Lecture Material

- Strings
- Input/output streams

Strings as Sequences of Characters

- A string is a sequence of characters
- The standard C++ library provides various string operations
  - indexing
  - assignment
  - comparison
  - appending
  - concatenation
  - substring search
- Characters are stored in the computer memory as numbers
  - Many possible character sets
  - Various 8-bit standards (Latin1, Latin2,...)
  - 16 and 32-bit characters (Unicode)

Character Traits

- The standard *string* requires, that the character type does not have the user-defined copy operation
  - Improves efficiency and simplifies implementation
  - Facilitates implementation of input/output operation
- The properties of character type are defined by its *char_traits*, it is a template specialization

```
template<class Ch> struct char_traits{ };
```

- All *char_traits* are defined in *std* namespace, and standard traits are provided by the <*string*> header

Character Traits

```
template<> struct char_traits<char> {
 typedef char char_type; // type of character
 static void assign(char_type&, const char_type&) ; // = for char_type
 // integer representation of characters:
 typedef int int_type; // type of integer value of character
 static char_type to_char_type(const int_type&) ; // int to char conversion
 static int_type to_int_type(const char_type&) ; // char to int conversion
 static bool eq_int_type(const int_type&, const int_type&) ; // ==
 // char_type comparisons:
 static bool eq(const char_type&, const char_type&) ; // ==
 static bool lt(const char_type&, const char_type&) ; // <
 // operations on s[n] arrays:
 static char_type* move(char_type* s, const char_type* s2, size_t n) ;
 static char_type* copy(char_type* s, const char_type* s2, size_t n) ;
 static char_type* assign(char_type* s, size_t n, char_type a) ;
 static int compare(const char_type* s, const char_type* s2, size_t n) ;
 static size_t length(const char_type*) ;
 static const char_type* find(const char_type* s, int n, const char_type&) ;
 // I/O related:
 typedef streamoff off_type; // offset in stream
 typedef streampos pos_type; // position in stream
 typedef mbstate_t state_type; // multibyte stream state
 static int_type eof() ; // endoffile
 static int_type not_eof(const int_type& i) ; // i unless i equals eof();
                                             // if not any value!=eof()
 static state_type get_state(pos_type p) ;
                                // multibyte conversion state of character in p
};
```

Character Traits

- The implementation of the standard string template, *basic_string*, uses these types and functions
- The character type used in *basic_string* must provide the specialization of *char_traits*, which provides all of them
- In order to fulfill the function of *char_type* by the character, it must be possible to obtain an integer value corresponding to each character. The type of this value is *int_type*. The conversion between it and a character type is performed using *to_char_type()* and *to_int_type()*. For *char* the conversion is trivial
- Both *move(s, s2, n)* and *copy(s, s2, n)* copy *n* characters from *s2* to *s*. *move()* works correctly even for overlapping strings, so *copy()* can be faster
- Method *assign(s, n, x)* writes *n* copies of *x* to *s*.
- The *compare()* method for comparison of characters uses *lt()* and *eq()* and returns an integer of type *int*, the return value convention is the same as in case of *strcmp()*
- Wide character, i.e. object of type *wchar_t* is similat to *char*, but occupies two or more bytes. The properties of *wchar_t* are described by *char_traits<wchar_t>*

```
template<> struct char_traits<wchar_t> {
 typedef wchar_t char_type;
 typedef wint_t int_type;
 typedef wstreamoff off_type;
 typedef wstreampos pos_type;
 // like char_traits<char>
};
```

basic_string

- The basis for string facilities in standard library is the *basic_string* template, which provides member types and operations similar to the ones provided by standard collections

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class std::basic_string {
 public:
 // ...
};
```

- This template and related features are defined in the *std* namespace and provided by the <*string*> header
- Two type definitions provide the conventional names for the popular string types

```
typedef basic_string<char> string;
typedef basic_string<wchar_t>wstring;
```

Types

- *basic_string* provides related types using the set of names of member types

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // types
 typedef Tr traits_type; // specific to basic_string
 typedef typename Tr::char_type value_type;
 typedef A allocator_type;
 typedef typename A::size_type size_type;
 typedef typename A::difference_type difference_type;
 typedef typename A::reference reference;
 typedef typename A::const_reference const_reference;
 typedef typename A::pointer pointer;
 typedef typename A::const_pointer const_pointer;
 typedef implementation_defined iterator;
 typedef implementation_defined const_iterator;
 typedef std::reverse_iterator<iterator> reverse_iterator;
 typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
 // ...
};
```

Iterators

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // iterators
 iterator begin() ;
 const_iterator begin() const;
 iterator end() ;
 const_iterator end() const;
 reverse_iterator rbegin() ;
 const_reverse_iterator rbegin() const;
 reverse_iterator rend() ;
 const_reverse_iterator rend() const;
 // ...
};
```

- Because *string* has the required member types and functions for aquisition of iterators, the strings can be used in connection with the standard algorithms

```
void f(string& s)
{
 string::iterator p = find(s.begin() ,s.end() ,´a´) ;
 // ...
}
```

- The most popular string operations are provided by the *string* class directly
- The range of string iterators is not checked

Element Access

- You can access the individual characters in a string by the means of indexing

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // element access
 const_reference operator[](size_type n) const; // unchecked access
 reference operator[](size_type n) ;
 const_reference at(size_type n) const; // checked access
 reference at(size_type n) ;
 // ...
};
```

- On the out-of-range access attempt, *at()* throws the *out_of_range* exception
- There are no *front()* nor *back()*
- The pointer/array equivalence is not valid for strings, *&s[0]* is not the same as *s*

Constructors

- The set of initialization and copying operations is different in many details from the set provided by other collections

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // constructors, etc. (a bit like vector and list)
 explicit basic_string(const A& a =A()) ;
 basic_string(const basic_string& s,
 size_type pos = 0, size_type n = npos, const A& a =A()) ;
 basic_string(const Ch* p, size_type n, const A& a =A()) ;
 basic_string(const Ch* p, const A& a =A()) ;
 basic_string(size_type n,Ch c, const A& a =A()) ;
 template<class In> basic_string(In first, In last, const A& a =A()) ;
 ~basic_string() ;
 static const size_type npos; // ``all characters'' marker
 // ...
};
```

Constructors

- The *string* can be initialized with a C-style string, another *string*, part of the C-style string, part of *string* or a sequence of characters. However, it cannot be initialized with a character nor an integer

```
void f(char* p,vector<char>&v)
{
 string s0; // the empty string
 string s00= ""; // also the empty string
 string s1= ´a´; // error: no conversion from char to string
 string s2 = 7; // error: no conversion from int to string
 string s3(7) ; // error: no constructor taking one int argument
 string s4(7,´a´) ; // 7 copies of ′a′; that is "aaaaaa"
 string s5= "Frodo"; // copy of "Frodo"
 string s6 = s5; // copy of s5
 string s7(s5,3,2) ; // s5[3] and s5[4]; that is "do"
 string s8(p+7,3) ; // p[7], p[8], and p[9]
 string s9(p,7,3) ; // string(string(p),7,3), possibly expensive
 string s10(v.begin() ,v.end()) ; // copy all characters from v
}
```

- The characters in a string are numbered starting from position 0, so a string is a set of characters numbered from 0 to *length()* -1
- *length()* is an alias of *size()*, both return the number of characters in a string
  - Not counting the terminating zero

Constructors

- The substrings are defined giving the character position and number of characters
- The copy constructor accepts four arguments, three of them have default values
- The most general constructor is a member template. It allows to initialize the string with any sequence of values, in particular the elements of a different character type, if the appropriate conversion exists

```
void f(string s)
{
 wstring ws(s.begin() ,s.end()) ; // copy all characters from s
 // ...
}
```

Error Handling

- Manipulation on substrings and characters can be the source of errors - attempts to read or write over the end of string
- *at()* throws *out_of_range* at an attempt of access over the end of string
- Most of string operations takes a character position and character count. Giving the position greater than the size of the strings results in *out_of_range* exception. Too large count is treated as if the user meant "the remaining characters"

```
void f()
{
 string s= "Snobol4";
 string s2(s,100,2) ; // character position beyond end of string:
                      // throw out_of_range()
 string s3(s,2,100) ; // character count too large:
                      // equivalent to s3(s,2,s.size()- 2)
 string s4(s,2,string::npos) ; // the characters starting from s[2]
}
```

- A negative number is a confusing way to submit a large positive number, as the *size_type* used to represent positions and counts is unsigned

```
void g(string& s)
{
 string s5(s,-2,3); // large position!: throw out_of_range()
 string s6(s,3,-2); // large character count!: ok
}
```

- The substring finding functions return *npos*, when they do not find anything. An attempt to use *npos* as a position will throw an exception
- For all strings *length()* < *npos*. Sometimes, when inserting one string into another, the result may be too long to be represented, *length_error* is thrown in that case.

Assignment

- Strings can be assigned

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // assignment (a bit like vector and list: §16.3.4):
 basic_string& operator=(const basic_string& s) ;
 basic_string& operator=(const Ch* p) ;
 basic_string& operator=(Ch c) ;
 basic_string& assign(const basic_string&) ;
 basic_string& assign(const basic_string& s, size_type pos, size_type n) ;
 basic_string& assign(const Ch* p, size_type n) ;
 basic_string& assign(const Ch* p) ;
 basic_string& assign(size_type n,Ch c) ;
 template<class In> basic_string& assign(In first, In last) ;
 // ...
};
```

Assignment

- *string* has a value semantics
  - By assigning one string to another, the string is copied, and after the operation two separate strings with the same value exist

```
void g()
{
 string s1= "Knold";
 string s2= "Tot";
 s1 = s2; // two copies of "Tot"
 s2[1] = ´u´; // s2 is "Tut", s1 is still "Tot"
}
```

- Assigning a single character to a string is allowed, even if it is not possible at the initialization time

```
void f()
{
 string s= ´a´; // error: initialization by char
 s= ´a´; // ok: assignment
 s= "a";
 s = s;
}
```

Conversion to C-style String

- *string* can be initialized with a C-style string and be assigned with such string. A copy of characters from *string* can be placed in an array.

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // conversion to C-style string:
 const Ch* c_str() const;
 const Ch* data() const;
 size_type copy(Ch* p, size_type n, size_type pos = 0) const;
 // ...
};
```

- The *data()* function copies the characters from the string to an array and returns the pointer to it.
  - The array is owned by *string* and the user should not attempt to delete it
  - He should not rely on the contents of this array after subsequent invocation of non-const method on string.

Conversion to C-style String

- The *c_str()* funcions is similar to *data()*, but terminates the C-style string with 0

```
void f()
{
 string s= "equinox"; // s.length()==7
 const char* p1 = s.data() ; // p1 points to seven characters
 printf("p1= %s\n",p1) ; // bad: missing terminator
 p1[2] = ´a´; // error: p1 points to a const array
 s[2] = ´a´;
 char c = p1[1] ; // bad: access of s.data() after modification of s
 const char* p2 = s.c_str() ; // p2 points to eight characters
 printf("p2= %s\n",p2) ; // ok: c_str() adds terminator
}
```

- The main task of this conversion function is to allow simple use of function with C-style strings as arguments. c_*str()* is more useful here than *data()*

```
void f(string s)
{
 int i = atoi(s.c_str()) ; // get int value of digits in string
 // ...
}
```

Conversion to C-style String

- Usually it is the best to leave characters in a *string* until they are needed. If they cannot be used immediately, they can be copied into an array, instead of leaving them in the buffer allocated by *c_str()* or *data()*

```
char* c_string(const string& s)
{
 char* p = new char[s.length()+1] ; // note: +1
 s.copy(p,string::npos) ;
 p[s.length()] = 0; // note: add terminator
 return p;
}
```

- A call *s.copy(p, n, m)* copies at mostj *n* characters to *p* starting from *s[m]*. If there is less than *n* characters to copy from *s*, the function copies all the characters till the end of string
- A *string* can contain the character with code 0. Functions dealing with C-style strings will interpret it as a terminator

Comparisons

- The *strings* can be compared with the *strings* of the same type and with the character arrays containing the characters of the same type

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 int compare(const basic_string& s) const; // combined > and ==
 int compare(const Ch* p) const;
 int compare(size_type pos, size_type n, const basic_string& s) const;
 int compare(size_type pos, size_type n,
               const basic_string& s, size_type pos2, size_type n2) const;
 int compare(size_type pos, size_type n, const Ch* p,
                                          size_type n2 = npos) const;
 // ...
};
```

- If *n* is given, only first *n* characters are compared. As a comparison criterion *compare()* from *char_traits<Ch>* is used..
- The user cannot provide a comparison criterion

Comparisons

- For *basic_string* objects the usual comparison operators ==, !=, >,<,>=,<= are available

```
template<class Ch, class Tr, class A>
                              bool operator==(const basic_string<Ch,Tr,A>&,
                                              const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
                 bool operator==(const Ch*, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
                 bool operator==(const basic_string<Ch,Tr,A>&, const Ch*) ;
   // similar declarations for !=, >, <, >=, and <=
```

- The comparison operators are not implemented as methods, so the conversions can be applied in the same way for both arguments. There are special versions of the operators to optimize comparisions with C-style strings.

```
void f(const string& name)
{
 if (name =="Obelix" || "Asterix"==name) { // use optimized ==
 // ...
 }
}
```

Insertion

- One of the most common modyfying operations on strings is appending some characters to their end. Insertion of characters in other places is less frequent.

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // add characters after (*this)[length()- 1]:
 basic_string& operator+=(const basic_string& s) ;
 basic_string& operator+=(const Ch* p) ;
 basic_string& operator+=(Ch c) ;
 void push_back(Ch c) ;
 basic_string& append(const basic_string& s) ;
 basic_string& append(const basic_string& s, size_type pos, size_type n) ;
 basic_string& append(const Ch* p, size_type n) ;
 basic_string& append(const Ch* p) ;
 basic_string& append(size_type n,Ch c) ;
 template<class In> basic_string& append(In first, In last) ;
 // insert characters before (*this)[pos]:
 basic_string& insert(size_type pos, const basic_string& s) ;
 basic_string& insert(size_type pos, const basic_string& s, size_type pos2,
                                                             size_type n) ;
 basic_string& insert(size_type pos, const Ch* p, size_type n) ;
 basic_string& insert(size_type pos, const Ch* p) ;
 basic_string& insert(size_type pos, size_type n,Ch c) ;
 // insert characters before p:
 iterator insert(iterator p,Ch c) ;
 void insert(iterator p, size_type n,Ch c) ;
 template<class In> void insert(iterator p, In first, In last) ;
 // ...
};
```

Insertion

- In principle, the set of operations provided for initialization and assignment is also available for appending and inserting the characters before the specified position
- The += operator describes the most common forms of appending characters to strings

```
string complete_name(const string& first_name, const string& family_name)
{
 string s = first_name;
 s += ´ ´;
 s += family_name;
 return s;
}
```

- Appending at the end can be significantly more efficient than insertion at other positions

```
string complete_name2(const string& first_name, const string& family_name)
                                                          // poor algorithm
{
 string s = family_name;
 s.insert(s.begin() ,´ ´) ;
 return s.insert(0,first_name) ;
}
```

Concatenation

- Concatenation - construction of string from two other strings by placing them next to each other - is a special case of appending. It is available as the + operator

```
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A>
 operator+(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(const Ch*, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(Ch, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, const Ch*) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&,Ch) ;
```

- As usual, + is implemented as a non-member
- The use of concatenation is obvious and convenient

```
string complete_name3(const string& first_name, const string& family_name)
{
 return first_name + ´ ´ + family_name;
}
```

- This notational convenience is obtained at a cost of some runtime overhead compared to complete_name(). The complete_name3() function needs one more temporary variable

Searching

- There is an entire family of search functions

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
 public:
 // ...
 // find subsequence (like search()):
 size_type find(const basic_string& s, size_type i = 0) const;
 size_type find(const Ch* p, size_type i, size_type n) const;
 size_type find(const Ch* p, size_type i = 0) const;
 size_type find(Ch c, size_type i = 0) const;
 // find subsequence searching backwards from the end (like find_end()):
 size_type rfind(const basic_string& s, size_type i = npos) const;
 size_type rfind(const Ch* p, size_type i, size_type n) const;
 size_type rfind(const Ch* p, size_type i = npos) const;
 size_type rfind(Ch c, size_type i = npos) const;
 // find character (like find_first_of()):
 size_type find_first_of(const basic_string& s, size_type i = 0) const;
 size_type find_first_of(const Ch* p, size_type i, size_type n) const;
 size_type find_first_of(const Ch* p, size_type i = 0) const;
 size_type find_first_of(Ch c, size_type i = 0) const;
 // find character from argument searching backwards from the end:
 size_type find_last_of(const basic_string& s, size_type i = npos) const;
 size_type find_last_of(const Ch* p, size_type i, size_type n) const;
 size_type find_last_of(const Ch* p, size_type i = npos) const;
 size_type find_last_of(Ch c, size_type i = npos) const;
```

Searching

```
  // find character not in argument:
  size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
  size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
  size_type find_first_not_of(const Ch* p, size_type i = 0) const;
  size_type find_first_not_of(Ch c, size_type i = 0) const;
  // find character not in argument searching backwards from the end:
  size_type find_last_not_of(const basic_string& s, size_type i=npos) const;
  size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
  size_type find_last_not_of(const Ch* p, size_type i = npos) const;
  size_type find_last_not_of(Ch c, size_type i = npos) const;
  // ...
};
```

- All those methods have the *const* attribute, i.e. allow to locate the substring for some purpose, but they do not modify its value
- The meaning of *basic_string::find()* can be understood based on equivalent generic algorithms

```
void f() {
  string s= "accdcde";
  string::size_type i1 = s.find("cd") ; // i1 = 2 s[2]=='c' && s[3]=='d'
  string::size_type i2 = s.rfind("cd") ; // i2 = 4 s[4]=='c' && s[5]=='d'
  string::size_type i3 = s.find_first_of("cd") ; // i3 = 1 s[1] == 'c'
  string::size_type i4 = s.find_last_of("cd") ; // i4 = 5 s[5] == 'd'
  string::size_type i5 = s.find_first_not_of("cd") ;
                                           // i5 = 0 s[0]!='c' && s[0]!='d'
  string::size_type i6 = s.find_last_not_of("cd") ;
                                           // i6 = 6 s[6]!='c' && s[6]!='d'
}
```

- If *find()* does not find a substring, it returns *npos* representing an illegal character position. If *npos* is used to indicate position,the *out_of_range* exception will be thrown

Replacing

- After identification of the position in the string, a single value can be changed using indexing, ot the entire substring can be replaced with new characters. It can be achieved using *replace()*

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // replace [ (*this)[i], (*this)[i+n] [ with other characters:
 basic_string& replace(size_type i, size_type n, const basic_string& s) ;
 basic_string& replace(size_type i, size_type n,
                      const basic_string& s, size_type i2, size_type n2) ;
 basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2) ;
 basic_string& replace(size_type i, size_type n, const Ch* p) ;
 basic_string& replace(size_type i, size_type n, size_type n2,Ch c) ;
 basic_string& replace(iterator i, iterator i2, const basic_string& s) ;
 basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n) ;
 basic_string& replace(iterator i, iterator i2, const Ch* p) ;
 basic_string& replace(iterator i, iterator i2, size_type n,Ch c) ;
 template<class In>basic_string& replace(iterator i,iterator i2, In j,In j2);
 // remove characters from string (``replace with nothing''):
 basic_string& erase(size_type i = 0, size_type n = npos) ;
 iterator erase(iterator i) ;
 iterator erase(iterator first, iterator last) ;
 // ...
};
```

Replacing

- The number of new characters does not have to be the same, as the number of characters in the original string. The size of string changes, adapting to the new substring
- The *erase()* just removes the substring and adjusts the size of the string accordingly

```
void f()
{
 string s= "but I have heard it works even if you don´t believe in it";
 s.erase(0,4) ; // erase initial "but "
 s.replace(s.find("even") ,4,"only") ;
 s.replace(s.find("don´t") ,5,"") ; // erase by replacing with ""
}
```

- Simple call of *erase()* without any arguments changes the string into an empty string. This operation is called *clear()* in generic collections
- The variety of *replace()* functions correspond to the variety of assignments; *replace()* is just an assignment to a substring

Substrings

- The *substr()* function allows to specify a substring by giving the position and lenght

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // address substring:
 basic_string substr(size_type i = 0, size_type n = npos) const;
 // ...
};
```

Size and Capacity

- The problems related to memory management are handled roughly the same, as for the vector class

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // size, capacity, etc.
 size_type size() const; // number of characters
 size_type max_size() const; // largest possible string
 size_type length() const { return size() ; }
 bool empty() const { return size()==0; }
 void resize(size_type n,Ch c) ;
 void resize(size_type n) { resize(n,Ch()) ; }
 size_type capacity() const; // like vector
 void reserve(size_type res_arg = 0) ; // like vector
 allocator_type get_allocator() const;
};
```

- Call of *reserve(res_arg)* throws a *length_error* exception if *res_arg>max_size()*

Input-output Operations

- Strings are used frequently as the result of input and the source of output

```
template<class Ch, class Tr, class A>
 basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&,
                                           basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&,
                                     const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&,
                                    basic_string<Ch,Tr,A>&,Ch eol) ;
template<class Ch, class Tr, class A>
 basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&,
                                           basic_string<Ch,Tr,A>&) ;
```

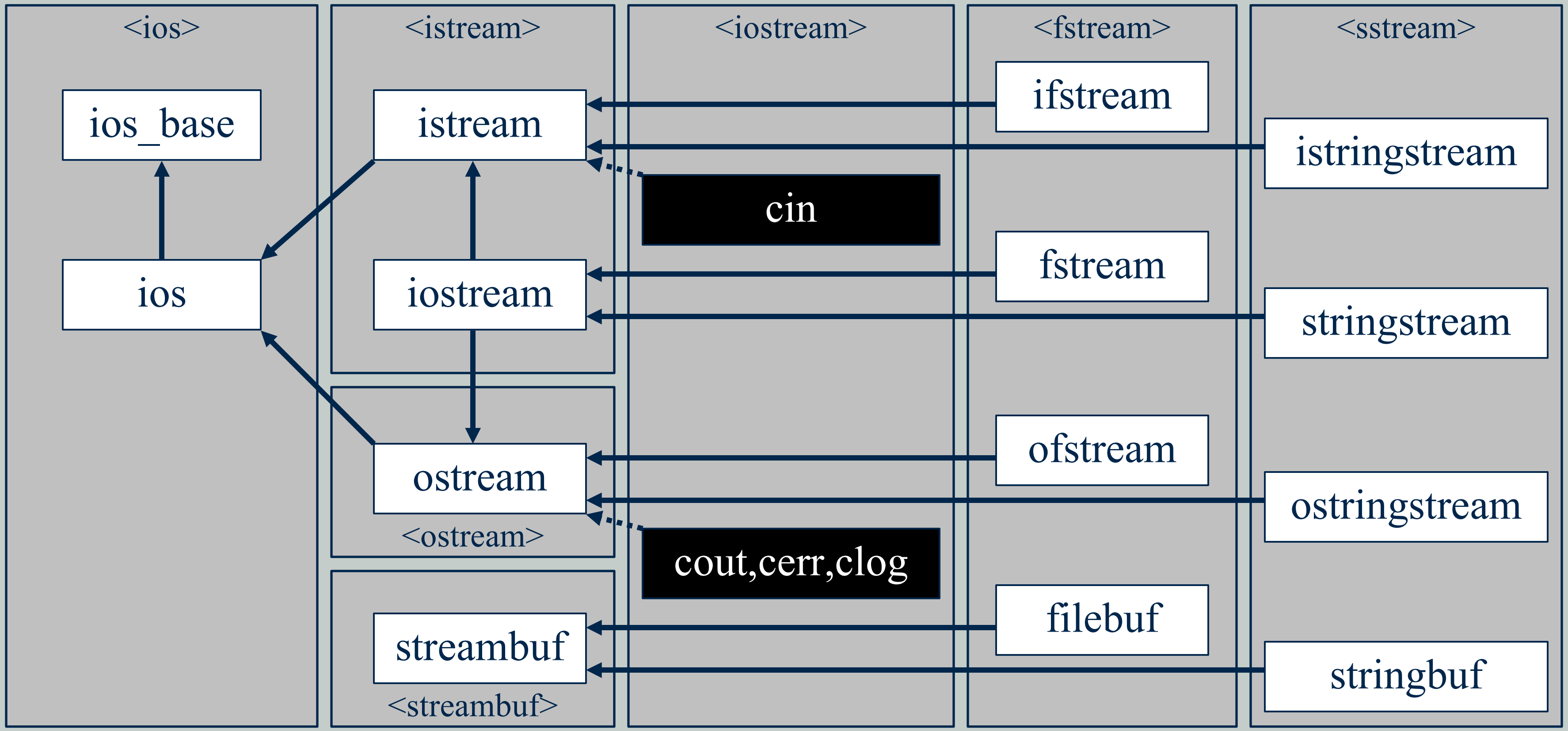
- Operator << prints the string to *ostream*, and >> reads a word terminated with a whitespace to a string, adjusting its size accordingly The leading whitespace is skipped, and the trailing whitespace is not put into the string
- The *getline()* reads into the string the entire line terminated with *eol*, expanding it accordingly. If *eol* is not given, the newline character *'\n'* is used. The terminator is removed from the stream, but not put into the string. As the string is expanded to fit the input, there is no need to leave the terminator in the stream nor provide the count of characters read, like *get()* i *getline()* for character arrays do.

Swap

- Similiarly as for vectors, the *swap()* wersion for strings can be much more effective, than the generic algorithm, so the special version is provided

```
template<class Ch, class Tr, class A>
 void swap(basic_string<Ch,Tr,A>&, basic_string<Ch,Tr,A>&);
```

The Stream Class Structure



Output

- Safe and uniform treatment of both built-in and user-defined types can be achieved using a single overloaded function name for the set of output functions

```
put(cerr,"x= ") ; // cerr is the error output stream
put(cerr,x) ;
put(cerr,´\n´) ;
```

- The more compact form of output operation is overloading of << operator, what gives more concise notation and allows the programmer to put a series of objects in a single instruction

```
cerr << "x= " << x << ´\n´;
```

- The priority of << is low enough to use arithmetic expressions without the need for parentheses

```
cout << "a*b+c=" << a*b+c << ´\n´;
```

- The parnetheses are still necessary to write expression containing operators with the lower priority

```
cout << "a^b|c=" << (a^b|c) << ´\n´;
```

- The << operator can be used in the output statement as the shift-left operator, but it must be put in parentheses

```
cout << "a<<b=" << (a<<b) << ´\n´;
```

Output Streams

- The *ostream* class provides a mechanism to covert values of different types to the strings of characters
- *ostream* is a specialization of the generic *basic_ostream* template

```
template <class Ch, class Tr = char_traits<Ch> >
 class std::basic_ostream : virtual public basic_ios<Ch,Tr> {
 public:
 virtual ~basic_ostream() ;
 // ...
};
```

- Each implementation directly supports streams realized using standard and wide characters

```
typedef basic_ostream<char>    ostream;
typedef basic_ostream<wchar_t> wostream;
```

Output Streams

- The base class *basic_ios* allows to control formatting, locale and buffer access. It also contains the definitions of a few types simplifying the notation

```
template <class Ch, class Tr = char_traits<Ch> >
 class std::basic_ios : public ios_base {
 public:
 typedef Ch char_type;
 typedef Tr traits_type;
 typedef typename Tr::int_type int_type; // type of integer value
                                         // of character
 typedef typename Tr::pos_type pos_type; // position in buffer
 typedef typename Tr::off_type off_type; // offset in buffer
 //...
};
```

- *basic_ios* does not allow copy constructor nor assignments – streams cannot be copied
- The *ios_base* class contains information independent of the character type used – it does not have to be a template

Output Streams

- The stream input/output library, apart from the type definitions from *ios_base*, uses the signed integer type, *streamsize*, to represent the number of characters transferred in the input/output operation and buffer sizes
- *streamoff* is used to express offsets in streams and buffers
- <*iostream*> declares a few standard streams
  - *cerr* and *clog* correspond to *stderr*
  - *cout* corresponds to *stdout*

```
ostream cout; // standard output stream of char
ostream cerr; // standard unbuffered output stream for error messages
ostream clog; // standard output stream for error messages
wostream wcout; // wide stream corresponding to cout
wostream wcerr; // wide stream corresponding to cerr
wostream wclog; // wide stream corresponding to clog
```

Intput Streams

- <*istream*> defines *basic_istream*

```
template <class Ch, class Tr = char_traits<Ch> >
 class std::basic_istream : virtual public basic_ios<Ch,Tr> {
 public:
 virtual ~basic_istream() ;
 // ...
};
```

- <*iostream*> provides two standard input streams

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t>wistream;
istream cin; // standard input stream of char
wistream wcin; // standard input stream of wchar_t
```

Stream State

- Each stream has an associated state. Correct setting and testing of the stream state makes it possible to handle errors and non-standard conditions

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ios : public ios_base {
 public:
 // ...
 bool good() const; // next operation might succeed
 bool eof() const; // end of input seen
 bool fail() const; // next operation will fail
 bool bad() const; // stream is corrupted
 iostate rdstate() const; // get io state flags
 void clear(iostate f = goodbit) ; // set io state flags
 void setstate(iostate f) {clear(rdstate()|f) ;} // add f to io state flags
 operator void*() const; // nonzero if !fail()
 bool operator!() const { return fail() ; }
 // ...
};
```

- If the state is *good()*, the previous operation was successfull, and the next one can succeed, otherwise it will fail
- If an attempt to read data to variable *v* fails, the value of *v* should not change (this is how the standard types handled by *istream* behave)
- There is a very subtle difference difference between the states *fail()* and *bad()*
  - in *fail()* state it is assumed, that the stream is not destroyed and no characters have been lost
  - in *bad()* state there is no such guarantee

Stream State

- The stream state is represented as a set of flags

```
class ios_base {
 public:
 // ...
 typedef implementation_defined2 iostate;
 static const iostate badbit, // stream is corrupted
 eofbit, // end-of-file seen
 failbit, // next operation will fail
 goodbit; // goodbit==0
 // ...
};
```

- The state flags can be manipulated directly

```
void f()
{
 ios_base::iostate s = cin.rdstate() ; // returns a set of iostate bits
 if (s & ios_base::badbit) {
   // cin characters possibly lost
 }
 // ...
 cin.setstate(ios_base::failbit) ;
 // ...
}
```

Stream State

- If the stream is used in place of a condition, the stream state is tested using the *void*()* operator or *operator!()*. The test is successfull, it the state is *!fail()* and *fail()*, respectively
- The generic copy function can be written as follows:

```
template<class T> void iocopy(istream& is, ostream& os)
{
 T buf;
 while (is>>buf) os << buf << ´\n´;
}
```

- The *is>>buf* operation returns the reference to the is stream, tested using *is::operator void*()*

```
void f(istream& i1, istream& i2, istream& i3, istream& i4)
{
 iocopy<complex>(i1,cout) ; // copy complex numbers
 iocopy<double>(i2,cout) ; // copy doubles
 iocopy<char>(i3,cout) ; // copy chars
 iocopy<string>(i4,cout) ; // copy whitespace-separated words
}
```

Unformatted Input

- The >> operator is used for formatted input, i.e. reading the objects of expected type and format. If this is not what we need, as we want to read characters as characters, and analyze them later, we use *get()*

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_istream : virtual public basic_ios<Ch,Tr> {
 public:
 // ...
 // unformatted input:
 streamsize gcount() const; // number of char read by last get()
 int_type get() ; // read one Ch (or Tr::eof())
 basic_istream& get(Ch& c) ; // read one Ch into c
 basic_istream& get(Ch* p, streamsize n) ;
           // newline is terminator, doesn't remove terminator from stream
 basic_istream& get(Ch* p, streamsize n, Ch term) ;
                       // reads at most n-1 characters, terminates with \0
 basic_istream& getline(Ch* p, streamsize n) ;
                 // newline is terminator, removes terminator from stream
 basic_istream& getline(Ch* p, streamsize n, Ch term) ;
 basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof()) ;
 basic_istream& read(Ch* p, streamsize n) ; // read at most n char
 // ...
};
```

Exceptions

- As the checking of correctness of each input/output operation is inconvenient, the frequent source of errors is ignoring this by the programmer
- The only function changing the stream state directly is *clear()*. Using the *exceptions()* function we can make it throw exceptions

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ios : public ios_base {
 public:
 // ...
 class failure; // exception class (see §14.10)
 iostate exceptions() const; // get exception state
 void exceptions(iostate except) ; // set exception state
 // ...
};
```

- The call

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit) ;
```

is a request for the *clear()* function to throw *ios_base::failure()* when *cout* state changes to *bad*, *fail* or *eof*

- Calling *exceptions()* without arguments returns the set of flags, that can cause an exception

Formatting

- Formatting of input/output is controlled by the set of flags and integers from the *ios_base* class

```
class ios_base {
 public:
 // ...
 // names of format flags:
 typedef implementation_defined1 fmtflags;
 static const fmtflags
   skipws, // skip whitespace on input
   left, // field adjustment: pad after value
   right, // pad before value
   internal, // pad between sign and value
   boolalpha, // use symbolic representation of true and false
   dec, // integer base: base 10 output (decimal)
   hex, // base 16 output (hexadecimal)
   oct, // base 8 output (octal)
   scientific, // floating-point notation: d.ddddddEdd
   fixed, // dddd.dd
   showbase, // on output prefix oct by 0 and hex by 0x
   showpoint, // print trailing zeros
   showpos, // explicit '+' for positive ints
   uppercase, // 'E', 'X' rather than 'e', 'x'
   adjustfield, // flags related to field adjustment
   basefield, // flags related to integer base
   floatfield, // flags related to floating-point output
   unitbuf; // flush output after each output operation
 fmtflags flags() const; // read flags
 fmtflags flags(fmtflags f) ; // set flags
 fmtflags setf(fmtflags f) { return flags(flags()|f) ; } // add flag
 fmtflags setf(fmtflags f, fmtflags mask)
                  { return flags(flags()|(f&mask)) ; } // add flag
 void unsetf(fmtflags mask) { flags(flags()&~mask) ; } // clear flags
 // ...
};
```

Formatting

- Defining the interface as the set of flags and operations to set and clear flags is quite an old-fashioned technique. Its main advantage is the possibility to arrange the set of options by the user

```
const ios_base::fmtflags my_opt =
                              ios_base::left|ios_base::oct|ios_base::fixed;
```

- This set of options can be copied and installed as needed

```
void your_function(ios_base::fmtflags opt)
{
 ios_base::fmtflags old_options = cout.flags(opt) ;
                                      // save old_options and set new ones
 // ...
 cout.flags(old_options) ; // reset options
}
void my_function()
{
 your_function(my_opt) ;
 // ...
}
```

- The *flags()* function returns the old set of options

Formatting

- The ability to set and clear all options allow also to set a single flag
- The statement

```
myostream.flags(myostream.flags()|ios_base::showpos) ;
```

  forces display of the + character before every positive number put into *myostream*, not affecting the remaining options. First the old options are read, and then the *showpos* bit is added to this set of options.
- The *setf()* function does exactly the same, so the above example can be rewritten as follows:

```
myostream.setf(ios_base::showpos) ;
```

- The set flags retains its value until explicitly cleared

Copying Stream State

- The full formatting state of the stream can be retrieved using *copyfmt()*

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ios : public ios_base {
 public:
 // ...
 basic_ios& copyfmt(const basic_ios& f) ;
 // ...
};
```

- The function does not copy the stream buffer nor its state. It copies the entire formatting state of the stream, including the requested exceptions and all user-provided extensions.

The Output for Integers

- The technique of adding a new option using the bitwise or with *flags()* or *setf()* works, if the feature is controlled by a single bit
- It is not so for the options defining the base used to print integers and setting the floating-point output style
  - The value specifying the style cannot be represented by a single bit or a set by individual bits
- The solution used in <*iostream*> consists in providing a version of the *setf()* function, which apart from the new value accepts a second argument indicating what kind of option we want to set

The Output for Integers

- The statements

```
cout.setf(ios_base::oct,ios_base::basefield) ; // octal
cout.setf(ios_base::dec,ios_base::basefield) ; // decimal
cout.setf(ios_base::hex,ios_base::basefield) ; // hexadecimal
```

  set the base of integer arithmetic, without affecting other features of the stream formatting state. It is valid till the next change

- Statements

```
cout << 1234 << ´ ´ << 1234 << ´ ´; //default: decimal
cout.setf(ios_base::oct,ios_base::basefield) ; // octal
cout << 1234 << ´ ´ << 1234 << ´ ´;
cout.setf(ios_base::hex,ios_base::basefield) ; // hexadecimal
cout << 1234 << ´ ´ << 1234 << ´ ´;
```

  give the result

```
1234 1234 2322 2322 4d2 4d2
```

- If we want to know, what base was used for each number, we can set *showbase*. If we add the following before the statements above

```
cout.setf(ios_base::showbase) ;
```

  we get:

```
1234 1234 02322 02322 0x4d2 0x4d2
```

Output for Floating-Point Numbers

- The floating-point output is controlled by format and precision
  - General format allows the implementation to choose the format presenting the output in a style, which best represents the value in the available space. The precision specifies the maximum number of digits. Corresponds to the %g option of *printf()*
  - Scientific format presents the value with one digit before the decimal point and an exponent. The precision specifies the maximum number of digits after the decimal point. Corresponds to the %e option of *printf()*
  - Fixed format presents the value as the integer part, followed by the decimal point and fractional part. The precision specifies the maximum number of digits after the decimal point. Corresponds to the %f option of *printf()*
- The format of floating-point output is controlled using the stream state modification function. In particular, one can set the floating-point output format without affecting other parts of stream state.

Output for Floating-Point Numbers

- The statements

```
cout << "default:\t" << 1234.56789 << ´\n´;
cout.setf(ios_base::scientific,ios_base::floatfield) ; // use scientific format
cout << "scientific:\t" << 1234.56789 << ´\n´;
cout.setf(ios_base::fixed,ios_base::floatfield) ; // use fixedpoint format
cout << "fixed:\t" << 1234.56789 << ´\n´;
cout.setf(0,ios_base::floatfield) ; // reset to default (that is, general format)
cout << "default:\t" << 1234.56789 << ´\n´;
```

produce

```
default: 1234.57
scientific: 1.234568e+03
fixed: 1234.567890
default: 1234.57
```

- The default precision (for all formats) is 6, it is controlled by the method of class *ios_base*

```
class ios_base {
 public:
 // ...
 streamsize precision() const; // get precision
 streamsize precision(streamsize n) ; // set precision (and get old precision)
 // ...
};
```

Output for Floating-Point Numbers

- The call of *precision()* influences all the input/output operations on floating-point numbers, till the next call of *precision()*
- The statements

```
cout.precision(8) ;
cout << 1234.56789 << ´ ´ << 1234.56789 << ´ ´ << 123456 << ´\n´;
cout.precision(4) ;
cout << 1234.56789 << ´ ´ << 1234.56789 << ´ ´ << 123456 << ´\n´;
```

produce

```
1234.5679 1234.5679 123456
1235 1235 123456
```

- The values are rounded, not truncated, *precision()* does not influence the output for integers
- The *uppercase* flag decides, if the scientific style uses e, or E to mark an exponent

Field Width

- Frequently we want to fill with text the given area in the output line
  - We want to use n characters and not less
- For this purpose we can specify the field width and the character, used to fill the remaining positions in the output

```
class ios_base {
 public:
 // ...
 streamsize width() const; // get field width
 streamsize width(streamsize wide) ; // set field width
 // ...
};
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
 public:
 // ...
 Ch fill() const; // get filler character
 Ch fill(Ch ch) ; // set filler character
 // ...
};
```

Field Width

- The *width()* function specifies the minimum number of characters, that will be used during the following operation << from the standard library, printing out the numerical value, *bool*, C-style string, a pointer, value of type *string* and *bitset*

```
cout.width(4) ;
cout << 12;
```

  will print the number 12 preceded by two spaces

- The fill character can be set using the *fill()* function

```
cout.width(4) ;
cout.fill(´#´) ;
cout << "ab"; // ##ab
```

- The default fill character is a space, and the default field size is 0, meaning „as many characters, as needed”. The default field size can be restored in the following way:

```
cout.width(0) ; // ``as many characters as needed''
```

Field Width

- The *width(n)* function sets the minimum number of characters. If more is provided, all will be printed. The statements:

```
cout.width(4) ;
cout << "abcdef";
```

Will produce abcdef and not only jedynie abcd

- The *width(n)* call influences only the single following output operation, so the statements

```
cout.width(4) ;
cout.fill(´#´) ;
cout << 12 << ´:´ << 13;
```

will produce ##12:13 and not ##12###:##13

Field Alignment

- The position of characters in the field can be controlled using *setf()*

```
cout.setf(ios_base::left,ios_base::adjustfield) ; // left
cout.setf(ios_base::right,ios_base::adjustfield) ; // right
cout.setf(ios_base::internal,ios_base::adjustfield) ; // internal
```

- The following statements can control the output alignment in the output field, whose size has been defined using *ios_base::width()*, without influencing other formatting options

```
cout.fill(´#´) ;
cout << ´(´;
cout.width(4) ;
cout << -12 << ") ,(";
cout.width(4) ;
cout.setf(ios_base::left,ios_base::adjustfield) ;
cout << -12 << ") ,(";
cout.width(4) ;
cout.setf(ios_base::internal,ios_base::adjustfield) ;
cout << - 12 << ")";
```

- The statements above will print (#-12), (-12#), (-#12)
- By default, the fields are right-aligned

Manipulators

- To protect the programmer from the necessity of setting the stream state using flags, the standard library provides a set of functions to manipulate this state
- The idea behind this is to insert the state-changing operation between read or written objects
- For example, we can explicitly request to flush the output buffer:

```
cout << x << flush << y << flush;
```

- At the appropriate moment the *cout.flush()* function will be called

Manipulators

- This is achieved by defining the version of << operator, which accepts the pointer to the function and calls it

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ostream : virtual public basic_ios<Ch,Tr> {
 public:
 // ...
 basic_ostream& operator<<(basic_ostream& (*f)(basic_ostream&))
   { return f(*this) ; }
 basic_ostream& operator<<(ios_base& (*f)(ios_base&)) ;
 basic_ostream& operator<<(basic_ios<Ch,Tr>& (*f)(basic_ios<Ch,Tr>&)) ;
 // ...
};
```

- In order for this to work, the function must be a standalone function or a static method of appropriate type
- *flush()* is defined as follows

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>& s)
{
 return s.flush() ; // call ostream's member flush()
}
```

Manipulators

- The preceding definitions ensure, that the statement

```
cout << flush;
```

is interpreted as

```
cout.operator<<(flush) ;
```

which causes invocation of

```
flush(cout) ;
```

which in turn calls

```
cout.flush() ;
```

- All this together allows to call *basic_ostream::flush()* using the notation *cout<<flush*

Manipulators

- There is a lot of operations, which should be executed just before or just after the input/output operation, e.g.:

```
cout << x;
cout.flush() ;
cout << y;
cin.noskipws() ; // don't skip whitespace
cin >> x;
```

- When they are written as the separate instructions, the logical relation between the operations is not obvious; therefore the code becomes less readable
- The concept of manipulators allows to insert such operations as *flush()* and *noskipws()* into the list of input/output operations, e.g.:

```
cout << x << flush << y << flush;
cin >> noskipws >> x;
```

- The class *basic_istream* contains the >> operators to invoke manipulators in a similar way as the *basic_ostream* class

Manipulators with Arguments

- Manipulators with arguments can be also useful. We can write e.g.:

```
cout << setprecision(4) << angle;
```

  to print the floating point value *angle* using four digits

- We need to make *setprecision* return an object initialized to 4, which can be called, and which then calls *cout.precision(4)*

- Such a manipulator is a function object, invoked not by (), but by <<. The object type is implementation dependent, but can have the following form:

```
struct smanip {
 ios_base& (*f)(ios_base&,int) ; // function to be called
 int i;
 smanip(ios_base& (*ff)(ios_base&,int) , int ii) : f(ff) , i(ii) { }
};
template<cladd Ch, class Tr>
ostream<Ch,Tr>& operator<<(ostream<Ch,Tr>& os, smanip&m)
{
 return m.f(os,m.i) ;
}
```

- The constructor of the *smanip* stores its arguments in *f* and *i*, and *operator<<* invokes *f(i)*

Manipulators with Arguments

- The *setprecision()* function can be defined as follows:

```
ios_base& set_precision(ios_base& s, int n) // helper
{
 return s.setprecision(n) ; // call the member function
}
inline smanip setprecision(int n)
{
 return smanip(set_precision,n) ; // make the function object
}
```

- Now we can write:

```
cout << setprecision(4) << angle;
```

Standard Manipulators

```
ios_base& boolalpha(ios_base&) ; // symbolic representation of true and false
                                 // (input and output)
ios_base& noboolalpha(ios_base& s) ; // s.unsetf(ios_base::boolalpha)
ios_base& showbase(ios_base&) ; // on output prefix oct by 0 and hex by 0x
ios_base& noshowbase(ios_base& s) ; // s.unsetf(ios_base::showbase)
ios_base& showpoint(ios_base&) ;
ios_base& noshowpoint(ios_base& s) ; // s.unsetf(ios_base::showpoint)
ios_base& showpos(ios_base&) ;
ios_base& noshowpos(ios_base& s) ; // s.unsetf(ios_base::showpos)
ios_base& skipws(ios_base&) ; // skip whitespace
ios_base& noskipws(ios_base& s) ; // s.unsetf(ios_base::skipws)
ios_base& uppercase(ios_base&) ; // X and E rather than x and e
ios_base& nouppercase(ios_base&) ; // x and e rather than X and E
ios_base& internal(ios_base&) ; // adjust
ios_base& left(ios_base&) ; // pad after value
ios_base& right(ios_base&) ; // pad before value
ios_base& dec(ios_base&) ; // integer base is 10
ios_base& hex(ios_base&) ; // integer base is 16
ios_base& oct(ios_base&) ; // integer base is 8
ios_base& fixed(ios_base&) ; // floatingpoint format dddd.dd
ios_base& scientific(ios_base&) ; // scientific format d.ddddEdd
template <class Ch, class Tr>
 basic_ostream<Ch,Tr>& endl(basic_ostream<Ch,Tr>&) ; // put '\n' and flush
template <class Ch, class Tr>
 basic_ostream<Ch,Tr>& ends(basic_ostream<Ch,Tr>&) ; // put '\0' and flush
template <class Ch, class Tr>
 basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>&) ; // flush stream
template <class Ch, class Tr>
 basic_istream<Ch,Tr>&ws(basic_istream<Ch,Tr>&) ; // eat whitespace
smanip resetiosflags(ios_base::fmtflags f) ; // clear flags
smanip setiosflags(ios_base::fmtflags f) ; // set flags
smanip setbase(int b) ; // output integers in base b
smanip setfill(int c) ; // make c the fill character
smanip setprecision(int n) ; // n digits after decimal point
smanip setw(int n) ; // next field is n char
```

File Streams

- This is the full program to copy one file to another. The file names are given as the command-line arguments

```
#include <fstream>
#include <cstdlib>
void error(const char* p, const char* p2= "")
{
 cerr << p << ´ ´ << p2 << ´\n´;
 std::exit(1) ;
}
int main(int argc, char* argv[])
{
 if (argc != 3) error("wrong number of arguments") ;
 std::ifstream from(argv[1]) ; // open input file stream
 if (!from) error("cannot open input file",argv[1]) ;
 std::ofstream to(argv[2]) ; // open output file stream
 if (!to) error("cannot open output file",argv[2]) ;
 char ch;
 while (from.get(ch)) to.put(ch) ;
 if (!from.eof() || !to) error("something strange happened") ;
}
```

File Streams

- The *basic_ofstream* class is declared in *<fstream>* in a following way

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ofstream : public basic_ostream<Ch,Tr> {
 public:
 basic_ofstream() ;
 explicit basic_ofstream(const char* p, openmode m = out) ;
 basic_filebuf<Ch,Tr>* rdbuf() const;
 bool is_open() const;
 void open(const char* p, openmode m = out) ;
 void close() ;
};
```

- As usual, the definitions are provided for the most common types

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
typedef basic_ifstream<wchar_t>wifstream;
typedef basic_ofstream<wchar_t>wofstream;
typedef basic_fstream<wchar_t>wfstream;};
```

File Streams

- The *ifstream* class is similar to *ofstream*, but it is derived from *istream* and open for reading by default. Apart from that, the standard library provides the *fstream* stream, which is similar to *ofstream*, but derived form *iostream* and by default open for reading and writing.
- The constructors of file streams accept the second argument, specifying the opening mode

```
class ios_base {
 public:
 // ...
 typedef implementation_defined3 openmode;
 static openmode app, // append
   ate, // open and seek to end of file (pronounced ``at end'')
   binary, // I/O to be done in binary mode (rather than text mode)
   in, // open for reading
   out, // open for writing
   trunc; // truncate file to 0-length
 // ...
};
```

String Streams

- The stream can be bound to a *string*, i.e. We can read from string and write to it, using the formatting facilities provided by streams. These streams are called *stringstream* and defined in *<sstream>*

```
template <class Ch, class Tr=char_traits<Ch> >
 class basic_stringstream : public basic_iostream<Ch,Tr> {
 public:
 explicit basic_stringstream(ios_base::openmode m = out|in) ;
 explicit basic_stringstream(const basic_string<Ch>& s, openmode m = out|in);
 basic_string<Ch> str() const; // get copy of string
 void str(const basic_string<Ch>& s) ; // set value to copy of s
 basic_stringbuf<Ch,Tr>* rdbuf() const;
};
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

String Streams

- We can use *ostringstream* to format text messages

```
extern const char* std_message[] ;
string compose(int n, const string& cs)
{
 ostringstream ost;
 ost<<"error("<< n << ") " <<std_message[n]<<" (user comment: "<<cs<< ´)´;
 return ost.str() ;
}
```

- We do not have to check for overflow, *ost* is expanded as needed.
- We can give the initial string value in the constructor

```
string compose2(int n, const string& cs)
{
 ostringstream ost("error(",ios_base::ate) ;
 ost << n << ") " << std_message[n] << " (user comment: " << cs << ´)´;
 return ost.str() ;
}
```

String Streams

- The *istringstream* class allows to read input stream from the *string*

```
#include <sstream>
void word_per_line(const string& s) // prints one word per line
{
 istringstream ist(s) ;
 string w;
 while (ist>>w) cout <<w << ´\n´;
}
int main()
{
 word_per_line("If you think C++ is difficult, try English") ;
}
```

- The string being the initializer is copied to *istringstream*. The end of string end the input.