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** orax **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 niezmienne 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 stały obiekt
  - `void mutate2(std::string *)`
    - przekazanie przez wskaźnik na stały 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 obiektu, 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 {}; }
};
```

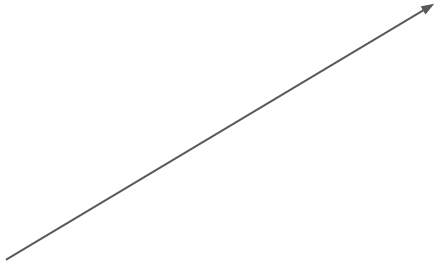
```
vector<unique_ptr<Staff>> teachers_and_technical_staff;
for (const auto& id: …) {
  // …
}
```

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(...);
}

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

czy brak studenta to jest sytuacja wyjątkowa?

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”
- 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 zawoła 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);
}
```