Programowanie Obiektowe 2
Semestr letni 2020/21
Materiały z wykładu i laboratorium

Przemysław Olbratowski

26 kwietnia 2021

Informacje organizacyjne oraz formularz do uploadu zadań domowych znajdują się na stronie `info.wsisiz.edu.pl/~olbratow`. Przy zadaniach domowych w nawiasach są podane terminy sprawdzeń.

Spis treści

1 Dziedziczenie: 6, 7, 20, 21 marca Dziedziczenie i polimorfizm

1.1 Wykład i laboratorium z działu Dziedziczenie

1.1.1 Figure: Kolorowe figury geometryczne

Graficzny interfejs pewnego programu dla dzieci składa się z kolorowych figur geometrycznych. Są one spisane w pliku tekstowym jak poniżej. Pierwsze słowo każdej linii określa rodzaj figury a drugie jej kolor. W przypadku koła po kolorze podany jest promień, zaś w przypadku prostokąta - szerokość i wysokość. Napisz klasę `Figure` reprezentującą figurę o kolorze opisanym łańcuchem tekstowym. Wyprowadź z niej klasę `Circle` reprezentującą koło o zadanym promieniu oraz klasę `Rectangle` reprezentującą prostokąt o zadanej szerokości i wysokości. Interfejs klasy `Figure` składa się z następujących elementów:

- Operator `<<`, po którego lewej stronie stoi referencja strumienia wyjściowego, po prawej - stała referencja figury, a wynikiem jest referencja podanego strumienia. Operator wypisuje do tego strumienia opis podanej figury jak w przykładzie poniżej.
- Konstruktor przyjmujący kolor figury.
- Stała bezargumentowa metoda `getColor` zwracająca kolor figury.
- Stała bezargumentowa metoda abstrakcyjna `getArea` zwracająca pole figury.

Na interfejs klasy `Circle` składają się:

- Konstruktor przyjmujący kolor i promień koła.
- Stała bezargumentowa metoda `getColor` zwracająca kolor koła.
- Stała bezargumentowa metoda `getArea` zwracająca pole koła.
- Stała bezargumentowa metoda `getRadius` zwracająca promień koła.

Interfejs klasy `Rectangle` to:

- Konstruktor przyjmujący kolor oraz szerokość i wysokość prostokąta.
- Stała bezargumentowa metoda `getColor` zwracająca kolor prostokąta.
- Stała bezargumentowa metoda `getArea` zwracająca pole prostokąta.
- Stałe bezargumentowe metody `getWidth` oraz `getHeight` zwracające odpowiednio szerokość i wysokość prostokąta.

Napisz funkcję `scan`, która przyjmuje referencję strumienia wejściowego ze spisem figur jak poniżej, odczytuje z niego jedną linię, tworzy dynamicznie odpowiednią figurę i zwraca jej adres. Napisz program `figure`, który wczytuje ze standardowego wejścia spis figur, tworzy na tej podstawie instancje odpowiednich klas, a ich adresy składuje w pojemniku typu wektor. Następnie wypisuje na standardowe wyjście w kolejnych liniach opis każdej figury oraz jej pole.

Przykładowe wykonanie

```
In: Rectangle red 5 4
In: Circle blue 1
In: Rectangle cyan 1.5 12
In: Circle green 3
In: Rectangle yellow 16 9
In: Circle gray 2.5
In: Circle magenta 4.2
In: Rectangle pink 10 5.5
```

```
Out: Rectangle red 5 4 20
Out: Circle blue 1 3.14159
Out: Rectangle cyan 1.5 12 18
Out: Circle green 3 28.2743
Out: Rectangle yellow 16 9 144
Out: Circle gray 2.5 19.635
Out: Circle magenta 4.2 55.4177
Out: Rectangle pink 10 5.5 55
```

Rozwiązanie

```
#include <cmath>
#include <iostream>
#include <string>
#include <vector>

constexpr double pi = 4. * std::atan(1.);

class Figure {
    friend std::ostream &operator <<(std::ostream &stream, const Figure &figure);
public:
    Figure(const std::string &color): color(color) {}
    const std::string &getColor() const {return color; }
    virtual double getArea() const = 0;
private:
    virtual std::ostream &print(std::ostream &stream) const {
        return stream << "Figure " << color;}
    std::string color; };

class Circle: public Figure {
public:
    Circle(const std::string &color, double radius): Figure(color), radius(radius) {}
    virtual double getArea() const {return pi * radius * radius; }
    double getRadius() const {return radius; }
private:
    virtual std::ostream &print(std::ostream &stream) const {
        return stream << "Circle " << getColor() << " " << radius;}
    double radius; };

class Rectangle: public Figure {
public:
    Rectangle(const std::string &color, double width, double height):
        Figure(color), width(width), height(height) {}
    virtual double getArea() const {return width * height; }
    double getWidth() const {return width; }
    double getHeight() const {return height; }
private:
    virtual std::ostream &print(std::ostream &stream) const {
        return stream << "Rectangle " << getColor() << " " << width << " " << height; }
    double width, height; };

std::ostream &operator <<(std::ostream &stream, const Figure &figure) {
    return figure.print(stream); }

Figure *scan(std::istream &stream) {
    std::string name, color;
```

```
    if (!(stream >> name >> color)) {
        return nullptr; }
    else if (name == "Circle") {
        double radius;
        stream >> radius;
        return new Circle(color, radius); }
    else if (name == "Rectangle") {
        double width, height;
        stream >> width >> height;
        return new Rectangle(color, width, height); }}

int main() {
    std::vector<Figure *> figures;
    while (Figure *figure = scan(std::cin)) {
        figures.push_back(figure); }
    for (const Figure *figure: figures) {
        std::cout << *figure << " " << figure->getArea() << std::endl; }
    for (const Figure *figure: figures) {
        delete figure; }}
```

1.2 Zadania domowe z działu Dziedziczenie (12, 19, 26 kwietnia)

1.2.1 Button: Przyciski graficznego interfejsu użytkownika - grupowo

Napisz klasę `Button` reprezentującą przycisk graficznego interfejsu użytkownika. Przycisk może być włączony lub wyłączony. Nowoutworzony przycisk jest wyłączony. Zadeklaruj:

- Stałą bezargumentową metodę `on`, która zwraca prawdę jeśli przycisk jest włączony oraz fałsz jeśli jest wyłączony.
- Bezargumentową metodę wirtualną `press` wywoływaną przy wciśnięciu przycisku.
- Bezargumentową metodę wirtualną `release` wywoływaną przy zwolnieniu przycisku.

Z klasy `Button` wyprowadź klasę `PushButton` reprezentującą przycisk, który włącza się przy każdym naciśnięciu i wyłącza przy każdym zwolnieniu. Wyprowadź też klasę `SwitchButton` reprezentującą przycisk, który włącza się przy jednym wciśnięciu a wyłącza przy następnym i tak dalej, zaś na zwolnienie nie reaguje. Klasy powinny być przystosowane do użycia w przykładowym programie poniżej. Klasy nie korzystają z żadnych plikó nagłówkowych.

Przykładowy program

```
int main() {
    std::cout << std::boolalpha;
    Button *pushButton = new PushButton;
    std::cout << pushButton->on() << " ";
    pushButton->press();
    std::cout << pushButton->on() << " ";
    pushButton->release();
    std::cout << pushButton->on() << std::endl;
    delete pushButton;
    Button *switchButton = new SwitchButton;
    std::cout << switchButton->on() << " ";
    switchButton->press();
    std::cout << switchButton->on() << " ";
    switchButton->release();
    std::cout << switchButton->on() << " ";
    switchButton->press();
    std::cout << switchButton->on() << std::endl;
    delete switchButton; }
```

Wykonanie

```
Out: false true false
Out: false true true false
```

1.2.2 Parallelogram: Równoległobok

Napisz klasę `Parallelogram` reprezentującą równoległobok. Zaimplementuj w niej:

- Konstruktor przyjmujący długość podstawy, długość drugiego boku, oraz wysokość.
- Stałą bezargumentową metodę `area`, która zwraca pole równoległoboku.
- Stałą bezargumentową metodę `perimeter`, która zwraca obwód równoległoboku.

Z klasy `Parallelogram` wyprowadź klasę `Rectangle` reprezentującą prostokąt. Zaimplementuj w niej konstruktor przyjmujący długość podstawy oraz długość drugiego boku. Z klasy `Rectangle` wyprowadź klasę `Square` reprezentującą kwadrat. Zaimplementuj w niej konstruktor przyjmujący długość boku. Klasy powinny być przystosowane do użycia w przykładowym programie poniżej. Klasy nie korzystają z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    Parallelogram *rectangle = new Rectangle(4, 5);
    std::cout << rectangle->area() << " " << rectangle->perimeter() << std::endl;
    delete rectangle;
    Rectangle *square = new Square(6);
    std::cout << square->area() << " " << square->perimeter() << std::endl;
    delete square; }
```

Wykonanie

```
Out: 20 18
Out: 36 24
```

1.2.3 Predicate: Predykat

Napisz klasę `Predicate` reprezentującą predykat. Zadeklaruj w niej stały abstrakcyjny operator `()`, który przyjmuje liczbę całkowitą i zwraca prawdę jeśli spełnia ona warunek predykatu albo fałsz w przeciwnym razie. Klasa powinna być przystosowana do użycia w przedstawionej niżej funkcji `count`. Funkcja ta przyjmuje stałą referencję wektora liczb całkowitych oraz stałą referencję predykatu i zwraca liczbę elementów wektora, które spełniają ten predykat. Z klasy `Predicate` wyprowadź klasy `Even` oraz `Negative` stwierdzające odpowiednio, czy liczba jest parzysta oraz ujemna. Klasy powinny być przystosowane do użycia w przykładowym programie poniżej. Program ten tworzy wektor o elementach całkowitych i wypisuje na standardowe wyjście liczbę elementów parzystych oraz ujemnych. Klasy nie korzystają z żadnych plików nagłówkowych.

Przykładowy program

```
int count(const std::vector<int> &elements, const Predicate &predicate) {
    int count = 0;
    for (int index = 0; index < elements.size(); ++index) {
        if (predicate(elements[index])) {
            ++count; }}
    return count; }

int main() {
    const std::vector<int> elements {-7, 12, -11, 2, 9, -4, -6, 5, 23, -1};
    std::cout << count(elements, Even()) << " " << count(elements, Negative()) << std::endl; }
```

Wykonanie

```
Out: 4 5
```

1.2.4 Event: Obsługa zdarzeń

Widgety graficznego interfejsu użytkownika muszą reagować na rozmaite zdarzenia, jak kliknięcie myszką lub naciśnięcie klawisza na klawiaturze. Każdy typ zdarzenia jest opisany oddzielną klasą. Przykładowo, klasa zdarzenia myszy przechowuje informację o tym, czy przycisk myszy został wciśnięty czy zwolniony, zaś klasa zdarzenia klawiatury pamięta kod ASCII znaku na wciśniętym klawiszu. Klasy wszystkich zdarzeń są wywiedzione z jednej klasy bazowej. Dzięki temu klasa każdego widgetu może zawierać jedną metodę obsługującą wszystkie zdarzenia. Metoda ta przyjmuje wskaźnik na zdarzenie, sprawdza jego typ i zależnie od tego podejmuje odpowiednie działania. Jest to metoda wirtualna reimplementowana w klasach poszczególnych widgetów. Nie jest ona jednak abstrakcyjna. Zamiast tego jej implementacja w bazowej klasie widgetu po prostu nic nie robi. Dzięki temu możliwe jest utworzenie instancji bazowej klasy widgetu. Zadeklaruj klasę bazową `Event` reprezentującą abstrakcyjne zdarzenie. Wyprowadź z niej klasy `Mouse` oraz `Key` opisujące odpowiednio kliknięcie myszy oraz naciśnięcie klawisza na klawiaturze. W klasie `Mouse` zaimplementuj:

- Konstruktor przyjmujący wartość logiczną. Prawda i fałsz oznaczają tu odpowiednio naciśnięcie i zwolnienie przycisku myszy.
- Stałą bezargumentową metodę `get` zwracającą prawdę albo fałsz jeśli zdarzenie reprezentuje odpowienio naciśnięcie lub zwolnienie przycisku myszy.

W klasie `Key` zaimplementuj:

- Konstruktor przyjmujący kod ASCII znaku na wciśniętym klawiszu.
- Stałą bezargumentową metodę `get` zwracającą kod ASCII znaku na wciśniętym klawiszu.

Do klasy `Widget` z poprzedniego zadania dodaj wirtualną metodę `handle`, która przyjmuje wskaźnik na dowolne zdarzenie i nic nie robi. Zreimplementuj tę metodę w klasach `Button` i `Edit`. Implementacja w klasie `Button` ignoruje zdarzenia klawiatury natomiast dla zdarzeń myszy wypisuje na standardowe wyjście nazwę widgetu oraz odpowienio napis `mouse pressed` lub `mouse released`, jak w przykładzie poniżej. Implementacja w klasie `Edit` ignoruje zdarzenia myszy natomiast dla zdarzeń klawiatury wypisuje nazwę widgetu, słowo `key`, oraz znak na wciśniętym klawiszu. Klasy zdarzeń oraz metoda `handle` powinny być przystosowane do użycia w przykładowym programie poniżej. Klasy korzystają tylko z plików nagłówkowych `iostream`, `string`, oraz `vector` lub `list`.

Przykładowy program

```
int main() {
    Mouse mouse(true);
    Key key('a');
    Button *button = new Button("button");
    Edit *edit = new Edit("edit");
    button->handle(&mouse);
    button->handle(&key);
    edit->handle(&mouse);
    edit->handle(&key);
    delete edit;
    delete button; }
```

Wykonanie

```
button mouse pressed
edit key a
```

1.2.5 Sequence: Ciąg liczbowy

Zadeklaruj abstrakcyjną klasę `Sequence` reprezentującą nieskończony ciąg liczbowy o wyrazach indeksowanych od zera. Zadeklaruj w niej stały abstrakcyjny operator indeksowania, który przyjmuje indeks wyrazu i zwraca jego wartość. Z klasy `Sequence` wyprowadź klasę `Arithmetic` reprezentującą ciąg arytmetyczny

$$a_n = a_0 + n\Delta$$

Zdefiniuj w niej konstruktor przyjmujący parametry a_0 oraz Δ. Z klasy `Sequence` wywiedź ponadto klasę `Fibonacci` reprezentującą ciąg Fibonacciego, określony zależnością rekurencyjną

$$a_0 = 0 \qquad a_1 = 1 \qquad a_{n+1} = a_n + a_{n-1}$$

Napisz jedną funkcję `print`, która przyjmuje stałą referencję ciągu oraz nieujemną liczbę całkowitą n i wypisuje na standardowee wyjście n pierwszych wyrazów tego ciągu. Klasy oraz funkcja powinny być przystosowane do użycia w przykładowym programie poniżej. Klasy nie korzystają z żadnych plików nagłówkowych, zaś funkcja korzysta tylko z pliku `iostream`.

Przykładowy program

```
int main() {
    print(Arithmetic(2., 3.), 10);
    print(Fibonacci(), 12); }
```

Wykonanie

```
Out: 2 5 8 11 14 17 20 23 26 29
Out: 0 1 1 2 3 5 8 13 21 34 55 89
```

1.2.6 Widget: Widgety graficznego interfejsu użytkownika - indywidualnie

Widgety to graficzne elementy interfejsu użytkownika, jak przyciski, suwaki, czy wyświetlacze. Są one często implementowane jako klasy wywiedzione z jednej klasy bazowej reprezentującej pusty widget. Skomplikowane widgety składają się z innych, na przykład edytor tekstu składa się z pola tekstowego oraz pionowego i poziomego suwaka. Całe okienko aplikacji również jest widgetem zawierającym wiele mniejszych kontrolek. Składowe dużego widgetu nazywamy jego dziećmi zaś sam ten widget nazywamy ich rodzicem. Graficzny układ widgetów opisany jest więc przez wielopoziomowe drzewo zawierania się jednych widgetów w innych. Korzeniem tego drzewa, czyli rodzicem wszystkich widgetów, jest okno aplikacji. Drzewo widgetów jest zaimplementowane jako dynamiczna struktura danych. Widgety są tworzone dynamicznie, a każdy z nich przechowuje listę wskaźników na swoje dzieci. Destruktor rodzica usuwa z pamięci wszystkie jego dzieci. Do usunięcia wszystkich widgetów aplikacji wystarczy więc usunięcie głównego okna. Każdy widget posiada unikatową nazwę umożliwiającą jego łatwe wyszukanie wśród wszystkich kontrolek aplikacji. Napisz klasę `Widget` reprezentującą widget o zadanej nazwie. Zaimplementuj:

- Konstruktor przyjmujący nazwę widgetu.
- Destruktor usuwający z pamięci wszystkie dzieci widgetu.
- Jedną metodę `add`, która przyjmuje wskaźnik na dowolny widget i dodaje go jako dziecko widgetu, na rzecz którego jest wołana.
- Jedną metodę `print`, która drukuje na standardowe wyjście nazwę widgetu oraz wszystkich jego dzieci jak w przykładzie poniżej. Dzieci drukowane są w kolejności ich dodawania metodą `add`.

Z klasy `Widget` wyprowadź klasy `Button` oraz `Edit` reprezentujące odpowiednio przycisk oraz pole tekstowe. W każdej z nich zaimplementuj jedynie konstruktor przyjmujący nazwę widgetu. Wszystkie klasy powinny być przystosowane do użycia w przykładowym programie poniżej. Klasy korzystają tylko z plików nagłówkowych `iostream`, `string`, oraz `vector` lub `list`.

Przykładowy program

```
int main() {
    Button *yes = new Button("yes");
    Button *no = new Button("no");
    Widget *panel = new Widget("panel");
    panel->add(yes);
    panel->add(no);
    Edit *edit = new Edit("edit");
    Widget *window = new Widget("window");
    window->add(panel);
    window->add(edit);
    window->print();
    delete window; }
```

Wykonanie

```
window
window panel
window panel yes
window panel no
window edit
```

2 Szablony: 10, 11, 24, 25 kwietnia Szablony funkcji i klas

2.1 Wykład i laboratorium z działu Szablony

2.1.1 Find: Wyszukiwanie elementu w dowolnym pojemniku

Napisz szablon funkcji `find`, która przyjmuje iterator początkowy i końcowy wycinka dowolnego pojemnika oraz stałą referencję wartości dowolnego typu i zwraca iterator pierwszego wystapienia tej wartości w wycinku. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej.

Przykładowy program

```
int main() {
    std::list<int> container {1, 5, -3, 6, 5, 2, 4};
    auto iterator = find(container.begin(), container.end(), -3);
    std::cout << std::distance(container.begin(), iterator) << std::endl; }
```

Wykonanie

```
Out: 2
```

Rozwiązanie

```
#include <iostream>
#include <iterator>
#include <list>

template<typename I, typename E>
I find(I begin, I end, const E &element) {
    for (; begin != end && *begin != element; ++begin);
    return begin; }

int main() {
    std::list<int> container {1, 5, -3, 6, 5, 2, 4};
    auto iterator = find(container.begin(), container.end(), -3);
    std::cout << std::distance(container.begin(), iterator) << std::endl; }
```

2.1.2 Find If: Wyszukiwanie warunkowe

Napisz szablon funkcji `find_f`, która przyjmuje iterator początkowy i końcowy wycinka dowolnego pojemnika oraz predykat przyjmujący stałą referencję elementu tego wycinka i zwracający wartość logiczną. Funkcja zwraca iterator pierwszego elementu wycinka, na którym predykat zwraca prawdę. Predykat może być zadany jako funkcja, obiekt funkcyjny lub wyrażenie lambda. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej.

Przykładowy program

```
int a1;

bool less(int n) {
    return n < a1; }

struct Less {
    Less(int a): a(a) {}
    bool operator ()(int n) const {return n < a; }
private:
```

```
    int a; };

int main() {
    std::vector<int> c {1, 5, -3, 6, 5, 2, 4};
    int a2, a3;
    std::cin >> a1 >> a2 >> a3;
    auto i1 = find_if(c.begin(), c.end(), less);
    std::cout << std::distance(c.begin(), i1) << std::endl;
    auto i2 = find_if(c.begin(), c.end(), Less(a2));
    std::cout << std::distance(c.begin(), i2) << std::endl;
    auto i3 = find_if(c.begin(), c.end(), [a3](int n) {return n < a3; });
    std::cout << std::distance(c.begin(), i3) << std::endl; }
```

Przykładowe wykonanie

```
 In: 0 0 0
Out: 2
Out: 2
Out: 2
```

Rozwiązanie

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

int a1;

bool less(int n) {
    return n < a1; }

struct Less {
    Less(int a): a(a) {}
    bool operator ()(int n) const {return n < a; }
private:
    int a; };

template<typename I, typename P>
I find_if(I b, I e, const P &predicate) {
    for (; b != e && !predicate(*b); ++b);
    return b; }

int main() {
    std::vector<int> c {1, 5, -3, 6, 5, 2, 4};
    int a2, a3;
    std::cin >> a1 >> a2 >> a3;
    auto i1 = find_if(c.begin(), c.end(), less);
    std::cout << std::distance(c.begin(), i1) << std::endl;
    auto i2 = find_if(c.begin(), c.end(), Less(a2));
    std::cout << std::distance(c.begin(), i2) << std::endl;
    auto i3 = find_if(c.begin(), c.end(), [a3](int n) {return n < a3; });
    std::cout << std::distance(c.begin(), i3) << std::endl; }
```

2.1.3 Vector: Pojemnik typu wektor

Napisz szablon klasy `Vector` reprezentującej uproszczony pojemnik typu wektor do przechowywania elementów typu zadanego parametrem szablonu. Zaimplementuj:

- Bezargumentowy konstruktor tworzący pusty wektor.
- Operator `[]`, który przyjmuje indeks elementu i zwraca jego referencję modyfikującą.
- Stałą bezargumentową metodę `size`, która zwraca liczbę przechowywanych elementów.
- Metodę `push_back`, która przyjmuje wartość typu zadanego parametrem szablonu i dodaje ją jako ostatni element wektora, w miarę potrzeby realokując pamięć.

Szablon powinien być przystosowany do użycia w przykładowym programie poniżej.

Przykładowy program

```
int main() {
    Vector<std::string> vector;
    for (std::string element; std::cin >> element;) {
        vector.push_back(element); }
    for (int index = 0; index < vector.size(); ++index) {
        std::cout << vector[index] << " "; }
    std::cout << std::endl; }
```

Przykładowe wykonanie

```
 In: ala ma kota
Out: ala ma kota
```

Rozwiązanie

```
template<typename Element>
class Vector {
public:
    Vector();
    ~Vector();
    Element &operator [](int index);
    int size() const;
    void push_back(Element element);
private:
    int _capacity, _size;
    Element *_data; };

template<typename Element>
Vector<Element>::Vector(): _capacity(), _size(), _data() {}

template<typename Element>
Vector<Element>::~Vector() {
    delete[] _data;
    _capacity = _size = 0;
    _data = nullptr; }

template<typename Element>
Element &Vector<Element>::operator [](int index) {
    return _data[index]; }

template<typename Element>
int Vector<Element>::size() const {
    return _size; }

template<typename Element>
```

```
void Vector<Element>::push_back(Element element) {
    if (_capacity == _size) {
        _capacity = (_capacity ? 2 * _capacity : 1);
        Element *data = new Element[_capacity];
        for (int index = 0; index < _size; ++index) {
            data[index] = _data[index]; }
        delete[] _data;
        _data = data; }
    _data[_size++] = element; }

int main() {
    Vector<std::string> vector;
    for (std::string element; std::cin >> element;) {
        vector.push_back(element); }
    for (int index = 0; index < vector.size(); ++index) {
        std::cout << vector[index] << " "; }
    std::cout << std::endl; }
```

2.2 Zadania domowe z działu Szablony (10, 17, 24 maja)

2.2.1 Binary: Stałe dwójkowe via wyrażenia stałe

Napisz funkcję `binary` przyjmującą nieujemną liczbę całkowitą, której zapis dziesiętny składa się z samych zer i jedynek. Funkcja traktuje je jak zapis dwójkowy i zwraca zadaną nim liczbę. Jeżeli wartość argumentu jest znana podczas kompilacji, wynik jest również obliczany w czasie kompilacji. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    constexpr int b = binary(101100);
    std::cout << b << std::endl; }
```

Wykonanie

```
Out: 44
```

2.2.2 Binary: Stałe dwójkowe via szablony funkcji

Napisz szablon bezargumentowej funkcji `binary` przyjmujący jako parameter nieujemną liczbę całkowitą, której zapis dziesiętny składa się z samych zer i jedynek. Szablon traktuje je jako zapis dwójkowy i zwraca zadaną nim liczbę. Wynik obliczany jest podczas kompilacji. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    constexpr int b = binary<101100>();
    std::cout << b << std::endl; }
```

Wykonanie

```
Out: 44
```

2.2.3 Binary: Stałe dwójkowe via szablony klas

Napisz szablon klasy `Binary` przyjmujący jako parameter nieujemną liczbę całkowitą, której zapis dziesiętny składa się z samych zer i jedynek. Szablon traktuje je jako zapis dwójkowy i zadaną nim wartość nadaje swej statycznej składowej `value`. Obliczenia są wykonywane podczas kompilacji. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    constexpr int b = Binary<101100>::value;
    std::cout << b << std::endl; }
```

Wykonanie

```
Out: 44
```

2.2.4 Bits: Drukowanie bitów

Napisz szablon funkcji `bits`, która przyjmuje liczbę typu danego parametrem szablonu i drukuje na standardowe wyjście wszystkie jej bity. Jako pierwszy drukuje bajt o największym adresie w pamięci, bajty oddziela spacjami, a bity w każdym bajcie drukuje poczynając od najbardziej znaczącego. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    bits((short)-1);
    bits((float)-1); }
```

Wykonanie przy zapisie Little Endian

```
Out: 11111111 11111111
Out: 10111111 10000000 00000000 00000000
```

2.2.5 Count: Zliczanie argumentów

Napisz szablon funkcji `count`, która przyjmuje dowolną liczbę argumentów dowolnych typów i zwraca liczbę tych argumentów. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    std::cout << count(1) << " " << count(1, 3.1415, "hello") << std::endl; }
```

Wykonanie

```
Out: 1 3
```

2.2.6 Euclid: Największy wspólny dzielnik argumentów

Napisz szablon funkcji `euclid`, która przyjmuje dowolną liczbę dodatnich liczb całkowitych i zwraca ich największy wspólny dzielnik. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    std::cout << euclid(8) << " " << euclid(30, 105, 210) << std::endl; }
```

Wykonanie

```
Out: 8 15
```

2.2.7 Integer: Arytmetyka całkowita na dowolnej liczbie bajtów

Napisz szablon klasy `Integer` reprezentującej liczbę całkowitą bez znaku, zapisaną na tylu bajtach, ile wynosi parametr szablonu. Zaimplementuj:

- Operatory arytmetyczne `+` i `*`.
- Operatory `<<` i `>>`.

Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    Integer<10> a, b;
    std::cin >> a >> b;
    std::cout << a + b << " " << a * b << std::endl; }
```

Przykładowe wykonanie

```
 In: 549755813888 549755813888
Out: 1099511627776 302231454903657293676544
```

2.2.8 Lesser: Mniejszy z dwóch argumentów

Napisz szablon funkcji `lesser`, która przyjmuje stałe referencje dwóch obiektów typu zadanego parametrem szablonu i zwraca stałą referencję mniejszego z nich. Szablon nie korzysta z żadnych plików nagłówkowych. Oprócz tego napisz funkcję `lesser`, która przyjmuje dwie stałe napisowe i zwraca adres tej z nich, która wypada pierwsza w kolejności alfabetycznej. Funkcja korzysta tylko z pliku nagłówkowego `cstring`. Szablon i funkcja powinny być przystosowane do użycia w przykładowym programie poniżej.

Przykładowy program

```
int main() {
    std::cout << lesser(3.12, 2.13) << std::endl;
    std::cout << lesser(std::string("eisenhower"), std::string("einstein")) << std::endl;
    std::cout << lesser("eisenhower", "einstein"); }
```

Wykonanie

```
Out: 2.13
Out: einstein
Out: einstein
```

2.2.9 Limit: Największe wartości całkowite

Napisz szablon bezargumentowej funkcji `limit` zwracającej największą wartość typu całkowitego zadanego parametrem szablonu. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Program ten wypisuje na standardowe wyjście największe wartości wszystkich predefiniowanych typów całkowitych ze znakiem i bez znaku, znalezione przy pomocy szablonu `limit` oraz dane odpowiednimi stałymi z biblioteki standardowej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    std::cout << CHAR_MAX << " " << (int)limit<signed char>() << std::endl;
    std::cout << UCHAR_MAX << " " << (int)limit<unsigned char>() << std::endl;
    std::cout << SHRT_MAX << " " << limit<signed short>() << std::endl;
    std::cout << USHRT_MAX << " " << limit<unsigned short>() << std::endl;
    std::cout << INT_MAX << " " << limit<signed int>() << std::endl;
    std::cout << UINT_MAX << " " << limit<unsigned int>() << std::endl;
    std::cout << LONG_MAX << " " << limit<signed long>() << std::endl;
    std::cout << ULONG_MAX << " " << limit<unsigned long>() << std::endl;
    std::cout << LLONG_MAX << " " << limit<signed long long>() << std::endl;
    std::cout << ULLONG_MAX << " " << limit<unsigned long long>() << std::endl; }
```

Wykonanie w systemie Windows

```
Out: 127 127
Out: 255 255
Out: 32767 32767
Out: 65535 65535
Out: 2147483647 2147483647
Out: 4294967295 4294967295
Out: 2147483647 2147483647
Out: 4294967295 4294967295
Out: 9223372036854775807 9223372036854775807
Out: 18446744073709551615 18446744073709551615
```

2.2.10 Minimum: Najmniejszy argument

Napisz szablon funkcji `minimum`, która przyjmuje dowolną liczbę argumentów typu zadanego parametrem szablonu, i zwraca najmniejszy z nich. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    std::cout << minimum(2) << " " << minimum(3, 1, 4) << std::endl; }
```

Wykonanie

```
Out: 2 1
```

2.2.11 Modulo: Arytmetyka modularna - indywidualnie

Artymetyka modularna o module n działa na nieujemnych liczbach całkowitych mniejszych od n, a wynik każdego działania zastępuje resztą z jego dzielenia przez n. Przykładowo, dla $n = 16$ mamy $15 + 2 = 1$, $15 * 2 = 14$. Napisz szablon klasy `Modulo` reprezentującej liczbę w arytmetyce modularnej z modułem n zadanym parametrem szablonu. Zaimplementuj:

- Konstruktor inicjalizujący obiekt wartością zero.
- Operatory preinkrementacji oraz postinkrementacji modulo n
- Operatory `+` oraz `*` obliczające sumę oraz iloczyn modulo n.
- Operatory `<<` i `>>`. Jeżeli liczba wczytywana operatorem `>>` jest większa lub równa n, to operator zastępuje ją resztą z dzielenia przez n.

Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon korzysta tylko z plików nagłówkowych `istream` oraz `ostream`.

Przykładowy program

```
int main() {
    Modulo<16> modulo1, modulo2;
    std::cin >> modulo1 >> modulo2;
    std::cout << modulo1 + modulo2 << " " << modulo1 * modulo2 << std::endl;
    std::cout << ++modulo1 << " " << modulo2++ << std::endl; }
```

Przykładowe wykonanie

```
 In: 15 2
Out: 1 14
Out: 0 2
```

2.2.12 Power: Potęga całkowita via wyrażenia stałe

Napisz funkcję `power`, która przyjmuje nieujemną liczbę całkowitą oraz liczbę rzeczywistą i zwraca wyznaczoną nimi całkowitą potęgę liczby rzeczywistej. Jeżeli wartości argumentów są znane podczas kompilacji, wynik jest również obliczany w czasie kompilacji. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    constexpr double p = power(2, 1.2);
    std::cout << p << std::endl; }
```

Przykładowe wykonanie

```
Out: 1.44
```

2.2.13 Power: Potęga całkowita via szablony funkcji

Napisz szablon funkcji `power`, która przyjmuje liczbę rzeczywistą i zwraca jej potęgę o nieujemnym wykładniku całkowitym zadanym parametrem szablonu. Jeżeli argument funkcji jest znany podczas kompilacji, potęga jest również obliczana w czasie kompilacji. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    constexpr double p = power<2>(1.2);
    std::cout << p << std::endl; }
```

Wykonanie

```
Out: 1.44
```

2.2.14 Power: Potęga całkowita via szablony klas

Napisz szablon klasy `Power` posiadającej statyczną metodę `value`, która przyjmuje liczbę rzeczywistą i zwraca jej potęgę o nieujemnym wykładniku całkowitym zadanym parametrem szablonu. Jeżeli wartość argumentu metody jest znana podczas kompilacji, potęga jest również obliczana w czasie kompilacji. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    constexpr double p = Power<2>::value(1.2);
    std::cout << p << std::endl; }
```

Wykonanie

```
Out: 1.44
```

2.2.15 Print: Drukowanie argumentów

Napisz szablon funkcji `print`, która przyjmuje dowolną liczbę argumentów dowolnego typu i drukuje je wszystkie na standardowe wyjście. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szblon korzysta tylko z pliku nagłówkowego `iostream`.

Przykładowy program

```
int main() {
    print(1);
    print(1, 3.1415, "hello"); }
```

Wykonanie

```
Out: 1
Out: 1 3.1415 hello
```

2.2.16 Rotate: Bitowe przesunięcie cykliczne

Napisz szablon funkcji `rotate`, która przyjmuje referencję zmiennej całkowitej x typu danego parametrem szablonu, a także nieujemną liczbę całkowitą n, i przesuwa bity zmiennej x cyklicznie o n pozycji w lewo. Funkcja jest przeznaczona jedynie dla wbudowanych typów całkowitych bez znaku. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    unsigned char integer = 129;
    rotate(integer, 2);
    std::cout << (int)integer << std::endl; }
```

Wykonanie

```
Out: 6
```

2.2.17 Stack: Stos dowolnych elementów - grupowo

Napisz szablon klasy `Stack` opisującej stos elementów typu danego parametrem szablonu. Zaimplementuj:

- Konstruktor bezargumentowy tworzący pusty stos.
- Destruktor usuwający ze stosu wszystkie elementy.
- Stałą bezargumentową metodę `empty`, która zwraca prawdę jeśli stos jest pusty albo fałsz w przeciwnym razie.
- Metodę `push`, która przyjmuje stałą referencję elementu i odkłada na stos jego kopię.
- Metodę `pop`, która zdejmuje ze stosu element i zwraca jego kopię.

Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    Stack<std::string> stack;
    std::string string;
    while (std::cin >> string) {
        stack.push(string); }
    while (!stack.empty()) {
        std::cout << stack.pop() << " "; }
    std::cout << std::endl; }
```

Przykładowe wykonanie

```
 In: lorem ipsum dolor sit amet
Out: amet sit dolor ipsum lorem
```

2.2.18 Sum: Sumowanie argumentów

Napisz szablon funkcji `sum`, która przyjmuje dowolną liczbę argumentów typu zadanego parametrem szablonu, i zwraca ich sumę. Szablon powinien być przystosowany do użycia w przykładowym programie poniżej. Szablon nie korzysta z żadnych plików nagłówkowych.

Przykładowy program

```
int main() {
    std::cout << sum(6) << " " << sum(1, 2, 3) << std::endl; }
```

Wykonanie

```
Out: 6 6
```

3 Algorytmy: 15, 16, 29, 30 maja Algorytmy biblioteki standardowej

3.1 Przykłady laboratoryjne z działu Algorytmy

3.1.1 Phones: Książka telefoniczna

Pewien plik tekstowy przechowuje dane abonentów telefonicznych zapisane jak poniżej. Pierwsza kolumna zawiera numery telefonów, druga nazwiska, trzecia imiona. Kolumny są oddzielone przecinkami. Dane w pliku są zapisane nieporządnie. Niektóre linie są puste, w wielu miejscach znajdują się niepotrzebne spacje, a wielkość liter jest nieuporządkowana. Napisz program `phones`, który wczytuje ze standardowego wejścia sekwencję cyfr lub liter. Jeżeli wpisno sekwencję cyfr, program wypisuje na standardowe wyjście dane wszystkich abonentów, których numery telefonów zawierają tę sekwencję. Jeżeli podano sekwencję liter, program wypisuje dane wszystkich abonentów, których nazwiska zawierają tę sekwencję, przy czym wielkość liter nie ma znaczenia. Abonenci są wypisywani w alfabetycznej kolejności nazwisk. Wydruk jest sformatowany jak poniżej. Cyfry nie są niczym oddzielone, spacje się nie powtarzają, nazwiska są drukowane dużymi literami, pierwsze litery imion są duże a pozostałe małe. Następnie program wczytuje kolejną sekwencję cyfr lub liter i tak do napotkania końca pliku.

Przykładowy plik z danymi

```
523 12 39 64, mozart  ,   wolfgang amadeus
657228198,    von  GOETHE, johann    Wolfgang

910375523   , de   Cervantes SAAVEDRA, MIGUEL
123-45-67-89, CHOPIN    , frederic  francois
836 287-556 , picasso,  Pablo RUIZ


33 31 23  555,van  der waals,johannes DIDERIK
563 - 765 - 230     ,   de la  tour,georges
```

Przykładowe wykonanie

```
 In: 523
Out: 910375523, DE CERVANTES SAAVEDRA, Miguel
Out: 563765230, DE LA TOUR, Georges
Out: 523123964, MOZART, Wolfgang Amadeus
 In: aa
Out: 910375523, DE CERVANTES SAAVEDRA, Miguel
Out: 333123555, VAN DER WAALS, Johannes Diderik
```

3.2 Zadania domowe z działu Algorytmy (7, 14, 21 czerwca)

3.2.1 Combination: Losowa kombinacja

Napisz program `combination`, która czyta ze standardowego wejścia nieujemne liczby całkowite do napotkania końca pliku i wypisuje na standardowe wyjście w losowej kolejności tyle zer, jedynek i tak dalej, ile wynoszą kolejne wczytane liczby. Program zawiera po jednym wywołaniu funkcji `std::accumulate`, `std::fill_n` oraz `std::random_shuffle` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `cstdlib`, `ctime`, `algorithm`, `iostream`, `numeric` i `vector`.

Przykładowe wykonanie

```
 In: 3 0 5 2
Out: 2 3 2 0 2 3 2 0 0 2
```

3.2.2 Duplicates: Duplikaty - indywidualnie

Napisz program `duplicates`, który czyta ze standardowego wejścia liczby całkowite do napotkania końca pliku i wypisuje na standardowe wyjście w kolejności rosnącej wszystkie liczby powtarzające się, każdą tylko raz. Program zawiera po jednym wywołaniu funkcji `std::adjacent_find`, `std::find_if` oraz `std::sort` i nie używa innych algorytmów bibliteki standardowej. Program załącza tylko pliki nagłówkowe `algorithm`, `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 13 7 2 13 5 2 1 13
Out: 2 13
```

3.2.3 Iota: Uogólnienie algorytmu iota

Napisz funkcję `iota`, która przyjmuje modyfikujący iterator początkowy i końcowy wycinka wektora liczb całkowitych, a także wartość początkową oraz krok i wpisuje do kolejnych komórek wycinka liczby z zadanym krokiem począwszy od podanej wartości początkowej. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja zawiera jedno wywołanie algorytmu `std::for_each` i nie używa innych algorytmów biblioteki standardowej. Funkcja korzysta tylko z plików nagłówkowych `algorithm` i `vector`.

Przykładowy program

```
int main() {
    std::vector<int> vector(10);
    iota(vector.begin(), vector.end(), 7, 2);
    std::for_each(vector.begin(), vector.end(),
                  [](int element){std::cout << element << " "; });
    std::cout << std::endl; }
```

Wykonanie

```
Out: 7 9 11 13 15 17 19 21 23 25
```

3.2.4 Median: Mediana

Medianę z kilku liczb rzeczywistych obliczamy następująco. Zapisujemy liczby w kolejności niemalejącej. Jeżeli jest ich nieparzysta liczba, to mediana jest środkową z nich. W przeciwnym razie mediana jest średnią arytmetyczną dwóch środkowych liczb. Napisz program `median`, który czyta za standardowego wejścia liczby rzeczywiste do napotkania końca pliku i wypisuje na standardowe wyjście ich medianę. Program zawiera jedno wywołanie funkcji `std::sort` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `algorithm`, `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 10.5 -12.3 5.7 9.9 -1 15
Out: 7.8
```

3.2.5 Memory: Gra karciana Memory

W prostej jednoosobowej wersji gry memory rekwizytami są dwie karty z literą A, dwie z literą B i tak dalej. Karty tasujemy i układamy w jednym rzędzie koszulkami do góry. Odkrywamy losowo dwie karty. Jeżeli są na nich różne litery, to zakrywamy je z powrotem, a w przeciwnym razie zabieramy. Celem gry jest zabranie wszystkich kart w jak najmniejszej liczbie prób. Napisz program `memory` symulujący taką grę. Program wyświetla aktualne ułożenie kart w następujący sposób:

```
Out:  1  B     C     6
```

Liczby to kolejne numery kart licząc od lewej, litery to odkryte karty, a puste miejsca to już zabrane karty. Program przyjmuje jako argument wywołania liczbę liter, tasuje karty i wyświetla początkowe ułożenie ze wszystkimi kartami zakrytymi. Następnie wczytuje numery dwóch kart i wyświetla ułożenie z tymi kartami odkrytymi. Po trzech sekundach drukuje tyle pustych linii, aby dotychczasowy wydruk zniknął z ekranu. Następnie wyświetla kolejne ułożenie odpowiednio zakrywając lub usuwając ostatnio odkryte karty i tak dalej. Po zabraniu wszystkich kart program automatycznie kończy działanie.

Przykładowe wykonanie

```
  Linux: ./memory 2
Windows: memory.exe 2

Out: 1  2  3  4
 In: 1 2
Out: A  B  3  4

Out: 1  2  3  4
 In: 3 4
Out: 1  2  B  A

Out: 1  2  3  4
 In: 2 3
Out: 1  B  B  4

Out: 1        4
 In: 1 4
Out: A        A
```

3.2.6 Mode: Najczęściej występująca liczba

Napisz program `mode`, który czyta ze standardowego wejścia wartości całkowite do napotkania końca pliku i wypisuje na standardowe wyjście wartość najczęściej się powtarzającą oraz liczbę jej wystąpień. Jeżeli takich wartości jest kilka, program wypisuje najmniejszą z nich. Program zawiera jedno wywołanie funkcji `std::sort` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `algorithm`, `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 2 1 7 7 5 2 0 2 7 1
Out: 2 3
```

3.2.7 Neighbors: Najbliżsi sąsiedzi

Napisz program `neighbors`, który czyta ze standardowego wejścia liczby rzeczywiste do napotkania końca pliku i wypisuje na standardowe wyjście w kolejności niemalejącej te dwie spośród nich, które się najmniej różnią. Program zawiera jedno wywołanie funkcji `std::sort` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `cmath`, `algorithm`, `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 10 -1.2 2.3 5.5 -5.5 7.1 0.4
Out: 5.5 7.1
```

3.2.8 Nth Element: Element n-ty co do wartości

Napisz program `nth_element`, który przyjmuje jako argument wywołania nieujemną liczbę całkowitą n, czyta ze standardowego wejścia liczby rzeczywiste do napotkania końca pliku i wypisuje na standardowe wyjście liczbę n-tą co do wartości. Program zawiera jedno wywołanie funkcji `std::nth_element` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `cstdlib`, `algorithm`, `iostream` i `vector`.

Przykładowe wykonanie

```
  Linux: ./selection 5
Windows: selection.exe 5
 In: 12.3 0.7 -0.1 8.3 5.7 0.7 8.3
Out: 8.3
```

3.2.9 Parity: Sortowanie według parzystości

Napisz program `parity`, który czyta ze standardowego wejścia liczby całkowite do napotkania końca pliku i wypisuje na standardowe wyjście najpierw wszystkie parzyste w kolejności niemalejącej, a potem wszystkie nieparzyste w takiej samej kolejności. Program zawiera jedno wywołanie funkcji `std::sort` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `algorithm` i `vector`.

Przykładowe wykonanie

```
 In: 1 12 5 17 3 10 7 8
Out: 8 10 12 1 3 5 7 17
```

3.2.10 Partial Sum: Sumy częściowe

Napisz funkcję `partial_sum`, która przyjmuje modyfikujący iterator początkowy i końcowy wycinka wektora liczb całkowitych i do każdego elementu tego wycinka dodaje sumę wszystkich go poprzedzających. Funkcja powinna być przystosowana do użycia w przykładowym programie poniżej. Funkcja zawiera jedno wywołanie algorytmu `std::for_each` i nie używa innych algorytmów biblioteki standardowej. Funkcja korzysta tylko z plików nagłówkowych `algorithm` i `vector`.

Przykładowy program

```
int main() {
    std::vector<double> vector {1.2, -0.5, 0.3, 2.5, -1};
    partial_sum(vector.begin(), vector.end());
    std::for_each(vector.begin(), vector.end(),
        [](double element){std::cout << element << " "; });
    std::cout << std::endl; }
```

Wykonanie

```
Out: 1.2 0.7 1 3.5 2.5
```

3.2.11 Permutation: Losowa permutacja - grupowo

Napisz program `permutation`, który wczytuje ze standardowego wejścia nieujemną liczbę całkowitą i wypisuje na standardowe wyjście losową permutację mniejszych od niej nieujemnych liczb całkowitych. Program zawiera po jednym wywołaniu funkcji `std::iota` oraz `std::random_shuffle` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `cstdlib`, `ctime`, `algorithm`, `iostream`, `numeric` i `vector`.

Przykładowe wykonanie

```
 In: 5
Out: 2 0 4 1 3
```

3.2.12 Permutations: Wszystkie permutacje

Napisz program `permutations`, który wczytuje ze standardowego wejścia nieujemną liczbę całkowitą i wypisuje na standardowe wyjście wszystkie permutacje mniejszych od niej nieujemnych liczb całkowitych. Program zawiera po jednym wywołaniu funkcji `std::iota` oraz `std::next_permutation` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `cstdlib`, `ctime`, `algorithm`, `iostream`, `numeric` i `vector`.

Przykładowe wykonanie

```
 In: 3
Out: 0 1 2
Out: 0 2 1
Out: 1 0 2
Out: 1 2 0
Out: 2 0 1
Out: 2 1 0
```

3.2.13 Puzzle: Układanka piętnastka

Układanka *Piętnastka* składa się z planszy 4×4, na której znajduje się 15 kwadratowych klocków ponumerowanych od 1 do 15 oraz jedno puste miejsce. Celem gry jest uporządkowanie wymieszanych klocków poprzez przesuwanie ich po jednym na puste miejsce, przy czym przesunąć można tylko klocek sąsiadujący z tym miejscem. Napisz program `puzzle` symulujący taką grę. Po uruchomieniu program rozmieszcza klocki losowo, wyświetla początkowe ułożenie, a następnie wczytuje z klawiatury numer klocka. Jeśli to możliwe, przesuwa go na puste miejsce, drukuje planszę po przesunięciu, i tak dalej. Po uporządkowaniu klocków program automatycznie kończy wykonanie.

Końcówka przykładowej rozgrywki

```
Out: 01 02 03 04
Out: 05 06 07 08
Out: 09 10 11 12
Out: 13 14    15

 In: 15

Out: 01 02 03 04
Out: 05 06 07 08
Out: 09 10 11 12
Out: 13 14 15
```

3.2.14 Tictactoe: Kółko i krzyżyk

Grę kółko i krzyżyk na klasycznej planszy 3×3 zaczynają zawsze krzyżyki. Napisz program `tictactoe`, który przyjmuje jako argument wywołania znak użytkownika, `X` lub `O`, po czym wyświetla pustą planszę. Jeżeli wypada ruch użytkownika, program wczytuje współrzędne pola, na którym użytkownik stawia swój znak, na przykład `b3`. Jeżeli wypada ruch programu, wypisuje współrzędne pola, na którym sam stawia swój znak. Potem wyświetla odpowiednio zaktualizowaną planszę i tak dalej. Program sam wykrywa zakończenie gry i wypisuje informację o wyniku. Program nigdy nie przegrywa i wygrywa zawsze, kiedy to możliwe. Program załącza tylko pliki nagłówkowe `cstdlib`, `algorithm`, `iostream` i `vector`.

Początek przykładowego wykonania

```
  Linux: ./tictactoe X
Windows: tictactoe.exe X

Out:  abc
Out: 1
Out: 2
Out: 3

 In: a1

Out:  abc
Out: 1X
Out: 2
Out: 3

Out: c2

Out:  abc
Out: 1X
Out: 2  O
Out: 3
```

3.2.15 Triangles: Liczba trójkątów

Napisz program `triangles`, który czyta ze standardowego wejścia długości odcinków do napotkania końca pliku i wypisuje na standardowe wyjście liczbę trójkątów, które można z nich zbudować. Każdego odcinka można użyć w jednym trójkącie tylko raz, a dwa odcinki o jednakowych długościach uważamy za różne. Program zawiera jedno wywołanie funkcji `std::sort` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `algorithm`, `iostream` i `vector`.

Przykładowe wykonanie

```
 In: 2.5 1 3.7 3
Out: 3
```

3.2.16 Variation: Losowa wariacja

Napisz program `variation`, który wczytuje ze standardowego wejścia nieujemne liczby całkowite k oraz n i drukuje na standardowe wyjście k niepowtarzających się losowych, nieujemnych liczb całkowitych mniejszych od n. Program zawiera po jednym wywołaniu funkcji `std::iota` oraz `std::random_shuffle` i nie używa innych algorytmów biblioteki standardowej. Program załącza tylko pliki nagłówkowe `cstdlib`, `ctime`, `algorithm`, `iostream`, `numeric` i `vector`.

Przykładowe wykonanie

```
 In: 10 100
Out: 78 15 23 50 49 87 2 35 98 69
```

4 Pojemniki: 12, 13, 26, 27 czerwca
Pojemniki biblioteki standardowej