Struct in C++

For C++, the only difference between *struct* and *class* is that by default in *struct* the members are *public* and in class *private*.

```
class Date
{
    int Day, // no access modifier, consequently private
    iMonth,
    Year;
    Date() { } // error – private constructor is useless
    ……….
};
struct Date
{
    int Day, // no access modifier, consequently public
    iMonth,
    Year;
    Date() { }
    ……….
};
```

In practice, however, the *struct* is used for small classes containing only attributes. Or in other words – *struct* in C++ has the same meaning as in C.

Copy constructor (1)

```
class Date
{
   int Day, iMonth, Year;
   public: Date(int, int, int);
    ……..……..
};
void PrintDate(Date d)
{
  printf("%d-%d-%d\n", d.GetDay(), d.GetMonth(), d.GetYear());
}
```

Each class has default copy constructor that copies byte by byte from original object into the new object:

```
Date d1(27, 5, 2019);
Date d2 = d1; // d2 is created by default copy constructor
Date *pd3 = new Date(27, 5, 2019);
Date d4 = *pd3; // d4 is created by copy constructor, pd3 points to the original
PrintDate(d1);  // argument d is created by copy constructor from d1
PrintDate(*pd3); // argument d is created by copy constructor, pd3 points to the original
PrintDate(Date(27, 5, 2019)); // contructs an object without name
                                    // this nameless object is the original in creating d
```

Copy constructor (2)

```
class Date
{
   int Day;
   char *pMonth = 0;
   int Year;
public:
    Date(int, const char *, int);
    ~Date() { if (pMonth) delete pMonth; }
……….……….……….……….
};
```

Let:

```
static Date d1(27, "May", 2019); // global lifetime
Date d2 = d1; // local lifetime
```

Problem: as the default copy constructor copies attribute by attribute, the pointers *d1.pMonth* and *d2.pMonth* point to the same memory field. Consequently, when *d2* as a local variable is deleted, *d1* looses its value for attribute *pMonth*.

To overcome this and similar problems, we have to write our own copy constructor.

Copy constructor (3)

```
class Date
{
  int Day;
  char *pMonth = 0;
  int Year;
public:
        Date(int, const char *, int);
        ~Date() { delete pMonth; }
        Date (const Date &Original)
        {  // overloads the default copy constructor
                Day = Original.Day; Year = Original.Year;
                int n;
                pMonth = new char[n = strlen(Original.pMonth) + 1];
                strcpy_s(pMonth, n, Original. pMonth);
        }
        ……..
};
Date d2 = d1; // When the copy constructor is working, Original is the synonym of d1;
                  // Day, Year and pMonth are the members of d2.
```

Pointer *this*

By default, each class has a member called as *this*. It is the pointer which points to the current object itself. For example:

```
class Date
{
    ……..
    Date (const Date &Original)
    {  // overloads the default copy constructor
        *this = Original; // At first copy everything, then modify
                          // Default assignment operator (discussed later) is applied
       int n;
        pMonth = new char[n = strlen(Original.pMonth) + 1];
        strcpy_s(pMonth, n, Original.pMonth);
    }
    int GetDay() { return this->Day; }
     // some programmers mark all the methods and variables that are class members
     // with this->. Reason: they want to distinguish the local variables from class
     // members
    ……..
};
```

Friends (1)

Let us have class Date and the similar to it class Time:

```
class Time {
  int Hour, Min, Sec;
public:
  Time(int, int, int);
  ……….……
};
```

Let us have also class Timestamp:

```
class Timestamp {
Date date; // aggregation
Time time;
public:
Timestamp();
~Timestamp();
Timestamp(int d, int mn, int y, int h, int m, int s) : date(d, mn, y), time(h, m, s) { }
void PrintTimestamp() {
      printf("%02d-%d-%d %02d:%02d:%02d\n", date.GetDay(), date.GetMonth(),
            date.GetYear(), time.GetHour(), time.GetMinutes(), time.GetSeconds());
      }
};
```

Friends (2)

If the author of all these 3 classes is the same programmer, then in class *Timestamp* he would like to access the members of *Date* and *Time* directly. In C++ it is possible if classes *Time* and *Date* are declared as friend classes of *Timestamp* :

```
class Time {
   ……….
   friend class Timestamp;
};
class Date {
   ……….
   friend class Timestamp;
};
```

Now:

```
class Timestamp {
  ……….
void PrintTimestamp() { // accessor functions not needed
      printf("%02d-%d-%d %02d:%02d:%02d\n", date. Day, date.iMonth,
               date. Year, time. Hour, time.Min, time. Sec);
      }
};
```

Friends (3)

If class A declares that class B is its friend, class B has free access to all the members of class A. But it does not mean that A can also access non-public members of B. Here classes *Time* and *Date* allow class *Timestamp* to work with its private attributes. As *Timestamp* has not declared friendship with Time and Date, those classes have no free access to *Timestamp* private and protected members.

Friendship is not inherited. Also, if B declares that C is its friend, C has access to non-public members of B but not to non-public members of A.

A class may also declare that a function out of classes is its friend. Example:

```
class Time; // put it at the beginning of file Date.h to explain the compiler
            // the meaning of word Time
class Date {
……….
friend void PrintTimestamp(Date *, Time *);
};
class Date; // put it at the beginning of file Time.h to explain the compiler
            // the meaning of word Date
class Time {
……….
friend void PrintTimestamp(Date *, Time *);
};
```

Friends (4)

Now function *PrintTimestamp* has access to all the members of classes *Date* and *Time*:

```
void PrintTimestamp(Date *pd, Time *pt)
{
   printf("%02d-%d-%d %02d:%02d:%02d\n", pd->Day, pd->iMonth, pd->Year,
                                          pt->Hour, pt->Min, pt->Sec);
}
```

Usage:

```
Date d(8, 3, 2019);
Time t(11, 3, 56);
PrintTimestamp(&d, &t);
```

Operator overloading (1)

```
class complex
{
  public:  double Re, Im; // real part and imaginary part
           complex(double d1 = 0, double d2 = 0) { Re = d1;  Im = d2; }
           complex operator+(complex &c)
                    { return complex (Re + c.Re, Im + c.Im); }
           int operator==(complex &c)
                    { return Re == c.Re && Im == c.Im ? 1 : 0; }
           complex operator!() { return complex(Re, -Im); }
           ……..
};

complex x(5, 6), y(1,2); // x = 5 + j6, y = 1 + j2
complex z1 = x + y; // Actually z1 = x.operator+(y);  we get z1 = 6 + j8.
           // When the operator method is working, c is the synonym of y; Re and
           // Im are the members of x.  The return value is a new nameless
           // complex  number. From it the default copy constructor creates z1.
if (x == y) // actually x.operator==(y)
   printf("Equal\n");
complex z2 = !x;  // actually z2 = x.operator!(); we get z2 = 5 - j6 (conjugate of x)
```

Operator overloading (2)

Alternative solution:

```
class complex {
  public:  double Re, Im;
           complex(double d1 = 0, double d2 = 0) { Re = d1;  Im = d2; }
           friend complex operator+(complex &, complex &);
           friend int operator==(complex &, complex &);
           friend complex operator!(complex &);
           ……..
};
complex operator+(complex &a, complex &b) {
    return complex(a.Re + b.Re, a.Im + b.Im);
}
int operator==(complex &a, complex &b) {
     return (a.Re == b.Re && a.Im == b.Im) ? 1 : 0;
}
complex operator!(complex &a) {
  return complex(a.Re, -a.Im);
}
```

Operator overloading (3)

```
complex x(5, 6), y(1,2); // x = 5 + j6, y = 1 + j2
complex z1 = x + y; // actually z1 = operator+(x, y);  we get z1 = 6 + j8
                    // complex operator+(complex &a, complex &b) {
                    // return complex(a.Re + b.Re, a.Im + b.Im); }
                    //
                    // When the operator method is working, a is the synonym of
                    // x and b is the synonym of y. The return value is a new
                    // nameless complex  number. From it the default copy
                    // constructor creates z1.
if (x == y) // actually operator==(x, y)
     printf("Equal\n");
complex z2 = !x;  // actually z2 = operator!(x); we get z2 = 5 - j6
```

Operator overloading (4)

It is not possible to:

1. Introduce new operators not specified in C++ standard.
2. Change the priorities.
3. Overload the *sizeof* operator, the scope resolution operator (*::*), the conditional operator (*?:*) and the member selection operator (.).

Overloading of operators like *new*, *delete*, function call (*( )*), array element reference (*[ ]*), comma (,), assignment (=) and type cast may be tricky.

Let us take class *Date*:

Date d1(20, 10, 2019); // constructor called
Date d2 = d1; // default copy constructor called
Date d3; // constructor without arguments called
d3 = d1; // here we need operator overloading function for assignment

Each class has default assignment overloading function providing byte-by-byte copy. Rather often it is not acceptable and we have to write our own assignment overloading function replacing the default one.

Operator overloading (5)

Let us have:

```
class Date {
   int Day;
   char *pMonth = 0;
   int Year;
public:
    Date() { }
    Date(int, const char *, int);
    ~Date() { if (pMonth) delete pMonth; }
……….
};
```

Let:

```
Date *pd1 = new Date(8, "March", 2019);
Date *pd2 = new Date; // constructor without arguments called
*pd2 = *pd1; // default assignment overloading function called
…….
delete pd1;
```

Problem: as the default assignment overloading function copies attribute by attribute, two objects of class *Date* share common memory field for month.. Consequently, deleting one of them corrupts the other.

Operator overloading (6)

```
Date &Date::operator =(const Date &Right)  // here & - specifies the reference type
{
  if (this == &Right) // here & - address operator
      return *this; // necessary for expressions like d1 = *pd where pd points to d1
   Day = Right.Day;  Year = Right.Year;
   if (pMonth)
      delete pMonth;
   int n;
   pMonth = new char[n = strlen(Right.pMonth) + 1];
   strcpy_s(pMonth, n, Right. pMonth);
   return *this;
}
```

d1 = d2; // actually d1.operator=(d2);

i.e. *this* points to *d1* and *Right* is the synonym of *d2*

d1 = d2 = d3; // d1 = d2.operator=(d3) → d1.operator=(d2.operator=(d3));

Therefore void Date::operator=(Date &Right) {…} does not work – the operator= function must return the object.

Operator overloading (7)

```
class Date {
private:    int Day, Year;
            char *pMonth = 0, *pText = 0;
public:     ……….
            operator char *()     // operator function to overload type casting
                                  // no return value, the word "operator" is followed
                                  // by the new type specifier
            {
               pText = new char[64];
               sprintf_s(pText, 64, "%d %s %d", Day, pMonth, Year);
               return pText;
            }
};
Date d (27, "May", 2020);
if (strcmp(d, "28 June 2020")) {
        // actually the operator char *() function associated with object d is called
   printf(“Do not match\n”);
}
printf("%s\n", (char *)d);
```

Static members(1)

```
class Base
{
public:     static int Counter;   // declaration, but definition for initialization is also needed
            Base() { Counter++; }
            ~Base() { Counter--; }
};
int Base::Counter = 10; // definition, must be outside of functions and class declarations
                                    // applicable to public, protected and private members
Base b;
printf("%d\n", b.Counter);  // not recommended
printf("%d\n", Base:: Counter); // correct
```

Static attributes get memory only once. They are shared between all the objects of that class and also objects of classes derived from that class. The static attributes exist even when there are no any objects defined yet.

```
class Derived : public Base {……..};
Derived d;
printf("%d\n", d.Counter); // not recommended
printf("%d\n", Derived::Counter); // correct
```

Here *Counter* presents the current total number of objects of class *Base* plus objects of class *Derived*.

Static members(2)

```
class Base
{
private:    static int Counter;
public:      Base() {Counter++; }
             ~Base() {Counter--; }
             static int GetCounter() { return Counter; }
};
int Base::Counter=0; // although private
class Derived : public Base {……..};
Derived d;
printf("%d\n", d.GetCounter()); // not recommended
printf("%d\n", Derived::GetCounter()); // correct
printf("%d\n", Base::GetCounter()); // correct
```

Static functions of a class cannot operate with non-static members of that class. They can be called even when there are no any objects defined yet.

All the non-static functions have access to any of the static members, the restrictions depend only on the access specifiers (*public*, *private*, *protected*).

Constant members

The value of static or non-static value attribute may be declared as constant. In that case they must be initialized right in the declaration. Later changes, of course, are not possible.

Example:

```
class Date
{
 ……….
 const char MonthNames[12][4] = {
                                    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
                                  };
 ……….
};
```

Constant objects (1)

Let us have

```
class Test
{
private:
    int Value;
    int Counter;
public:
   Test(int i) { Value = i; Counter = 0; }
   void SetValue(int i) { Value = i; }
   int GetValue() { return Value; }
};
```

and

```
const Test *pt = new Test(1);
```

In that case:

```
pt->SetValue(5); // error: the object and consequently its attributes are constants
int i = pt->GetValue(); // also error!
```

If an object is constant, it is not possible to call methods associated with it.

Constant objects (2)

Solution:

```
int GetValue() const { return Value; } // const member function
```

Now:

```
const Test *pt = new Test(1);
int i = pt->GetValue(); // works
```

But the *const* member function cannot so simply change the state of object:

```
int GetValue() const { Counter++; return Value; } // compile error
```

To solve the problem cast the *this* pointer to non-const:

```
int GetValue() const
{
  ((Test *)this)->Counter++;
   return Value;
 }
```

or better

```
int GetValue() const
{
  (const_cast<Test *>(this))->Counter++;
   return Value;
 }
```

Casts (1)

The traditional explicit C cast (*new type) expression* is still in use:

```
double d = 5.6;
int i;
i = (int)d;
```

C++ has 4 new casting operators:

static_cast <*new type*> (*expression*)
dynamic_cast <*new type*> (*expression*)
reinterpret_cast <*new type*> (*expression*)
const_cast <*new type*> (*expression*)
Turn attention that the expression is always in parentheses.

The C-style cast is suitable for conversions between primitive data types. For conversions between pointers the C++ new casting operators are preferred.

Generally, the *static*, *reinterpret* ja *const* casts do the same as the C-style cast but allow more control over how the conversion should be performed. They are also easier to find in the source code.

Dynamic cast correctness is checked during run-time.

Casts (2)

The *static_cast* checks a bit more that C-style cast and is therefore more secure.

```
double d = 5.6;
int i;
i = static_cast<int>(d) // the same as i = (int)d;
class Base { ….. };
class Derived : public Base { ….. };
Derived *pd = new Derived;
Base *pb = pd; // implicit cast
pd = pb; // compile error, implicit cast not allowed
pd = (Derived *)pb; // legal, but also a possible source of run-time errors
pd = static_cast<Derived *>(pb); // legal and possible source of run-time errors
```

But

```
class Class1 {……};
class Class2 {……};
Class1 *pc1 = new Class1;
Class2 *pc2 = new Class2;
pc2 = (Class2 *)pc1; // legal, but also a source of run-time errors
pc2 = static_cast<Class2 *>(pc1); // compile error, static cast not allowed
```

The *static_cast* checks whether the pointer and pointee data types are compatible.

Casts (3)

The *reinterpret_cast* checks nothing and allows to cast a pointer to any other type of pointer (exactly as C-style cast):

```
class Class1 {……};
class Class2 {……};
Class1 *pc1 = new Class1;
Class2 *pc2 = new Class2;
pc2 = (Class2 *)pc1; // legal, but also a source of run-time errors
pc2 = reinterpret_cast<Class2 *>(pc1); // legal, but also a source of run-time errors
```

Using the *reinterpret_cast* instead of C-style cast the programmer emphasizes that he knows about the possible risks. If the program has crashed, places where *reinterpret_cast* (they are easy to find) is used are good start points for searching the bugs.

Casts (4)

The *const_cast* is used to convert a constant to non-constant. Example:

```
void alien (char *); // a third-party function we have to use
void fun (const char *p)
{  // our function, by specification its argument must be const char *
  ……….
  alien(p); // compile error
  alien((char *)p); // legal, but may crash if p points to a string constant
  alien(const_cast<char *>(p)); // legal , but may crash if p points to a string constant
  ……..
}
fun("I am John"); // crashes when function alien tries to change this text
```

Generally, if you try to change a value declared as *const*, the behavior is undefined but mostly the program crashes.

```
char *pc = new char[10];
strcpy(pc, "I am John");
const char *cpc = pc;
fun(cpc); // works because cpc points to memory field that is not constant
```

Casts (5)

The *const_cast* is safer because it can adjust the qualifier but not change the underlying type:

```
class Class1 {……};
class Class2 {……};
Class1 c1;
const Class1 *pc1 = &c1;
Class2 *pc2 = const_cast<Class2 *>(pc1); // compile error, pc1 is from different type
```

Casts (6)

The *dynamic_cast* provides pointers run-time check (not compile-time as the other casts) on casts within an inheritance hierarchy.

```
class Base
{
  virtual void base_fun(); // the hierarchy must contain at least one virtual method
   ……..
};
class Derived : public Base { ….. };
Derived *pd;
Base *pb = new Base;
pd = static_cast<Derived *>(pb); // legal and possible source of run-time errors
pd = dynamic_cast<Derived *>(pb); // no compile error but when the program
                                  // runs, the result is null-pointer
if (!pd)
{
   ……..
}
pb = dynamic_cast<Base *>(pd); // legal, no any errorrs
```

If the hierarchy does not contain virtual functions, a compile error will follow.

New variable types (1)

In C any variable of any type is interpreted as false if its value is zero and as true if its value is not zero. This is still correct in C++.
To improve the readability of code, preprocessor definitions like

```
#define TRUE  1
#define FALSE 0
```

are used. In C++ there is an additional built-in type: bool

```
bool b1 = true, b2 = false;
```

Actually, b1 is stored as integer 1 and b2 as integer 0. Boolean variables are implicitly (i.e. automatically) converted into integers and vice versa:

```
int i = b1; // i is now 1
b1 = 10; // b1 is now true
```

Examples of usage:

```
while (b1 == true) {……}
while (b1) {……}
while (!b2) {……}
bool fun()
{ …..
   return true;
}
```

New variable types (2)

Pointer that points to nothing has value 0:

```
char *p = 0;
```

Rather often:

```
#define NULL 0
char *p = NULL;

void fun(char *p) {…….}
void fun(int i) {…….}
```

Problem:

```
fun(0); // as 0 is an integer, always the second function is called
```

Solution:

```
fun(nullptr); // the first function is called
fun(0); // the second function is called
```

nullptr is introduced in C++ v 11. Advised to use instead 0 when working with pointers.

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);
```

I/O streams (7)

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()*.

I/O streams (8)

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();
```

I/O streams (9)

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
```

I/O streams (10)

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
    }
  ……..……..
}
```

I/O streams (11)

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
 ……..……..
}
```

I/O streams (12)

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
}
```