Templates (1)

Let us have:

```
class Array
{
 protected: int Size,  * pArray;
 public:     Array(int n) {  Size = n;   pArray = new int[n]; }
             virtual ~Array()  { delete  pArray; }
             int GetSize() { return  Size; }
             int Get(int);
             void Set(int, int);
};

int Array::Get(int i)
{
  if (i < 0 || i >  Size - 1)  throw "Illegal index";
  else return *( pArray + i);
}
void Array::Set(int Value, int i)
{
  if (i < 0 || i >  Size - 1) throw  "Illegal index";
  else *( pArray + i) = Value;
}
```

Templates (2)

Generic programming: how to write class *Array* so that one of the users could apply it as a container of double numbers, another user for storing of pointers to strings, etc.

The class template defines a class where the types of some attributes, return values of methods and/or parameters of methods are specified as parameters.

```
template<typename T> class Array // deprecated: template<class T>
{ // template<typename T> is the template specifier.
  // Word "Array" here is the class template name (not the class name) .
  // T is the placeholder for actual types like int, double, etc.
protected: int Size;
             T *pArray;
public:     Array(int n) { Size = n; pArray = new T[n]; }
            virtual ~Array()  { delete pArray; }
            int GetSize() { return Size; }
            T Get(int);
            void Set(T, int);
};
```

Templates (3)

```
template<typename T> T Array<T>::Get(int i)
{// template<typename T> is the template specifier, it says that we have a template,
  // not a traditional class
  // Array<T> refers to class template with parameter T and name Array
  // Name Array without following to it <T> is meaningless
  // Array<T>::Get(int i) means that Get() is a member function of class template
  // T is the type of Get() return value.
  if (i < 0 || i > Size - 1)  throw "Illegal index";
  else return *(pArray + i);
}

template<typename T> void Array<T>::Set(T Value, int i)
{
  if (i < 0 || i > m_Size - 1) throw "Illegal index";
  else *(pArray + i) = Value;
}
```

Here *T* may be a simple variable or a class. In the last case the assignment operator overloading must be implemented.

Templates (4)

```
int main( )
{
  Array<int> IntArr(100); // instantiate the template
  try
  {
    for (int i = 0;  i < 100;  i++)
          IntArr.Set(i, i); // use any an ordinary object
     cout << IntArr.Get(5) << endl;
  }
  catch(char *pMsg)
  {
       cout << pMsg << endl;
  }
  return 0;
}
```

Important: the compiler checks the template code syntax, but does not compile it. The compiling is performed when the actual type is specified. Therefore, in the example above the compiler needs the complete code of template Array<T>.

Templates (5)

```
template<typename T> class Array
{
 ……………………………………………………..
   Array<T>(const Array<T> &Original)
   { // copy constructor
     Size = Original. Size;
      pArray = new T[ Size];
      memcpy(pArray, Original. pArray, sizeof(T) *  Size);
   }
   Array<T> &operator=(const Array<T> &Right)
   { // overloading assignment
      Size = Right. Size;
      delete  pArray;
       pArray = new T[Size];
      memcpy( pArray, Right. pArray, sizeof(T) *  Size);
      return *this;
   }
  …………………………………………………….
};
```

Remember: instead of class name *Array* here we write class template name as *Array<T>*.

Templates (6)

```
template<typename T, int SIZE> class Array
{ // non-type parameters can only be integrals (char, int, etc.), pointers and references
protected: T *pArray;
public:     Array() {  pArray = new T[SIZE]; } // constructor
            Array<T, SIZE>(const Array<T, SIZE> &Original) // copy constructor
            {  pArray = new T[SIZE];
              memcpy( pArray, Original. pArray, sizeof(T) * SIZE); }
            virtual ~Array()  { delete  pArray; } // destructor
            Array<T, SIZE> &operator=(const Array<T, SIZE> &Right) // overloading =
            { memcpy( pArray, Right. pArray, sizeof(T) * SIZE);
               return *this; }
            int GetSize() { return SIZE; } // get the number of elements
            T Get(int i) // get an element
            { if (i < 0 || i > SIZE) throw "Illegal index";
               else return *(m_pArray + i); }
            void Set(T Value, int i) // set value to an element
            { if (i < 0 || i > SIZE) throw "Illegal index";
              else *(m_pArray + i) = Value; }
};
// Array<int, 10> IntArr; // array of integers, the length is 10
```

Templates (7)

C++ supports also templates for functions:

```
template<typename T> T Larger(T a, T b)
{
        return a > b ? a : b;
}
```

Usage:

```
double x, y, z;
z = Larger<double>(x,y);
```

This function is applicable for types for which the "greater than" operation is defined.

The arguments and return values may be from different types:

```
template <typename T1, typename T2, typename T3> void Fun(T1 a, T2 b, T3 c)
{
……………..
}
```

Usage:

```
double x, y;
int i;
Fun<double, int, double>(x, i, y);
```

New variable types (1)

In C and C++ prior to version 11 keyword *auto* meant that the variable has automatic duration (i.e. it will be created and destroyed automatically):

```
auto int i; // "auto" was almost always omitted
```

In C++ v11 and later keyword *auto* means that the compiler has to deduct the actual type:

```
auto i = 10; // i is of type int
auto j = 10L; // j is of type long int
auto k; // error -  compiler is unable to deduct the type
```

auto simplifies the work of code writers. In templates its usage may be inevitable.

Example:

```
template <typename T1, typename T2> void Fun(T1 a, T2 b)
{
      auto c = a + b;
      ………………….
}
```

If *T1* and *T2* are both *int*, *c* is also *int*. But if *T1* is *double* and *T2* is *int*, *c* is *double*. Consequently, when writing the code, we do not know the type of *c* and therefore using the auto deduction is the only way out.

New variable types (2)

Length depends on the implementation of compiler:

```
long long int ll; // in Visual Studio 64 bits
unsigned long long int ull; // in Visual Studio 64 bits
wchar_t wct; // in Visual Studio 16 bits
long double ld; // in Visual Studio 64 bits, i.e. the same as double
```

Length is specified in standard:

```
char16_t c16; // 16-bits, for UTF-16 characters
char32_t c32; // 32-bits, for UTF-32 characters
```

Additional built-in types defined by Microsoft:

```
signed __int8 i8; // 8-bit signed integer
signed __int16 i16; // 16-bit signed integer
signed __int32 i32; // 32-bit signed integer
signed __int64 i64; // 64-bit signed integer
unsigned __int8 i8; // 8-bit unsigned integer
unsigned __int16 i16; // 16-bit unsigned integer
unsigned __int32 i32; // 32-bit unsigned integer
unsigned __int64 i64; // 64-bit unsigned integer
```

Visual Studio does not support 128 bit variables.

New variable types (3)

To write platform-independent code that will be compiled using different compilers and will run under different operating system we may need aliases:

int8_t i8; // 8 bits, also there are types int16_t, int32_t, int64_t

uint8_t ui8; // 8 bits, also there are types uint16_t, uint32_t, uint64_t

(u)intX_t is the alias for signed or unsigned type occupying exactly X bits. For example, if we write *int x*, we get a variable that on one platform occupies 32 bits but on another one may occupy 16 bits. However, if we need a 32 bit variable in any case, we need to write *int32_t x*.

int_fast8_t if8; //at least 8 bits, also there are types int_fast16_t, int_fast32_t, int_fast64_t

uint_fast8_t uif8;

// at least 8 bits, also there are types uint_fast16_t, uint_fast32_t, uint_fast64_t

(u)int_fastX_t is the alias for signed or unsigned type occupying at least X bits and on the current platform guaratees the fastest operating. With Visual Studio on a 32-bit processor, for example, *int_fast16_t* corresponds to *__int32*.

int_least8_t il8;

//at least 8 bits, also there are types int_least16_t, int_least32_t, int_least64_t

uint_least8_t uil8;

// at least 8 bits, also there are types uint_least16_t,uint_least32_t,uint_least64_t

(u)int_leastX_t is the alias for the smallest signed or unsigned type occupying at least X bits.

New variable types (4)

```
intmax_t imax;
uintmax_t uimax;
```

(u)intmax_t is the alias for the largest signed or unsigned integer type. In Visual Studio, for example, *intmax_t* corresponds to *__int64*.

```
intptr_t ip;
uintptr_t uip;
```

(u)intptr_t is the alias for the largest signed or unsigned integer that is large enough to hold a pointer.

To work with aliases, the programmer may need to know their maximum and minimum value. Those limits are defined by macros like *INT8_MIN*, *INT8_MAX*, *UINT8_MAX*, etc. See more on https://en.cppreference.com/w/cpp/types/integer.

To know more about the properties of numeric types use template *numeric_limits<T>*.

Example: // see https://www.cplusplus.com/reference/limits/numeric_limits/

```
cout << numeric_limits<double>::min() << ' ' << numeric_limits<double>::max() <<
endl;   // prints 2.22507e-308 1.79769e+308
```

To work with aliases and limits you need:

```
#include <cstdint>
#include <limits>
```

New variable types (5)

Keyword *decltype* specifies the type from the result of expression:

```
decltype (expression) variable_name = initial_value;
```

The initial value is optional. Examples:

```
Date d;
decltype (d) d1; // d1 is also of type Date
decltype (d.GetYear()) i; // i is of type int
decltype (d.GetYear()) i = 2018; // i is of type int and gets initial value 2018
```

As *auto*, *decltype* also simplifies the work of code writers. But mostly it is used in templates.

Example:

```
template <typename T1, typename T2> void Fun(T1 a, T2 b)
{
      typedef decltype(a + b) T;
      T x, y;
      ………………….
}
```

Remark that this slide presents only a simplified definition of *decltype*. A detailed discussion may be found on http://thbecker.net/articles/auto_and_decltype/section_01.html

Run time type information (1)

Sometimes it is difficult or even impossible to specify the type of pointers. In that case we may declare the type as *auto*. But later (especially during debugging but for other reasons too) we may need to know what is the actual type.

Operator typeid(expression) returns an object of standard class *type_info*. Function *name()* of this class returns a string specifying the type of result of the expression.

Examples:

```
int i;
Date d, *pd = new Date;
cout << typeid(i).name() << endl; // prints "int"
cout << typeid(d).name() << endl; // prints "class Date"
cout << typeid(pd).name() << endl; // prints "class Date *"
cout << typeid(*pd).name() << endl; // prints "class Date"
```

The *type_info* objects can be compared. Example:

```
Date *pd1 = new Date, *pd2 = new Date;
cout << boolalpha << (typeid(*pd1)) == typeid(*pd2)) << endl; // prints "true"
```

If you have a chain of inherited classes then the *typeid* operator works correctly only if the base class has at least one virtual function (for example, the destructor).

Run time type information (2)

It is always better to apply the *typeid* operator not to the pointer but (using dereference operator) to the object itself.

Example:

```
class Base {…..};
class Derived : public Base { …..};
Derived *pd = new Derived;
Base *pb = pd;
cout << typeid(pb).name() << " " << typeid(pd).name() << endl;
 // Prints "class Base * class Derived *"
 // Formally correct, but actually pb points to an object of class derived
cout << typeid(*pb).name() << " " << typeid(*pd).name() << endl;
 // Prints " class Derived class Derived"
 // Here we have got the actual situation in memory
```

Numerics library (1)

Here we speek about common mathematical functions partly inherited from classical C, special mathematical functions introduced in C++ v. 17 and mathematical constants introduced in C++ v. 20.

By default, Visual Studio is set to compile code written in C++ v. 14. To upgrade, open the project properties and set the C++ language standard to *ISO C++ 17* or *ISO C++ 20* (in Visual Studio 2019 *Features from the latest C++*).

To use common mathematical functions write:

```
#include <cmath>
```

The complete list is on: https://en.cppreference.com/w/cpp/numeric/math . Some examples:

```
x = sin(y); // the argument is in radians
            // if the argument is float, the return value is also float
            // to emphasize it, you may use instead of sin function sinf
            // if the argument is double or any kind of integer, the result is double
x = pow(y, z); // calculates y^z
               // if the base (i.e. y) is float, the return value is also float. The exponent
               // (i.e. z) may be float or any kind of integer.
               // to emphasize that the both arguments are float, you may use function powf
               // if the base is double, the return value is also double. The exponent
               // may be double or any kind of integer.
```

Numerics library (2)

```
x = fmod(y, z); // calculates the remainder of x/y, for example 12 / 10 the remainder is 2
                // if the arguments are float, the return value is also float
                // to emphasize it, you may use instead of fmodf function fmodf
                // if the arguments are double, the result is also double
x = modf(y, &z); // decomposes y into integral part (z) and fractional part (x), for example
                 // if y = 2.5 then v gets value 0.5 and z gets value 2.0
                 // if the arguments are float, the results are also float
                 // to emphasize it, you may use instead of modf function modff
                 // if the arguments are double, the results are also double
x = ceil(y); // returns the smallest integer value not less than argument for example
             // if y = 2.5 then x gets value 3.0
             // if the arguments are float, the results are also float
             // to emphasize it, you may use instead of ceil function ceilf
             // if the argument is double, the result is also double
x = floor(y); // returns the largest integer value not greater than argument for example
              // if y = 2.5 then x gets value 2.0
              // if the arguments are float, the results are also float
              // to emphasize it, you may use instead of ceil function floorf
              // if the argument is double, the result is also double
```

Numerics library (3)

If you are working with common mathematical functions, study carefully their behavior and return value in case of errors. Example:

```
double x, y;
y = -1;
x = log(y); // x = ln(y) (natural logarithm, base is e), no crash, returns NAN
cout << boolalpha << isnan(x) << endl; // prints true
y = 0;
x = log(y); // no crash, returns INFINITY
cout << boolalpha << isinf(x) << endl; // prints true
```

Instead of *isnan* and *isinf*, the result may be checked with functions *isfinite* (i.e. not NAN, not INFINITY) or *isnormal* (i.e. not NAN, not INFINITY, not zero).

To get an error message, use global variable *errno* (inherited from C). It is defined in

```
#include <cerrno> // see https://cplusplus.com/reference/cerrno/errno/
```

Before call to a function set *errno* to zero. If there is something abnormal, the function assigns to *errno* an error code (see the list on https://cplusplus.com/reference/system_error/errc/).

Example:

```
errno = 0;
y = -1;
x = log(y); // errno gets value EDOM
cout << stderror(errno) << endl; // prints "Domain error"
```

Numerics library (4)

The *errno* mechanism works only if

```
cout << boolalpha << (bool)(math_errhandling & MATH_ERRNO) << endl; // prints true
```

It is so in Visual Studio. Macro *math_errhandling* is defined in <*cmath*> header.

There is another error handling mechanism (also supported in Visual Studio) that works if

```
cout << boolalpha << (bool)(math_errhandling & MATH_ERREXCEPT) << endl; // prints true
```

Here the tools from floating point environment are used:

```
#include <cfenv> // see https://en.cppreference.com/w/cpp/numeric/fenv
```

If during floating-point calculations an exceptional circumstance occurs, a floating-point exception (it is not a C++ exception) is raised, it means that a flag is set. Example:

```
double x, y;
…………………… // calculates y
feclearexcept(FE_ALL_EXCEPT); // reset all flags
x = log(y);
if (fetestexcept(FE_INVALID)) // checks is the flag set
{
   cout << "Invalid value exception raised" << endl; // here it means that y was negative
}
else if (fetestexcept(FE_DIVBYZERO))
{
   cout << "Division-by-zero exception raised" << endl; // here it means that y was zero
}
```

Numerics library (5)

Special mathematical functions introduced in C++ v. 17 are declared also in

```
#include <cmath>
```

There are methods for Bessel functions, Legendre polynomials, Gamma functions, etc. The details fall outside the scope of this course.

C++ v. 20 adds several mathematical constants.

Examples (see the complete list on https://en.cppreference.com/w/cpp/numeric/constants):

```
#include <numbers>
cout << numbers::pi <<endl; // ¶
cout << numbers::inv_pi << endl; // (1 / ¶)
cout << numbers::sqrt2 << endl; // √2
```

Complex numbers (1)

In *template <class T> class complex* variable *T* may be *float*, *double* or *long double*. The class has member functions *real()*, *imag()* and operator functions for arithmetics and comparing. See details from https://www.cplusplus.com/reference/complex/complex/.

Examples:

```
#include <complex>
complex<double> c1(3.4, 5.6), c2(10, 20);
cout << c.real() << ' ' << c.imag() << endl; // prints 3.4 5.6
cout << c << endl; // prints (3.4,5.6)
complex<double> c3 = c1 + c2, c4 = c1 * c2;
cout << c3 << ' ' << c4 << endl; // prints (13.4,25.6) (-78,124)
cout << boolalpha << (c1 != c2) << endl; // prints true
```

Arithmetical operations between complex and non-complex values are also allowed, for example:

```
cout << (2.5 + c1) << endl; // prints (5.9, 5.6)
cout << (c1 + 2.5) << endl; // prints (5.9, 5.6)
cout << (c2 * 2.0) << endl; // prints (20, 40)
```

In addition, there is the complex numbers library: a set of standard functions for operating with complex numbers. See https://www.cplusplus.com/reference/complex/.

Complex numbers (2)

Examples:

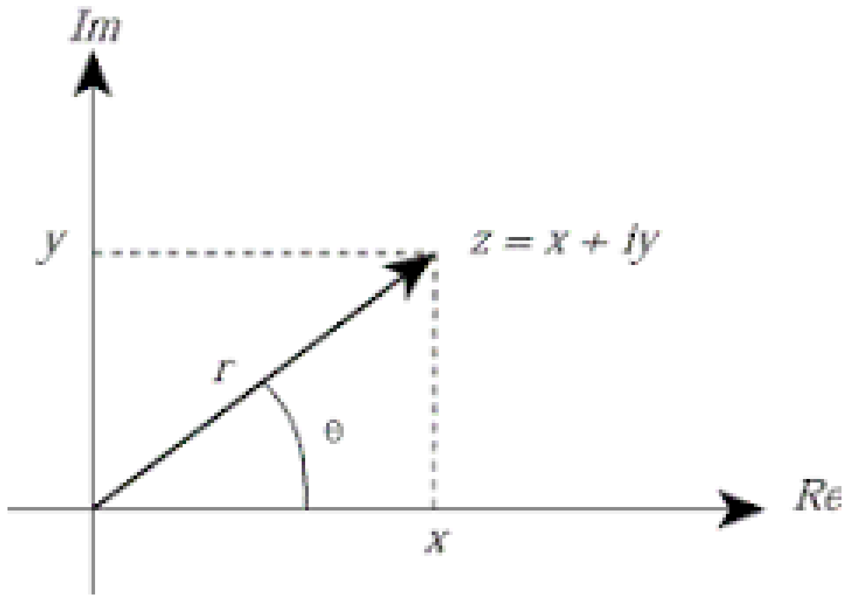
```
#include <complex>
#include <numbers>
using namespace std;
complex<double> c(10, 20);
cout << conj(c) << endl; // prints the conjugate (10, -20)
cout << abs(c) << endl; // prints the absolute value or modulus sqrt(Re² + Im²)
                                         // 22.3607
cout << norm(c) << endl; // prints the norm (Re² + Im²) or modulus²
                                           // 500
cout << arg(c) << endl; // prints the phase angle in radians
```

A complex number may be in cartesian format x + i *y or in polar format (r, Θ).

To get the polar format components from complex number presented in cartesian format use methods *abs* and *arg*.

To get a complex number in cartesian format from complex number presented in polar format use method *polar*:

```
cout << polar(25.0, 45 * (numbers::pi / 180.0)) << endl;
// prints (17.6777,17.6777)
```



Byte

Type *std::byte* was introduced in C++ version 17. Earlier, if we wanted to work with memory bytes we had to use types *signed char* or *unsigned char*. Their difference with *byte* is that a *byte* cannot have a numeric or character interpretation: it is just a sequence of 8 bits and nothing more. The *byte* supports comparing, bitwise operations and shifting, but not arithmetic operations. Explicitly it can be casted to integers and vice versa. Examples:

```
// include <cstddef> // see more on https://en.cppreference.com/w/cpp/types/byte
byte b1 { 0xFF }, b2 { 255 }, b3 = static_cast<byte>(0xFF); // but b3 = 0xFF is an error
byte  b4 { 0b11110000 };
byte b5 { 0 }; // all the bits are 0
byte b6 { 1 } ; // all the bits are 1
cout << hex << static_cast<int>(b1) << endl; // prints ff
cout << hex << to_integer<unsiged int>(b1) << endl; // prints ff
byte b7 = b1 << 1;
byte b8 = b2 | b4;
if (b5 == byte { 0 } ) // but not if (!b5)
{ ……….. }
```

To print a byte in binary format you need to use bitsets (see more in chapter "Containers"):

```
byte by { 0b10101010 };
unsigned long int lu = to_integer<unsigned long>(by);
bitset<8> bits(lu);
cout << bits << endl;  // prints 10101010
```

Any (1)

An instance of class *any* can hold a value of any type or no value at all. This feature was first introduced in C++ version 17. Examples:

```
#include <any> // see https://en.cppreference.com/w/cpp/utility/any
any a1; // no value
any a2 = 10; // has value 10, type is int
any a3 = string("Hello"); // has value "Hello", type is string
```

To know the type of value stored in *any* use method *type* and operator *typeid*. To retrieve the value stored in *any* use *any_cast*. Example:

```
if (a3.type() == typeid(string)) {
      string s = any_cast<string>(a3); // copies the contents of a3 into s
      ………….. // do something with s
}
```

If you do not check the type, you may get an exception:

```
try {
    int i = any_cast<int>(a3);
}
catch (bad_any_cast &e) {
   cout << e.what() << endl;
}
```

Any (2)

You can change the value stored in *any* to another value of the same type or some other type. Example:

```
any a = string("Hello");
cout << any_cast<string>(a) << endl; // prints "Hello"
a = string("Goodbye");
cout << any_cast<string>(a) << endl; // prints "Goodbye"
a = 10;
cout << any_cast<int>(a) << endl; // prints 10
```

To access the value stored into *any* directly, cast to pointer or reference. Example:

```
any a = string("Hello");
string* p = any_cast<string>(&a); // not "any_cast<string *>"
p->insert(5, " world");
cout << any_cast<string>(a) << endl; // prints "Hello world"
string& r = any_cast<string&>(a);
r.insert(11, " champion");
cout << any_cast<string>(a) << endl; // prints "Hello world champion"
```

Turn attention that

```
any a = "Hello";
cout << a.type().name() << endl; // prints "const char *" and not "string"
```

Any (3)

To remove the contents of *any* use method *reset()*:

```
a.reset();
cout << a.type().name() << endl; // prints "void"
```

To check whether there is a value in *any* use method *has_value()*:

```
cout << boolalpha << a.has_value() << endl; // prints "false"
auto p = any_cast<string>(&a); // p is nullptr
```

Usage example: suppose we need to write function that needs an integer as input value. But this integer may be presented as variable of type *int* or as an object of class *string*. Due to *any* we may instead of two functions

```
bool fun(int);
bool fun(string);
```

write only one:

```
bool fun(any);
```

and call it like:

```
fun(200);
```

or

```
fun(string("100"));
```

The implementation is on the following slide.

Any (4)

```
bool fun(any a) {
 int i;
 if (a.type() == typeid(string)) {
     try {
             i = stoi(any_cast<string>(a));
     }
      catch (exception &e) {
             cout << e.what() << endl;
             return false; // string does not present an integer
     }
 }
 else if (a.type() != typeid(int)) {
      return false; // input value is neither integer nor string
 }
 else {
      i = any_cast<int>(a);
 }
  …… // do something with variable i
  return true;
}
```

Optional (1)

Object specified by template *optional<T>* holds an object of class *T* or nothing at all. This template was first introduced in C++ version 17.

If a function must return the pointer to result but fails, it returns *nullptr*. If a function must return the resulting object itself but fails, it may return value *nullopt*.

Example (see also https://en.cppreference.com/w/cpp/utility/optional):

```
#include <optional>
optional<int> convert(string s) {
   try  {
        return stoi(s);
   }
   catch (exception)  {
        return nullopt;
    }
}
```

Usage:

```
optional<int> oi = convert("xxx");
if (!oi)
    cout << "Failed" << endl;
else
    cout << *oi << endl;
```

Optional (2)

Alternative solution:

```
optional<int> convert(string s)
{
  optional<int> result; // automatically initializes to nullopt
  try {
       result = stoi(s);
   }
   catch (exception) { }
   return result;
}
```

Alternative usage:

```
optional<int> oi = convert("xxx");
if (!oi.has_value())
    cout << "No result" << endl;
else
    cout << oi.value() << endl;
```

If we call method *value()* but the value is not present, *bad_optional_accesss* expression is thrown. If we use deferencing to retrieve the non-existing value, the result is unpredictable.

Due to template *optional* we do not need to use tricks for expressing the failure (for example returning values like -*1*, *""*, etc. symbolizing the absence of result).

Optional (3)

Class attributes or function parameters may be also optional. Example:

```
void PrintName(string first, optional<string> middle, string last) {
  cout << first << ' ';
  if (middle.has_value()) {
      cout << middle.value() << ' ';
  }
   cout << last << endl;
}
```

Usage:

```
PrintName("John", "Edward", "Smith");
PrintName("James", nullopt, "Sailor");
```

In a class:

```
class Name {
   string First;
   optional<string> Middle;
   string Last;
   Name(string s1, optional<string>s2, string s3) : First(s1), Middle(s2), Last(s3) { }
……………………..
};
```

Optional (4)

Examples about defining and initializing of optional values:

```
optional<int> oi; // nullopt
optional<string> os1("Hello"), os2 = "Hello";
optional<int> oi1(10), oi2 = 10, oi3 = make_optional(10), oi4 = oi3;
optional<Date> od1(Date(1, 1, 2021)), od2 = Date(1, 1, 2021), od3 { Date { 1, 1, 2021 } };
```

It is possible to compare optional values (actually to compare values wrapped into template):

```
if (o4 == o3)
    cout << "Equal" << endl;
```

Read also: https://www.bfilipek.com/2018/05/using-optional.html

Constant expressions

Keyword *constexpr* specifies that it is possible to evaluate the result of a function or the value of a variable at compile time. Example:

```
#include <numbers> // see https://en.cppreference.com/w/cpp/numeric
constexpr double CircleArea(double radius) {   return numbers::pi * radius * radius; }
// now function CircleArea() can be called from constant expressions
```

The following expression is an constant expression:

```
constexpr double a1 = CircleArea(10); // value of a1 will be calculated at compile time
a1 += 10; // error – a1 is constant
```

The following expressions are not constant expressions:

```
const double a2 = CircleArea(10); // value of a2 will be calculated at run time
double a3 = CircleArea(10); // value of a3 will be calculated at run time
double a4;
cin >> a4;
double a5 = CircleArea(a4); // value of a5 will be calculated at run time
```

Constant expressions may improve the application performance. See more at https://en.cppreference.com/w/cpp/language/constant_expression and https://en.cppreference.com/w/cpp/language/constexpr

Initializing (1)

Starting from C++ v 11, the member variables may be initialized directly in the class definition. Example:

```
class Matrix
{
 private: int nRows = 0, nColumns = 0; // default values
              double **ppMatrix = nullptr; // default value
  public: Matrix () { }
              Matrix(int, int);
…………………………….
};
Matrix *pm1 = new Matrix; // empty constructor is called, attributes get default values
Matrix *pm2 = new Matrix(10, 10); // attributes get values corresponding to the
                                                       // constructor actual parameters
```

Default value may be presented by any expression that is executable when the object is created. Example:

```
class Time
{
  private: time_t Now = time(&Now);
 …………………………..
};
```

Initializing (2)

There are several cases when the constructors written in traditional mode do not work. Examples:

```
class Test1
{
 public:
    const int ciValue = 0;
    Test2 test2; // class  Test2 has no constructor without arguments
    int &ri; // error, it is not possible to declare a reference without initialization
    Test1(int i)
    {
        ciValue = i;  // error, it is not possible to change a constant
        test2.SetInitialValues(); // error, object test2 was not created
    }
};
```

Comment: the constructor of *Test1* must at first create all the attributes and after that execute the initialization defined in its body. But to create attribute *test2* it needs to call the constructor of *Test2*. However, *Test2* has no constructor without arguments.

Initializing (3)

The constructor initializer is defined as:

class_name::class_name(list_of_arguments) : attribute_initializer_list { body }

where the comma-separated components of attribute initializers list are:

- if the attribute is not an object: attribute_name(attribute_initial_value)
- if the attribute is an object: attribute_name(constructor_arguments)

Attribute initial values and constructor arguments may be constants, elements from the constructor argument list or any other executable expressions.

Examples:

```
class Point
{
public: int x, y;
        Point(int i, int j) : x(i), y(j) { } // x gets value of i, y gets value of j, body is empty
};
class Rectangle
{
public: Point p1, p2;
        Rectangle(int x1, int y1, int x2, int  y2) : p1(x1, y1), p2(x2, y2) { }
}; // attribute initializer list contains calls to constructors of attribute objects
   // remark that class Point does not have constructor without arguments
```

Initializing (4)

Constructor initializer is necessary when:

- Some attributes are objects of classes without default (i.e. not having arguments) constructor (like *Point* on previous slides).
- Some attributes are objects of classes having constructor with arguments (already discussed earlier, see the problems with aggregation).
- A constant attribute or a reference attribute must be initialized.

```
class Test1
{
public: const int ciValue;
            int &ri;
            Test1(int i, int &j) : ciValue(i), ri(j) {  }
};
```

The classical constructor first creates all the attributes and after that executes the initializations defined in its body. The constructor initializer creates an attribute and right after that initializes it.

Mixed constructors in which some of the initializations are specified in the attribute initializers list and the others in the constructor body are also allowed.

Initializing (5)

```
class Circle
{
public:  const double pi = 3.14159;
         Point centre;
         int radius;
         double area;
         Circle(int x, int y, int r) : radius(r), centre(x, y), area (pi * radius * radius) { }
};
```

Important: the attributes are initialized in the order that they appear in class definition. So, although in the list attribute *radius* is the first, attribute *centre* is initialized before it.

```
class Circle
{
public:  const double pi = 3.14159;
         double area; // error, when area is initialized, radius has no value
         Point centre;
         int radius;
Circle(int x, int y, int r) : radius(r), centre(x, y), area (pi * radius * radius) { }
};
```

Initializing (6)

Let us have

```
class Circle {
public:  const double pi = 3.14159;
         Point centre;
         int radius;
         double area;
         Circle(int x, int y, int r) : radius(r), centre(x, y), area (pi * radius * radius) { }
};
struct Date {
  int Day,
      Month,
      Year;
};
```

Traditionally we define an objects of class *Circle* and *Date* like:

```
Circle c1(0, 0, 10);
Circle *pc = new Circle(0, 0, 10);
Date d1; // compiler-created default empty constructor is applied
Date *pd1 = new Date;
Date d2(); // not an error but for compiler it is the prototype of a function without
           // parameters returning object of class Date
```

Initializing (7)

It is less known that we can write also:

```
Circle c2 = Circle(0, 0, 10);
Date d2 = Date(); // but Date d3 = Date; is an error
Date *pd2 = new Date();
```

From introductory courses we know that

```
int m1[5] = { 0, 1, 2, 3, 4 };
int m2[] = { 0, 1, 2, 3, 4 }; // dimension omitted
int *pm1 = new int[5] { 0, 1, 2, 3, 4 };
int *pm2 = new int[] { 0, 1, 2, 3, 4 };
```

Actually, this is the uniform initialization that has two formats:

```
type object { initial_values_or_constructor_arguments }
type object =  { initial_values_or_constructor_arguments }
```

So:

```
int m3[5] { 0, 1, 2, 3, 4 };
int m4[] { 0, 1, 2, 3, 4 };
Circle c3 = { 0, 0, 10 }; // constructor is called
Circle c4 { 0, 0, 10 }; // constructor is called
Circle *pc5 = new Circle { 0, 0, 10 };
Date d4 = { };
Date d5 { };
int i1 = { 10 }, i2 { 10 }; // int i1 = 10, i2 = 10;
```

if / else **with initializing**

Let us have code snippet:

```
int n = fun();
if (n > 0) {
    ….. // perform some operations with n
}
else {
   ...... // perform some other operations with n
}
```

Starting from C++ version 17 we may write this snippet as follows:

```
if (int n = fun();  n > 0) {
    ….. // perform some operations with n
}
else {
   ...... // perform some other operations with n
} // from this point variable "n" is out of scope
```

Variable defined and initialized in *if*-statement is visible and has memory:

- in conditional expression of *if*-statement as well as in the conditional expressions of the following *if-else*-statements;
- in the body of *if*-statement as well as in the body of the following *if-else*-statements and also in the body of final *else*-statement

switch with initializing

Similarly to *if / else*, in C++ version 17 the *switch*-statement may also include definition and initialization of variables. Example:

```
enum class colors { Red, Green, Blue };
colors GetColor() { …… }

switch (colors wall = GetColor(); wall)
{
  case colors::Red:
        …….  // do something with variable wall
        break;
  case colors::Blue:
         ……. // do something with variable wall
        break;
 case colors::Green:
        ……. // do something with variable wall
        break;
} // from this point variable "wall" is out of scope
```

Default constructors (1)

Default constructor has no arguments. Its body may be (but not must be) empty.

If the class declaration does not contain constructors, the compiler itself generates a default constructor having empty body. But sometimes you may need a class in which there are no constructors at all. In that case write:

```
class Test {
  public: Test() = delete; // explicitly deleted default constructor
  ………………
};
```

If the class declaration contains constructors (with or without arguments), the compiler does not generate its own constructor.

It is also possible to forbid the automatic generation of default copy constructor and default *operator*= for assignment overloading:

```
class Test {
  ……………..
  public: Test(const Test &) = delete;
             Test& operator=(const Test &) = delete;
  ………………
};
```

Default constructors (2)

It may happen that the programmer does not see any need to include a default constructor into his / her class declaration (example: class *Point* on slide *Initializing (3)*). But for example the C++ standard containers operate only with objects from classes having the default constructor. In that case we need to add to the declaration of our class our own empty default constructor:

```
class Test {
  public: Test() = default; // explicitly defaulted constructor
                            // we may also write Test() { }
  ………………
};
```

Shorthand *return* (1)

Let us have:

```
struct Date {
  int day, month, year;
  Date(); // default constructor implemented in file Date.cpp calls the computer's clock
  Date(int d, int m, int y); // implemented in file Date.cpp
};
```

Then instead of

```
Date GetDate() {
     Date d; // default constructor is called
     return d;
}
```

we may write

```
Date GetDate() {
     return Date(); // default constructor is called
}
```

or

```
Date GetDate() {
     return { };
}
```

return { } means that the default constructor of the return value type is called.

Shorthand *return* (2)

Similarly instead of

```
Date GetDate() {
    Date d(26, 5, 2023); // default constructor is called
    return d;
}
```

we may write

```
Date GetDate() {
    return Date(26, 5, 2023);
}
```

or

```
Date GetDate() {
    return { 26, 5, 2023 };
}
```

Conversion constructors (1)

Let us have class

```
class Test1
{
public:
  int value;
  Test1(int i) : value(i) { }
};
```

and function

```
void TestFun1(Test1 t)
{
      cout << t.value << endl;
}
```

Then

```
TestFun1(10);  // prints 10
```

is correct because the compiler handles the constructor as a casting method: it casts integer 10 to object *t* of class *Test1*. Of course, the equivalent expression

```
TestFun1(Test1(10));
```

is better to understand. In C++ version 11 any constructor with arguments may be interpreted as casting operator or in other words, is a conversion constructor. In the earlier versions a conversion constructor had to have default values for all except one of its arguments.

Conversion constructors (2)

Let us have class

```
class Test2
{
public:
  int value1, value2;
  Test2(int i, int j) : value1(i), value2(j) { }
};
```

and function

```
void TestFun2(Test2 t)
{
      cout << t.value1 <<  ' '  << t.value2 << endl;
}
```

Then

```
TestFun2( { 10, 20 } ); // prints 10 20
```

is equivalent with

```
TestFun2(Test2(10, 20));
```

To prevent interpreting a constructor as casting operator declare it with keyword *explicit*, for example:

```
explicit Test2(int i, int j) : value1(i), value2(j) { }
```

After that:

```
TestFun2( { 10, 20 } ); // compile error
```

Pointers to functions (1)

Pointer to a variable holds the address of the first byte of memory field on which the variable is located. Pointer to a function holds the address of the byte from which the function code starts.

Declaring a pointer to variable we have to specify the type of data to which it will point.
Declaring a pointer to function we have to specify:

- the type of function return value
- the number of parameters
- the types of parameters

Generally, the declaration to a pointer to function is:
return_value_type (*pointer_name)(parameter_list);

Examples:

```
void (*pf)(char *); // pf will point to functions with prototype void XXX(char *)
                    // where XXX is any identifier
double **(*pfn)(int, int); //pfn will point to functions with prototype double **XXX(int, int)
```

Pointers to functions (2)

To assign values to pointer to functions use function names:

```
pointer_to_function = function_name;
```

Example: suppose we have

```
void ToUpper(char *);
void ToLower(char *);
```

then we may write

```
void (*pf)(char *);
pf = ToLower;
```

or

```
pf = ToUpper;
```

Call to a function using pointer:

```
(pointer_to_function)(parameter_list);
```

Example:

```
char Buf[81];
cout << "Type some text" << endl;
gets_s(Buf);
cout << "press '\u\' to convert the text to uppercase or any other key to lowercase" << endl;
pf = _getche() == 'u' ? ToUpper : ToLower;
(pf)(Buf);
```

Pointers to functions (3)

Suppose we have to write a function that is able to sort array containing records of any type. There are several well-known algorithms (insertion sort, bubble sort, quick sort, etc.) but they all need to compare the records. As the values in array may be of any type, we cannot build the comparison directly into the code. The only way to solve the problem is to implement the comparison with pointer to function that can compare two records:

```
void sort(void *pArray, int RecordLength, int nRecords, int (*pCompare)(void *, void *));
```

If the records are of type

```
struct Student
{
  char *pName;
  ……
};
```

the comparing function may be:

```
int CompareStudentNames(void *pStud1, void *pStud2)
{
  return strcmp((char *)((Student *)pStud1)->pName, (char *)((Student *)pStud2)->pName);
}
```

and the call to sorting function may be like:

```
sort(pStudentGroup, sizeof(Student), nGroup, CompareStudentNames);
```

Pointers to functions (4)

Let us have a function for solution of quadratic equation $ax^2 + bx + c$, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
void QuadEq(double a, double b, double (*pf)(), double *px1, double *px2)
{ // coefficient c is the output of any function with no parameters and double as return value
  double d = b * b – 4 * a * (pf)();
  if (d < 0 || !a)
      throw exception("No solution");
   *px1 = (-b +sqrt(d)) / 2 * a;
   *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage example:

```
double tester() {   return 6.0;  }
double x1, x2;
try {
  QuadEq(1, 5, tester, &x1, &x2); // roots are -2 and -3
  // QuadEq(1, 5, tester(), &x1, &x2); // error, here tester is a pointer, not function
 }
catch (const exception &e) {
  cout << e.what() << endl;
}
```

Pointers to functions (5)

Another example:

```
void QuadEq(double a, double b, double (*pf)(double, double), double d1, double d2,
                    double *px1, double *px2)
{ // coefficient c is the output of any function with no parameters and double as return value
  double d = b * b – 4 * a * (pf)(d1, d2);   // In function sort from slide Pointers to functions 3
   // the input parameters for pCompare are in pArray. Here we need to specify the input
   // parameters for pf as input parameters of QuadEq
  if (d < 0 || !a)
      throw exception("No solution");
   *px1 = (-b +sqrt(d)) / 2 * a;
   *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage:

```
double tester(double d1, double d2) {   return d1 + d2;  }
double x1, x2;
try {
  QuadEq(1, 5, tester, 3, 3, &x1, &x2); // roots are -2 and -3
 }
catch (const exception &e) {
  cout << e.what() << endl;
}
```

Pointers to functions (6)

But if we have

```
class Tester
{
private:
        double Value = 6;
public:
        double GetValue() const { return Value; }
        void SetValue(double d) { Value = d; }
};
```

we cannot call function *QuadEq* from slide *Pointers to functions (4)*:

```
QuadEq(1, 5, Tester::GetValue, &x1, &x2); // error
```

because to use a member function we must also specify the object.

```
 class Tester
{
  public: static double GetValue() const { return 6.0; }
};
```

Now

```
QuadEq(1, 5, Tester::GetValue, &x1, &x2);
```

works because *GetValue()* is now *static* and for static member function it's enough to specify just the class.

Pointers to functions (7)

Pointers to member functions are defined in another way:

return_value_type (class_name::*pointer_name)(parameter_list);

Example:

```
double (Tester::*pf)(); // pf points to functions from class Tester, those functions
                        // have no arguments and they return a double value
```

To assign value to a pointer to member function you must specify also the class:

pointer_name = &class_name::member_function_name

Example:

```
pf = &Tester::GetValue;
```

Calls using the pointers to member functions:

(object_name.*pointer_name)(parameter_list);

(pointer_to_object->*pointer_name)(parameter_list);

Examples:

```
Tester t, *pt = new Tester;
cout << (t.*pf)() << endl;
cout << (pt->*pf)() -> endl;
```

Problem: we have no pointers that can point to functions from any class.

Pointers to functions (8)

Consequently, we cannot use function *QuadEq* from slide *Pointers to functions (4)* with member functions. The proper definition is:

```
void QuadEq(double a, double b, double(Tester::*pf)(), Tester *pt, double *px1, double *px2)
{// Problem: function QuadEx is applicable only for class Tester
 double d = b * b - 4 * a * (pt->*pf)();
 if (d < 0 || !a)
     throw exception ("No solution“);
 *px1 = (-b + sqrt(d)) / 2 * a;
 *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage example:

```
Tester *pt = new Tester;
double x1, x2;
try
{
  QuadEq(1, 5, &Tester::GetValue, pt, &x1, &x2); // roots are -2 and -3
 }
catch (const exception &e)
{
  cout << e.what() << endl;
}
```

Lambda expressions (1)

The lambda (the term is from LISP language) is a short nameless function defined in the body of another function.

The simplest lambda definition is:

```
[ ] (formal_parameter list) { body }
```

To execute lambda expression immediately add the list of actual parameters:

```
[ ] (formal_parameter list) {  body } (actual_parameter list);
```

The type of return value is deduced by the expression following the *return* keyword. If there is no *return* statement, the return type is *void*. If necessary, the programmer may specify the return type explicitly:

```
[ ] (formal_parameter list) -> return_type {  body }
```

Examples:

```
int x = []() { return 6; } (); // define lambda and execute immediately, x gets value 6
double x1 = -2, x2 = -3;
double max = [](double a, double b) { return a <= b ? b : a; } ( x1, x2);
                                        // define lambda and execute immediately, max gets value -2
int arr[] = { 1, 2, 3, 4, 5, 6 };
cout << boolalpha << [](int *p, int n, int m) -> bool
{ int i = 0;  for (; i < n && *(p + i) != m;  i++);  return i != n; } (arr, 6, 10) << endl;
  // prints false because 10 was not found
```

Lambda expressions (2)

To execute a lambda several times declare pointers to lambda expressions:

```
auto pointer_name = lambda_definition;
```

Example:

```
auto pl = [](double a, double b) { return a <= b ? b : a; };
// auto is very useful here because we do not need to guess the type
```

To call a lambda expression by its pointer:

```
pointer_name(actual_parameter_list);
```

Example:

```
double x = pl(x1, x2);
```

Lambda expressions may use variables from the enclosing scope. The brackets at the beginning of lambda are to define the capture block.

Capture block *[=]* means that all the variables may be used by value. Example:

```
double x1 = -2, x2 = -3;
double max = [=]() { return x1 <= x2 ? x2 : x1; } (); // max is -2
```

Capture block *[&]* means that all the variables may be used by reference. Example:

```
double x1 = -2, x2 = -3;
double max = [&]() { return x1 <= x2 ? x2 : x1; } (); // max is -2
```

Lambda expressions (3)

Call by value means that

```
double x1 = -2, x2 = -3;
double max = [=]() { return x1 <= x2 ? x2 : x1; } ();
```

brown and magenta variables have the same names but they are not the same: x1 is the copy of x1. Also, magenta variables are constants:

```
double max = [=]() {x1 = -1; return x1 <= x2 ? x2 : x1; } (); // error, x1 cannot be changed
```

To specify the copies as not constants use keyword *mutual*:

```
[ =] (formal_parameter list) mutual -> return_type { body }
```

Example:

```
double x1 = -2, x2 = -3;
double max = [=]() mutual{x1 = -1; return x1 <= x2 ? x2 : x1; } (); // max is -1
cout << x1 << endl; // still -2 because x1 is just the copy of x1.
```

Call by reference means that lambda can change the values defined in the enclosing block.

Example:

```
double x1 = -2, x2 = -3;
double max = [&]() { x1 = -1; return x1 <= x2 ? x2 : x1; } (); // max is -1
cout << x1 << endl; // x1 is now -1
```

Lambda expressions (4)

Capture blocks *[=]* and *[&]* allow the lambda to use all the variables defined in the enclosing scope. To decide selectively which variables the lambda may capture, specify the capture list.

Examples:

```
double x1 = -2, x2 = -3;
double max = [x1](double b)  {  return x1 <= b ? b : x1; } (x2);
            // lambda can use the copy of x1
max = [&x1](double b)  {  return x1 <= b ? b : x1; } (x2);
            // lambda can use the reference to x1
max = [&x1, x2]() {return x1 <= x2 ? x2 : x1; } ();
             // lambda can use the reference to x1 and the copy of x2
max = [&, x2]() {return x1 <= x2 ? x2 : x1; } ();
             // lambda can use all the variables by reference except x2 that is captured by value.
max = [=, &x2]() {return x1 <= x2 ? x2 : x1; } ();
             // lambda can use all the variables by value except x2 that is captured by reference.
max = [=, &x1, &x2]() {return x1 <= x2 ? x2 : x1; } ();
             // lambda can use all the variables by value except x1 and x2 that are captured by
             // reference.
```

Capture block *[this]* allows lambda to access all the members of the current class.

Lambda expressions (5)

Lambda expressions are often used to replace the pointers to functions. Examples:

```
double x1, x2;
auto pl = [] () -> double { return 6; };
try
{   // QuadEq is defined on slide Pointers to functions (4)
    // especially convenient for testing QuadEx: no additional test functions needed
  QuadEq(1, 5, []() -> double { return 6; }, &x1, &x2); // lambda is defined in call statement
  QuadEq(1, 5, pl, &x1, &x2); // alternative, pointer to lambda is used
}
catch (const exception &e)
{
  cout << e.what() << endl;
}
```

Of course, the lambda used in call statement must have the types and number of input parameters as well as the type of return value that correspond to the function prototype. For example, to call function *QuadEx* we can use only lambdas that have no parameters and return a double value.

However, lambda expressions with capture cannot replace the pointers to functions. Example:

```
Tester *pt = new Tester; // defined on slide Pointers to functions (5)
QuadEq(1, 5, [pt]() -> double { return pt->GetValue(); }, &x1, &x2); // error
```

Function wrappers (1)

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

Let us rewrite QuadEq defined on slide *Pointers to functions (4):*

```
void QuadEq(double a, double b, function<double()>pf, double *px1, double *px2)
{  // pointer to function is replaced by function wrapper
  double d = b * b – 4 * a * (pf)();
  if (d < 0 || !a)
      throw exception("No solution");
   *px1 = (-b +sqrt(d)) / 2 * a;
   *px2 = (-b - sqrt(d)) / 2 * a;
}
```

function<double()>pf means that, using standard class templates, we build a wrapper object *pf* for any callable object (function, lambda with or without capture) that has no input parameters and returns a double number. Wrapper object is used (i.e. the corresponding function or lambda is called) as a regular pointer to function.

Generally:

`function < return_value_type (list_of_input_parameter_types) > wrapper name`

Function wrappers (2)

```
double tester()
{
   return 6.0;
}

double x1, x2;
Tester *pt = new Tester; // defined on slide Pointers to functions (5)
try
{
  QuadEq(1, 5, tester, &x1, &x2); // normal function out of classes
  QuadEq(1, 5, []() -> double { return 6; }, &x1, &x2); // lambda without capture
  QuadEq(1, 5, [pt]() -> double { return pt->GetValue(); }, &x1, &x2);
                                                                    // lambda with capture
}
catch (const exception &e)
{
  cout << e.what() << endl;
}
```

Thus, we have now instruments for transferring functions out of classes as well as member functions. To transfer a function out of classes we may use its name. To transfer member functions we need to create a simple lambda.

Function wrapper3 (3)

Let us also rewrite QuadEq defined on slide *Pointers to functions (5):*

```
void QuadEq(double a, double b, function<double(double, double)>pf, double d1, double d2,
                        double *px1, double *px2)
{ // here pf is a wrapper for functions with two double arguments, it returns also a double
 double d = b * b - 4 * a * (pf)(d1, d2);
 if (d < 0 || !a)
     throw exception("No solution");
 *px1 = (-b + sqrt(d)) / 2 * a;
 *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage:

```
double x1, x2;
try
{
  QuadEq(1, 5, [](double z1, double z2) { return z1 <= z2 ? z2 : z1; }, 1, 6, &x1, &x2);
}
catch (const exception &e)
{
   cout << e.what() << endl;
}
```

Functors (1)

Functors or function objects are objects that can be treated as though they are functions. An object of a class is a functor if in its class the function call is overloaded. Example:

```
class FunctorClass {
private:
    double Value;
public: // a class may include several operator functions having different signatures.
    FunctorClass(double d) : Value(d) { }
    double operator() () { return Value; } // overloads call to function that has no parameters
                                            // and returns a double
    void operator() (double); // overloads call to function that has a double parameter
                              // and returns nothing
};
void FunctorClass::operator() (double d) {
   Value = d;
}
```

Now

```
FunctorClass fn(5.0), *pfn = new FunctorClass(10.0);
fn(6); // actually fn.operator() (6) and Value is now 6, fn is an object treated as function
(*pfn)(6); // *pfn gives us an object
cout << fn() << endl; // actually cout << fn.operator()() and prints 6
cout << (*pfn)() << endl; // prints 6
```

Functors (2)

Generally the operator that overloads the function call is written as:

```
return_value_type operator() (input_parameter_list) { function_body }
```

or if we have separate **.h* and **.cpp* files:

```
return_value_type operator() (input_parameter_list); // prototype in *.h
return_value_type class_name::operator() (input_parameter_list)
{  function_body  } // definition is *.cpp
```

As a functor is an object, it has state (the collection of values of attributes). A function using variables with global lifetime has also state but ... A function has only one instance and the global variables it uses are freely attached and maybe modified by the other components of application or by the function itself:

```
int x = 0;
void fun() {
static int y = 0;
……………. // may modify x and / or y
}
int main() {
fun(); // after each call the state may be changed
……………..
```

Advantage of functors: we may create any number of functors and each of them has its own encapsulated state.

Functors (3)

```
class FunctorModifier {
private:
    int Coeff;
public:
    FunctorModifier(int i) : Coeff(i) { }
    int operator() (int i) { return i + Coeff; }
};
FunctorModifier fm1(1);  // modifies with coefficient 1
FunctorModifier fm2(2); // modifies with coefficient 2
cout << fm1(10) << ' ' << fm2(20) << endl;  // prints 11 and 22
                                            // actually fm1.operator() (10) and
                                            // fm2.operator() (20) were called
```

When using functions:

```
int Coeff = 1; // global
int FunModifier(int i)
{
     return i + Coeff;
}
cout << FunModifier(10) << endl;  // prints 11
Coeff = 2; // changes the global coefficient, serious side effects may occur if Coeff is also
                 // used elsewhere in the application
cout << FunModifier(20) << endl; // prints 22
```

Functors (4)

Functors may be used instead of pointers to functions. Let us have

```
void ProcessArray(int *p, int n, int i1, int i2, function<void(int)>pf)
{  // pf is a function wrapper
  ……………………… // checks input, throws exception if errors
  for (int i = i1;  i <= i2;  i++)   {
        (pf)(*(p + i)); // do something with each member of array
   }
}
class FunctorPrint {
  public: // no constructor, here we have just one method – the operator overloading
        void operator() (int x) const { cout << x << ' '; }
 };
```

Now:

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
FunctorPrint print; // compiler-created default constructor is applied
ProcessArray(arr, 9, 1, 5, print); // prints 2, 3, 4, 5
```

Value of argument *pf* is *print* – a functor, i.e. object of class in which call to function is overloaded. From this class an *operator()* with single argument of type *int* and without return value is searched and called.

Functors (5)

More examples:

```
void QuadEq(double a, double b, function<double()>pf, double *px1, double *px2)
{  // from slide Function wrappers (1)
  double d = b * b – 4 * a * (pf)();
  ………………………………….
}
FunctorClass fn(6.0), *pfn = new FunctorClass(6.0); // see slide Functors (1)
double x1, x2;
QuadEq(1, 5, fn, &x1, &x2); // actually d = b * b – 4 * a * fn()
                                                // or d = b * b – 4 * a * fn.operator()()
QuadEq(1, 5, fn() , &x1, &x2); // error
QuadEq(1, 5, *pfn, &x1, &x2); // dereferencing is applied
QuadEq(1, 5, FunctorClass(6.0), &x1, &x2); // creates nameless functor object, see
                                                                              // slide Initializing (7)
QuadEq(1, 5, FunctorClass { 6.0 }, &x1, &x2);
QuadEq(1, 5, FunctorClass ffn = { 6.0 }, &x1, &x2); // error
```

Similarly:

```
ProcessArray(arr, 9, 1, 5, FunctorPrint());
ProcessArray(arr, 9, 1, 5, FunctorPrint { });
```

Functors (6)

C++ defines several templates for functors that may replace simple functions and lambdas. Example:

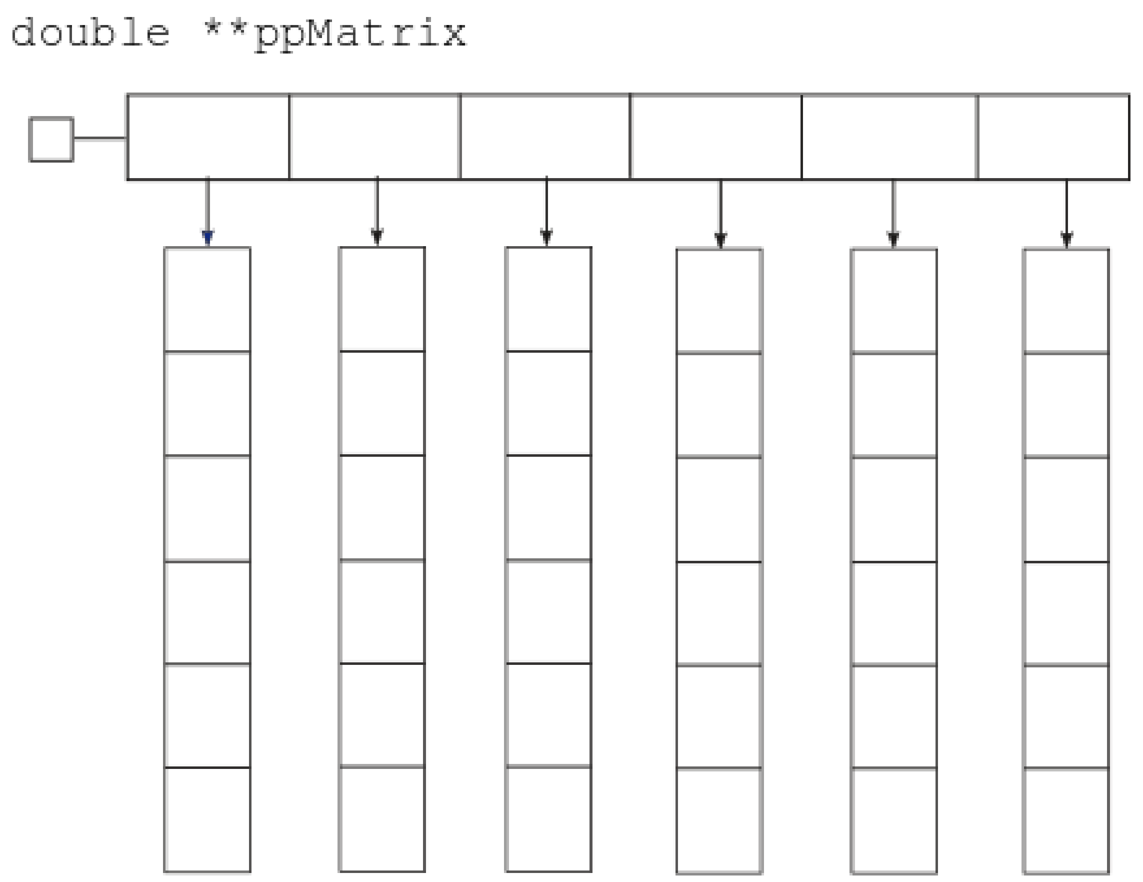
```
void QuadEq(double a, double b, function<double(double, double)>pf, double x, double y,
                        double *px1, double *px2)
{ // Here pf is a wrapper for functions with two double arguments, it returns also a double
   // The complete code is on slide Function wrapper (3)
  double d = b * b - 4 * a * (pf)(x, y);
  ………………………………………..
 }
plus<double> add; // template plus for adding two doubles, add is the functor name
double x1, x2;
try {
  QuadEq(1, 9, add, 7, 13, &x1, &x2);
  // here (pf)(x, y) is actually add(7, 13) and we get 20
}
catch (exception ex) {
   cout << ex.what() << endl;
}
```

There are also standard functors for other arithmetical operations (*minus*, *multiplies*, etc.), for comparison (*equal_to*, *less*, *greater*, etc.), logical operations and bitwise operations. See more on http://www.cplusplus.com/reference/functional/

Move semantics (1)

Suppose we have class Matrix:

```
class Matrix
{
private:
    int nRow = 0;
    int nColumn = 0;
    double **ppMatrix = nullptr;
public:
    Matrix() { }
    Matrix(int, int);
    Matrix(const Matrix &);
    ~Matrix();
    Matrix &operator=(const Matrix &);
    Matrix operator+(const Matrix &);
……………………………………….
};
```



Move semantics (2)

Suppose we have also a function with prototype:

```
Matrix Sum(Matrix &, Matrix &);
```

and code snippet

```
Matrix a, b;
Matrix c = Sum(a, b); // the same as c = a + b
```

As we use call by reference, *Sum* has access to *a* and *b*, therefore copying of arguments is not needed. But *Sum* has to create and return the temporary result matrix that is in turn the argument of copy constructor for *c*. The copy constructor copies all the values from result to *c*. At last the result as the local variable of *Sum* is removed (its destructor is called):

```
Matrix::Matrix(const Matrix &m)
{ // copy constructor, "this" is matrix c and m is the temporary return value of Sum
  this->nRow = m.nRow;
  this->nColumn = m.nColumn;
  this->ppMatrix = new double *[nRow];
  for (int i = 0; i < this->nRow; i++)
    *(this->ppMatrix + i) = new double[nColumn];
  for (int i = 0; i < this->nRow; i++)
     for (int j = 0; j < this->nColumn; j++)
         *(*(this->ppMatrix + i) + j) = *(*(m.ppMatrix + i) + j);
}
```

So we have 4 matrices: *a*, *b*, *c* and a temporary.

Move semantics (3)

But actually there is no need to allocate new vectors for matrix *c*, copy everything and at last destroy the temporary matrix from *Sum*. It is more reasonable to copy only the numbers of rows and columns and the pointer *ppMatrix*, i.e. simply capture the vectors from heap and use them in *c*. It is said that instead of making a copy we move heap data (actually we copy the pointers to them) from one object to another.

As at the end of *Sum* the destructor for its local temporary matrix is called anyway, during moving we must refuse to delete the data vectors. If the destructor is written in the following way:

```
Matrix::~Matrix()
{
  if (ppMatrix)
 { // if ppMatrix is set to 0, deletes nothing
      for (int i = 0; i < nRow; i++)
     {
           if (*(ppMatrix + i))
              delete *(ppMatrix + i);
      }
      delete ppMatrix;
  }
}
```

we must simply set the *ppMatrix* to *nullptr*.

Move semantics (4)

But the old copy constructor is still needed because the heap data moving is possible only when the original is a temporary matrix not needed afterwards. Consequently, we need two constructors: almost obligatory copy constructor and optional move constructor:

```
Matrix::Matrix(Matrix &&m)
{ // "this" is matrix c and m is the temporary return value of Sum
   this->nRow = m.nRow;
   this->nColumn = m.nColumn;
   this->ppMatrix = m.ppMatrix; // move data on heap
   m.ppMatrix = nullptr; // when the temporary matrix is removed, data on heap is kept
}
```

&& specifies a new data type: rvalue reference. The ordinary reference (&) or lvalue reference may refer only to lvalues located on a memory field that can be identified (by identifier, by array index, by pointer, etc.). The rvalue reference may refer to temporary objects we cannot identify. For example:

```
Matrix c = a + b;
```

Here a temporary matrix presenting the result of addition is created, but for us it has no name and cannot be handled. This matrix is an rvalue and it is wise to create *c* with the move constructor.

```
Matrix c = a;
```

Here we must create *c* with the copy constructor.

Move semantics (5)

The C++ compiler is able to detect whether to use copy constructor (argument is lvalue reference) or move constructor (argument is rvalue reference):

Matrix c = a + b; // if present, the move constructor is called; if not then the copy constructor
Matrix c = a; // the copy constructor is called

However,

Matrix c = Sum(a, b); // the copy constructor is called
The problem is that the compiler does not know what function *Sum* actually does and returns. For example, it may return not result of addition but one of the inputs. To force the call to move constructor, write:
Matrix c = move(Sum(a, b)); // std::move, if necessary, converts lvalue to rvalue

Unnecessary copying may also take place in *operator*= assignment overloading function. Therefore it may be wise to overload assignment twice: one with copying and the other with moving. The main ideas and the technique are the same as in case of constructors.

However, the move assignment operator function has an important difference: it must capture the heap data from temporary object standing right of the = sign, but it must also release its own heap data that has become outdated.

Matrix a(5, 5), b(5,5), c(5, 5);
………………. // set values to elements of a
b = a; // actually b.operator=(a); copy assignment needed
c = a + b; // actually c.operator(a +b); move assignment may be used

Move semantics (6)

```
Matrix &Matrix::operator=(const Matrix &m)
{ // b = a; here "this" means matrix b and m means matrix a
  if (this == &m)
      return *this;
   if (!this->ppMatrix || !m.ppMatrix)
      throw new exception("Empty operand(s)");
   if (m.nRow != this->nRow || m.nColumn != this->nColumn)
      throw new exception("Dimensions do not match");
  for (int i = 0; i < this->nRow; i++)
    for (int j = 0; j < this->nColumn; j++)
           *(*(this->ppMatrix + i) + j) = *(*(m.ppMatrix + i) + j); // overwrites the old values
  return *this;
}
Matrix &Matrix::operator=(Matrix &&m)
{// c = a + b; here "this" means matrix c and m means temporary matrix got after addition
  ……………………………. // the same as above written in brown font
   this->Destroy(); // the body is equivalent with the body of destructor
                            // removes the old values
   this->ppMatrix = m.ppMatrix;
   m.ppMatrix = nullptr;
   return *this;
}
```

Smart pointers (1)

Objects of smart pointer class (i.e. the smart pointers) automatically deallocate the memory to which they point. In the simplest cases it happens when the smart pointer goes out of its scope:

```
unique_ptr<item_type> pointer_name (memory_allocation_with_new_operator);
```

Example:

```
void fun()
{
 ……………………………………..
 unique_ptr<Date> pDate(new Date); // local variable pDate points to an object of class Date
 cout << pDate->GetYear() << endl; // operator -> is supported
 Date date = *pDate; // dereference is supported
 if (date == Date(1, 1, 2019)
      throw new exception("Not working day"); // pDate memory automatically released
 ……………………………………………………….
} // pDate memory automatically released
```

But there is no smart pointer arithmetics:

```
unique_ptr<double> pd(new double[10]); // allowed
for (int i = 0; i < 10; i++)
   *(pd + i) = 10; // compiler error, operations like pd++, pd[i], etc. not allowed
```

Smart pointers (2)

Copying of *unique_ptr* smart pointers is not allowed. Example:

```
unique_ptr<Date> pDate(new Date(29, 11, 2018));
unique_ptr<Date> pDate1 = pDate; // compile error
```

If you need several smart pointers to point to the same memory field, use *shared_ptr*:

```
shared_ptr<Date> pDate(new Date(29, 11, 2018));
shared_ptr<Date> pDate1 = pDate; // allowed
```

Example:

```
void fun(shared_ptr<Date>pd) {……} // usage of unique_ptr not possible
int main()
{
  shared_ptr<Date> pDate(new Date(29, 11, 2018));
  fun(pDate);
    // formal parameter pd of function fun is now out of scope but the memory of pDate
    // is not released. shared_ptr has a counter incremented each time when a new pointer
    // points to the resource and decremented when destructor of object is called. If the
    // counter becomes 0, the memory is released. Here when function fun is running, this
    // couter is 2
   return 0;
}
```

Older C++ versions define *auto_ptr* smart pointer. It is now deprecated.

Random numbers (1)

Software-based random number generators rely on some mathematical formulas and are therefore pseudo-random numbers. To get truly random numbers we need some hardware attached to the computer.

Random number engine *random_device* tries to find a hardware generator and in case of failure selects a software algorithm. The standard does not specify which algorithm: the choice is up to the library designer. The other engines generate only pseudo-random numbers. To declare an engine without serious mathematical background is very difficult. Therefore C++ has several predefined engines.

In addition to engines we need also distributions that describe how the random numbers are distributed within a range. The C++ standard specifies 20 distribution classes.

Example:

```
#include <random> // see http://www.cplusplus.com/reference/random/
default_random_engine generator; // the simplest predefined engine, no parameters needed
int lower_bound = 0, upper_bound = 100;
uniform_int_distribution<int> distribution(lower_bound, upper_bound);
for (int i = 0;  i < 10;  i++)
   cout << distribution(generator) << endl;
            // prints 10 pseudo-random numbers from range 0…100
```

Random numbers (2)

Example:

```
mt19937 generator(static_cast<unsigned long int>(time(nullptr)));
    // mt19937 is a predefined engine of type Mersanne_twister_engine
    // Mersanne_twister_engine is considered to generate the highest quality of randomness
    // It needs seed, here the current time from computer clock
double mean = 0, deviation = 1.0;
normal_distribution<double> distribution(mean, deviation);
for (int i = 0;  i < 10;  i++)
   cout << distribution(generator) << endl;
                  // prints 10 pseudo-random numbers
```

Rational numbers (1)

In mathematics, a rational number can be expressed as fraction *a / b*, where *a* is called as numerator and *b* as denominator. The decimal expansion of a rational number may have finite number of digits like *1.234*. But it may also have endless number of digits in which a sequence of digits is repeating over and over, like *7 / 3 = 2.33333…..*

An irrational number like *sqrt(2)*, *π*, *e* has also endless decimal expansion, but without repeating.

Problems with endless rational numbers:

```
double x = 2.3333333; // actually in specification this expression is written as 7 / 3
double y = x * 3; // we get 6.9999999, but the correct value is 7
```

To get results of calculations that are as exact as possible, we need to use template *ratio*:

```
typedef <numerator_as_integer_constant, denominator_as_integer_constant> ratio_name;
```

The denominator has default value 1. Examples:

```
#include <ratio> // see http://www.cplusplus.com/reference/ratio/ratio/
const int numerator = 7, denominator = 3; // must be constant expression
typedef ratio<numerator, denominator> test1;
typedef ratio<7, 3> test2;
```

To access numerator and denominator, use public members *num* and *den*, for example:

```
cout << test1::num << ' ' << test1::den<< endl;
```

Rational numbers (2)

The following expression is for adding ratios:

```
typedef ratio_add<addend_1 _as_ratio, addend_2_ as_ratio> ratio_name;
```

Example:

```
typedef ratio<7, 3> test1;
typedef ratio<5, 6> test2;
typedef ratio_add<test1, test2> sum;
cout << sum::num << ' ' << sum::den << endl; // prints 19 6
```

ratio_subtract, *ratio_multiply* and *ratio_divide* are similar.

An *integral_constant* is a standard class (better to say *struct*) template that stores the type and constant value. For example, *integral_constant<bool, true>* stores a boolean value *true* and *integral_constant<int, 100>* stores integer *100*. It has two members: *type* and *value*.

To compare two ratios write expression:

```
typedef ratio_equal<ratio_1, ratio_2> integral_constant_name;
```

The results is *integral_constant<bool, true>* or *integral_constant<bool, false>*

Example:

```
typedef ratio<7, 3> test1;
typedef ratio<5, 6> test2;
typedef ratio_equal<test1, test2> res;
cout << (res::value ? "Equal" : "Not equal") << endl;
```

Rational numbers (3)

ratio_not_equal, *ratio_less*, *ratio_less_equal*, *ratio_greater*, *ratio_greater_equal* are similar.

All the ratio templates are evaluated at compile time. The values for numerator and denominator cannot be calculated at run time, for example:

```
int x;
cin >> x;
typedef ratio <x, 2> test; // error
```

There are no C++ operations between rational numbers and integers or doubles. So, if we have

```
typedef ratio<5, 6> test2;
```

and we want to multiply it with 2, we need to write

```
typedef ratio<2, 1> test3; // or simply ratio<2>
typedef ratio_multiply<test2, test3> test4;
cout << test4::num << ' ' << test4::den << endl; // prints 5 3
```

C++ has several predefined ratios, for example *micro* (i.e. 1 / 1e6), *milli* (i.e. 1 / 1e3), *kilo* (i.e. 1e3 / 1), *mega* (i.e. 1e6 / 1), etc.

Time handling (1)

In classical C the reading of current time from the system clock is performed as follows:

```
#include "time.h"
time_t now; // time_t is specified by typedef, in Visual Studio it is is a 64-bit integer
time(&now); // the number of seconds since January 1, 1970, 0:00 UTC
```

To get the current date and time understandable for humans use the standard *struct tm*:

```
struct tm // do not declare it in your code, it is already declared by time.h
{
    int tm_sec;   // seconds after the minute - [0, 60] including leap second
    int tm_min;   // minutes after the hour - [0, 59]
    int tm_hour;  // hours since midnight - [0, 23]
    int tm_mday;  // day of the month - [1, 31]
    int tm_mon;   // months since January - [0, 11], attention: January is with index 0
    int tm_year;  // years since 1900, attention, not from the birth of Christ
    int tm_wday;  // days since Sunday - [0, 6], attention: Sunday is with index 0, Monday 1
    int tm_yday;  // days since January 1 - [0, 365]
    int tm_isdst; // daylight savings time flag
};
```

To fill this struct:

```
struct tm now_tm;
localtime_s(&now_tm, &now);
```

Time handling (2)

Example:

```
printf("Today is %d.%d.%d\n",
   now_tm.tm_mday, now_tm.tm_mon + 1, now_tm.tm_year + 1900);
```

Function *asctime_s* converts the *struct tm* to string:

```
char buf[100];
asctime_s(buf, 100, &now_tm);
printf("%s\n", buf); // prints like Thu Jan 23 14:26:42 2020
```

but here we cannot set the format. Better is to use function *strftime*, for example:

```
strftime(buf, 100, "%H:%M:%S %d-%m-%Y", &now_tm);
printf("%s\n", buf);   // prints according to Estonian format 14:26:42 23-01-2020
```

The complete reference of *strftime* is on http://www.cplusplus.com/reference/ctime/strftime/

The attributes of *struct tm* may be modified. For example, if we want to know what date is after 100 days, do as follows:

```
struct tm future_tm = now_tm;
future_tm.tm_mday += 100; // add 100 days
time_t future = mktime(& future_tm); // convert back to time_t
localtime_s(&future_tm, &future); // convert once more to struct tm
asctime_s(buf, 100, &future_tm);
printf("%s\n", buf); // prints like Sat May 2 15:26:42 2020
```

Time handling (3)

In C++ we have more powerful but complicated tools:

```
#include <chrono> // See http://www.cplusplus.com/reference/chrono/
using namespace std::chrono; // do not forget!
```

Namespace *chrono* includes five concepts: *system_clock*, *steady_clock*, *high_resolution_clock*, *time_point* and *duration*. *Duration* and *time_point* are components of clocks.

- *system_clock* represents timepoints associated with the computer usual real-time clock.
- *steady_clock* guarantees that it never gets adjusted.
- *high_resolution_clock* represents the clock with the shortest possible tick period. In Visual Studio equivalent with the *system_clock*.

To read the current time:

```
system_clock::time_point now = system_clock::now();
```

Turn attention, that a *time_point* is always associated with a clock:

```
time_point<system_clock> t; // correct
system_clock::time_point t; // correct
steady_clock::time_point t = steady_clock::now();
time_point t; // error, clock not specified
```

The time_point has epoch (or origin, 01.01.1601 in case of Windows, 01.01.1970 in case of Linux). Its value is actually the duration from the epoch (measured in 100ns units in case of Windows and seconds in case of Linux).

Time handling (4)

It seems to be more convenient to continue with C time handling tools:

```
time_t now_t = system_clock::to_time_t(now); // convert to time_t
struct tm now_tm;
localtime_s(&now_tm, &now_t);
struct tm future_tm = now_tm;
future_tm.tm_mday += 100; // add 100 days
time_t future_t = mktime(& future_tm);
```

There is a standard function std::put_time to create from *struct tm* time strings for *iostream* and *sstream*:

```
#include <iomanip>
cout << put_time(&future_tm, "%d-%m-%Y %H:%M:%S") << endl;
```

or

```
stringstream sout;
sout << put_time(&future_tm, "%d-%m-%Y %H:%M:%S") << endl;
cout << sout.str() << endl;
```

See more from http://www.cplusplus.com/reference/iomanip/put_time/

To turn back to C++ tools:

```
system_clock::time_point future = system_clock::from_time_t(future_t);
```

Time handling (5)

Template *duration* (see http://www.cplusplus.com/reference/chrono/duration/) represents an interval between two timepoints:

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

Here *T1* is used for variable storing the number of ticks (*int*, *long int*, *double*, etc.) and *T2* is for ratio presenting the period of one tick. The default value for *T2* is *ratio<1, 1>* (or simply *ratio<1>*). Examples:

```
duration<long int> d1; // ratio has default value, it means that tick is one second
duration<long int , ratio<60, 1> > d2; // tick is one minute
duration <long int, milli>  d3; // tick is one millisecond
duration <long long int, ratio<1, 10>>  d4; // tick is one tenth of second
```

Constructor without parameters does not initialize the number of ticks.

```
duration <long int, milli>  d3(1000); // now the initial duration is 1000 ms
```

There are several typedefs for typical durations. Examples:

```
hours d1(24);              // declares time interval 24 hours
minutes d2(10);            // declares time interval 10 minutes
seconds d3(20);            // declares time interval 20s
milliseconds d4(1500);     // declares time interval 1500ms
microseconds d5(1500);     // declares time interval 1500µs
nanoseconds d6(1500);      // declares time interval 1500ns
```

Time handling (6)

Method *count* returns the value of ticks, for example:

```
cout << d1.count() << endl;
```

Duration has a large set of operator functions for arithmetics and comparison. The full list is on http://www.cplusplus.com/reference/chrono/duration/operators/. The operands may be of different types. Examples:

```
milliseconds d1(1000);
milliseconds d2(2000);
milliseconds d3 = d1 + d2; // get time interval 3000ms
cout << boolalpha << (d1 < d2) << endl;
seconds d5(1);
nanoseconds d6 = d5 + d3; // different units, get time interval 4000000000ns
milliseconds d7 = d3 * 2; // get time interval 6000ms
hours d9(1); //one hour
seconds d10 = (seconds)d9; // casting, get time interval 3600s
```

but

```
milliseconds d11 = (milliseconds)d6; // error
```

The simple casting is possible if there is implicit cast between types used for ticks. Here the *milliseconds* uses *long int* and *nanoseconds* uses *long long int*. But there is a special cast template (see http://www.cplusplus.com/reference/chrono/duration_cast/):

```
milliseconds d11 = duration_cast<milliseconds>(d6);
```

Time handling (7)

Actually, *time_point* (see http://www.cplusplus.com/reference/chrono/time_point/) is a template:

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

Here *T1* is used for clocks (*system_clock*, *etc*.) and *T2* for duration, i.e. the interval between the current moment and the epoch. For example, if we write:

```
time_point<system_clock, duration<long long int, ratio<1, 1> > > t;
```

then *t* measures the number of seconds from epoch, the value is retrieved from system clock. Theoretically we may declare timepoints in many different ways but actually the duration parameters (epoch and tick period) are built into clock. Consequently, each clock must have its own standard for timepoint:

```
system_clock::time_point now = system_clock::now();
```

To know which ratio is used in the duration of your system clock, write the following code snippet:

```
cout << system_clock::period().num << " " <<  system_clock::period().den << endl;
```

On the instructor's computer the result was *1 10000000*.

To know what is the type for ticks in the duration of your system clock, write the following code snippet:

```
cout << typeid(system_clock::rep).name() << endl;
```

On the instructor's computer the result was *__int64*.

Time handling (8)

Timepoint has a set of operator functions for arithmetics and comparison. The only operation between two timepoints is subtraction, its result is a duration:

```
system_clock::time_point start = system_clock::now();
int i;
cin >> i; // to introduce a pause
system_clock::time_point end = system_clock::now();
auto diff = end - start;
cout << typeid(diff).name() << endl;
```

the result is *class std::chrono::duration<__int64,struct std::ratio<1,10000000> >* , i.e. the type of duration presenting the difference between two timepoints is the same as the duration in system_clock::time_point.

Due to casting problems we may convert implicitly the difference into nanoseconds but not to milliseconds or seconds:

```
nanoseconds dn = (nanoseconds)diff;
milliseconds dm = (milliseconds)diff;
seconds ds = (seconds)diff;
seconds ds = duration_cast<seconds>(diff); // duration_cast template works
cout << dn.count() << "ns" << endl; // prints 3669534600 nanoseconds
cout << ds.count() << "s" << endl; // prints 3 seconds
```

Time handling (9)

There are operator functions for operations between timepoints and durations. Examples:

```
system_clock::time_point now = system_clock::now();
system_clock::time_point future = now + hours(1);
system_clock::time_point past = now - hours(365 * 24);
cout << boolalpha << (now < future) << endl;
```

A very detailed discussion about time handling problems in C++ can be found on page
http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2