New variable types (1)

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 (2)

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 (3)

```
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>
```

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.

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:

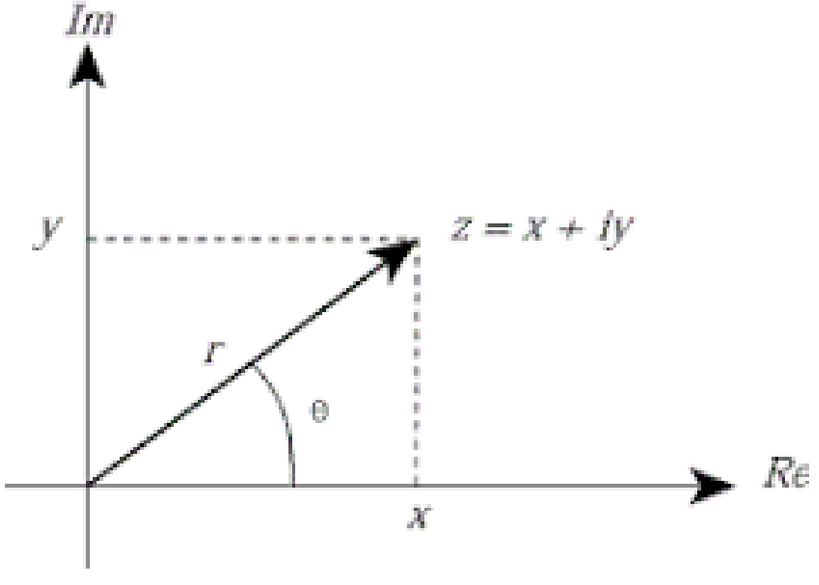
```
#include <complex>
#include <numbers>
using namespace std;
complex<double> c(10, 20);
cout << conj(c) << endl; // prints the conjugate (10, -20)
cout << abs(c) << endl; // prints the absolute value or modulus sqrt(Re^2 + Im^2)
                        // 22.3607
cout << norm(c) << endl; // prints the norm (Re^2 + Im^2) or modulus^2
                         // 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)
```



Value arrays (1)

Value arrays are containers designed for storing numeric values. Examples:

```
#include <valarray> // see http://www.cplusplus.com/reference/valarray/valarray/
valarray<int> va1; // empty
valarray<double> va2(10); // initialized with zeroes
valarray<int> va3 = { 1, 2, 3 }; // va3[0] is 1, va3[1] is 2, va3[2] is 3
initializer_list<int> il = { 4, 5, 6 };
valarray<int> va4(il); // va4[0] is 4, va4[1] is 5, va4[2] is 6
```

There are also copy and move constructors. The large set of operator functions allows very efficiently to operate with value arrays. Some examples:

```
valarray<int> va1 = { 1, 2, 3 }, va2 = { 4, 5, 6 };
valarray<int> va3 = va1 + va2; // get value array 5, 7, 9
valarray<int> va4 = -va1; // get value array -1, -2, -3
valarray<int> va5 = va1 - va2; // get value array -3, -3, -3
valarray<int> va6 = va1 * va2; // get value array 4, 10, 18
va6 += va2; // get value array 8, 15, 24
valarray<int> va7 = va6 - 1; // get value array 7, 14, 23
va7 -= 2; // get value array 7, 14, 23
va7 = 2; // get value array 2, 2, 2
valarray<double> va8 = { 16, 25, 36 };
valarray<double> va9 = sqrt(va8); // get value array 4, 5, 6
```

Value arrays (2)

Examples about methods implemented for value arrays:

```
valarray<int> va10 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
cout << va10.min() << ' ' << va10.max() << ' ' << va10.sum() << endl; // prints 1, 9, 45
valarray<int> va11 = va10.shift(3); // get 4, 5, 6, 7, 8, 9, 0, 0, 0
valarray<int> va12 = va10.shift(-3); // get 0, 0, 0, 1, 2, 3, 4, 5, 6
valarray<int> va13 = va10.cshift(3); // get 4, 5, 6, 7, 8, 9, 1, 2, 3 (circular shift)
valarray<int> va14 = va10.cshift(-3); // get 7, 8, 9, 1, 2, 3, 4, 5, 6
```

For non-standard operations use method *apply()*:

```
valarray<int> va15 = va10.apply([](int i) { return i % 2 ? i : -i; }); // get 1, -2, 3, -4, 5, -6, 7, -8, 9
```

Value array is a dynamic array (i.e. its size is changeable):

```
cout << va15.size() << endl; // prints 9
va15.resize(5); // get 0, 0, 0, 0, 0
va15.resize(3, 10); // get 10, 10, 20
```

Value array objects do not have iterators. But there are standard functions *std::begin()* and *std::end()* taking *valarray* objects as arguments:

```
for (auto it = begin(va10);  it != end(va10);  it++)
{
    cout << *it << ' ';
}
```

Value arrays (3)

A new value array may be created from an existing value array by selecting the elements to include. For that we need slice selector:

```
slice slice_name(starting_index, size, stride);
```

Example (see also http://www.cplusplus.com/reference/valarray/slice/):

```
valarray<int> va16(20);
int j = 0;
for (int &i : va16) {
    i = ++j; // get 1, 2, 3, …., 20
}
slice slicer(1, 3, 5);
valarray<int> va17 = va16[slicer]; // get 2, 7, 12
```

Comment: as the starting index is 1, the first element of selection is *va16[1]*. As the stride (i.e. step) is 5, the following elements of selection are *va16[6]*, *va16[11]*, … The size of selection here is 3.

Selection can be provided also with mask.

Example (see also http://www.cplusplus.com/reference/valarray/mask_array/):

```
valarray<bool> mask(20);
mask[5] = mask[7] = mask[11] = true;
valarray<int> va18 = va16[mask]; // get 6, 8, 12
```