

ZPR

Witold Wysota

Stałość



const-correctness

- prawidłowe używanie `const` do chronienia stałych obiektów przed zmianami
- zadeklarowanie stałości parametru jest przejawem bezpieczeństwa typów
- zapobiega przypadkowej modyfikacji danych
- koncepcyjnie można traktować **`const X`** oraz **`X`** jako dwa różne typy
- użycie **`const`** w wielu przypadkach może doprowadzić do uzyskania wydajniejszego kodu
- niektóre iteratory posiadają swoje wersje `const_`
(`std::vector<T>::const_iterator`) będące iteratorami na stałą wartość
 - uzyskiwane np. przez `cbegin()`, `cend()`

Funkcje biorące niezmiennie wartości

- stworzenie funkcji w taki sposób, żeby zagwarantować wołającym, że funkcja nie zmieni przekazanych danych
 - `void inspect1(const std::string&)`
 - przekazanie przez referencję na stały obiekt
 - `void inspect2(const std::string*)`
 - przekazanie przez wskaźnik na stały obiekt
 - `void inspect3(std::string)`
 - przekazanie przez wartość (kopię)
- sprawdzenie poprawności na etapie kompilacji (brak narzutu na pamięć ani szybkość)

Funkcje biorące zmienne wartości

- stworzenie funkcji w taki sposób, żeby poinformować wołających, że funkcja może (nie musi) zmienić obiekt
 - `void mutate1(std::string&)`
 - przekazanie przez referencję na mutowalny obiekt
 - `void mutate2(std::string *)`
 - przekazanie przez wskaźnik na mutowalny obiekt

Metody

- oznaczenie metody jako const oznacza gwarancję, że metoda nie zmienia stanu this

```
struct K {  
    void inspect() const;  
    void mutate();  
};  
  
void foo(K& changeable, const K& unchangeable) {  
    changeable.inspect();    // OK  
    changeable.mutate();    // OK  
  
    unchangeable.inspect(); // OK  
    unchangeable.mutate(); // ERROR  
}
```

Metody - przeciążanie

- metoda może mieć przeciążenia z const i bez const

```
struct K {  
    Foo& operator[](size_t index);  
    const Foo& operator[](size_t index) const;  
};
```

Stan logiczny

Interfejs

Użytkownicy klasy widzą tylko publiczny interfejs obiektu

Niezależność

Zmiana stanu interfejsu może nie wpływać na zmianę stanu fizycznego obiektu



Stan interfejsu

Ten interfejs może nieść ze sobą jakiś stan

Niezależność

Zmiana stanu fizycznego obiektu może nie wpływać na zmianę stanu interfejsu

Stan fizyczny

Ułożenie w pamięci

Faktyczne składowe obiekty, które znajdują się w konkretnym miejscu w pamięci

Niezależność

Niektóre składowe mogą nie mieć odzwierciedlenia w publicznym interfejsie obiektu



Zmiany stanu

- publiczny interfejs klasy odzwierciedla stan logiczny
 - na “const” patrzymy z punktu widzenia użytkownika klasy
- są sytuacje, kiedy stan fizyczny się zmienia mimo braku zmian w stanie logicznym
 - np. trudno wyliczalną wartość można zapisać na boku

```
class Container {  
public:  
    // ...  
    double averageValue() const;  
};
```

- mutable - oznacza konkretne pole obiektu jako modyfikowalne z const
- const_cast - używać tylko jeśli jest pewność, że obiekt jest fizycznie modyfikowalny

Zasady pisania kodu

- o const-correctness trzeba dbać od początku
 - dopisanie “const” na późniejszym etapie może spowodować lawinę zmian
- koncepcyjnie (a może i realnie) pisząc nowy kod oznaczajmy każdą zmienną i metodę jako const
 - jeżeli “nie działa”, to ewentualnie usuńmy const
 - nigdy odwrotnie
- są języki programowania (np. Rust), gdzie każda zmienna jest domyślnie const

Jak czytać const

- const west vs east const
 - `const std::string`
 - `std::string const`
- czytamy zawsze od prawej do lewej
 - `const char * a`
 - *a to wskaźnik na char, który jest const*
 - `char const * a`
 - *a to wskaźnik na const char*
 - `char * const a`
 - *a to const wskaźnik na char*
 - `char const * const a`
 - *a to const wskaźnik na const char*
 - `const char * const a`
 - *a to const wskaźnik na char, który jest const*
- analogicznie dla referencji (przy czym referencja zawsze jest stała)

Sprawdzian

- `const X** foo`
- `const X* const* bar`
- `const X* const* const baz`
- `const X** const qux`
- `const X const** const quux`

Sprawdzian

- `X const** foo`
- `X const* const* bar`
- `X const* const* const baz`
- `X const** const qux`
- `X const const** const quux`

Sprawdzian

- `X const** foo`
 - *wskaźnik na wskaźnik na const X*
- `X const* const* bar`
 - *wskaźnik na const wskaźnik na const X*
- `X const* const* const baz`
 - *const wskaźnik na const wskaźnik na const X*
- `X const** const qux`
 - *const wskaźnik na wskaźnik na const X*
- ~~`X const const** const quux`~~
 - *const wskaźnik na wskaźnik na const const X (warning - podwojone const)*

Język zorientowany na wartości



Zadanie

Stworzyć funkcję fabryki, która zwraca instancję studenta na podstawie identyfikatora.

Stworzyć funkcję fabryki, która zwraca instancję nauczyciela na podstawie identyfikatora.

```
struct Student {  
    std::string name;  
    double average;  
};
```

```
struct Teacher {  
    std::string name;  
    Rooms roomKeys;  
    Courses courses;  
};
```

Fabryka studentów

```
Student makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        // ... ???  
    }  
    return Student(...);  
}
```

Fabryka studentów

```
std::unique_ptr<Student> makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        return nullptr;  
    }  
    return std::make_unique<Student>(...);  
}
```

Zadanie

Stworzyć kolekcję (np. vector) studentów i nauczycieli.

Chcemy mieć możliwość sprawdzać, jak się nazywają, jaką średnią mają studenci i do których pokojów mogą się dostać nauczyciele i jakie przedmioty prowadzą.

```
struct Person {  
    virtual ~Person() = default;  
    virtual std::string name() const = 0;  
    virtual double average() const = 0;  
    virtual Rooms roomKeys() const = 0;  
    virtual Courses courses() const = 0;  
};
```

```
struct Student : Person {  
    std::string name() const override { return m_name; }  
    double average() const override { return m_average; }  
    Rooms roomKeys() const override { return {}; }  
    Courses courses() const override { return {}; }  
};
```

```
struct Teacher : Person {  
    std::string name() const override { return m_name; }  
    double average() const override { return {}; }  
    Rooms roomKeys() const override { return ...; }  
    Courses courses() const override { return ...; }  
};
```

```
vector<unique_ptr<Person>> studentsAndTeachers;

for (const auto id: ...) {
    if (auto student = makeStudent(id)) {
        studentsAndTeachers.push_back(std::move(student));
    } else if (auto teacher = makeTeacher(id)) {
        studentsAndTeachers.push_back(std::move(teacher));
    }
}
```

Wypisanie imion i nazwisk, wypisanie średnich studentów

```
for (const auto &person: students_and_teachers) {  
    print("{}\n", person.name());  
}
```

```
for (const auto &person: students_and_teachers) {  
    if (!person.average()) {  
        continue;  
    }  
    print("{}\n", person.average());  
}
```

Wypisanie pokojów (analogicznie dla przedmiotów)

```
for (const auto &person: students_and_teachers) {  
    if (person->roomKeys().empty()) {  
        continue;  
    }  
  
    print("{}\n", person->roomKeys());  
}
```



Trochę problem, bo może być nauczyciel bez dostępu do żadnego pokoju...

W sumie to student bez średniej też...

No trudno...

Zadanie

Stworzyć kolekcję nauczycieli i pracowników technicznych.

```
struct TechnicalStaff {  
    std::string name;  
    Rooms roomKeys;  
};
```

```
struct Staff : Person {  
    Rooms roomKeys() const override { return ...; }  
};  
  
struct Teacher : Staff {  
    Courses courses() const override { return ...; }  
};  
  
struct TechnicalStaff : Staff {  
    Courses courses() const override { return {};}  
};
```

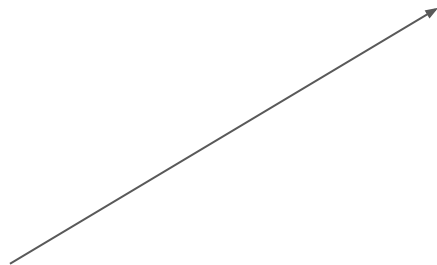
```
vector<unique_ptr<Staff>> teachers_and_technical_staff;  
for (const auto& id: ...) {  
    // ...  
}
```

Zadanie

Stworzyć kolekcję studentów i pracowników technicznych.

```
vector<unique_ptr<Person>> students_and_technical_staff;
```

Ale tu się też da dodać nauczyciela...



```
vector<...> students_and_technical_staff; // don't add teachers
```

```
vector<...> students_and_technical_staff; // don't add teachers
```

SERIO?

Gdzie popełniliśmy błąd?

Zadanie

Stworzyć kolekcję (np. vector) **studentów** i **nauczycieli**.

Chcemy mieć możliwość sprawdzać, jak się nazywają, jaką średnią mają studenci i do których pokojów mogą się dostać nauczyciele i jakie przedmioty prowadzą.

Person? Staff?

```
using StudentsAndTeacher = union { Student* student; Teacher* teacher };  
  
vector<StudentsAndTeacher> studentsAndTeachers;
```

```
using StudentsAndTeacher = union { Student* student; Teacher* teacher };  
  
vector<StudentsAndTeacher> studentsAndTeachers;
```

Dwa problemy:

1. nie da się tego sensownie zrobić na `std::unique_ptr`
2. nie wiemy co jest zapisane w danym elemencie wektora

Gdzie (tak naprawdę) popełniliśmy błąd?

Założyliśmy, że będziemy używać wskaźników

- wbrew obiegowej opinii C++ (tak samo C) nie jest językiem zorientowanym na wskaźniki
- [książka Bjarne Stroustroupa](#) opisująca C++17 ma około 1230 stron (1070 treści, 27 rozdziałów)
 - wskaźniki pojawiają się na stronie 588 w rozdziale 17
 - m.in. po rozdziale o GUI... którego nawet w ogóle nie ma w języku C++
 - przed szczegółowym opisem szablonów, biblioteki standardowej, testowania i standardu C

*Do as an **int** would do*

- C++ jest zorientowany na wartości
- wskaźnik jest adresem wartości
- referencja jest wskazaniem na wartość

Typy algebraiczne



Typy algebraiczne

- rodzaj typów złożonych
 - typy iloczynowe (opierające się na mnożeniu)
 - typy sumowe (opierające się na dodawaniu)
- operandami są typy
- wartościami jest liczba różnych stanów, które typ reprezentuje

Typ iloczynowy - struktura

- Ile różnych wartości może przechować?
 - `uint8_t x;`
 - `bool y;`

Typ iloczynowy - struktura

- Ile różnych wartości może przechować?

- `uint8_t x;`

- `bool y;`

- `struct {`

- `uint8_t x;`

- `bool y;`

- `} rekord; // 256 * 2`

- `std::tuple<bool, bool, bool> soMuchPain; // 2 * 2 * 2`

Powrót do fabryki - pozbycie się wskaźników

```
unique_ptr<Student> makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        return nullptr;  
    }  
    return make_unique<Student>(...);  
}
```

```
unique_ptr<Student> student = makeStudent(...);  
if (!student) {  
    // ...  
}
```

Powrót do fabryki - analiza problemu

- dwa problemy

1. utworzenie obiektu może się nie udać (np. nie ma studenta o danym id)
2. wartości różnych typów chcemy zmieścić w tym samym pudełku

```
unique_ptr<Student> makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        return nullptr;  
    }  
    return make_unique<Student>(...);  
}
```

```
unique_ptr<Student> makeStudent(uint32_t);  
unique_ptr<Teacher> makeTeacher(uint32_t);
```

Powrót do fabryki - pozbycie się wskaźników

```
Student makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        throw ...;  
    }  
    return Student(...);  
}
```

czy brak studenta to jest
sytuacja wyjątkowa?



```
Student student = makeStudent(...);  
// ...
```

Co gdybyśmy zwracali tylko imię i nazwisko studenta?

```
std::string makeStudentName(uint32_t id) {  
    if (!Student::exists(id)) {  
        return "";  
    }  
    return ...;  
}
```

```
std::string studentName = makeStudentName(...);  
if (studentName.empty()) {  
    // ...  
}
```



można sprawdzić, czy jest imię
i nazwisko

Powrót do fabryki

```
Student makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        return Student(); // default means invalid  
    }  
    return Student(...);  
}
```

```
Student student = makeStudent(...);  
if (!student.isValid()) {  
    // ...  
}
```

Powrót do fabryki

```
std::pair<Student, bool> makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        return {Student(), false};  
    }  
    return {Student(...), true};  
}  
  
auto [student, valid] = makeStudent(...);  
if (!valid) {  
    // ...  
}
```

std::optional

```
std::optional<Student> makeStudent(uint32_t id) {  
    if (!Student::exists(id)) {  
        return std::nullopt;  
    }  
    return Student(...);  
}
```

```
std::optional<Student> student = makeStudent(...);  
if (!student) {  
    // ...  
}
```

std::optional

- podobne do wartości + bool (typ algebraiczny z sumowaniem!)
- jeżeli nie przypiszemy wartości, to nie zostanie ona nigdy stworzona
- nie potrzebujemy żadnej specjalnej wartości w typie do oznaczenia braku
- specjalna stała „nullopt” odpowiada za ustawienie „braku wartości”
- nie służy do sygnalizacji błędów (od tego jest std::expected)
- optional
 - jest rzutowalny na bool (sprawdzenie, czy wartość jest ustawiona)
 - ma metodę has_value()
 - ma metodę value() oraz operator *
 - próba dostania się do wartości, gdy jej nie ma, kończy się rzuceniem wyjątku

Wektor studentów i nauczycieli

- pomysł z unią nie był zły
- unii nie da się w ten sposób zastosować sensownie w C++
 - nie wiemy, która składowa unii jest prawidłowa
 - unia nie zwoła konstruktora ani destruktora
- pierwszy problem dałoby się rozwiązać, dodając informację, która składowa jest obecnie wykorzystywana

std::variant

- bezpieczna unia różnych typów
- w każdym momencie przechowuje wartość jednego ze zdefiniowanych typów
 - może się zdarzyć na skutek błędu, że nie przechowuje żadnej
- wartość przechowywana jest w ramach pamięci zaalokowanej na wariant (nie ma żadnego zewnętrznego odwołania do sterty)

```
using StudentOrTeacher = std::variant<Student, Teacher>;
std::vector<StudentOrTeacher> studentsAndTeachers;
for (const auto id: ...) {

    if (auto student = makeStudent(id)) {
        studentsAndTeachers.push_back(*student);
    } else if (auto teacher = makeTeacher(id)) {
        studentsAndTeachers.push_back(*teacher);
    }

}
```

```
using StudentOrTeacher = std::variant<Student, Teacher>;  
using Staff             = std::variant<Teacher, TechnicalStaff>;  
using Person            = std::variant<Student, Teacher, TechnicalStaff>;
```

std::monostate

- typ, który można dodać jako pierwszy element variant, żeby oznaczyć „brak wartości”
- domyślnie skonstruowany wariant posiada domyślnie skonstruowaną wartość pierwszego ze swoich podtypów

std::holds_alternative

- pozwala na sprawdzenie, czy w wariancie przechowywana jest wartość spodziewanego typu

```
if (std::holds_alternative<Student>(element)) {  
    // ...  
}
```

std::get

- Pozwala pobrać wartość określonego typu

```
Teacher person = std::get<Teacher>(element);
```

```
Teacher person = std::get<0>(element);
```

- std::variant pozwala, żeby ten sam typ znajdował się więcej niż raz, get<int> pozwala określić, do którego chcemy się odwołać
- std::get działa też na std::pair czy std::tuple, a nawet na naszych własnych typach
- rzuca wyjątek, jeśli w wariancie nie znajduje się oczekiwana wartość

std::get_if

```
if (Student* student = std::get_if<Student>(&element)) {  
    // ...  
}
```

- pozwala na warunkowy dostęp do danych

std::visit

- pozwala na zaimplementowanie wzorca projektowego Visitor na wariancie
- podajemy funkcję i wariant (lub kilka z nich)
- sprawdzi, jakiego typu wartość jest w wariancie i wywoła funkcję dla tego typu

std::visit

```
auto printName = [](const auto &person) {  
    std::cout << person.name << std::endl;  
}
```

```
using TeacherOrStudent = std::variant<Teacher, Student>;
```

```
std::vector<TeacherOrStudent> people;  
for (const auto &person: people) {  
    std::visit(printName, person);  
}
```

Szablony



Problem

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

```
int min(int a, int b) { return a < b ? a : b; }
```

Sukces! Czy nie?

```
auto a = min(4.5, 5.7);  
std::cout << a << endl; // wynik to '4'
```

min() rzutuje argumenty na int! co więcej, zwraca int!

Problem

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

- Należy zdefiniować rodzinę funkcji (przeciążenia)

- `int min(int a, int b)` { `return a < b ? a : b;` }
- `float min(float a, float b)` { `return a < b ? a : b;` }
- `double min(double a, double b)` { `return a < b ? a : b;` }
- `uint64_t min(uint64_t a, uint64_t b)` { `return a < b ? a : b;` }

- Kompilator dobiera odpowiednie przeciążenie na podstawie parametrów.
- Jeżeli przeciążenie nie istnieje, zwróci błąd
 - np. podaliśmy dwa niekompatybilne typy do `min()`

Szablony funkcji

- wzorzec klasy, funkcji lub zmiennej, który może zostać wykorzystany przez kompilator do utworzenia prawdziwej klasy, funkcji lub zmiennej
- parametryzowany typami, wartościami lub szablonami

```
template<typename T>
```

```
const T& min(const T& a, const T &b) { return a < b ? a : b; }
```

Szablony funkcji

- szablon funkcji to nie funkcja tylko ogólny opis algorytmu
- narzędzie do generowania funkcji
- deklaracja szablonu bierze udział w rozwiązywaniu przeciążeń funkcji
- jeżeli kompilator wybierze szablon, będzie potrzebował jego ciała, żeby wyprodukować funkcję
- lista parametrów szablonu funkcji
 - lista parametrów szablonu
 - lista parametrów funkcji
- T jest zastępnikiem jakiegoś typu, który zostanie określony przez kompilator podczas kompilacji

```
template <typename T>  
const T& min(const T &a, const T &b) {  
    // ...  
}
```

lista parametrów szablonu

lista parametrów funkcji

Szablony funkcji

- szablon może mieć dowolną liczbę parametrów szablonych

```
template<typename U, typename V>  
V foo(U u, V v, int a) { ... }
```

- wywołanie też będzie zawierać te parametry

```
int x = foo<K1, K2>(i1, i2, 7);
```

- zamiast “typename” można używać “class”

- niezależnie czy to, co podajemy, jest klasą czy typem wbudowanym
- jest jedna sytuacja (przed C++17), gdzie tych słów nie można używać zamiennie

Szablony klas

- szablon klasy to nie typ danych tylko generalizacja typu
- przepis, którego kompilator może użyć, żeby tworzyć podobne (niezależne) klasy

```
template<typename T>
class rational {
public:
    rational();
    rational(T n);
    rational(T n, T d);
    // ...
private:
    T num, den;
};
```

Parametry szablonów

- argumentem szablonu są nie tylko typy, ale i wyrażenia

```
template<size_t N> class IntArray { ... };
```

- parametr nie będący typem może być:
 - liczbą całkowitą
 - wyliczeniem
 - wskaźnikiem
 - wskaźnikiem do składowej klasy
 - referencją
 - nullptr
 - liczbą rzeczywistą (C++20)
 - obiektem klasy literału (C++20)

Dedukcja typów argumentów szablonów

Szablony funkcji

```
i = min(a, b);
```

- kompilator sprawdzi typy argumentów wywołania funkcji (ale nie zwracany typ)
- na tej podstawie “uzupełni” brakujące parametry szablonów

```
template <typename T> T min(T a, T b);
```

- `min(10, 12);` // wywoła `min<int>(10, 12)`
- `min(10.1, 12.2);` // wywoła `min<double>(10.1, 12.2)`
- `min(10, 10.1);` // błąd kompilacji -> różne typy
- `min<int>(10, 10.1);` // wywoła `min<int>(10, 10.1)`

Dwie fazy kompilacji szablonów

- kompilator zbiera dostępne deklaracje szablonów
 - nie generuje kodu
 - nie interesuje go, jakie są parametry szablonów
 - sprawdza poprawność składni tej części ciała, która nie zależy od parametrów szablonu
- kompilator instancjonuje potrzebny mu szablon dla konkretnej kombinacji parametrów szablonowych
 - sprawdza poprawność składni ciała szablonu, dla konkretnych parametrów szablonowych

Typy zależne (typename)

```
template<typename T>
T::size_type process(const T& arg) {
    T::size_type * iter (T::len);
}
```

- funkcja działa tylko dla typu T, który zawiera składowe size_type oraz len
- czym jest T::size_type?
- czym jest T::len?
- czym jest iter?
- czym jest *?

Typy zależne (typename)

```
template<typename T>
typename T::size_type process(const T& arg) {
    typename T::size_type * iter (T::len);
}
```

- w razie wątpliwości standard nie pozwala kompilatorowi założyć, że coś jest typem
- musimy mu to wskazać
- często kompilator wie, co mieliśmy na myśli i sugeruje rozwiązanie
- w nowszych standardach niektóre typename nie są już potrzebne

Specjalizacja szablonu

- podstawienie pewnej konkretnej kombinacji argumentów za parametry szablonu
- domniemana
 - kiedy kompilator dopasowuje argumenty do szablonu
- jawna
 - kiedy programista wymusza konkretną implementację funkcji lub klasy dla danego zestawu argumentów szablonoowych

Specjalizacje szablonu

```
auto x = min<const char *>("def", "abc");
```

```
const char * min(const char *a, const char *b) { return a < b ? a : b; }
```

funkcja porówna wskaźniki

```
template<>
```

```
const char * min(const char *a, const char *b) {
```

```
    return strcmp(a, b) < 0 ? a : b;
```

```
}
```

Specjalizacje szablonu

- częściowa
 - część parametrów szablonych jest zastępowana konkretnymi argumentami
- pełna
 - wszystkie parametry szablone są zastępowane konkretnymi argumentami

Szablony bez szablonów - C++20 + auto

```
auto min(auto lhs, auto rhs) {  
    return lhs < rhs ? lhs : rhs;  
}
```

Biblioteka standardowa, programowanie generyczne



Składowe języka C++

- core language features
 - elementy, które wymagają zmiany składni języka
- biblioteka standardowa
 - elementy, które da się stworzyć poprzez zbudowanie z istniejących elementów

Typy danych

- pair
- tuple
- kolekcje
 - vector
 - deque
 - map
 - unordered_map
 - set
 - unordered_set
 - list

vector - pojemnik pierwszego wyboru

- w nowoczesnych komputerach vector będzie średnio najszybszy
 - elementy sąsiadująco w pamięci
 - https://quick-bench.com/q/IPNBidd8q5jajdSQoaWvx44r_nQ
- najważniejsze aspekty
 - rośnie “od ogona”
 - dopychanie na koniec względnie tanie
 - dopychanie na początek (lub w środku) względnie drogie
 - warto wołać `reserve()`, jeżeli znamy liczbę elementów, której potrzebujemy
 - wydajność mocno zależy od charakterystyki przechowywanego typu
 - dodanie elementu potencjalnie dokonuje realokację pamięci
 - istniejące elementy są memcpy/przesuwane/kopiuwane na nowe miejsce
 - bardzo łatwo popsuć wydajność -
<https://quick-bench.com/q/5Cw9DIGobK4MnWIJdm6Tz7HCv7E>

vector - `emplace_back` vs `push_back`

- `push_back`, `insert`
 - podajemy element, który zostanie dodany do wektora
 - element tworzony jest zanim zostanie przekazany (kopia/przeniesienie) do wektora
- `emplace_back`, `emplace`
 - podajemy argumenty wywołania konstruktora elementu, który ma być dodany
 - element tworzony jest już w wektorze (nie ma kopii ani przeniesienia)
 - zwraca referencję na element w wektorze

span (C++20)

- Jak napisać funkcję, co pobiera zestaw elementów sąsiadujących w pamięci?
 - `char tab[], const char*, std::string, std::string_view, std::vector<std::string>, ...?`
- wszystkie te typy to w zasadzie wskaźnik + rozmiar
 - `void foo(char *ptr, size_t N)`
- łatwo zapomnieć/przegapić/pomylić się w podaniu rozmiaru i będzie kłopot
 - wynika to z przekazywania oddzielnie wskaźnika i rozmiaru
 - a gdyby one były scalone w jeden obiekt?
- `std::span`
 - `void foo(std::span<char> str);`
 - automatyczna konwersja z kompatybilnych typów
- dynamiczny (`std::span<int>`) vs statyczny (`std::span<int, N>`) rozmiar



Co to jest za algorytm?

```
void algorithm1(int n, int t[])  
{  
    for(int i=0; i<n; i++) {  
        int k=i;  
        for(int j=i+1; j<n; j++)  
            if(t[j]<t[k]) k=j;  
        int tmp = t[k];  
        t[k] = t[i];  
        t[i] = tmp;  
    }  
}
```

Co to jest za algorytm?

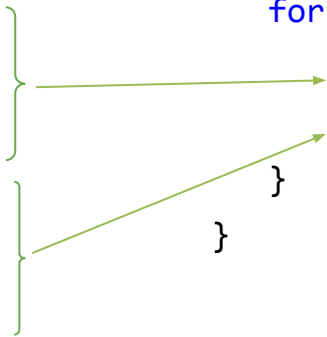
```
void algorithm1(auto from, auto to)
{
    for (; from != to; ++from) {
        auto iter = std::min_element(from, to);
        std::iter_swap(from, iter);
    }
}
```



Porównanie - selection sort

```
void selection_sort(int n, int t[])  
{  
    for(int i=0; i<n; i++) {  
        int k=i;  
        for(int j=i+1; j<n; j++) {  
            if(t[j]<t[k]) k=j;  
        }  
        int tmp = t[k];  
        t[k] = t[i];  
        t[i] = tmp;  
    }  
}
```

```
void selection_sort(auto from, auto to)  
{  
    for (; from != to; ++from) {  
        auto iter = std::min_element(from, to);  
        std::iter_swap(from, iter);  
    }  
}
```



Co to jest za algorytm?

```
void algorithm2(int n, int arr[])
{
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

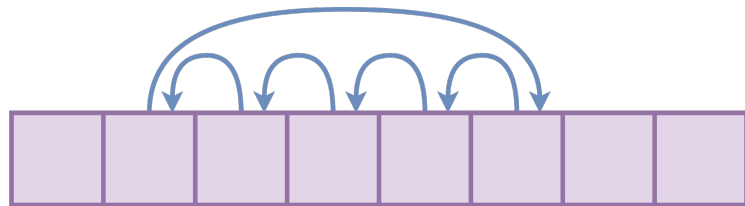
Co to jest za algorytm?

```
void algorithm2(auto from, auto to)
```

```
{
```

```
    for (auto curr = from; curr != to; ++curr) {  
        auto position = std::upper_bound(from, curr, *curr);  
        std::rotate(position, curr, curr+1);  
    }
```

```
}
```



Porównanie

```
void insertion_sort(int n, int arr[])
{
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```
void insertion_sort(auto from, auto to)
{
    for (auto curr = from; curr != to; ++curr) {
        auto position = std::upper_bound(from, curr, *curr);
        std::rotate(position, curr, curr+1);
    }
}
```

Quicksort?



Algorytmy

- Std C++ posiada ponad 100 algorytmów ogólnego przeznaczenia
- w C++20 dochodzą wersje części algorytmów w postaci uzakresowanej
- poza algorytmami jest mnóstwo funkcji pomocniczych
- algorytmy opierają się na iteratorach

Zalety używania gotowych algorytmów

- **czytelność**
 - dobrze nazwane funkcje zamiast ściany kodu
 - Clean Code → funkcja wołająca działa na jednym poziomie abstrakcji
- **odporność na błędy**
 - przetestowane przez miliony użytkowników w ciągu kilkudziesięciu lat
 - automatycznie dostajemy ewentualne poprawki przy rekompilacji z nowszą STD
- **wydajność**
 - algorytmy dopasowują się do charakterystyki danych, na których operują
 - kompilator “widzi” kod (szablony), co pozwala mu go zoptymalizować

<https://www.youtube.com/watch?v=W2tW0dzgXHA> (“No raw loops”, Sean Parent)

Iterator

- Zastosowania

- dostęp do zawartości złożonego obiektu bez ujawniania jego wewnętrznej struktury
- możliwość jednoczesnego przechodzenia przez agregat przez kilka źródeł
- uproszczenie interfejsu agregatu
- możliwość wielokrotnej iteracji po agregacie
- możliwość zróżnicowania sposobu przejścia przez agregat (np. przeglądanie drzewa wszerz lub włąb) bez modyfikacji interfejsu agregatu

Adaptery iteratorów

- `reverse_iterator`
 - `make_reverse_iterator`
- `move_iterator`
 - `make_move_iterator`
- `back_insert_iterator`
 - `back_inserter`
- `front_insert_iterator`
 - `front_inserter`
- `insert_iterator`
 - `inserter`

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <vector>
```

```
int main() {
```

```
    std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
    std::vector<int> clone;
```

```
    std::copy(v.begin(), v.end(), std::back_inserter(clone));
```

```
    std::copy(v2.begin(), v2.end(), std::ostream_iterator<int>(std::cout, " ")); std::cout << '\n';
```

```
    return 0;
```

```
}
```

zakres

iterator
wyjściowy

Algorytmy z modyfikatorami - kopiowanie warunkowe

```
bool filter(int value) {  
    return value > 0 && value < 10;  
}
```

```
const std::set<int> s = {1, 5, 3, 8, 9, 15, 14, 0};  
std::array<int, 8> result{};  
std::copy_if(s.begin(), s.end(), result.begin(), filter);
```



Algorytmy z modyfikatorami - kopiowanie warunkowe

```
const std::set<int> s = {1, 5, 3, 8, 9, 15, 14, 0};  
std::array<int, 8> result{};  
std::copy_if(s.begin(), s.end(),  
             result.begin(), [](int value) { return value > 0 && value < 10; }  
);
```



Wyrażenia lambda

Wyrażenie lambda definiuje *domknięcie* - anonimową funkcję, która przechowuje dane ze swojego środowiska, które są jej potrzebne do obliczeń

- Można ją przypisać do zmiennej (najczęściej auto)
- Można jej używać jak funkcji
- Realizacja przez obiekt funkcyjny (lambda to klasa a nie funkcja)

Lambda - składnia

[przechwycenia] (parametry) { ciało funkcji }

```
auto fun = [] (int a, int b) { return a > b; };
```

```
bool res = fun(2, 1);
```

```
std::vector<int> vec;
```

```
int cnt = std::count_if(vec.begin(), vec.end(), [](int v) { return v < 0; });
```

Lambda - przechwylenia

```
[przechwylenia] ( parametry ) { ciało funkcji }
```

Przechwylenia:

- = - wszystko przez wartość (kopia)
- & - wszystko przez referencję
- nazwa_zmiennej - pojedyncza zmienna przez wartość
- &nazwa_zmiennej - pojedyncza zmienna przez referencję
- definicje rozdzielone przecinkami
 - [&x, =] - x przez referencję, wszystko inne przez wartość
 - [this] - this przez wartość (od C++20 '=' nie obejmuje 'this')

Przechwycenia - przykład

```
int count_less_than(auto from, auto to, int threshold) {  
    auto predicate = [threshold] (int value) { return value < threshold; };  
    return std::count_if(from, to, predicate);  
}
```

Lambda - realizacja

```
auto lambda = [x, y, &z](int a, int b) {  
    return x+y+z-a-b;  
};  
lambda(1, 2);
```

```
class FunObj {  
public:  
    FunObj(int _x, int _y, int &_z)  
        : x(_x), y(_y), z(_z) {}  
    int operator()(int a, int b) const {  
        return x+y+z-a-b;  
    }  
private:  
    int x, y, &z;  
};
```

```
auto funobj = FunObj(x, y, z);  
funobj(1, 2);
```



Lambda - Przechwytywanie wyrażeń

- C++11
 - [=,&v], [v,&], ... przechwytywanie przez wartość lub referencję istniejących zmiennych
- C++14
 - [&r = z, x = 7, ptr = std::move(o)]
 - deklaruje zmienne jako auto
 - przypisuje im wyniki wyrażeń

```
std::unique_ptr<Object> ptr = ...;
std::function<void(void)> f = [o = std::move(ptr)] () {
    consume(o.get()); };
// ...
f();
```

Generalizacja wyrażeń lambda

```
[ ] (auto v1, auto v2) { ... }
```

- zasadniczo działa identycznie jak szablon:
 - `template<class U, class T1, class T2> U funkcja( T1 v1, T2 v2)`
- pozwala na stworzenie rodziny funkcji, które działają dla dowolnych danych zgodnych ze składnią ciała funkcji

```
auto l = [ ] (auto v1, auto v2) { return std::min(v1, v2); }
```

- `l` można zwołać na czymkolwiek, co posiada przeładowanie `std::min()`

Lambda w C++20

[przechwycenia] <specyfikacja typów> (parametry) { ciało funkcji }

```
auto min_cpp17 = [](auto left, auto right) { // left i right mogą być różnych typów
    return left < right ? left : right;
};
```

```
auto min_cpp20 = [] <typename T> (T left, T right) { // left i right są tego samego typu
    return left < right ? left : right;
};
```

Algorytmy - sprawdzanie

- any_of
- all_of
- none_of

Algorytmy - wyszukiwanie

- find
- find_if
- adjacent_find
- mismatch
- search

Algorytmy - wyszukiwanie binarne

- lower_bound
- upper_bound
- binary_search
- equal_range

Algorytmy - zliczanie

- count
- count_if

Algorytmy - modyfikacja

- copy
- copy_if
- copy_n
- copy_backward
- move
- move_backward
- fill
- fill_n
- transform
- generate
- generate_n
- replace
- replace_if
- reverse
- rotate
- unique

Algorytmy - numeryczne

- `iota`
- `accumulate`
- `inner_product`
- `partial_sum`

Algorytmy - usuwanie

- `remove`
- `remove_if`

Algorytmy - losowość

- generatory losowości
 - random_device
 - mt19937
 - rozkłady zmiennych losowych
- algorytmy
 - shuffle
 - sample

Algorytmy

... i wiele innych

<https://en.cppreference.com/w/cpp/algorithm>

<https://en.cppreference.com/w/cpp/numeric>

<https://www.youtube.com/watch?v=2olsGf6JlKU>

(105 STL algorithms in less than an hour)

Algorytmy - wersje uzakresowione

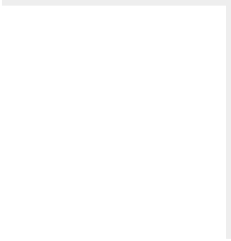
- Od C++20 możemy używać zakresów (ranges)
- dwa oddzielne obiekty iteratorów zastąpione są jednym (iterator+strażnik)
- daje to całkiem nowe możliwości, o których tu nie mamy czasu powiedzieć...
- ... ale daje też nowe wersje algorytmów
 - zakres zamiast pary iteratorów → `std::ranges::sort(v);`
 - odwzorowania (projections) → `std::ranges::find(people, "Piotr", &Person::name);`

Pozostałe

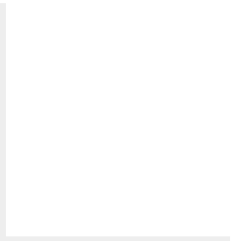
- biblioteka standardowa to nie tylko kontenery i algorytmy
 - funkcje pomocnicze
 - wielowątkowość
 - czas i kalendarz
 - metaprogramowanie
 - formatowanie tekstu
 - liczby zespolone
 - system plików
 - wyrażenia regularne

Programowanie uogólnione

- Tworzenie kodu programu
 - bez znajomości typów danych
 - zachowujące kontrolę typów
- W skrócie
 - używamy typów wg tego co potrafią, a nie jakimi konkretnie typami są i po czym dziedziczą
- Realizacja
 - polimorfizm statyczny
 - span
 - iteratory
 - auto



Na czym polega polimorfizm?



Polimorfizm

- przeciążanie funkcji
- szablony
- warianty
- polimorfizm dynamiczny (funkcje wirtualne)

Dedukcja typu zmiennej

auto var = <wyrażenie>

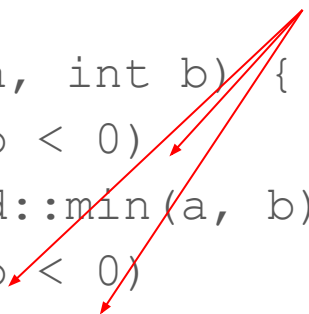
- używane gdy typ wyniku wyrażenia jest
 - nieistotny
 - trudny do odgadnięcia
 - niewygodny do zapisania
 - słowu kluczowemu auto mogą towarzyszyć modyfikatory (const, &, *, itd.) pomagające w dedukcji
-
- `for (auto iter = container.begin(); iter != container.end(); ++iter) { ... }`
 - `auto lambda = [...](...) { ... }`

Dedukcja typu wyniku funkcji

auto funkcja(...) { ... }

- typ funkcji określany jest na podstawie operandu **return**
- ewentualne wywołania rekurencyjne nie mogą być przed pierwszym **return**
- wszystkie **return** muszą zwracać ten sam typ

```
auto funkcja(int a, int b) {  
    if (a < 0 && b < 0)  
        return std::min(a, b);  
    if (a > 0 && b < 0)  
        return a + b;  
    return std::max(a, b);  
}
```



Almost-Always-Auto

- Preferowanie dedukowanych typów nad jawnie określonymi
- Używanie auto, decltype, decltype(auto)
- Tworzenie kodu dopasowanego do interfejsu a nie implementacji

- Jeżeli typ ma podążać za zmianami w kodzie, polegamy na dedukcji

```
auto a = init;
```

- Jeżeli typ ma być niezmienny, polegamy na jawnym określeniu

```
auto b = Type{ init };
```

```
Type c{ init };
```

