C++11 w domu i zagrodzie

Bartosz Szreder

O czym **nie** zamierzam mówić

O czym **nie** zamierzam mówić

. . . co nie znaczy, że nie mogę.

O czym **nie** zamierzam mówić

. . . co nie znaczy, że nie mogę.

- `<regex>`
- `<memory>`: `unique_ptr`, `shared_ptr`, `weak_ptr`
- λ-abstrakcja
- *variadic templates*, `<tuple>`, metaprogramowanie
- wielowątkowość
- *rvalue reference*, *move semantics*
- delegacja konstruktorów

Ostateczne rozwiązanie kwestii NULL-pointerów

NULL nie jest częścią języka, a makrodefinicją.

```
//<cstddef>
#ifndef __cplusplus
#define NULL ((void *)0)
#else   /* C++ */
#define NULL 0
```

Problem

```
void f(int);
void f(char *);
```

Problem

```
void f(int);
void f(char *);
```

Co się stanie jak wywołamy `f(NULL)`?

Problem

```
void f(int);
void f(char *);
```

Co się stanie jak wywołamy `f(NULL)`?

Żeby nie było takich sytuacji, mamy od teraz `nullptr`:

```
f(nullptr);
int *p = nullptr;
```

„Specyfikatory" override i final

Jawnie deklarujemy metody jako *przeciążające* bazowe metody wirtualne:

```
class A {
public:
    virtual void f(int);
    virtual void g(int);
};

class B : public A {
public:
    void f(int) override;          //OK
    void g(long long) override; //compile error
};
```

Możemy też zabronić przeciążania:

```
class A {
public:
    virtual void f(int);
    virtual void g(int) final;
};

class B : public A {
public:
    void f(int) override; //OK
    void g(int) override; //compile error
};
```

Możemy nawet zabronić dziedziczenia:

```
class A final { ... };
class B : A { ... }; //compile error
```

Gdzie to się może przydać?

Gdzie to się może przydać?

Przykład 1.

Zmiana sygnatury metody w klasie bazowej i niezrobienie tego w klasie pochodnej. Mamy złapanie błędu na etapie kompilacji.

Gdzie to się może przydać?

Przykład 1.

Zmiana sygnatury metody w klasie bazowej i niezrobienie tego w klasie pochodnej. Mamy złapanie błędu na etapie kompilacji.

Przykład 2.

Real-life problem, który udało mi się wygenerować programując w Qt:

```
class SomeQtClass {
public:
    virtual int rowCount(const QModelIndex &) const;
};

class MyClass : public SomeQtClass {
public:
    int rowCount(const QModelIndex &);
};
```

Gdzie to się może przydać?

Przykład 1.

Zmiana sygnatury metody w klasie bazowej i niezrobienie tego w klasie pochodnej. Mamy złapanie błędu na etapie kompilacji.

Przykład 2.

Real-life problem, który udało mi się wygenerować programując w Qt:

```
class SomeQtClass {
public:
    virtual int rowCount(const QModelIndex &) const;
};

class MyClass : public SomeQtClass {
public:
    int rowCount(const QModelIndex &);
};
```

Gdzie jest błąd?

Operatory rzutowania/konwersji explicit

Dobra praktyka z poprzedniego standardu: niektóre konstruktory warto prefiksować słowem kluczowym `explicit`. Bez tego kompilator ma prawo dokonać niejawnej konwersji np. w takiej sytuacji:

```
class MyClass {
public:
    MyClass(int);
};

void f(MyClass);

f(42);              //OK
MyClass x = 42;     //OK
```

Wersja z konstruktorem explicit

```
class MyClass {
public:
    explicit MyClass(int);
};

f(42);                          //compile error
f(MyClass(42));                 //OK
MyClass x = 42;                 //compile error
MyClass x(42);                  //OK
MyClass x = MyClass(42);        //OK
```

Poprzedni standard dozwalał też na implementację operatorów konwersji:

```
class MyClass {
public:
    operator int() const
    {
        return value;
    }

private:
    int value;
};

void f(int);

MyClass x;
f(x);       //OK
int a = x; //OK
```

Można w ten sposób konwertować w dowolnie złożone typy, np. `QString` lub własne. Przydatne chociażby w konwersji do napisów celem wypisek debugowych. Silne, ale niebezpieczne. Nowy standard umożliwia definiowanie takich operatorów jako `explicit`.

Można w ten sposób konwertować w dowolnie złożone typy, np. `QString` lub własne. Przydatne chociażby w konwersji do napisów celem wypisek debugowych. Silne, ale niebezpieczne. Nowy standard umożliwia definiowanie takich operatorów jako `explicit`.

```
class MyClass {
public:
    explicit operator int() const;
};

void f(int);

MyClass x;
f(x);                           //compile error
f(static_cast<int>(x));         //OK
f(int(x));                      //OK
int a = x;                      //compile error
int a = static_cast<int>(x);    //OK
int a = int(x);                 //OK
int a(x);                       //OK
```

Wsparcie dla napisów UTF

W poprzednim standardzie oprócz `"standardowych"` napisów typu `char *` umieszczanych w cudzysłowie można było użyć przedrostka `L` i pisać `L"napisy"` typu `wchar_t *`.

Wsparcie dla napisów UTF

W poprzednim standardzie oprócz `"standardowych"` napisów typu `char *` umieszczanych w cudzysłowie można było użyć przedrostka `L` i pisać `L"napisy"` typu `wchar_t *`.

W zasadzie nikt nie wie co to jest `wchar_t` (*wide char*).

Wsparcie dla napisów UTF

W poprzednim standardzie oprócz `"standardowych"` napisów typu `char *` umieszczanych w cudzysłowie można było użyć przedrostka `L` i pisać `L"napisy"` typu `wchar_t *`.

W zasadzie nikt nie wie co to jest `wchar_t` (*wide char*).

Od teraz można też definiować napisy:

- UTF-8 przedrostkiem `u8` (typ `char *`),
- UTF-16 przedrostkiem `u` (nowy typ `char16_t *`),
- UTF-32 przedrostkiem `U` (nowy typ `char32_t *`).

Pojedyncze „krzaki” można wstawiać ciągiem heksadecymalnym. Zamiast zwyczajowego `0x` wstawiamy przedrostek:

- dla UTF-8 i UTF-16 cztery cyfry (2 bajty) `\u1234`,
- dla UTF-32 osiem cyfr (4 bajty) `\U12345678`.

Pojedyncze „krzaki” można wstawiać ciągiem heksadecymalnym. Zamiast zwyczajowego `0x` wstawiamy przedrostek:

- dla UTF-8 i UTF-16 cztery cyfry (2 bajty) `\u1234`,
- dla UTF-32 osiem cyfr (4 bajty) `\U12345678`.

W zasadzie jeśli używamy Qt, to pewnie lepiej jest korzystać z tamtejszych wynalazków, jak np. `QString::fromUtf8()`. Ale nie wszyscy korzystają z Qt...

Raw string literals

Jeśli musimy wpisać do kodu źródłowego tekst o sporej liczbie śmiesznych znaczków typu apostrofy, cudzysłowy, backslashe itp. (regexpy, interpretery, parsery, HTML, XML, OMG, WTF...), to szybko można się pogubić w *escapingu*.

Raw string literals

Jeśli musimy wpisać do kodu źródłowego tekst o sporej liczbie śmiesznych znaczków typu apostrofy, cudzysłowy, backslashe itp. (regexpy, interpretery, parsery, HTML, XML, OMG, WTF...), to szybko można się pogubić w *escapingu*.

```
const QString regexp = "\\\*(\'|\")\*\\";
```

Raw string literals

Jeśli musimy wpisać do kodu źródłowego tekst o sporej liczbie śmiesznych znaczków typu apostrofy, cudzysłowy, backslashe itp. (regexpy, interpretery, parsery, HTML, XML, OMG, WTF...), to szybko można się pogubić w *escapingu*.

```
const QString regexp = "\\\*(\'|\")\*\\";
```

Lepiej:

```
const QString regexp = R"(\*('|")*\)";
```

Raw string literals

Jeśli musimy wpisać do kodu źródłowego tekst o sporej liczbie śmiesznych znaczków typu apostrofy, cudzysłowy, backslashe itp. (regexpy, interpretery, parsery, HTML, XML, OMG, WTF...), to szybko można się pogubić w *escapingu*.

```
const QString regexp = "\\\*(\'|\")\*\\";
```

Lepiej:

```
const QString regexp = R"(\*('|")*\)";
```

(To wyrażenie powyżej najpewniej nie ma żadnego sensu. Nothing to see here, move along.)

- *Raw strings* umieszczamy w ciągu `R"(...)"`.

- *Raw strings* umieszczamy w ciągu `R"(...)"`.
- Możemy też w ten sposób: `R"separator(...)separator"`, gdzie `separator` to maksymalnie 16-znakowy ciąg niemalże dowolnych znaków, poza nawiasami okrągłymi, backslashem i znakami kontrolnymi.

- *Raw strings* umieszczamy w ciągu `R"(...)"`.
- Możemy też w ten sposób: `R"separator(...)separator"`, gdzie `separator` to maksymalnie 16-znakowy ciąg niemalże dowolnych znaków, poza nawiasami okrągłymi, backslashem i znakami kontrolnymi.
- Użycie separatora pozwala na zawarcie nawiasów okrągłych wewnątrz *raw string*.

- *Raw strings* umieszczamy w ciągu `R"(...)"`.
- Możemy też w ten sposób: `R"separator(...)separator"`, gdzie `separator` to maksymalnie 16-znakowy ciąg niemalże dowolnych znaków, poza nawiasami okrągłymi, backslashem i znakami kontrolnymi.
- Użycie separatora pozwala na zawarcie nawiasów okrągłych wewnątrz *raw string*.
- *Raw strings* można łączyć z UTF, np. `u8R"(tekst)"`.

Listy inicjujące

Weźmy taką klasę:

```
struct student {
    std::string nazwisko;
    float srednia_ocen;
};
```

Listy inicjujące

Weźmy taką klasę:

```
struct student {
    std::string nazwisko;
    float srednia_ocen;
};
```

Możemy inicjować obiekty tej klasy w ten sposób:

```
student a{"Kowalski", 5.0};
student b = {"Nowak", 3.5};
```

Listy inicjujące

Weźmy taką klasę:

```
struct student {
    std::string nazwisko;
    float srednia_ocen;
};
```

Możemy inicjować obiekty tej klasy w ten sposób:

```
student a{"Kowalski", 5.0};
student b = {"Nowak", 3.5};
```

Czyli w klamerkach i po przecinku podajemy wartości kolejnych pól obiektu.

Listy inicjujące działają „rekurencyjnie”:

```
std::pair <student, student> a {
    {"Kowalski", 5.0},
    {"Nowak", 3.5}
};
student b[] {
    {"Kowalski", 5.0},
    {"Nowak", 3.5}
};
std::vector <student> c {
    {"Kowalski", 5.0},
    {"Nowak", 3.5}
};
```

Listy inicjujące działają „rekurencyjnie”:

```
struct student {
    std::string nazwisko;
    float srednia_ocen;
};

struct indeks {
    int numer;
    student s;
    std::string uczelnia;
};

indeks i{123456, {"Kowalski", 5.0}, "MIMUW"};
```

Korzystając z powyższej klamerkowej składni, ujednolicono inicjowanie obiektów. Od teraz możemy wywoływać konstruktory w klamerkach, a nie tylko w nawiasach:

```
int x{6};
std::string s{"napis"};
std::pair <double, char> p{1.0, 'w'};

struct student {
    std::string nazwisko;
    float srednia_ocen;

    student(std::string n, float so)
        : nazwisko{n}, srednia_ocen{so} {}
};
```

Jeszcze jeden ciekawy przykład:

```
struct student {
    std::string nazwisko;
    float srednia_ocen;
};

student f()
{
    return {"Kowalski", 5.0};
}
```

Możemy tworzyć const obiekty skomplikowanych typów, np. std::map:

```
const std::map <std::string, float> tabela_studentow {
    {"Kowalski", 5.0},
    {"Nowak", 3.5}
};
```

Możemy tworzyć const obiekty skomplikowanych typów, np. std::map:

```
const std::map <std::string, float> tabela_studentow {
    {"Kowalski", 5.0},
    {"Nowak", 3.5}
};
```

Przydatne nie tylko w wolnostojących zmiennych, ale także w obiektach.

Czym pod maską są listy inicjujące?

Czym pod maską są listy inicjujące?

```
template <typename T> std::initializer_list <T>
```

Czym pod maską są listy inicjujące?

```
template <typename T> std::initializer_list <T>
```

- Możemy robić własne funkcje, które przyjmują w argumencie `initializer_list`.

Czym pod maską są listy inicjujące?

```
template <typename T> std::initializer_list <T>
```

- Możemy robić własne funkcje, które przyjmują w argumencie `initializer_list`.
- Takie listy da się jedynie tworzyć w kodzie, a nie budować „w locie”, np. dokładając po jednym elemencie.

Czym pod maską są listy inicjujące?

```
template <typename T> std::initializer_list <T>
```

- Możemy robić własne funkcje, które przyjmują w argumencie `initializer_list`.
- Takie listy da się jedynie tworzyć w kodzie, a nie budować „w locie”, np. dokładając po jednym elemencie.
- Da się je „zapamiętać” na zmienną — inaczej nie dałoby się ich obsługiwać jako argumentów funkcji, ale są oczywiście *read-only*.

Czym pod maską są listy inicjujące?

```
template <typename T> std::initializer_list <T>
```

- Możemy robić własne funkcje, które przyjmują w argumencie `initializer_list`.
- Takie listy da się jedynie tworzyć w kodzie, a nie budować „w locie”, np. dokładając po jednym elemencie.
- Da się je „zapamiętać” na zmienną — inaczej nie dałoby się ich obsługiwać jako argumentów funkcji, ale są oczywiście *read-only*.
- Mają metody `begin()` i `end()`, zwracające iteratory (jak w kontenerach z STL), oraz `size()`.

Przykład z życia: chcę mieć mapę działającą w dwie strony (`BiMap`).

Przykład z życia: chcę mieć mapę działającą w dwie strony (BiMap).

```
template <typename T, typename U> class BiMap {
public:
    BiMap(std::initializer_list <std::pair <T, U> >
        initList);

    U operator[](const T &leftKey) const;
    T operator[](const U &rightKey) const;
    void insert(const T &leftKey, const U &rightKey);

private:
    std::map <T, U> left;
    std::map <U, T> right;
};
```

```
template <typename T, typename U>
BiHash <T, U>::BiHash(
    std::initializer_list <std::pair <T, U> >
    initList)
{
    for (const std::pair <T, U> &p : initList)
        insert(p.first, p.second);
}

template <typename T, typename U>
void BiHash <T, U>::insert(const T &leftKey,
    const U &rightKey)
{
    left.insert(leftKey, rightKey);
    right.insert(rightKey, leftKey);
}
```

Mogę teraz bez problemu definiować stałe obiekty typu BiMap:

```
const BiMap <float, std::vector <std::string> > oceny {
    {5.0, {"Einstein", "Newton", "Feynman"}},
    {3.0, {"Kowalski", "Nowak", "Malinowski"}}
};
```

Nowa postać pętli for

Jeśli chcemy dokonać iteracji po elementach jakiejś struktury (np. tablicy), to możemy od teraz pisać pętle w taki sposób:

```
int tab[5]{1, 2, 3, 4, 5};
int sum = 0;
for (int x : tab)
    sum += x;

vector <pair <int, int> > v{{1, 2}, {3, 4}};
for (pair <int, int> p : v)
    sum += p.first * p.second;
```

Nowa postać pętli for

Jeśli chcemy dokonać iteracji po elementach jakiejś struktury (np. tablicy), to możemy od teraz pisać pętle w taki sposób:

```
int tab[5]{1, 2, 3, 4, 5};
int sum = 0;
for (int x : tab)
    sum += x;

vector <pair <int, int> > v{{1, 2}, {3, 4}};
for (pair <int, int> p : v)
    sum += p.first * p.second;
```

- Iterator może być typu referencyjnego. Wtedy modyfikując go, zmieniamy zawartość struktury.

Nowa postać pętli for

Jeśli chcemy dokonać iteracji po elementach jakiejś struktury (np. tablicy), to możemy od teraz pisać pętle w taki sposób:

```
int tab[5]{1, 2, 3, 4, 5};
int sum = 0;
for (int x : tab)
    sum += x;

vector <pair <int, int> > v{{1, 2}, {3, 4}};
for (pair <int, int> p : v)
    sum += p.first * p.second;
```

- Iterator może być typu referencyjnego. Wtedy modyfikując go, zmieniamy zawartość struktury.
- Oprócz tablic w stylu C, możemy tego użyć na dowolnej strukturze definiującej odpowiednio metody `begin()`, `end()` oraz iteratory.

Nowa postać pętli for

Jeśli chcemy dokonać iteracji po elementach jakiejś struktury (np. tablicy), to możemy od teraz pisać pętle w taki sposób:

```
int tab[5]{1, 2, 3, 4, 5};
int sum = 0;
for (int x : tab)
    sum += x;

vector <pair <int, int> > v{{1, 2}, {3, 4}};
for (pair <int, int> p : v)
    sum += p.first * p.second;
```

- Iterator może być typu referencyjnego. Wtedy modyfikując go, zmieniamy zawartość struktury.
- Oprócz tablic w stylu C, możemy tego użyć na dowolnej strukturze definiującej odpowiednio metody `begin()`, `end()` oraz iteratory.
- To znaczy, że możemy użyć np. `initializer_list`.

Z życia wzięte: mamy (np. w konstruktorze) kilka widgetów Qt tego samego typu. Wszystkie te widgety inicjujemy w ten sam sposób, poprzez wywołanie na każdym kilku metod, ustawiające różne własności i atrybuty.

Z życia wzięte: mamy (np. w konstruktorze) kilka widgetów Qt tego samego typu. Wszystkie te widgety inicjujemy w ten sam sposób, poprzez wywołanie na każdym kilku metod, ustawiające różne własności i atrybuty.

```
spinBox_1 = new QSpinBox;
spinBox_1->setAlignment(Qt::AlignRight);
spinBox_1->setWidth(80);

//analogicznie dla spinBox_2, spinBox_3 itd.
```

Z życia wzięte: mamy (np. w konstruktorze) kilka widgetów Qt tego samego typu. Wszystkie te widgety inicjujemy w ten sam sposób, poprzez wywołanie na każdym kilku metod, ustawiające różne własności i atrybuty.

```
spinBox_1 = new QSpinBox;
spinBox_1->setAlignment(Qt::AlignRight);
spinBox_1->setWidth(80);

//analogicznie dla spinBox_2, spinBox_3 itd.
```

- Kopypasta.

Z życia wzięte: mamy (np. w konstruktorze) kilka widgetów Qt tego samego typu. Wszystkie te widgety inicjujemy w ten sam sposób, poprzez wywołanie na każdym kilku metod, ustawiające różne własności i atrybuty.

```
spinBox_1 = new QSpinBox;
spinBox_1->setAlignment(Qt::AlignRight);
spinBox_1->setWidth(80);

//analogicznie dla spinBox_2, spinBox_3 itd.
```

- Kopypasta.
- Jak będziemy chcieli np. zmienić typ zmiennych, bo robimy własnego widgeta dziedziczącego po `QSpinBox`, to musimy poprawić *n* miejsc.

Z życia wzięte: mamy (np. w konstruktorze) kilka widgetów Qt tego samego typu. Wszystkie te widgety inicjujemy w ten sam sposób, poprzez wywołanie na każdym kilku metod, ustawiające różne własności i atrybuty.

```
spinBox_1 = new QSpinBox;
spinBox_1->setAlignment(Qt::AlignRight);
spinBox_1->setWidth(80);

//analogicznie dla spinBox_2, spinBox_3 itd.
```

- Kopypasta.
- Jak będziemy chcieli np. zmienić typ zmiennych, bo robimy własnego widgeta dziedziczącego po `QSpinBox`, to musimy poprawić *n* miejsc.
- Tak samo jeśli sposób inicjowania będzie trzeba zmienić.

Lepiej:

```
for (QSpinBox **spinBox : {&spinBox_1, ... }) {
    *spinBox = new QSpinBox;
    (*spinBox)->setAlignment(Qt::AlignRight);
    (*spinBox)->setWidth(80);
}
```

Typ tablicowy array

- Nowy standard wprowadza bardziej wysokopoziomowy typ tablicowy `std::array`.

Typ tablicowy array

- Nowy standard wprowadza bardziej wysokopoziomowy typ tablicowy `std::array`.
- Jest to typ szablonowy:

```
template <typename T, std::size_t N>
std::array <T, N>
```

Typ tablicowy array

- Nowy standard wprowadza bardziej wysokopoziomowy typ tablicowy `std::array`.
- Jest to typ szablonowy:

```
template <typename T, std::size_t N>
std::array <T, N>
```

- `T` oznacza typ przechowywany w tablicy, `N` jej rozmiar.

Typ tablicowy array

- Nowy standard wprowadza bardziej wysokopoziomowy typ tablicowy `std::array`.
- Jest to typ szablonowy:

```
template <typename T, std::size_t N>
std::array <T, N>
```

- `T` oznacza typ przechowywany w tablicy, `N` jej rozmiar.
- **Tablica jest stałego rozmiaru.** Ale przynajmniej ma np. `begin()`, `end()` i `size()`.

Typ tablicowy array

- Nowy standard wprowadza bardziej wysokopoziomowy typ tablicowy `std::array`.
- Jest to typ szablonowy:

```
template <typename T, std::size_t N>
std::array <T, N>
```

- `T` oznacza typ przechowywany w tablicy, `N` jej rozmiar.
- **Tablica jest stałego rozmiaru.** Ale przynajmniej ma np. `begin()`, `end()` i `size()`.
- Oprócz standardowego operatora tablicowego ma też metodę `at()`, która robi *bounds-checking*.

Silne typy wyliczeniowe

- W starszym C++ typ wyliczeniowy (`enum`) był w zasadzie nazwanym zbiorem stałych, o niewielkiej nadbudowie typologicznej.

```
enum Fruit {
    Apple,
    Orange
};

Fruit f = Apple;
```

Silne typy wyliczeniowe

- W starszym C++ typ wyliczeniowy (`enum`) był w zasadzie nazwanym zbiorem stałych, o niewielkiej nadbudowie typologicznej.

```
enum Fruit {
    Apple,
    Orange
};

Fruit f = Apple;
```

- Niewiele było wiadomo o „fizycznym” typie danych użytym do implementacji typu wyliczeniowego. Był *implementation defined*, w praktyce zwykle znaczyło to `int`.

- Nazwa typu wyliczeniowego nie była zasięgiem (*scope*), co powodowało konflikty nazw, gdy w tym samym *scope* chcieliśmy zdefiniować więcej niż jeden typ wyliczeniowy, zawierający wewnątrz identycznie nazwaną stałą.

```
enum Fruit {Apple, Orange};
enum Cake {Cheese, Apple};
```

- Nazwa typu wyliczeniowego nie była zasięgiem (*scope*), co powodowało konflikty nazw, gdy w tym samym *scope* chcieliśmy zdefiniować więcej niż jeden typ wyliczeniowy, zawierający wewnątrz identycznie nazwaną stałą.

```
enum Fruit {Apple, Orange};
enum Cake {Cheese, Apple};
```

- Żeby dodać *scoping* robiło się różne hacki:

```
struct Fruit {
    enum FruitEnum {
        Apple,
        Orange
    };
};

Fruit::FruitEnum f = Fruit::Apple;
```

W C++11 pojawiły się dwa ulepszenia typów wyliczeniowych:

W C++11 pojawiły się dwa ulepszenia typów wyliczeniowych:

1. Można specyfikować działający pod spodem typ całkowitoliczbowy.

W C++11 pojawiły się dwa ulepszenia typów wyliczeniowych:

1. Można specyfikować działający pod spodem typ całkowitoliczbowy.
2. Można dodać przedrostek `class` (albo `struct`, bez różnicy) przed nazwą typu wyliczeniowego. Powoduje to jego wzmocnienie.

Typ wewnętrzny specyfikujemy w taki sposób:

```
enum Fruit : int {
    Apple,
    Orange
};

enum Cake : quint8 {
    Cheese,
    Apple
};
```

Typ wewnętrzny specyfikujemy w taki sposób:

```
enum Fruit : int {
    Apple,
    Orange
};

enum Cake : quint8 {
    Cheese,
    Apple
};
```

- W tym przykładzie nadal mamy błąd kompilacji, bo powtarza się identyfikator `Apple` w tym samym zasięgu widoczności.
- Jeśli nie wyspecyfikujemy typu wewnętrznego, domyślnie jest to `int`.

Wzmacniamy typy wyliczeniowe:

```
enum class Fruit : int {
    Apple,
    Orange
};

enum class Cake : quint8 {
    Cheese,
    Apple
};

Fruit f = Fruit::Apple;
Cake c = Cake::Apple;
```

Wzmacniamy typy wyliczeniowe:

```
enum class Fruit : int {
    Apple,
    Orange
};

enum class Cake : quint8 {
    Cheese,
    Apple
};

Fruit f = Fruit::Apple;
Cake c = Cake::Apple;
```

- Przedrostek z nazwą typu wyliczeniowego jest obligatoryjny przy odwoływaniu się do samych wartości.

Wzmacniamy typy wyliczeniowe:

```
enum class Fruit : int {
    Apple,
    Orange
};

enum class Cake : quint8 {
    Cheese,
    Apple
};

Fruit f = Fruit::Apple;
Cake c = Cake::Apple;
```

- Przedrostek z nazwą typu wyliczeniowego jest obligatoryjny przy odwoływaniu się do samych wartości.
- **Moim zdaniem jeden z ważniejszych dodatków do języka.** Przestałem korzystać ze „słabych” typów wyliczeniowych.

Wyrażenia stałe (constexpr)

- Wyrażenia stałe to takie, które zawsze generują ten sam wynik i można ten wynik otrzymać już na etapie kompilacji.

Wyrażenia stałe (constexpr)

- Wyrażenia stałe to takie, które zawsze generują ten sam wynik i można ten wynik otrzymać już na etapie kompilacji.
- Niestety w sensie poprzedniego standardu znaczy to tyle, że wyrażenia są stałe tylko wtedy, gdy są całkowitoliczbowe i nie odpalają po drodze żadnej funkcji. Nawet, jeśli ta funkcja zawsze zwraca tę samą wartość.

Wyrażenia stałe (constexpr)

- Wyrażenia stałe to takie, które zawsze generują ten sam wynik i można ten wynik otrzymać już na etapie kompilacji.
- Niestety w sensie poprzedniego standardu znaczy to tyle, że wyrażenia są stałe tylko wtedy, gdy są całkowitoliczbowe i nie odpalają po drodze żadnej funkcji. Nawet, jeśli ta funkcja zawsze zwraca tę samą wartość.

```
int f() {return 3;}
int a[3];          //OK
int b[3 + 3];      //OK
int c[3 + f()]; //compile error
```

Wyrażenia stałe (constexpr)

- Wyrażenia stałe to takie, które zawsze generują ten sam wynik i można ten wynik otrzymać już na etapie kompilacji.
- Niestety w sensie poprzedniego standardu znaczy to tyle, że wyrażenia są stałe tylko wtedy, gdy są całkowitoliczbowe i nie odpalają po drodze żadnej funkcji. Nawet, jeśli ta funkcja zawsze zwraca tę samą wartość.

```
int f() {return 3;}
int a[3];          //OK
int b[3 + 3];      //OK
int c[3 + f()]; //compile error
```

- Jeśli prefiksujemy funkcję `f()` słowem `constexpr`, to zacznie być OK.

Ograniczenia i możliwości:

- Funkcje `constexpr` muszą w wyniku zawierać coś innego niż `void`.

Ograniczenia i możliwości:

- Funkcje `constexpr` muszą w wyniku zawierać coś innego niż `void`.
- W ciele funkcji nie można deklarować zmiennych i definiować nowych typów.

Ograniczenia i możliwości:

- Funkcje `constexpr` muszą w wyniku zawierać coś innego niż `void`.
- W ciele funkcji nie można deklarować zmiennych i definiować nowych typów.
- W zasadzie najlepiej, jakby zawierała po prostu jedno `return` z obliczeniem wyniku na podstawie ewentualnych argumentów.

Ograniczenia i możliwości:

- Funkcje `constexpr` muszą w wyniku zawierać coś innego niż `void`.
- W ciele funkcji nie można deklarować zmiennych i definiować nowych typów.
- W zasadzie najlepiej, jakby zawierała po prostu jedno `return` z obliczeniem wyniku na podstawie ewentualnych argumentów.

```
constexpr int g(int a, int b) {return a * b;}
```

Ograniczenia i możliwości:

- Funkcje `constexpr` muszą w wyniku zawierać coś innego niż `void`.
- W ciele funkcji nie można deklarować zmiennych i definiować nowych typów.
- W zasadzie najlepiej, jakby zawierała po prostu jedno `return` z obliczeniem wyniku na podstawie ewentualnych argumentów.

```
constexpr int g(int a, int b) {return a * b;}
```

- Jeśli wywołamy funkcję `constexpr` z argumentami nieznanymi w czasie kompilacji, to wszystko zadziała OK, ale funkcja już nie jest `constexpr`. Tzn. argumenty funkcji muszą być `constexpr`, bo inaczej tracimy tę własność.

Ograniczenia i możliwości:

- Funkcje `constexpr` muszą w wyniku zawierać coś innego niż `void`.
- W ciele funkcji nie można deklarować zmiennych i definiować nowych typów.
- W zasadzie najlepiej, jakby zawierała po prostu jedno `return` z obliczeniem wyniku na podstawie ewentualnych argumentów.

```
constexpr int g(int a, int b) {return a * b;}
```

- Jeśli wywołamy funkcję `constexpr` z argumentami nieznanymi w czasie kompilacji, to wszystko zadziała OK, ale funkcja już nie jest `constexpr`. Tzn. argumenty funkcji muszą być `constexpr`, bo inaczej tracimy tę własność.
- Możemy też deklarować konstruktory własnych typów jako `constexpr`. Możemy ich wtedy używać w wyrażeniach tego typu bez straty stałości.

Ciekawy skutek „uboczny”: dawniej dało się definiować stałe statyczne w klasach już w definicji klasy, ale tylko pod warunkiem, że były to stałe typów całkowitoliczbowych.

```
class A {
    static const int I = 5;
    static const double D = 5.0; //compile error
};
const double A::D = 5.0; //OK
```

Ciekawy skutek „uboczny”: dawniej dało się definiować stałe statyczne w klasach już w definicji klasy, ale tylko pod warunkiem, że były to stałe typów całkowitoliczbowych.

```
class A {
    static const int I = 5;
    static const double D = 5.0; //compile error
};
const double A::D = 5.0; //OK
```

```
class A {
    static const int I = 5;
    static constexpr double D = 5.0; //OK
};
```

User-defined literals

W C++ istnieje kilka predefiniowanych literałów, zarówno przedrostkowych jak i przyrostkowych:

- Przedrostkowe: `0x` na liczby szesnastkowe, `0` na liczby ósemkowe, `u8` na napisy UTF-8...
- Przyrostkowe: `f` jako `float`, `L` jako `long`, `LL` jako `long long`, `i` jako część urojona w liczbach zespolonych...

W nowym standardzie można definiować własne literały przyrostkowe za pomocą specjalnego operatora.

W nowym standardzie można definiować własne literały przyrostkowe za pomocą specjalnego operatora.

```
typedef qint32 LengthUnit;
static const int LengthUnitPerMm = 10;

inline constexpr LengthUnit
    operator"" _mm(long double mm) {
        return mm * LengthUnitPerMm;
}

inline constexpr LengthUnit
    operator"" _cm(long double cm) {
        return cm * 10_mm;
}

inline constexpr LengthUnit
    operator"" _in(long double in) {
        return in * 2.54_cm;
}
```

Składnia:

```
wynik operator"" _sufiks(argument);
```

Składnia:

```
wynik operator"" _sufiks(argument);
```

- Sufiks powinien zaczynać się od podkreślnika. Literały bez podkreślnika są zarezerwowane na potrzeby przyszłych wersji języka.

Składnia:

```
wynik operator"" _sufiks(argument);
```

- Sufiks powinien zaczynać się od podkreślnika. Literały bez podkreślnika są zarezerwowane na potrzeby przyszłych wersji języka.
- Dopuszczalne typy argumentu to: `unsigned long long`, `long double`, `char`, `const char *` (standardowy napis null-terminated). Ewentualnie dwa argumenty, kolejno `const char *`, `size_t` – napis i długość.

Składnia:

```
wynik operator"" _sufiks(argument);
```

- Sufiks powinien zaczynać się od podkreślnika. Literały bez podkreślnika są zarezerwowane na potrzeby przyszłych wersji języka.
- Dopuszczalne typy argumentu to: `unsigned long long`, `long double`, `char`, `const char *` (standardowy napis null-terminated). Ewentualnie dwa argumenty, kolejno `const char *`, `size_t` – napis i długość.

Użycie:

```
const LengthUnit Metr = 100_cm;
```