Namespaces (1)

As in most cases a software product includes modules developed by different programmers, it is almost impossible to ensure the uniqueness of classnames. To overcome this difficulty, namespaces were introduced:

```
namespace  namespace_name
{
  body
} // semicolon not needed
```

The body may include class declarations, global variable declarations, function declarations, etc. A namespace can be defined in several parts spread over multiple files. Example:

```
namespace TimeDate // file *.h
{
  class Date { …….. };
  class Time { …….. };
  class Timestamp { …….. };
}
namespace TimeDate // file *.cpp
{
  Date::Date(int d, int, m, int y) {…….. }
  ……..
}
```

Namespaces (2)

If we do not specify a namespace, our code is still in a namespace: it is the anonymous global namespace.

Namespaces may be nested: a namespace declaration may contain other namespaces.

The complete names of classes, functions and variables include also the complete list of namespaces separated by scope resolution operator (*::*), for example *TimeDate::Date* or *Coursework::TimeDate::Timestamp*.

```
namespace TimeDate
{
  ..................................................
// In this code section for classes, functions and variables declared in namespace TimeDate
// the complete name is not needed.
// For classes, functions and variables not declared in namespace TimeDate the complete
// name is necessary.
// For classes, functions and variables declared in anonymus namespace the complete
// name starts with ::
}
```

The C++ standard classes are from namespace *std*.

Namespaces (3)

Example:

```
int main()
{
  string abc("ABC"); // error
                     // our code is in anonymous namespace, but C++ standard class
                     // string is from namespace std.
  std::string def("DEF");  //correct
  ……..
}
```

To facilitate the code writing put directive *using* to the beginning of your source code:

```
using namespace namespace_name;
```

or

```
using namespace_name::class_name;
```

Example:

```
using namespace std;
  // if string is the only standard class you need, you may write using std::string
int main()
{
  string abc("ABC"); // now correct
……..
}
```

C++ standard library

Standard classes for:

- Input and output
- String processing
- Exception handling
- Containers (vectors, linked lists, etc.)
- Algorithms for container handling
- Clocks and timers
- Multithreading
- Threads synchronization
- Random numbers
- Complex numbers
- Internationalization
- Regular expressions

I/O streams (1)

To *stdout* (command prompt window): *printf*, *wprintf*

```
printf("%d\n", i);
wprintf (L"%d\n", i);
```

To a stream: *fprintf*, *fwprintf*

```
fprintf(stderr, "%s\n", "Error");
fprintf(stdout, "%s\n", "Error"); // the same as printf("%s\n", "Error");
FILE *pFile = fopen("c:\\temp\\data.txt", "wt+");
fwprintf(pFile, L"%d\n", i);
```

To a memory field: *sprintf*, *swprintf*, *sprintf_s*, *swprintf_s*

```
char pc[20];
sprintf(pc, "%d\n", i);
sprintf_s(pc, 20, "%d\n", i);
wchar_t pwc[40];
swprintf(pwc, L"%d\n", i);
swprintf_s(pwc, 40, L"%d\n", i);
```

If the buffer is too short, *sprintf_s* and *swprintf_s* return empty string, but *sprintf* and *swprintf* crash.

I/O streams (2)

#include <iostream> // obligatory
#include <iomanip> // may be needed for manipulators
cin – global object of class *istream*, reads data from keyboard (more officially, from input console).
cout – global object of class *ostream*, writes data to the command prompt window (more officially, to output console).
cerr – global object of class *ostream*, writes data to the error console (mostly the same as output console).

Class *istream* has operator overloading function operator>>. Class *ostream* has operator overloading function operator<< . Those two functions are for formatted I/O, thus replacing *scanf* and *printf*. The full description of *istream* and *ostream* is on pages:
http://www.cplusplus.com/reference/iostream/
http://www.cplusplus.com/reference/istream/istream/
http://www.cplusplus.com/reference/ostream/ostream/

Examples:

```
int i = 10, j = 20; double d = 3.14159; char *p = "abc";
cout << i; // printf("%d", i);
cout << i << ' ' << d << ' ' << p << endl; // printf("%d %lg %s\n", i, d, p);
                    // possible because the return value of operator<< is ostream&
                    // endl means line feed
```

I/O streams (3)

```
int i = 10, j = 20;
double d = 3.14159;
cout << "i = " << i << " j = " << j << endl; // printf("i = %d j = %d\n", i, j);
cerr << "Unable to open file" << endl; // fprintf(stderr, "Unable to open file\n");
char buf[100];
cout << "Type your name" << endl;
cin >> buf;
```

cin and *cout* support basic types like *char*, *char **, *int*, *long int*, *double*, etc. For more sophisticated formatting use manipulators. Some examples of them:

To get integers in hexadecimal format use *hex*:

```
cout << hex << i << endl; // printf("%x\n", i);
```

The next integers will be also printed as hexadecimal numbers. To return to decimal use manipulator *dec*:

```
cout << dec << j << endl; // printf("%d\n", j);
```

To set the output field width for numerical data use *setw*:

```
cout << setw(6) << i << ' ' << j << endl; // printf("%6d %d\n", i, j);
```

To set the number of decimal places use *setprecision*:

```
cout << setprecision(4) << d << endl; // printf("%.4lg\n", d); we get 3.142
```

I/O streams (4)

To specify the character used for padding use *setfill*:

```
int i = 255;
cout << setfill('0') << setw(6) << i << endl; // printf("%06d\n", i);
```

Class *ostream* has also two functions: *write* to print a block of data and *put* to print just one character:

```
char *p;
cout.write(p, 2); // prints the first 2 characters, no formatting
cout.put(*p); // prints the first character , no formatting
```

Input with *cin* has a problem: whitespace is considered as the end of token:

```
char buf[100];
cout << "Type your name" << endl;
cin >> buf; // types John Smith
cout << buf << endl; // prints John
```

Solution:

```
char buf1[100], buf2[100];
cout << "Type your name" << endl;
cin >> buf1 >> buf2;
cout << buf1 << ' ' << buf2;
```

I/O streams (5)

There is another (and better) solution – use *istream* function *get()*:

```
char buf[100];
cout << "Type your name" << endl;
cin.get(buf, 100); // types John Smith
cout << buf << endl; // prints John Smith
```

Generally:

```
char c, buf[256], delim = ' ';
cin.get(c); // reads the typed character, reading starts when the user has pressed ENTER
if (cin.peek() != EOF) { // \'n' stays in cin, we need to get rid of it.
    cin.get(c);                  // with peek() we can check wether the cin is empty
                                 // peek() returns the first character in cin but does not pop it out
                                 // if the cin is empty, peek() returns constant EOF
}
cin.get(buf, sizeof buf); // reads max (sizeof buf – 1) characters, stores the result as C string
                          // reading starts when the user has pressed ENTER
if (cin.peek() != EOF) { // \'n' stays in cin, we need to get rid of it.
    cin.get(c);
}
cin.get(buf, sizeof buf, delim); // as previous, but reads until delimiter, here until space
do { // delimiter and following to it characters stay in cin
   cin.get(c);
} while (c != '\n');
```

I/O streams (6)

It is more comfortable to use *istream* function *getline()*:

```
cin.getline(buf, sizeof buf); // Reads max (sizeof buf – 1) characters, stores the result as C
// string. Reading starts when the user has pressed ENTER.  '\n' is removed from cin.
cin.get(buf, sizeof buf, delim); // As previous, but reads until delimiter, here until space.
// Delimiter is removed from cin but the following to it characters stay.
```

For our own classes we may write our own *operator>>* and *operator<<* functions. Example:

```
ostream &operator<<(ostream &ostr, const Date &d)
{ // friend, out of classes
  const char MonthNames[12][12] = { "January", "February", "March", "April", "May",
                                    "June", "July", "August",
                                    "September", "October", "November", "December" };
  ostr << d.Day << ' ' << MonthNames[d.iMonth - 1] << ' ' << d.Year << endl;
  return ostr;
}
Date d(11, 3, 2019);
cout << d << endl; // prints 11 March 2019
```

To use Unicode and *wchar_t*, use *wcout* and *wcin*, for example:

```
int i = 10, j = 20;
wcout << L"i = " << i << L" j = " << j << endl; // printf(L"i = %d j = %d\n", i, j);
```

File operations in C (1)

To work with a disk file, our first task is to open it:

FILE <pointer_to_struct_typedefed_as_FILE> = fopen(<filename_as_string_constant>, <mode_as_string_constant>);

Example:

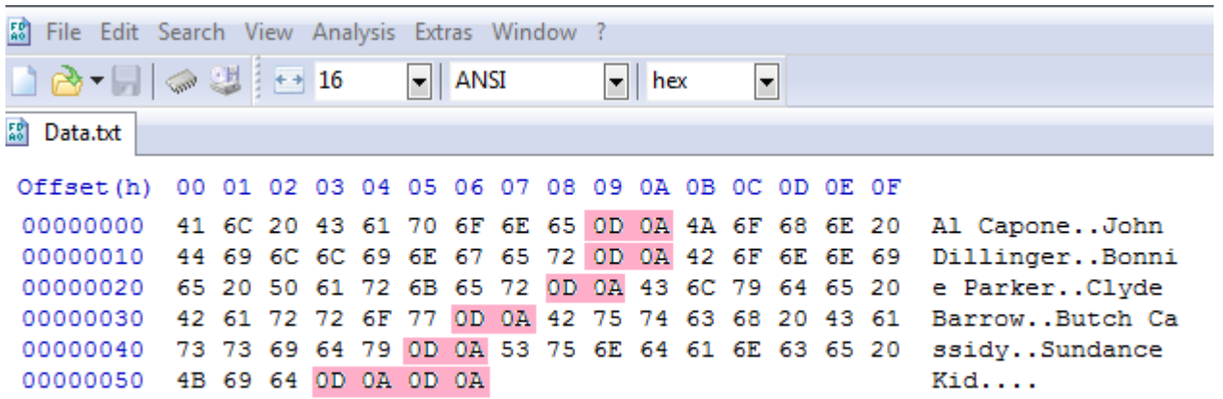
```
FILE *pFile = fopen("c:\\temp\\data.txt", "wt+"); // open text file data.txt for writing and
                                                  // reading
```

struct with typedef name *FILE* is defined in *stdio.h*. We do not need to know its attributes.

To avoid problems specify the complete path to the file. Do not forget that backslash as character constant is '\\'.

Binary files (character *'b'* in mode string) are handled as byte sequences. Text files (character *'t'* in mode string) consist of rows of text. Each row is terminated by two characters: carriage return or CR or *'\r'* (0x0D) and line feed or LF or *'\n'* (0x0A).

To see the contents of file use freeware utility HxD (https://mh-nexus.de/en/hxd/):



File operations in C (2)

The access modes are:

Mode	Access
"r"	For reading only. If the file was not found, *fopen* returns null pointer.
"r+"	For reading and writing. If the file was not found, *fopen* returns null pointer.
"w"	For writing only. If the file was not found, creates it. If the file already exists, deletes its contents.
"w+"	For reading and writing. If the file was not found, creates it. If the file already exists, deletes its contents.
"a"	For writing only. If the file was not found, creates it. If the file already exists, its contents is kept and the new data is appended.
"a+"	For reading and writing. If the file was not found, creates it. If the file already exists, its contents is kept and the new data is appended.

The *fopen* mode string must specify the file type (binary or text) as well as the access mode. Examples: *"rb"*, *"at+"* .

If you have finished the operations with file, close it:

```
fclose(pFile);
```

File operations in C (3)

To write into a file use function *fwrite*:

<number_of_written_items> = fwrite(<pointer_to_data_to_write>, <size_of_data_item>, <number_of_items_to_write>, <pointer_to_FILE_struct>);

Example:

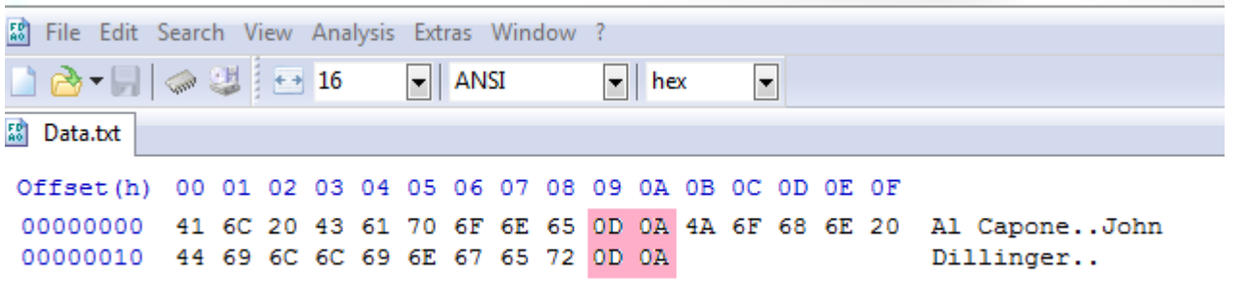
```
#pragma warning (disable: 4996) // for Visual Studio function fopen is unsafe. To
                                // suppress the compiler error message use this pragma
char *pData = (char *)malloc(100);
…… // fills the array with data
FILE *pFile = fopen("c:\\temp\\data.txt", "wt");
if (!pFile)
{  // Good programming practice: check always
    printf("Failure, the file was not open\n");
    return;
}
int n = fwrite(pData, 1, 100, pFile); // 100 characters, one byte each
if (n != 100)
{ // Good programming practice: check always
    printf("Failure, only %d bytes were written\n", n);
}
fclose(pFile);
```

File operations in C (4)

If you want to store a string, remember that to mark the end of row use '\n' only. '\r' will be added automatically.

Example:

```
const char *pData[] = { "Al Capone\n", "John Dillinger\n" };
FILE *pFile = fopen("C:\\Temp\\Data.txt", "wt+");
if (pFile)
{
	fwrite(pData[0], 1, strlen(pData[0]), pFile);
			// Stores without terminating zero.
			// fwrite(pData[0], 1, strlen(pData[0]) + 1, pFile); // with terminating zero
	fwrite(pData[1], 1, strlen(pData[1]), pFile);
	fclose(pFile);
}
```



File operations in C (5)

Function *fwrite* may not store the data immediately. There is an inaccessible for us system buffer and the data is collected into it. The writing is automatically performed when the buffer is full. In this way time is economized. Function *fflush* forces the system to perform the writing immediately:

fflush(<pointer_to_FILE_struct>);

The file has an associated with it inner pointer that specifies the location to where the first written byte will be placed. If the opening mode was *"w"*, then right after opening the pointer points to the beginning of file. If the opening mode was *"a"*, right after opening the pointer points to the first byte after the end of file. After each writing the system shifts the pointer to the byte following the last written byte.

If the opening mode was *"w"*, you may select the location to where the first written byte will be placed or in other words, you may shift the pointer before writing:

fseek(<pointer_to_FILE_struct>, <offset>, <origin>);

Origin is specified by constants defined in file *stdio.h*. They are *SEEK_CUR* (current position), *SEEK_END* (end of file) and *SEEK_SET* (beginning of file). Offset specifies the number of bytes from the origin. Examples:

```
fseek(pFile, 10, SEEK_SET); // put the pointer on the 10-th byte of file
fseek(pFile, -sizeof(struct Date), SEEK_END); // shift the pointer back to overwrite the last
                                              // struct Date
```

If the opening mode was *"a"*, the new data is always appended. Shifting with *fseek* is ignored.

File operations in C (6)

To read from a file use function *fread*:

<number of_read_items> = fread(<pointer_to_buffer_for_read_data>, <size_of_data_item>, <number_of_items_to_write>, <pointer_to_FILE_struct>);

Example:

```
char *pData = (char *)malloc(100);
FILE *pFile = fopen("c:\\temp\\data.txt", "wt+");
if (pFile)
{
    int n = fread(pData, 1, 100, pFile); // 100 characters, one byte each
    if (n != 100)
    { // it may be not a failure, simply there was no data
        printf("Only %d bytes were read\n", n);
    }
    fclose(pFile);
}
```

In case of text files the carriage return – line feed pairs (*"\r\n"*) at the row ends are replaced by line feeds.

Use *fseek* to specify the location of the first byte to read. It is possible in each mode, even in case of *"a+"*. After reading the file pointer is shifted to the first not read byte.

File operations in C++ (1)

For input into files and output from files:
#include <fstream> // http://www.cplusplus.com/reference/fstream/fstream/
fstream File; // File is an object of class fstream

To open file use method open:
open(file_name_string, mode)

Filename: *const char* *, in Windows also *const wchar_t* *.

fstream static members for modes:

1. *app* - set the stream position indicator to the end of stream before each output operation.
2. *ate* - set the stream position indicator to the end of stream on opening.
3. *binary* - consider the stream as binary rather than text.
4. *in* - allow input operations on the stream.
5. *out* - allow output operations on the stream.
6. *trunc* – discard the current content, assume that on opening the file is empty.

To join the modes use bitwise OR. Example:
File.open("c:\\temp\\data.bin", fstream::out | fstream::in | fstream::binary);

To check the success call right after opening method *good()*. It returns 0 (failed) or 1 (success).

To close the file use method *close()*.

File operations in C++ (2)

To operate with text files, *fstream* has overloaded functions *operator>>* and *operator<<* that work like the corresponding functions of *ostream* and *istream*. For example:

```
fstream File;
File.open("C:\\Temp\\data.txt", fstream::out | fstream::in | fstream::trunc); // not binary!
if (!File.good())
{
      return;
}
int arr1[10], arr2[10];
for (int i = 0; i < 10; i++)
    arr1[i] = i;
for (int i = 0; i < 10; i++)
    File << arr1[i] << ' '; // converts into text, in file 0 1 2 3 4 5 6 7 8 9
File.seekg(ios_base::beg); // shifts the position indicator to the beginning, see the next slide
for (int i = 0; i < 10; i++)
    File >> arr2[i]; // converts into integers
for (int i = 0; i < 10; i++)
    cout << arr2[i]; // prints 0123456789
File.close();
```

File operations in C++ (3)

For reading from binary files (i.e. without formatting) use method *read()*.

fstream_object_name.read(pointer_to_buffer, number_of_bytes_to_read);

To check the success use method *good()*. Method *gcount()* returns the number of bytes that were actually read. Example:

```
int arr[100];
fstream File;
File.open("C:\\Temp\\data.bin", fstream::in | fstream::binary);
File.read(arr, 100 * sizeof(int));
if (!File.good())
    cout << "Error, only " << File.gcount() << " bytes were read!" << endl;
```

To shift the reading position indicator use method *seekg()*:

```
int n;
File.seekg(ios_base::beg + n); // n bytes from the beginning
File.seekg(ios_base::end - n); // n bytes before the end
File.seekg(ios_base::cur + n); // n bytes after the current position
File.seekg(ios_base::cur - n); // n bytes before the current position
n = File.tellg(); // returns the current position
```

File operations in C++ (4)

To read from text or binary files byte by byte use method *get()*. Example:

```
int arr[100];
fstream File;
File.open("C:\\Temp\\data.bin", fstream::in | fstream::binary);
for (int i = 0; File.good() && i < 100 * sizeof(int); i++) {
    *((char *)arr + i) = File.get();
}
```

Suppose our text file consists of words separated by space. When a word is retrieved, we want to analyze it immediately:

```
char buf[1024];
fstream File;
File.open("C:\\Temp\\data.txt", fstream::in);
while(1) {
  for (int i = 0;  i < 1024; i++)  {
        if (File.peek() == ' ') // see what is there but do not read out
             break; // jump to analyze the current word
        else
            *(arr + i) = File.get(); // read the next character of the current word
    }
  ……..……..
}
```

File operations in C++ (5)

If the text in file is divided into rows separated by ´\n', we may use method getline():

```
char buf[256], delim = ' ';
File.getline(buf, sizeof buf); // reads max (sizeof buf – 1) characters, stores the result as C
                               // string. '\n' is discarded
File.getline(buf, sizeof buf, delim); // here '\n' is replaced by another delimiter
```

Also, the reading stops when the end of file is reached. Example:

```
while (true)
{
   File.getline(buf, sizeof buf);
   ……..……..  // process the retrieved text
   if (File.eof())
   {
        break;  // end of file is true, stop processing
    }
}
```

You can also check the reading result with methods *good()* and *fail()*:

```
File.getline(buf, sizeof buf);
if (File.fail())
{  // the buffer is full but '\n' or other delimiter was not found
 ……..……..
}
```

File operations in C++ (6)

For writing into binary files (i.e. without formatting) use method *write*.

```
fstream_object_name.write(pointer_to_data_write, number_of_bytes_to_write);
```

To check the success use method *good()*. Example:

```
int arr[100];
fstream File;
File.open("C:\\Temp\\data.bin", fstream::out | fstream::binary);
File.write((char *)arr, 100 * sizeof(int));
if (!File.good())
    cout << "Error, failed to write!" << endl;
```

To shift the writing position inidicator use method *seekp()* (similar to *seekg()*). To get the current location use method *tellp()*.

To write byte by byte use method *put()*:

```
for (int i = 0;  i < sizeof(int) * 100;  i++)
     File.put(*((char *)arr + i));
```

The data to write are accumulated in an inner buffer and will be actually written when the buffer is full, when the stream is closed or goes out of scope. Method *flush()* explicitly tells the stream to write into file immediately:

```
File.flush();
```

C++ standard exceptions (1)

```
#include <exception> // see http://www.cplusplus.com/reference/exception/exception/
try {
   ……..…… // may throw an object of class exception
}
catch(const exception &e) {
  ……..…….. // processing the exception
}
```

Example:

```
try {
   int n;
   cin >> n;
   if (n <= 0)
       throw exception("Wrong length"); // create exception object, specify the error message
    ……..……
}
 catch(const exception &e) {
      cout  << e.what() << endl; // what() returns the error message
      return;
}
```

C++ standard exceptions (2)

We may derive from the standard *exception* class our own exception, adding attributes that describe the abnormal situation. C++ standard presents also some additional classes derived from *exception*, for example *invalid_argument*, *out_of_range*, *system_error*, *overflow_error*, *underflow_error*, etc. Those classes do not introduce new members additional to members inherited from *exception.* Throwing of exceptions of different classes simply help to ascertain the reason of failure without analyzing the text in error message. Example:

```
try {
 …… // some code
 }
 catch(const invalid_argument &e1) {
 …… // do something
}
 catch(const out_of_range &e2) {
 …… // do something;
}
 catch(const exception &e3) { // all the other possible exception types
…… // do something
}
```

C++ standard exceptions (3)

The C++ standard allows to add to function header the list of exceptions that this function may throw, for example:

```
void fun(void *p, int i) throw (out_of_range, invalid_argument)
{           // throw list informs the user about exceptions the function may throw
  if (i < 0)
      throw out_of_range("Failure, index is negative");
  if (!p)
      throw invalid_argument ("Failure, no object");
  ……….
}
```

The throw list is not compulsory. If present, it must be included also into the prototype:

```
void fun(void *, int) throw (out_of_range, invalid_argument);
```

To emphasize that the current function does not throw exceptions, you may replace the throw list with keyword *noexcept*, for example:

```
void fun() noexcept;
```

In Visual Studio the throw list may cause compiler warnings. To suppress them write at the beginning of your file:

```
#pragma warning( disable : 4290 )
```

C++ strings (1)

```
#include <string> // see http://www.cplusplus.com/reference/string/
```

Constructors:

```
string s1("abc"), // s1 contains characters a, b and c
       s2("abc", 2), // s2 contains characters a and b (the first two)
       s3(5, 'a'), // s3 contains 5 characters 'a'
       s4(s1), // s4 is identical with s1
       s5(s1, 1), // s5 contains characters b and c (from position 1)
       s6 = s1,  // copy constructor, s6 is also "abc"
       s7;  // empty string, the alternative is s7("");
```

Examples with memory allocation:

```
string *ps1 = new string("abc"), // ps1 points to string that contains characters a, b and c
       *ps2 = new string(*ps1), // ps2 points to string that contains characters a, b and c
        *ps3 = new string; // ps3 points to empty string, alternative is new string("")
```

In case of Unicode use *wstring*:

```
wstring ws(L"abc"), pws = new wstring(5, L'a');
```

C++ strings (2)

Get string in C format:

```
string s1("abc");
const char *p = s1.c_str();
```

However, if we later change the string *s1*, *p* may start to point to a wrong place. In case of Unicode:

```
wstring ws1(L"abc");
const wchar_t *p = ws1.c_str();
```

Input and output:

```
string s1("abc");
cout << s1<< endl;
cout << s1.c_str() << endl;
wstring ws1(L"abc");
wcout << ws1 << endl;
wcout <<  ws1.c_str() << endl;
string s2;
cin >> s2; // reads until space
std::getline(cin, s2); // reads until ENTER , not a member of a class
wstring ws2;
wcin >>  ws2;
std::getline (wcin, ws2);
```

C++ strings (3)

Prototypes of functions for conversions from string:

```
int std::stoi(string_or_wstring);  // but stou returning unsigned int is not defined
long int std::stol(string_or_wstring);
long long int std::stoll(string_or_wstring);
unsigned long int std::stoul(string_or_wstring);
unsigned long long int std::stoull(string_or_wstring);
float std::stof(string_or_wstring);
double std::stod(string_or_wstring);
```

In case of failure those functions throw *invalid_argument* or *out_of_range* exception.
Example:

```
cout << "Type the length of array" << endl;
string s;
int n;
getline(cin, s);
try {
    n = std::stoi(s);
}
catch (const exception &e) { // absolutely needed, the human operator may hit a wrong key
     cout << "Wrong" << endl;
}
```

C++ strings (4)

Prototypes of functions for conversions to string:
string std::to_string(argument);
wstring std::to_wstring(argument);
The argument may be any integer, float or double.

Capacity:

```
string s1("abc");
int n = s1.length(); // number of characters in string
if (s1.empty())
{…… } // true if no characters in string
s1.clear(); // s1 is now empty string
```

Access:

```
string s1("abc");
char c1 = s1.at(0); // c1 gets value 'a'. If index is out of scope, throws out_of_range exeption
char c2 = s1[0]; // c2 gets value 'a'. If index is out of scope, the behavior is undefined
s1[0] = 'x'; // s1 is now "xbc"
s1[3] = 'y'; // error, corrupts memory
char c3 = s1.front(); // the first character
char c4 = s1.back(); // the last character
```

C++ strings (5)

Arithmetics:

```
string s3 = s1 + s2; // s3 is "abcdef"
s3 += s1; // s6 is now "abcdefabc"
s1 += "y"; // get "abcy", "y" is automatically converted to object of class string
```

Comparisons:

```
if (s1 == s2) // also !=, >, <,  >=, <=
{……….……}
```

Example:

```
string name;
if (name != "John")  // automatically converts C string constant to string object
    cout << "Unknown person " << name << endl;
```

Another option is to use function *compare*:

```
int i = s1.compare(s2);
if (i == 0)
    cout << "s1 and s2 are identical" << endl;
else if (i < 0)
    cout << "s1 is less than s2" << endl;
else
    cout << "s1 is greater that s2" << endl;
```

C++ strings (6)

Find:

```
int position = find(character_to_find, position_to_start_search);
int position = find(pointer_to_C_string_to_find, position_to_start_search);
int position = find(reference_to_string_to_find, position_to_start_search);
```

If nothing was found, the return value is *string::npos*. Examples:

```
string s("abcdef"), s1 = string("de");
const char *pBuf = "cd";
cout << s.find('e', 0) << endl; // prints 4
cout << s.find(pBuf, 0) << endl; // prints 2
cout << s.find(s1, 0) << endl; // prints 3
if (s.find("klm", 0) == string::npos)
    cout << "not found" << endl;
```

Function *find* is to search the first occurrence. With function *rfind* we may get the last occurrence.

```
int position = find_first_of(pointer_to_C_string_of_characters_to_find,
                                        position_to_start_search);
int position = find_first_not_of(pointer_to_C_string_of_characters_to_find,
                                            position_to_start_search);
string s("ka3djvn5po9gn");
cout << s.find_first_of("0123456789", 0) << endl; // prints 2 – the position of the first digit
cout << s.find_first_not_of("0123456789", 0) << endl; // prints 0 – the first that is not digit
```

C++ strings (7)

Copy the contents into buffer:

```
copy(pointer_to_destination_buffer, number_of_bytes_to_copy,
        position_of_the first_character_to_copy);
```

Example:

```
string s("abc");
char buf[10];
s.copy(buf, 2, 0);
buf[2] = 0; // to get a C string, we have to append the terminating zero ourselves
cout << buf << endl; // prints "ab"
```

Cut a section:

```
substr(position_of_the_first_character, length);
```

returns a string consisting of the specified section of original string.

Example:

```
string name; // first name, middle name, last name like John Edward Smith
int n1 = name.find(' ', 0);
int n2 = name.find(' ', n1 +1);
cout << "The middle name is " << name.substr(n1, n2 – n1) << endl;
```

C++ strings (8)

Insert:

```
insert(position_to_insert, reference_to_string_to_insert);
insert(position_to_insert, pointer_to_C_string_to_insert);
```

The additional characters will be inserted right before the indicated position. Example:

```
string name("John Smith");
name.insert(5, "Edward ");
cout << "The complete name is " << name << endl; // prints John Edward Smith
```

Replace:

```
replace(position_to_start_replacing, number_of_characters_to_replace,
        reference_to_string_that_replaces);
replace(position_to_insert, position_to_start_replacing, number_of_characters_to_replace,
        pointer_to_C_string_that_replaces);
```

The number of characters that replace the specified section may be any. Examples:

```
string name("John Edward Smith");
name.replace(5, 7, "");
cout << name << endl; // prints John Smith
name.replace(5, 0, "Edward ");
cout << name << endl; // prints John Edward Smith
```

C++ strings (9)

Erase:

erase(position_to_start_erasing, number_of_characters_to_erase);

Example:

```
string name("John Edward Smith");
name.erase(5, 7);
cout << name << endl; // prints "John Smith"
```

String streams (1)

#include<sstream> // see http://www.cplusplus.com/reference/sstream/stringstream/

String output streams format data exactly as ordinary output streams. But instead of immediate output they store the formatted data in a string allowing to output them later.
Example:

```
stringstream sout; // not a predefined object
int nError;
……..
sout << "Failure, error is " << nError << endl; // resulting string is stored in sout
sout << "Press ESC to continue, ENTER to break" << endl;
                                                    // appended to the contents of sout
```

Thus, with string stream we can collect a longer text. To get the string stored in *sout* use method *str()*, for example:

```
cout << sout.str(); // prints the result
```

Method *str()* with argument of type string replaces the current contents of *sout*:

```
string name("John Smith");
sout.str(name); // sout contains only text "John Smith"
sout.str(""); // clears the contents, the argument is implicitly converted to string object
```

String streams (2)

String input streams are useful for parsing. Example:

```
void fun(string name) // name like "John Smith"
{
   string first_name ="", last_name = "";
   stringstream name_stream(name);
   name_stream >> first_name >> last_name;
            // now first_name is "John", last_name is "Smith"
   ……..
            // if the last name is not present, last_name remains empty
}
```