Templates

Templates provide the ability to create generic functions and generic classes. In generic functions (classes), the type of data the function (class) operates on is referred to as a parameter. This means that you can create a function (class) adapted to work with several data types without explicitly programming a new version of the function (class) for each of the individual data types.

Generic functions

- Defines a set of operations to be performed on different data types.

- The data type is passed as a parameter.

- Many algorithms have exactly the same logic for different data types. For example, searching an object in an array (data structure), sorting objects in an array (data structure), adding an object to an array (to a data structure), removing an object from an array (from the data structure) ... .

- The use of generic functions (generic classes) is very efficient when data processing algorithms do not depend on data types. Then the code can be separated into procedures that do not depend on the data type (relate to algorithms) and into procedures that relate to a specific data type (relate to the data type).

- The part of the code that does not depend on the data type can be effectively written as generic functions (generic classes).

- When creating a generic function, a function is created that can overload itself.

- To create generic functions using the keyword *template*. At the same time, a skeleton of functions is created, in which appropriate data types will be substituted during compilation - the compiler itself generates each specific implementation for the given data type

```
template  <class Ttype>  typ_return_value  function_name(list of arguments)
{
   //function body
}
```

- Ttype - data type used by functions. This name can be used inside a function definition. This symbol is a template. All actions of the generic algorithm when executing it in this function will be performed using this symbol, and when creating a specific version of the function, the compiler will replace this template symbol with a real (substituted) data type. The word class may be replaced with *typename* :

```
template <typename Ttype>  typ_return_value  function_name(list of arguments)
{
   //function body
}
```

Example W21.

- A generic function (the definition of this function is preceded by the keyword *template*) is also called a *template function*. For each substituted data type, the compiler creates a specific *specialization* of that function - we say that the compiler creates a *generated function*. The process of generating a function is called *instantiation*.

- The template keyword may be on a different line:

```
template <class T>
void findmydata(T ob, size_t  len)
{
……..
}
```

- Between the template <class T> line and the void *findmydata*……. line no instructions can appear.

- More than one generic type can be used in a template:

```
template <class T1, class T2>  void myfunc(T1 a, T2 b)
{
……..
}
```

- When creating a function template, the compiler generates as many different versions of this function as needed to handle all the different function calls occurring in the program.

- Template functions are similar to overloaded functions, but they are more limited. With function overload, you can make any changes to code statements, template function must execute the same code statements for each version.

➢ Although the template function itself performs an overload as necessary, an explicit overload can be performer:

```
template <class T> void swapargs(T  &a,  T  &b)
{
      T    tmp;
      tmp = a;       // class T  must have an overload of the operator =
      a = b;
      b = tmp;
}

//Here the template function will remain redefined
void swapargs(double &a, double &b)
{
            double t = a;
            a = (a+b)*(a+b);
            b = t;
}

void main()
{
      double a = 10.0, b = 20.0;
      coord c1(0, 0), c2(1,1);
      swapargs(c1, c2);          //call to initial swapargs
      swapargs(a, b);             // call to overloaded swapargs
}
```

Example W22.

- The example provided is not typical. If your program logic is that each version of a function should execute its own algorithm, it is more natural to use overloaded functions. If the algorithm is the same for each version, and the difference is only in the data type, use templates naturally.

Generic classes

- The class contains the definitions of the algorithms it uses, but the data type on which the class operates will be passed to it as a parameter only when the object is created.

- Generic classes are useful when one set of algorithms must be used for different data types.

- Template class declaration:

temlate <*class* Ttype> *class* class_name
{
};

- Ttype - the name of the type that will be specified when creating an object of the class.

➢ Creating an instance of the class:

class_name <*typ*> objekt;

➢ *typ* - this is the data type on which the class is to operate. Functions - members of a generic class, automatically become generic functions, there is no need to precede their name with the *template* keyword when declaring their prototypes.

Example W23.

➢ Class templates provide the ability to create a general form of an algorithm that can be further used to handle any type of data. This deprives the programmer of writing multiple implementations of the same algorithm for different data types.

➢ The class template can have many data types:

```
template <class T1, class T2> class myclass
{
      ……..……
};
```

➢ The default data types associated with the standard data types can be used in the template class:

```
tempalte  <class T1, class T2 = int>  class   my_cl
{
…….....
};

void main()
{
      my_cl< coord, double >  aa(……..);   //T1 ← coord;  T2 ← double
      my_cl <coord>   bb(……….);            //T1 ← coord;  T2 ← int
}
```

Example W24.

- The *typename* keyword has two uses
  - serves as a replacement for the keyword *class*
  - informs the compiler that the name used in the template declaration is for the type name and not for the object name

```
typename X::Name someObj;
```

- X::Name is treated as a name type.

- The *export* keyword can precede template declarations. It gives the possibility to put the template declaration in one file, and the definitions - in another.

- ATTENTION! The Visual Studio compiler does not support the *export* keyword. For compound programs, the template declarations and definitions must be placed in the *.h file, and in those files where this template must be made available, the header file must be included.

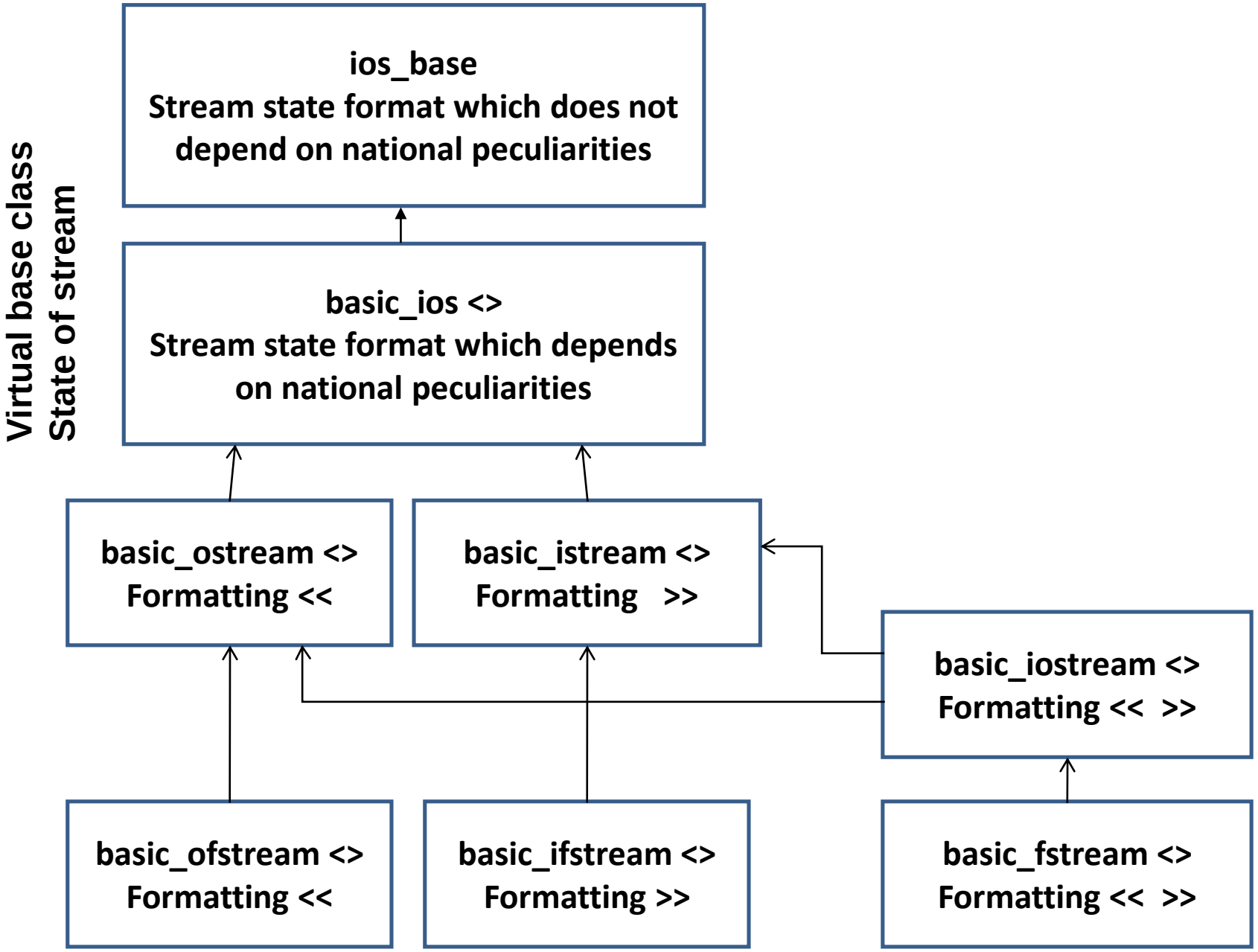
Input-output in C ++

- C ++ supports two complete I / O systems
  - I / O of C based on CRT library.
  - I / O of C++.

- The main advantage of C ++ I / O is the overload operators <<, >> for the classes you create. This makes it possible to build data types created by programmer into I / O systems.
- Stream - is a logical device that produces or retrieves information.
- All I / O streams behave the same - the I / O system presents the same interface to different physical devices. This means, for example, that the same function will output to monitor, file, printer,….
- The streams *stdin*, *stdout*, *stderr* are created when the C program is started.

➢ The I / O library creates two separate versions of the class hierarchy - one is 8-bit and the other is 16-bit.

Template class	Class for 8-bit version
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream


Virtual base class
State of stream
ios_base
Stream state format which does not depend on national peculiarities
basic_ios <>
Stream state format which depends on national peculiarities
basic_ostream <>
Formatting <<
basic_istream <>
Formatting >>
basic_iostream <>
Formatting << >>
basic_ofstream <>
Formatting <<
basic_ifstream <>
Formatting >>
basic_fstream <>
Formatting << >>

- For access to an important ios class, you need to include an <iostream> header.

Formatting of the Input \ Outputs operations

I \ O formatting can be done:

- with the use of *ios* class members;
- with the use of manipulators - special functions that can be placed in expressions concerning I \ O.

Formatting with *ios* class-members

- Each stream has a set of formatting flags that control how the information is formatted. These are bit masks.
- The fmtflags bitmask (*ios_base* class-members - see MSDN) is declared in *ios* class.

- Enumeration *fmtflags*:

adjustfield	**floatfield**	**right**	**skipws**
basefild	**hex**	**scientific**	**unitbuf**
boolalpha	**internal**	**showbase**	**uppercase**
dec	**left**	**showpoint**	
fixed	**oct**	**showpos**	

flag	set	not set
skipws	Leading symbols (spaces, tabs, newline) are ignored on output to the stream	are not ignored
left	Left edge alignment	If all these flags are ignored - defaults
right	Right edge alignment	
internal	Numeric values are padded to occupy the entire field - spaces are inserted between digits and the sign	

flag	set	not set
oct	displaying values in octal system	Default – the decimal system
hex	displaying values in hexadecimal	
dec	restore to the decimal system	----------
showbase	show the basis of the counting system	does not show
uppercase	in scientific notation we display 'E', hexadecimal - 'X'	default: ‘e’, ‘x’
showpos	'+' sign is displayed	is not displayed
showpoint	displays the decimal point and trailing zero everywhere	is not displayed
scientific	floating-point values are displayed in scientific notation 3.141592e + 00	the compiler chooses the appropriate notation by itself
fixed	floating-point values are displayed in ordinary notation 3.141592	
unitbuf	the buffer is flushed after each insertion operation	Is not flushed
boolalpha	logical values can be entered and outputted as false, true	numbers 0, 1
basefield	dec | hex | oct	not

flag	set	not set
adjustfield	**internal | left | right**	**not**
floatfield	**fixed | scientific**	**not**

ios_base class-members:

fmtflags setf(fmtflags flags)	Returns the previous flag setting and sets the listed arguments. *stream*.setf(ios::showpos | ios::scientific); *stream* - the stream to which the flags belong is an object of a class derived from *ios_base*. There is no global formatting in C ++, all formatting is done on a current stream. showpos, scientific - these are enumerated values in the *ios* class, for this the range operator is necessary, otherwise the compiler does not "see" these values.
void unsetf(fmtflags flags)	Flags in the *flags* list are zeroed. All other flags - unchanged.

fmtflags flags();	Returns the current state of the flags, does not change anything in set flags: ios::fmtflags f = cout.flags();

fmtflags flags(fmtflags *f*);	Set any flags defined in the *fmtflags f* template, return the previous setting of the flags:
	
	ios::fmtflags f = ios::showpos | ios::showbase;
	ios::fmtflags f_prev = cout.flags(f);

streamsize width(streamsize w);	By default, when outputting a value, it takes as many positions as it has symbols. The width () functions give the minimum field width w, returns the previous field width. After each output is made, the field width remains set as the default value. The rest of the field (not occupied by symbols) will be filled with the fill symbol (by default - space). If the number of symbols exceeds the set field width, the number of symbols equal to the value will be output.
streamsize precision(streamsize p);	By default, floating-point numbers are output with a precision of 6 digits. The precision can be changed using the precision() function. Returns the previous precision.

Function fill () - a member of a derived class basic_ios:

template <class Elem, class Traits> class basic_ios : public ios_base

char fill(char ch);	By default, empty items are filled with spaces. The fill () function allows you to fill these items with the ch symbol.

Example W25.

I / O Manipulators

Manipulators - these are special functions that can be included in I / O expressions. To access parameter manipulators, you need to add the <iomanip> header.

Standard manipulators:

manipulator	destiny	I \ O
boolalpha	Turns on the *boolalph* flag	I \ O
dec	Turns on the *dec* flag	I \ O
endl	Outputs a newline and empties the stream	O
ends	Outputs null-terminating symbol '\0'	O
fixed	Turns on the *fixed* flag	O
flush	Empties the stream	O

manipulator	destiny	I \ O
hex	Turns on the *hex* flag	I \ O
internal	Turns on the *internal* flag	O
left	Turns on the *left* flag	O
noboolalpha	Turns out the *boolalpha* flag	O
noshowbase	Turns out the *showbase* flag	O
noshowpoint	Turns out the *showpoint* flag	O
noskipws	Turns out the *skipws* flag	O
nounitbuf	Turns out the *unitbuf* flag	O
nouppercase	Turns out the *uppercase* flag	O
oct	Turns out the *oct* flag	I \ O
resetiosflags(fmtflags f)	Disables the flags listed in *f*	O
right	Turns on the *right* flag	O
scientific	Turns on the *scientific* flag	O
setbase(int base)	Assigns the value shown in base to the base of the counting system	I \ O
setfill(int ch)	Defines as a fill sign ch	O
setiosflags(fmtflags f)	Enables the flags listed in *f*	I \ O

manipulator	destiny	I \ O
setprecision(int p)	Sets the precision value	O
setw(int w)	Defines the field width as *w*	O
showbase	Turns on *showbase* flag	O
showpoint	Turns on *showpoint* flag	O
showpos	Turns on *showpos* flag	O
skipws	Turns on *skipws* flag	O
unitbuf	Turns on *unitbuf* flag	O
uppercase	Turns on uppercase flag	O
ws	Skip empty leading characters	I

Example:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << hex << 100  << endl;
    cout << setfill('?') << setw(10) << 2343.0;
}
output:  64
         ??????2343
```

- Manipulators appear in complex expressions and are often found to be more useful than members of the *ios* class.

- If the manipulator takes no arguments (for example, endl, hex), no parentheses are present. This is because it is a function address passed to the overloaded operator << .

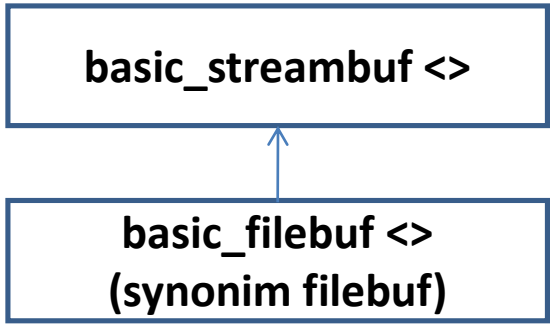
- I / O manipulators only affect the stream for which they are enabled. Other streams open in the program are not affected.

Example W26.

I / O operations, performed on files

- Files have many unique features, so in the C ++ I / O system there are special classes created for working with files ofstream, ifstream, fstream (see the class hierarchies). Header <fstream> is needed.

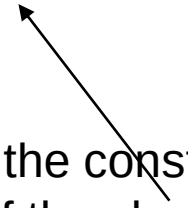
- There is the *basic_filebuf* template class for lower-level management of file streams (binary files handling).



Opening and closing a file.

- When you open a file, you connect it to a stream. There are three types of streams:

  - input (handled by ifstream class)
  - output (ofstream class)
  - input \ output (fstream class)



- Opening the file is produced when the constructor of the appropriate class is called or when the *open*(...) method of the class is explicitly used.

Example W27.

- The open method:

void basic_ifstream::open(const char *_Filename, ios_base::openmode _Mode = ios_base::in, int _Prot = (int)ios_base::_Openprot);

void basic_ofstream::open(const char *_Filename, ios_base::openmode _Mode = ios_base::out, int _Prot = (int)ios_base::_Openprot);

void basic_fstream::open(const char *_Filename, ios_base::openmode _Mode = ios_base::in | ios_base::out, int _Prot = (int)ios_base::_Openprot);

Opening and closing a file.

- The file open mode depends on the value of ios_base openmode _Mode.

```
class ios_base {
public: typedef implementation-defined-bitmask-type openmode;
static const openmode in; static const openmode out; static const openmode ate; static const
openmode app; static const openmode trunc; static const openmode binary; // ... };
```

ios_base::app	Before each write to the file, the file position indicator will be moved to the end of the file (*add* mode at the end of the file)
ios_base::ate	When opening the file, the position indicator will be moved to the end of the file, because writing \ reading can be done anywhere in the file
ios_base::binary	Binary mode. By default, files are opened in text mode. The file, when opened in text mode, "sees" formatting and special characters. The file, opened in binary mode, "does not see" the formatting.
ios_base::in	The file is open for reading (extraction)
ios_base::out	The file is open for writing (insertion)
ios_base::trunc	The contents of the existing file are destroyed, the file will be truncated to zero length while being opened

basic_filebuf::open

ios_base::in	becomes "r" (open existing file for reading).
ios_base::out **ios_base::out | ios_base::trunc**	becomes "w" (truncate existing file or create for writing).
ios_base::out | ios_base::app	becomes "a" (open existing file for appending all writes)
ios_base::in | ios_base::out	becomes "r+" (open existing file for reading and writing).
ios_base::in | ios_base::out | ios_base::trunc	becomes "w+" (truncate existing file or create for reading and writing).
ios_base::in | ios_base::out | ios_base::app	becomes "a+" (open existing file for reading and for appending all writes).

_Prot - the default file opening protection, equivalent to the *shflag* parameter in *_fsopen*(…..).

Unformatted and binary input/output

- In the classic C / C ++ I / O functions realize the byte operations. Since sizeof (*char*) = 1 B, the type *char* is used in the prototypes of these functions.

- We will open files in binary mode (ios_base binary).

- In C ++ Run-time library *put*, *get* functions for binary files save and read one byte of information:

istream & get (char & ch);
ostream & put (char ch);

Example W28.

- C ++ Run-time library *read*, *write* functions for writing / reading blocks:

```
basic_istream  & read( char_type *_Str,   streamsize _Count );
basic_ostream & write( const char_type *_Str,   streamsize _Count );
```

_Str is a pointer to a memory buffer cast to char * or const char *, and _Count gives the number of bytes to be written / read.

If the end of the file is read before the _Count bytes are read, the reading is interrupted and the _Str buffer contains as many bytes as actually was read. To check how many bytes have been read, there is a

```
streamsize basic_istream :: gcount( ) const;
```

- To unload the system I \ O buffer into file, use

```
basic_ostream basic_ostream flush ();
```

We do not use flashing the file unnecessarily - it slows down the operation of the program significantly. When closing the file, the data is automatically unloaded from the system buffer and saved to the file.

- Free (direct) access. In C ++, a file has 2 position indicators - one specifies the file positions where the next input will take place, and the other the output. For the shift of file position indicators we have 2 systems of functions xxxg () (get - read), xxxp () (put - write).

- Moves the file position indicator to the value _Off regarding _Way:

```
basic_istream & basic_istream::seekg   ( off_type _Off, ios_base::seekdir _Way );
basic_ostream& basic_ostream:: seekp ( off_type _Off, ios_base::seekdir _Way );
```

- _Way == ios_base :: beg - regarding the beginning of the file
- _Way == ios_base :: end - regarding the end of the file
- _Way == ios_base :: cur - regarding the current position

➢ Sets the file position indicator to the position given by _Pos

```
basic_istream & basic_istream::seekg  ( pos_type _Pos );
basic_ostream& basic_ostream:: seekp( pos_type _Pos );
```

➢ Indicates the position of the file position indicator:

```
pos_type  basic_istream  :: tellg( );
pos_type  basic_ostream :: tellp( );
```

Example:

```
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   ifstream file;
   char c, c1;

   file.open( "basic_istream_seekg.txt" );
   if(!file) return;
   file.seekg(2);   // chars to skip
   file >> c;
   cout << c << endl;

   file.seekg( 0, ios_base::beg );
   file >> c;
   cout << c << endl;

   file.seekg( -1, ios_base::end );
   file >> c1;
   cout << c1 << endl;
   file.close();
}
```

Text file:
basic_istream_seekg.txt:

0123456789

Output:

2
0
9

- Checking the status of input / output. The current state of the I \ O system is stored in an object of type *ios_base* :: *iostate*:

```
class ios_base {
public:
        typedef implementation-defined-bitmask-type iostate;
        static const iostate badbit;
        static const iostate eofbit;
        static const iostate failbit;
        static const iostate goodbit;
// ... };
```

The type is a bitmask type that describes an object that can store stream state information. The distinct flag values (elements) are:

badbit, to record a loss of integrity of the stream buffer.
eofbit, to record end-of-file while extracting from a stream.
failbit, to record a failure to extract a valid field from a stream.

In addition, a useful value is **goodbit**, where none of the previously mentioned bits are set (**goodbit** is guaranteed to be zero).

- Functions *ios_base::iostate basic_ios:: rdstate*( ) *const*; returns an *iostate* object containing these bit flags.

- failbit = 1 - for example, when entering int, a symbol turned out that cannot be converted correctly to an int type.

- Another way to check:

```
bool  basic_ios::bad( ) const;      //true, if flag badbit is set
bool  basic_ios::eof( )  const;     //true, if flag eofbit is set
bool  basic_ios::fail ( ) const;    // true, if flag failbit is set
bool  basic_ios::good ( ) const;  // true, if flag goodbit is set
```

- reset error flags:

```
void  basic_ios::clear( iostate _State = goodbit );
```

If *goodbit* flag is given as argument, all flags will be cleared. If another flag is specified as an argument, only that flag will be reset.

Example IO_1

Example W29.