- Inline variables

  inline static [type] [name] = [value];

  ~ special case

  static constexpr [type] [name] = [value];

  * constexpr jest od razu inline *

- Bit operators

  ~ | - bitwise OR

```
int a=5;          // 0101b
int b=12;         // 1100b
int c = a | b;    // 1101b      13
```

  ~ ^ - bitwise XOR

```
int a = 5;        // 0101b
int b = 9;        // 1001b
int c = a ^ b;    // 1100b      12
```

  ~ & - bitwise AND

```
int a = 6;        // 0110b
int b = 10;       // 1010b
int c = a & b;    // 0010b      2
```

  ~ << - left shift

```
int a = 1;        // 0001b
int b = a << 1;   // 0010b      2
```

  ~ >> - right shift

```
int a = 2;        // 0010b
int b = a >> 1;   // 0001b      1
```

- Bit fields

```
struct date {
    unsigned int year : 13;   // 2^13 = 8192
    unsigned int month : 4;   // 2^4 = 16
    unsigned int day : 5;     // 2^5 = 32
}
```

- Array size : ~~compile~~ type safe at compile time

```
typedef int array[5][24][60];
char cArray[] = { 'a', 'b', 'c', 'd' };

std::extent<array, 0>::value          // 5
std::extent<array, 2>::value          // 60
std::extent<decltype(cArray)>::value  // 4
```

- Iterators

```
std::vector<int> ve = { 1, 2, 3, 4, 5, 6, 7 };
std::vector<int>::iterator first = ve.begin();   // 1
auto last = ve.end();                            // ~~wyjątek pamięci~~
first++;                                         // 2
std::advance(first, 2);                          // 4
last--;                                          // 7
std::advance(last, -2);                          // 5
first = std::next(first, 3);                     // 7
last = std::prev(last, 2);                       // 3
auto distance = std::distance(first, last);      // -4
```

- Iteratory
  - Przykład na poprzedniej kartce sprawdza się w innych kontenerach
  - Gdy kontenery są const zostaje używany oraz zwracany const_iterator.
  - Gdy chcemy otrzymać const_iterator, a nie używamy const kontenera ~~się~~ korzystamy z cbegin() i cend()
  - Odwrotny iterator to reverse_iterator. W tym przypadku korzystamy z rbegin() i rend();
  - Zmiana reverse_iterator na iterator odbywa się poprzez metodę base().

```
std::vector<int>::iterator first = ve.begin();
std::vector<int>::reverse_iterator rFirst = ve.rbegin();
first = rFirst.base();
```

- File I/O

std::istream for reading text >>
std::ostream for writing text; <<
std::streambuf for reading or writing characters.

~ Zapisywanie do pliku

```
std::ofstream os("foo.txt");
if(os.is_open())
{os << "Hello World";
 lub
{char text[] = "Hello World";
os.write(text, 12);
 if(os.bad())
      // failed to write
```

~ Odczytywanie z pliku

```
std::ifstream os("foo.txt");
// Załóżmy wygląd 1 linii: Mateusz Rzeczyca 7 8 1999
std::string name, surname;
int day, month, year;
os >> name >> surname >> day >> month >> year;
```

~ Opening modes

```
std::fstream os("foo.txt", [mode] | [mode]);
```

std::ios::app	// append	std::ios::trunc	// truncate
std::ios::binary		std::ios::ate	// atend
std::ios::in	//input		
std::ios::out	// output		

~ Kopiowanie pliku

```
std :: ifstream src ("foo1.txt", std::ios::binary);
std:: ofstream dst ("foo2.txt", std::ios :: binary);
dst << src . rdbuf ();
// C++17
std :: filesystem:: copy_file ("source", "destination");
```

~ Zamykanie pliku

```
os. close ();
```

- C++ streams

```
# include <sstream>
```

~ string stream

```
ostringstream ss;                    // regular stream
ss << "Hello World";
std :: string result = ss. str ();
```

~ Printing collections

```
std :: ostream_iterator              // prints content of STL container
```

```
std :: copy (ve. begin (), ve. end (), std:: ostream :: iterator <type> (std:: cout, " "));
```

* przykład podzielności liczb przez 2 int, string, anything

```
std :: transform (ve. begin (), ve. end (), std :: ostream:: iterator <bool> (std::cout, " "), ->
                  -> [](int val) { return (val % 2) == 0; });
```

- Stream manipulators

```
#include <iomanip>
```

std:: boolalpha → zamienia 1 - true, 0 - false

std:: noboolalpha → zamienia true - 1, false - 0.

std:: showbase
std:: noshowbase } pokazuje / nie pokazuje systemu liczbowego

std:: oct
std:: dec
std:: hex } zamienia na ... system liczbowy

std:: setw(n) - ile spacji ma wykonać
std:: setfill(' ') - czym te spacje wypełnić

```
#include <locale>
```

std:: left
std:: right
std:: internal } pozycje dla std:: setfill(' ') std:: setw(n)

std:: fixed
std:: scientific
C++11 { std:: hexfloat
C++11 { std:: defaultfloat } zmienia format dla float

std:: showpoint
std:: noshowpoint } ustawia kropkę przy typie float

std:: showpos
std:: noshowpos } gdy liczba dodatnia pokazuje (lub nie) '+' przy niej

std:: setprecision(n) - ustawia ilość liczb po przecinku

std:: ends - dodaje '\0' na koniec

std:: flush - czyści output stream i nie przechodzi do next line

---

- Avoid duplication of code in const and non-const getter methods

```
struct foo
{
    Bar& GetBar(/* arguments */)
    {
        return const_cast<Bar&>(const_cast<const Foo*>
                (this)->GetBar(/*arguments */));
    }
    const Bar& GetBar(/* arguments */) const
    {
        /* some calculations */
        return foo;
    }
```

- mutable
  - ~ lambda

```
int a=0;
auto godCounter = [a]() mutable
{
} return a++;
```

  - ~ non-static class member modifier

```
class foo
{
  public:
    foo getFoo() const
    {
      mVariable = true;
      /* other stuff */
    }
  private:
    mutable bool mVariable = false;
```

- friend
  - ~ function

```
void friendF();
class Foo
{
} friend void friendF();
```

  - ~ method

```
class Foo
{
} void getFoo();

class another
{
} friend void Foo::getFoo();
```

  - ~ class

```
class Foo
{
};

class another
{
} friend class Foo;
```

- Different keywords
  - ~ void* points to everything that is not declared const or volatile
  - ~ volatile can be modified by the hardware
  - ~ virtual - function or ~~class~~ base class
  - ~ noexcept - checks if operand can propagate an exception

- Returning several values from a function
  - ~ std::tuple # include <tuple>

```
std::tuple<int, int, int, int> foo(int a, int b)
{
    return std::make_tuple(a+b, a-b, a*b, a/b);   // C++11
    return {a+b, a-b, a*b, a/b};                  // C++17
}

int a = std::get<0>( foo(5, 5) );   // 10

int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 5);
std::tie(add, sub, std::ignore, div) = foo(10, 2);
                          ↑ if not needed
auto [add, sub, mul, div] = foo(12, 3);   // C++17
```

∩ structured bindings (C++17)

```
std::map<std::string, int> m;
auto [iterator, success] = m.insert({"Hello", 42});

for (auto const& [key, value] : m)
    std::cout << "The value for" << key << "is" << value << "\n";

std::pair<int, int> foo(int a, int b)
{
    return std::make_pair(a+b, a-b);
    return {a+b, a-b};
}

std::pair<int, int> p;
p = foo(5, 12);
std::cout << p.first << " " << p.second << "\n"; //17 -7
```

- Make it obviously which parameter should be

```
return;

# define OUT

int void calc(int& a, int& b, OUT int& c)
{
    c = a+b;
    return c;
}
```

• Polymorphism

```
class shape
{
public:
    virtual ~shape() = default;
    virtual double get_surface() const = 0;
};
```

```
class square : public shape
{
public:
    double get_surface() override;
};
```

~ Polymorphic class should always have default destructor.

~ Function ended with = 0 is pure virtual function

~ You can have only pointers or references of an abstract class

~ dynamic_cast for polymorphic classes

• Copy constructor vs Assignment constructor

```
Foo(const Foo& rhs)
{
    data = rhs.data;
}
```

```
Foo& operator=(const Foo& rhs)
{
    data = rhs.data;
    return *this;
}
```

```
Foo a(2);   // normal constructor  Foo(int data)
Foo b = a;  // Copy constructor called
Foo c(42);
c = a;      // Assignment constructor called
```

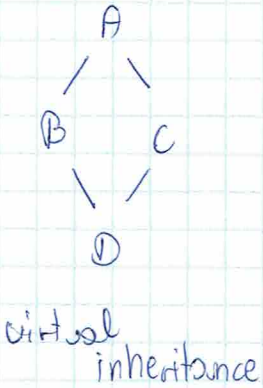
- Final classes / structs

```
class A final
{
};
```

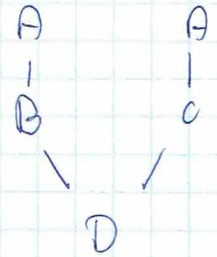
~ cannot inheritate

- Virtual inheritance

```
class A: public virtual B
{
};
```



virtual inheritance



normal inheritance

- Unnamed struct / class

```
struct
{
} center;
```

which means center is variable of this struct

- Member function cv-qualifier overloading

```
class Integer
{
  void print(int);
  void print(int) const;
};
```

```
Integer i;
const Integer &iC = i;
i.print(5);   // print(int)
iC.print(6);  // print(int) const
```

- Operator overloading
  - outside of class / struct

```
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}
```

  - inside of class / struct

```
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}
```

- Final virtual functions

```
class A
{
    virtual int get();
};

class B : public A
{
    int get() final;
};

class C : public B
{
virtual int get(); // error
};
```

- Default constructor

```
class A
{
    A() = delete;
};
```

- default constructor is explicitly deleted

```
class A
{
    A() = default;
};
```

```
std::is_default_constructible<  >()
                  ↑
              type name
```

* same with destructor *

```
std::is_destructible<  >()
                ↑
            type name
```

- Encapsulation.

```
class A
{
    int a;
public: int getA() const { return a; }
    int setA(int e) { a = e; }
};
```

- Redeclaring members from base class to avoid name hiding

```
struct A
{
    void foo(int);
};
```

```
struct B : A
{
    using A::foo;
    void foo(std::string);
};
B objectB;
objectB.foo(42); // correct
```

- Inheriting constructors

```
struct A
{
    A(int, std::string);
};
```

```
struct B : A
{
    B(int x, std::string s) : A(x, 5) {}
};
```

```
struct C : A
{
    using A::A;
};
```

- Std:: string to (const) char*

```
std:: string str;                    cstr = str.c_str();
const char* cstr;                    cstr = str.data();
```

```
std:: unique_ptr<char []> cstr = std:: make_unique<char []>
                                 (str.size() + 1);
```

```
char* unsafe = new char [str.size() +1];

std:: copy (str.data(), str.data() + str.size(), cstr);
cstr [str.size()] = '\0';
```

- Converting to string

std:: ostringstream can be used to convert any streamable type to a string representation. Example:

```
int val =4;
std:: ostringstream str;
str << val;
std:: string s = str.str();
```

- std::array

```
std::array<int, 3> a{0, 1, 2};

a.at(0)
a[0]
std::get<0>(a)
```
} the same command

- std::vector

```
std::vector<int> v{0, 1, 2};

v.at(0)
v[0]
```
} the same command

* Remember about their functionality *

- std::map

```
std::map<std::string, int> m{ std::make_pair("Mateusz", 19),
                              std::make_pair("Cpp", 6)};

std.cout<< m.at("Mateusz");
```

There is also std::multimap

- std::optional

```
struct Animal
{
};
    std.string name;

struct Person
{
    std::string name;
    std::optional<Animal> pet;
};
```

True when pet is definied —>

False if not definied —>

```
int main()
{
    Person mateusz;
    mateusz.name = "Mati";
    if (mateusz.pet)
        std::cout << "Mateusz has a pet";
    else
        std::cout << "He doesn't have a pet";
}
```

std::optional - continuation

~ You can return it instead of nullptr without resorting memory allocation.

- std::function #include <functional>

```
std::function<void(int)> myFunc;
void setInteger(int x) { a = x; }
int main()
{
    myFunc = setInteger;
}   myFunc(10);
```

- std::forward_list #include <forward_list>

~ provides more space efficient storage when bidirectional iteration is not needed. (compared to std::list)

- std::pair #include <utility>

```
std::pair<int, int> p = std::make_pair(1, 2);
```

- std::atomic #include <atomic>

- std:: variant

  In this we can store on of either type in it

```
std:: variant<int, std:: string> var;
var = "hello";
auto * sVar = std:: get_if<std: string>(var); // *sVar = "hello"
auto * iVar = std:: get_if<int>(var); // compiler issue

auto lVar = [](auto &&e) {
        std :: cout << e << std:: endl;
};

std:: visit(lVar, var); // hello
```

- #include <iomanip>

  std:: setprecision , std:: setfill , std:: setiosflags
  std:: setw

- std:: any

```
std:: any aObject { std:: string ("Hello World") };
if ( aObject. has_value() )
        std:: cout << std:: any_cast<std:: string>(aObject); // HelloWorld
try {
} std:: cout << stol:: any_cast<int>(aObject);
catch ( std :: bad_any_cast &) {
} std:: cout << " Wrong Type ";
                                        // Wrong Type
```