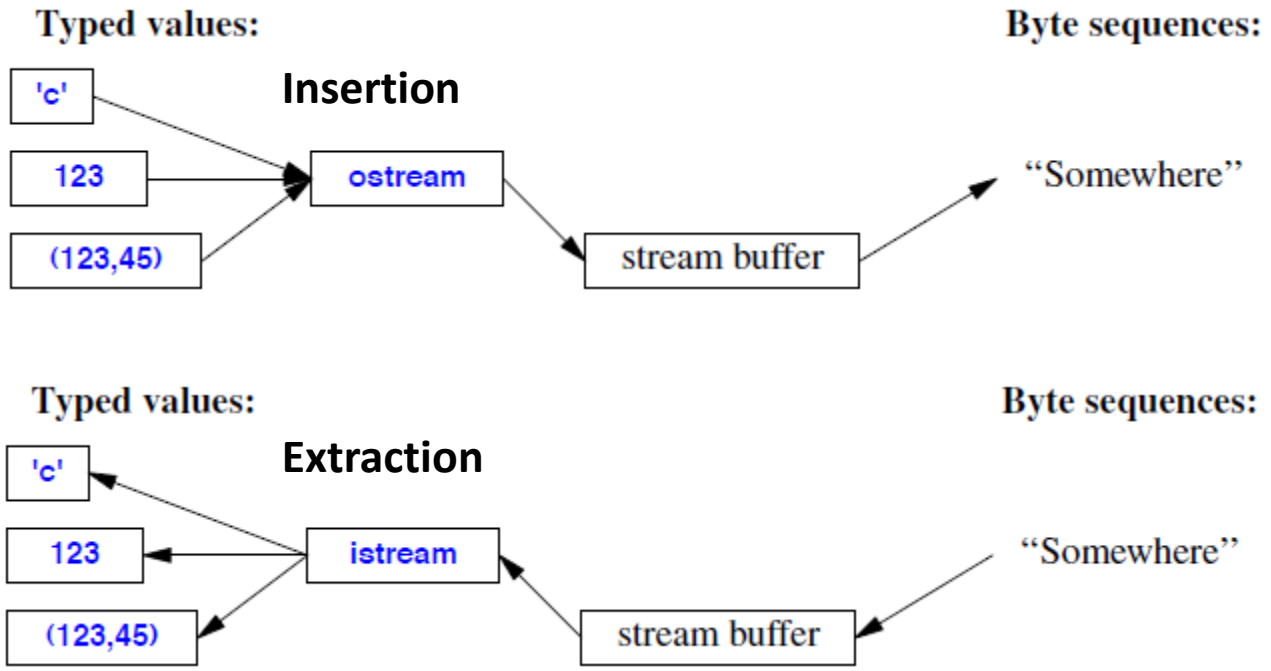
Compare →

n=s.compare(s2)	A lexicographical comparison of s and s2; using char_traits<C>::compare() for comparison; n=0 if s==s2; n<0 if s<s2; n>0 if s2>s; noexcept;
n2=s.compare(pos,n,s2)	n2=basic_string{s,pos,n}.compare(s2)
n2=s.compare(pos,n,s2,pos2,n2)	n2=basic_string{s,pos,n}.compare(basic_string{s2,pos2,n2})

I/O Streams

- Provide formatted and unformatted buffered I/O of text and numeric values

- An ostream converts typed objects to a stream of bytes
- An istream converts a stream of bytes to typed objects
- An iostream can act as both an istream and an ostream
- The buffers are represented by streambufs

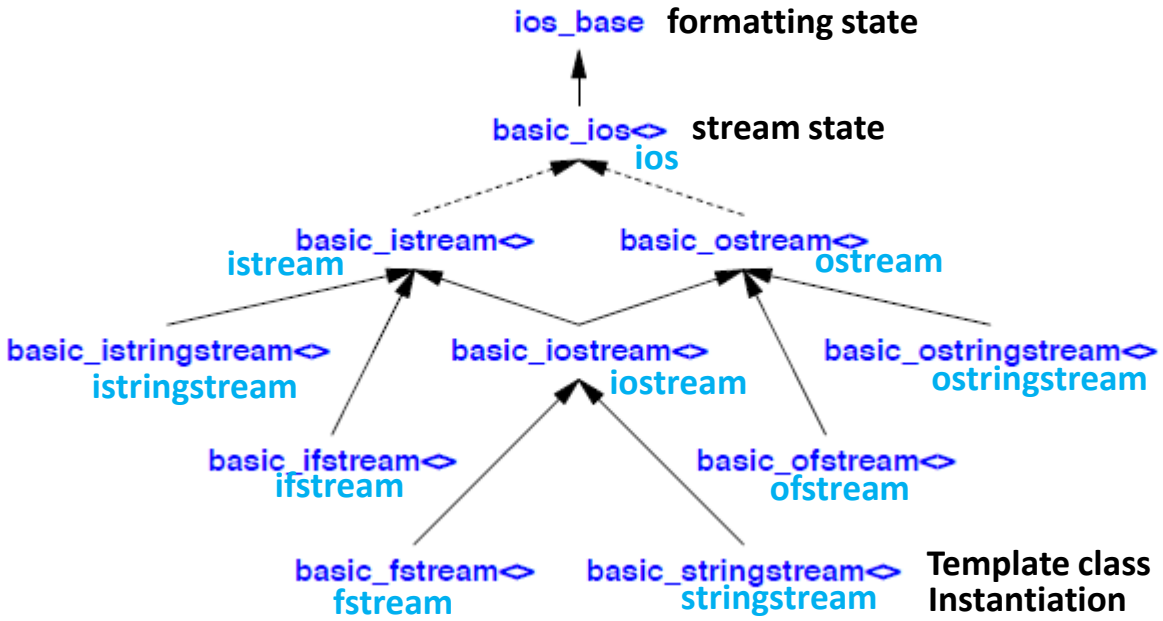


The I/O Stream Hierarchy

- Involved header files
  - <ios>
  - <istream>
  - <ostream>
  - <iostream>
  - <fstream>
  - <sstream>

- From **<iostream>**

cout	The standard character output (often by default a screen)
cin	The standard character input (often by default a keyboard)
cerr	The standard character error output (unbuffered)
clog	The standard character error output (buffered)



File Streams <fstream>

- fstream operations
  - **fstream fs {};** **fs** is a file stream not attached to a file, default mode **m** = **ios_base::in** | **ios_base::out**
  - **fstream fs {s,m};** **fs** is a file stream opened for a file called **s** with mode **m**; **s** can be a **string** or a C-style string
  - **fs.is_open()** Is **fs** open?
  - **fs.open(s,m)** Open a file called **s** with mode **m** and have **fs** refer to it; sets **fs**'s **failbit** if it couldn't open the file; **s** can be a **string** or a C-style string
  - **fs.close()** Close the file associated with **fs** (if any)
- Stream Modes
  - **ios_base::app** Append (i.e., add to the end of the file)
  - **ios_base::ate** ''At end'' (open and seek to the end)
  - **ios_base::binary** Binary mode; beware of system-specific behavior
  - **ios_base::in** For reading
  - **ios_base::out** For writing
  - **ios_base::trunc** Truncate the file to 0 length
- Usage Example

```
ofstream ofs("target"); // ''o'' for ''output'' implying ios_base::out
if (!ofs)
    error("couldn't open 'target' for writing");
fstream ifs; // ''i'' for ''input'' implying ios_base::in
ifs.open("source", ios_base::in);
if (!ifs)
    error("couldn't open 'source' for reading");
```

String Streams <sstream>

- stringstream operations

stringstream ss {m};	**ss** is an empty string stream with mode **m**
stringstream ss {};	Default constructor: **stringstream ss {ios_base::out|ios_base::in};**
stringstream ss {s,m};	**ss** is a string stream with its buffer initialized from the **string s** with mode **m**
stringstream ss {s};	**stringstream ss {s, ios_base::out|ios_base::in};**
s=ss.str()	**s** is a **string** copy of the characters in **ss**: **s=ss.rdbuf()–>str()**
ss.str(s)	**ss**'s buffer is initialized from the **string s**: **ss.rdbuf()–>str(s)**; if **ss**'s mode is **ios::ate** (''at end'') values written to **ss** are added after the characters from **s**; otherwise values written overwrites the characters from **s**

- Usage Example

```
void test() {
    ostringstream oss {"Label: ", ios::ate}; // write at end
    cout << oss.str() << '\n';               // writes "Label: "
    oss<<"val";
    cout << oss.str() << '\n';               // writes "Label: val" ("val" appended after "Label: ")
    ostringstream oss2 {"Label: "};          // write at beginning
    cout << oss2.str() << '\n';              // writes "Label: "
    oss2<<"val";
    cout << oss2.str() << '\n';              // writes "valel: " (val overwrites "Label: ")
}
```

I/O Error Handling

- An iostream can be in one of four states

good()	The previous **iostream** operations succeeded
eof()	We hit end-of-input (‘‘end-of-file’’)
fail()	Something unexpected happened (e.g., we looked for a digit and found **'x'**)
bad()	Something unexpected and serious happened (e.g., disk read error)

- Any operation not in the good() state has no effect; it is a no-op.
- An iostream can be used as a condition (true when it’s good())

```
int i;
if (cin>>i) {
    // ... use i ...
} else if (cin.fail()){ // possibly a formatting error
    cin.clear(); // clear error state flags
    string s;
    if (cin>>s) { // we might be able to use a string to recover
        // ... use s ...
    }
}
```

Formatted vs Unformatted Input

- Formatted input

  in>>x – read from **in** into **x** according to **x**'s type

  getline(in, s) – read a line from **in** into string **s** **(note: it may read '\r' on Windows)**

- Unformatted input

  x=in.get() - read one character from **in** and return its integer value; return **EOF** for end-of-file

  in.get(c) - read a character from **in** into **c**

  in.getline(p,n,t) - read at most **n** characters from **in** into **[p:...)**; consider **t** a terminator; remove terminator

  in.ignore(n,d) - extract characters from **in** and discard them until either **n** characters have been discarded or **d** is found (and discarded)

Formatting

- Formatting state (defined in ios_base, there are more than listed here)
  - **boolalpha** Use symbolic representation of **true** and **false**
  - **dec** | **hex** | **oct** Integer base is 10 | 16 | 8
  - **fixed** Floating-point format dddd**.**dd
  - **scientific** Scientific format d**.**dddd**E**dd
  - **internal** Pad between a prefix (such as **+**) and the number
  - **left** | **right** Pad after | before the value
  - **showbase** On output, prefix octal numbers by **0** and hexadecimal numbers by **0x**
  - **showpoint** Always show the decimal point (e.g., **123.**)
- ios.unsetf(ios_base::boolalpha) or ios.setf(ios_base::fixed), etc changes these fmtflags settings, as well as ios.width(n) and ios.precision(n), etc
- *Precision* is an integer determining the # of digits for a floating-point number
  - The *general* format (defaultfloat) lets the implementation choose the best format, the precision specifies the max # of digits
  - The *scientific* format (scientific) presents a value with one digit before a decimal point and an exponent. The precision specifies the max # of digits after the decimal point
  - The *fixed* format (fixed) presents a value as an (integer.fraction) and the precision value now specifies the max # of digits after the decimal point

Formatting (cont’)

- Examples of formatting operations

```
cout.precision(8); // sticky setting
cout << 1234.56789 << ' ' << 1234.56789 << '\n'; // 1234.5679 1234.5679
cout.width(4); // minimum # of characters, not sticky, only applies once
cout.fill('#');
cout << "ab" << ' : ' << "ab"; // print ##ab : ab
```

- Standard Manipulators
  - It would be easier to allow these manipulators to be directly embedded within the << operation, instead of being member function calls of streams
  - Standard manipulators are defined in <ios>, <istream>, <ostream> (these are already included when you include <iostream>), and <iomanip> (need explicit inclusion) for manipulators that take arguments

Standard Manipulators

- Examples (#include <iomanip>, for setw, setfill and setprecision)

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n'; // print 1234,4d2,2322 (sticky)

constexpr double d = 123.456;
cout << d << "; “ << scientific << d << "; “ << fixed << d << "; " << '\n'; // sticky
// print 123.456; 1.234560e+002; 123.456000 (defaultfloat has precision of 6)

// for a fixed two decimal places, use a combo of fixed and setprecision()
cout << fixed << setprecision(2) << d << endl; // print 123.46

// again setw() is not sticky
cout << '(' << setw(4) << setfill('#') << 12 << ") (" << 12 << ")\n"; // print (##12) (12)
```