# Programowanie obiektowe

EiT 2011/2012

# Sprawy organizacyjne

- dr Wojciech Tylman, Katedra Mikroelektroniki i Technik Informatycznych PŁ
- B 18, lp., p. 56
- www.dmcs.p.lodz.pl
- tyl@dmcs.p.lodz.pl
- godziny przyjęć: WWW

# Tematyka

- Programowanie obiektowe na podstawie C++
- Inne języki zorientowane obiektowo - zarys:
  - Java
  - C#

# Literatura

- Jerzy Grębosz *Symfonia C++*
- Jerzy Grębosz *Pasja C++*
- Bjarne Stroustrup *Język C++*

# Organizacja zajęć

- Wykład: 15h (1h co tydzień)
- Laboratorium: 15h (2h co 2 tygodnie)

- Zaliczenie przedmiotu: zaliczenie wykładu i laboratorium, oceny brane z tą samą wagą

# Część I

## Wprowadzenie

# Kolejny język programowania...

- Rozwój języków – szybkość działania a funkcjonalność
- Związek z C
- Pozycja C++ wśród innych języków
  - co przedtem
  - co potem
  - czy nowe to lepsze?

# Dlaczego warto - w stosunku do starszych języków

- Język zorientowany obiektowo – wygodna i naturalna reprezentacja rzeczywistego świata
- Hermetyzacja
- Wielożywalność
- Ogranicza bałaganiarstwo przy programowaniu, ułatwia znajdowanie błędów
- Wygodniejszy podział prac w dużych projektach, łatwiejsze zmiany
- Łatwa migracja z C
- Wiele bibliotek pisanych w tym języku

# Dlaczego warto - w stosunku do nowszych języków

- Duża szybkość działania – język kompilowany w “klasyczny” sposób
- Dobra przenośność na poziomie kodu źródłowego
- Możliwość łatwej integracji z kodem w C
- Ogromna ilość narzędzi, literatury, baz wiedzy itp.

# Kiedy może być lepsza alternatywa

- Wymóg maksymalnej szybkości działania lub minimalnej objętości kodu wynikowego
- Przenośność na poziomie kodu wynikowego
- Specyficzne zastosowania: sztuczna inteligencja, analiza tekstów, bazy danych...
- Dalsze uproszczenie procesu programowania

# Historia

- Oparty na językach: C, Simula, Algol, Ada...
- Początek prac: 1979. Klasy, dziedziczenie, silne sprawdzanie typów
- Obecna nazwa: 1983. Również nowe możliwości, m. in. funkcje wirtualne
- Wersja 2.0: 1989. M. in. klasy abstrakcyjne i wielokrotne dziedziczenie
- Znormalizowany w 1998, kolejna wersja standardu w 2003, poprawki w 2005
- Norma ISO/IEC 14882

# Część 2

Klasy

# Po co klasy

- Klasa czyli typ
- Dlaczego warto? Klasa a świat rzeczywisty
- Co zawiera? Składniki klasy
- Powiązanie ze strukturami w C
- Typy wbudowane a typy definiowane przez użytkownika
- Typy prymitywne a klasy
- Klasa a obiekt
- Enkapsulacja

# Definicja klasy i tworzenie obiektów

```
class Circle
{
   public:
   // methods
   float Area();
   float Circumference();

   // fields
   float radius;
};
```

```
...
...
Circle smallCircle;
Circle bigCircle;
Circle circles[15];
Circle* circlePtr = &smallCircle;
...
...
```

# Dostęp do składowych

```
...
...
Circle smallCircle;
Circle bigCircle;
Circle* circlePtr = &smallCircle;

bigCircle.radius = 100;
circlePtr->radius = 5;

float bigCircleArea = bigCircle.Area();
float smallCircleArea = circlePtr->Area();

if (bigCircle.radius < smallCircle.radius)
{
    ...
    ...
}
```

**Ważne: działając na składowych klasy “od zewnątrz” trzeba podać obiekt na którym chcemy działać**

# Klasa od środka – implementacja metod

```
class Circle
{
   public:
   // methods
   float Area()
   {
         return 3.14159f * radius * radius;
   }
   float Circumference();

   // fields
   float radius;
};

float Circle::Circumference()
{
   return 2 * 3.14159f * radius;
}
```

**Ważne: działając na składowych klasy “od środka” domyślnie działamy na składowych obiektu w którym jesteśmy**

# Klasa od środka – wskaźnik `this`

```
class Circle
{
   public:
   // methods
   float Area()
   {
         return 3.14159f * this->radius * this->radius;
   }
   float Circumference();

   // fields
   float radius;
};

float Circle::Circumference()
{
   return 2 * 3.14159f * this->radius;
}
```

# Dygresja – co w którym pliku

- definicja **klasy**: plik .h lub .hpp
- definicja **funkcji składowych klasy**: w definicji klasy (czyli w pliku .h, .hpp) lub w pliku .cpp

# Dostęp do składowych klasy a enkapsulacja

- Po co ukrywać dane?
- Dostępne możliwości: public, protected, private
- Zalecany schemat postępowania
- Problemy

```
class Circle
{
   public:
   // methods
   float Area();
   float Circumference();

   private:
   // fields
   float radius;
};
```

# Konstruktor

```
Circle myCircle;
float area = myCircle.Area(); //???
```

- Funkcja wywoływana przy tworzeniu obiektu
- Nazwa jak nazwa klasy
- Nie ma typu zwracanego
- Może mieć argumenty
- Jeśli klasa nie ma **żadnego** konstruktora, zostanie wygenerowany automatycznie konstruktor bez argumentów

# Konstruktor 2

```
class Circle
{
   public:
   // constructors
   Circle();
   Circle(float radius_);
   // methods
   float Area();
   float Circumference();

   private:
   // fields
   float radius;
};
Circle::Circle()
{
   radius = 0;
}
Circle::Circle(float radius_)
{
   radius = radius_;
}
```

# Konstruktor – lista inicjalizacyjna

```
Circle::Circle():
radius(0)
{
}
Circle::Circle(float radius_):
radius(radius_)
{
}
```

# Dygresja: domyślna wartość argumentu

```
class Circle
{
   public:
   // constructors
   Circle(float radius_ = 0);
   // methods
   float Area();
   float Circumference();

   private:
   // fields
   float radius;
};

Circle::Circle(float radius_ /*= 0 */):
radius(radius_)
{
}
```

# Kiedy niezbędna jest lista inicjalizacyjna?

```
class Circle
{
public:
    Circle(float radius_);
    float Area();
    float Circumference();
private:
    float radius;
};

class TwoCircles
{
public:
    TwoCircles(float radius1_, float radius2_):
    circle1(radius1_),
    circle2(radius2_)
    {
    }
private:
    Circle circle1;
    Circle circle2;
};
```

# Program 1

main.cpp

```
#include <iostream>
#include "circle.hpp"

using namespace std;

int main()
{
    Circle first;
    Circle second(5);
    Circle* circlePtr;

    cout << first.Area() << endl;
    cout << second.Area() << endl;
    //cout << circlePtr->Area();
    circlePtr = &second;
    cout << circlePtr->Area() << endl;

    return 0;
}
```

circle.hpp

```
class Circle
{
public:
    Circle(float radius_ = 0);
    float Area();
    float Circumference();
private:
    float radius;
};
```

circle.cpp

```
#include "circle.hpp"

Circle::Circle(float radius_ /*= 0*/):
radius(radius_)
{
}
float Circle::Area()
{
    return 3.14159f * radius * radius;
}

float Circle::Circumference()
{
    return 2 * 3.14159f * radius;
}
```

# Alternatywna implementacja

```
class Circle
{
   public:
   Circle(float radius_ = 0);
   float Area()
   {
         return area;
   }
   float Circumference()
   {
         return circumference;
   }

   private:
   float area;
   float circumference;
};

Circle::Circle(float radius_ /* = 0*/):
area(3.14159f * radius_ * radius_),
circumference(2 * 3.14159f * radius_)
{
}
```

# Dynamiczna alokacja pamięci

- Obiekty automatyczne – alokacja ze stosu. Za obsługę pamięci odpowiada kompilator.
- Alternatywa: alokacja ze sterty. Za obsługę pamięci odpowiedzialny jest programista.
- W języku C dostępne są funkcje `malloc`, `free` i inne
- C++ rozszerza możliwości alokacji dynamicznej dzięki operatorom `new` i `delete`
- Zalety: bezpieczeństwo typów, automatyczne wywołanie konstruktora i destruktora.

# New i delete

```
Circle* circlePtr = new Circle(20);
if (circlePtr != NULL)
{
   cout << circlePtr->Area();
   delete circlePtr;
}
//cout << circlePtr->Circumference();
//delete circlePtr;
circlePtr = NULL;

int* array = new int[100];
for (int i = 0; i < 100; i++)
{
   array[i] = i;
}
//array = new int[50];
delete[] array;

circlePtr = new Circle(30);
Circle* secondCirclePtr = circlePtr;
delete secondCirclePtr;
//delete circlePtr;
```

# Destruktor

- Umożliwia “sprzątanie” po zniszczeniu obiektu
- Nie ma typu zwracanego ani argumentów
- Nazwa – jak klasy, poprzedzona znakiem ~
- Nie musi występować
- Szczególnie przydatny przy dynamicznej alokacji pamięci lub w przypadku użycia ograniczonych zasobów systemowych (np. timery, uchwyty plików, okien itp.)

# Destruktor i dynamiczna alokacja pamięci

```
class Array
{
    public:
    Array(int size_):
    size(size_),
    arrayPtr(new int[size])
    {
    }
    ~Array(){
        delete[] arrayPtr;
    }
    int Get(int index){
        return arrayPtr[index];
    }
    void Set(int index, int value){
        arrayPtr[index] = value;
    }
    int GetSize(){
        return size;
    }
    private:
    int* arrayPtr;
    int size;
};
```

# Przekazywanie argumentów do funkcji, referencje, atrybut const

- Do funkcji można przekazywać argumenty przez:
  - wartość
  - wskaźnik
    - jawnie
    - **jako referencję**
- Przekazywanie przez wskaźniki jest szybsze w przypadku dużych obiektów, ale może być:
  - niewygodne
  - niebezpieczne

# Przekazywanie przez wartość a przekazywanie przez wskaźnik

```
double Average(Array ar){
    double av = 0;
    for (int i = 0; i < ar.GetSize(); i++){
          av += ar.Get(i);
    }
    av /= ar.GetSize();
    return av;
}
double Average(Array* arPtr){
    double av = 0;
    for (int i = 0; i < arPtr->GetSize(); i++){
          av += arPtr->Get(i);
    }
    av /= arPtr->GetSize();
    return av;
}
double Average(Array& ar){
    double av = 0;
    for (int i = 0; i < ar.GetSize(); i++){
          av += ar.Get(i);
    }
    av /= ar.GetSize();
    return av;
}
```

# Jak uniemożliwić modyfikację argumentu w funkcji?

```
double Average(const Array ar){
    double av = 0;
    for (int i = 0; i < ar.GetSize(); i++){
          av += ar.Get(i);
    }
    av /= ar.GetSize();
    return av;
}
double Average(const Array* arPtr){
    double av = 0;
    for (int i = 0; i < arPtr->GetSize(); i++){
          av += arPtr->Get(i);
    }
    av /= arPtr->GetSize();
    return av;
}
double Average(const Array& ar){
    double av = 0;
    for (int i = 0; i < ar.GetSize(); i++){
          av += ar.Get(i);
    }
    av /= ar.GetSize();
    return av;
}
```

# Czy nasza klasa jest na to przygotowana?

```
class Array
{
    public:
    Array(int size_):
    size(size_),
    arrayPtr(new int[size])
    {
    }
    ~Array(){
        delete[] arrayPtr;
    }
    int Get(int index){
        return arrayPtr[index];
    }
    void Set(int index, int value){
        arrayPtr[index] = value;
    }
    int GetSize(){
        return size;
    }
    private:
    int* arrayPtr;
    int size;
};
```

# Metody const

```
class Array
{
    public:
    Array(int size_):
    size(size_),
    arrayPtr(new int[size])
    {
    }
    ~Array(){
        delete[] arrayPtr;
    }
    int Get(int index) const {
        return arrayPtr[index];
    }
    void Set(int index, int value){
        arrayPtr[index] = value;
    }
    int GetSize() const {
        return size;
    }
    private:
    int* arrayPtr;
    int size;
};
```

# Pola const

```
class Array
{
    public:
    Array(int size_):
    size(size_),
    arrayPtr(new int[size])
    {
    }
    ~Array(){
        delete[] arrayPtr;
    }
    int Get(int index) const {
        return arrayPtr[index];
    }
    void Set(int index, int value){
        arrayPtr[index] = value;
    }
    int GetSize() const {
        return size;
    }
    private:
    int* const arrayPtr;
    const int size;
};
```

# Inne zastosowanie referencji

```
...
...
Circle smallCircle;
Circle bigCircle;
Circle circles[15];
Circle* circlePtr = &smallCircle;
Circle& circleRef = smallCircle;
cout << circlePtr->Area();
cout << circleRef.Area();
...
...
```

# Konstruktor kopiujący

- Jako argument przyjmuje referencję do obiektu tej samej klasy, referencja powinna być `const`
- Jest konieczny w przypadku gdy tworzenie nowego obiektu na podstawie istniejącego wymaga czegoś więcej niż kopiowanie “składnik po składniku” (nie mylić z “bit po bicie”)
- Jeśli programista go nie utworzy, kompilator wygeneruje automatycznie konstruktor kopiujący działający na zasadzie “składnik po składniku”
- Jest wywoływany zawsze przy tworzeniu nowego obiektu na podstawie istniejącego (np. podczas przekazywania lub zwracania do/z funkcji przez wartość)

# Kiedy kompilator nie wygeneruje konstruktora kopiującego?

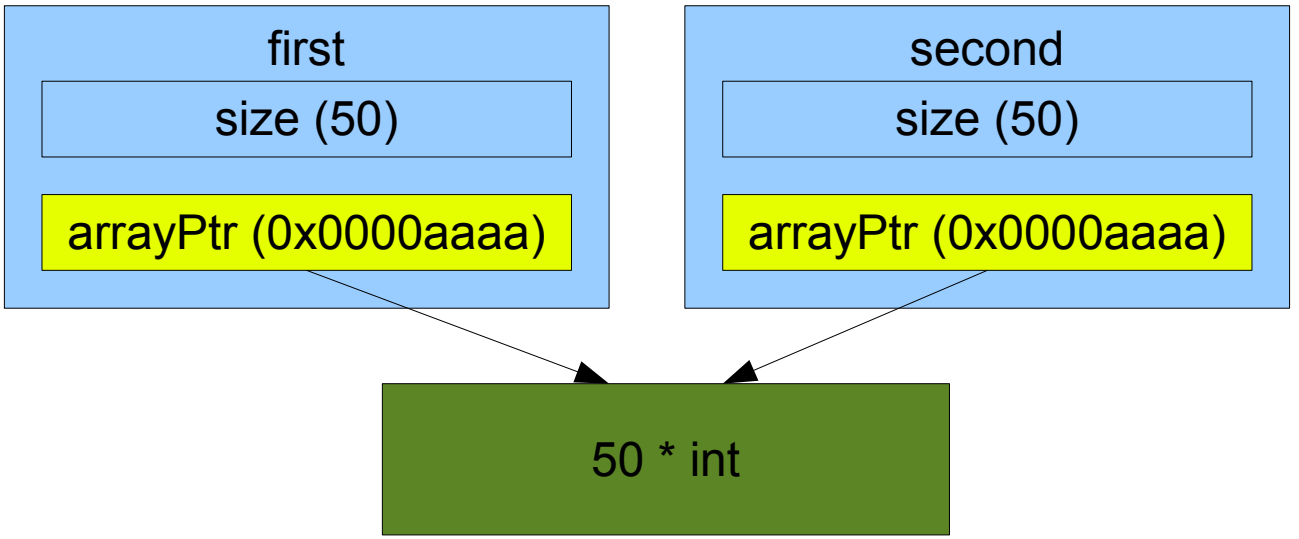
- Składnik klasy z modyfikatorem `const`
- Składnik klasy jest referencją
- Składnik lub klasa bazowa mają prywatny konstruktor kopiujący
- Składnik lub klasa bazowa nie mają konstruktora kopiującego

- Jak zabezpieczyć klasę przed kopiowaniem?

# Konstruktor kopiujący – konieczność użycia

```
Array first(50);
Array second = first;

class Array
{
    .
    .
    .
   private:
   int* arrayPtr;
   int size;
};
```



# Konstruktor kopiujący – przykładowa implementacja

```
class Array
{
    public:
    Array(int size_):
    size(size_),
    arrayPtr(new int[size]){
    }
    Array(const Array& right):
    size(right.size),
    arrayPtr(new int[size]){
        for (int i = 0; i < size; i++){
            arrayPtr[i] = right.arrayPtr[i];
        }
    }
    ~Array(){
        delete[] arrayPtr;
    }
    .
    .
    .
    private:
    int* arrayPtr;
    int size;
};
```

# Operatory

- Większość operatorów w C++ może zostać przeciążona
- Przeciążenie operatora polega na napisaniu własnej funkcji o postaci
  `return_type operator`*`symbol`*`(arguments){ /*...*/}`
- Nie można przeładować operatorów: `. .* :: ?:`
- Nie można zdefiniować nowych operatorów
- Nie można zmienić priorytetu, łączności i argumentowości
- Przynajmniej jeden argument musi być zdefiniowany przez użytkownika
- Mogą być funkcją składową lub globalne (z wyjątkami)

# Operator przypisania

- Funkcja podobna do konstruktora kopiującego, ale wywoływany w innych przypadkach (przypisanie do już istniejącego obiektu)
- Implementacja często zbliżona do konstruktora kopiującego, ale zwykle najpierw trzeba obiekt wyczyścić
- Jest zawsze funkcją składową klasy
- Czasem konieczne jest zabezpieczenie przed przypisaniem obiektu do samego siebie

# Przykładowy operator=

```
class Array
{
    .
    .
    Array(const Array& right):
    size(right.size),
    arrayPtr(new int[size]){
        for (int i = 0; i < size; i++){
            arrayPtr[i] = right.arrayPtr[i];
        }
    }
    const Array& operator=(const Array& right)
    {
        delete[] arrayPtr;
        size = right.size;
        arrayPtr  = new int[size];
        for (int i = 0; i < size; i++){
            arrayPtr[i] = right.arrayPtr[i];
        }
        return *this;
    }
    .
    .
};
```

# Zabezpieczenie przed przypisaniem samemu sobie

```
Array arr1;
arr1.Set(1, 10);
arr1 = arr1; /???
```

```
const Array& operator=(const Array& right)
{
    if (this != &right)
    {
        delete[] arrayPtr;
        size = right.size;
        arrayPtr  = new int[size];
        for (int i = 0; i < size; i++){
            arrayPtr[i] = right.arrayPtr[i];
        }
    }
    return *this;
}
```

# Operator indeksowania

- Musi być funkcją składową klasy
- Ma zawsze jeden argument
- Zazwyczaj stosowany do indeksowania tablic

# Przykładowy operator indeksowania

```
class Array
{
   public:
   .
   .
   int Get(int index) const {
      return arrayPtr[index];
   }
   void Set(int index, int value){
      arrayPtr[index] = value;
   }
   int& operator[](int index)
   {
      return arrayPtr[index];
   }
   int operator[](int index) const
   {
      return arrayPtr[index];
   }
   .
   .
};
```

# Operator wywołania funkcji

- Może mieć dowolną liczbę parametrów
- Musi być funkcją składową

# Operatory + - * / >> << itp.

- Operatory dwuargumentowe
- Mogą być funkcjami składowymi lub funkcjami globalnymi

# Operator + jako funkcja składowa

```
class Array
{
    public:
    .
    .
    Array operator+(const Array& right) const
    {
        if (size == right.size)
        {
            Array result(*this);
            for (int i = 0; i < size; i++)
            {
                result[i] += right[i];
            }
            return result;
        }
        return *this;
    }
};
```

```
Array arr1, arr2;
.
.
Array arr3 = arr1 + arr2;
```

# Operator + jako funkcja globalna

- Zadanie: napisać operator + dodający do wszystkich elementów tablicy w klasie `Array` podaną wartość typu int

```
Array arr1;
.
.
Array arr2 = arr1 + 12;
Array arr3 = 18 + arr1;
```

# Operator + jako funkcja globalna

```
Array operator+(const Array& left, int right)
{
    Array result(left);
    for (int i = 0; i < left.size; i++)
    {
        result[i] += right;
    }
    return result;
}

Array operator+(int left, const Array& right)
{
    return right + left;
}
```

# Funkcje zaprzyjaźnione

- Umożliwiają dostęp do prywatnych składników klasy
- Deklarację umożliwienia dostępu **musi** zawierać klasa do której dostęp jest umożliwiany
- Można umożliwić dostęp funkcji globalnej, funkcji składowej klasy lub całej klasie

# Przykład przyjaźni

```
class Array
{
    friend Array operator+(const Array& left, int right);
    friend class Circle;
    .
    .
};
```

# Konwersje

- Standardowe
- Zdefiniowane przez użytkownika

- Niejawne – np. przy wywołaniu funkcji, przy zwracaniu wartości przez funkcję, przy obliczaniu wyrażeń (w tym warunkowych) itp. Wykorzystywane jeśli nie jest możliwe idealne dopasowanie typów
- Jawne – wedle uznania programisty

# Konwersje standardowe niejawne

- Całkowite i zmiennoprzecinkowe rozszerzenie
- Konwersje liczb całkowitych z liczby bez znaku na liczbę ze znakiem i odwrotnie
- Całkowite i zmiennoprzecinkowe zawężenie
- Konwersje liczb całkowitych na zmiennoprzecinkowe i odwrotnie
- Konwersje dla arytmetycznych operatorów dwuargumentowych
- Konwersje wskaźników
- Konwersje związane z dziedziczeniem

# Konwersje wskaźników

- Zero całkowite na wskaźnik NULL
- Wskaźnik do obiektu dowolnego typu na wskaźnik do typu void (z wyjątkiem obiektów const i volatile)
- Tablica na wskaźnik do jej pierwszego elementu

# Konwersje jawne

- Wywoływane przy pomocy składni rzutowania lub wywołania funkcji (równoznaczne dla konwersji między typami wbudowanymi)
- Należy traktować jako ostateczność, gdyż wyłączają wbudowane sprawdzanie typów
- W nowszych implementacjach występują konwersje oparte o sprawdzanie typu w momencie wykonania (RTTI)

# Przykłady konwersji jawnych

- Z dowolnej liczby lub wskaźnika na liczbę całkowitą
- Z dowolnej liczby na liczbę zmiennoprzecinkową
- Z liczby całkowitej lub wskaźnika na wskaźnik do innego typu (w tym void)
- Z dowolnego typu na void

# Konwersje użytkownika

- Umożliwiają zdefiniowanie konwersji, która może być wywołana niejawnie lub jawnie, pod warunkiem że typ z którego lub do którego przebiega konwersji nie jest typem wbudowanym
- Za pomocą konstruktora – implementacja w klasie na którą konwertujemy
- Za pomocą funkcji konwertującej – implementacja w klasie z której konwertujemy

# Konstruktor konwertujący

```
float CircleToSquare(const Circle& c)
{
   return sqrt(c.Area());
}

float a = CircleToSquare(5);
```

# Jeśli nie chcemy aby konstruktor jednoargumentowy konwertował

```
class Array
{
    public:
    explicit Array(int size_):
    size(size_),
    arrayPtr(new int[size]){
    }
    .
    .
};
```

# Funkcja konwertująca

- Funkcja składowa klasy
- Jest rodzajem operatora
- Nazwa taka jak typ na który następuje konwersja
- Nie ma typu zwracanego
- Nie ma argumentów

# Przykładowa funkcja konwertująca

```
class Circle
{
public:
    Circle(float radius_);
    float Area() const;
    float Circumference();
    operator float() const;
private:
    float radius;
};

Circle::operator float() const
{
    return radius;
}

Circle second(5);
cout << second << endl;
```

# Konwersje użytkownika w konwersjach niejawnych

- Kompilator może wykorzystać konwersje użytkownika jeśli bez nich nie jest w stanie dopasować typów i znajdzie jednoznaczny sposób konwersji, z uwzględnieniem pewnych reguł
- Jeśli występuje niejednoznaczność, zostanie zgłoszony błąd, jednak tylko jeśli w programie pojawi się próba konwersji uwidaczniająca tę niejednoznaczność

# Zasady wykorzystania konwersji użytkownika

- Tylko jedna konwersja w łańcuchu może być konwersją użytkownika
- Dwa łańcuchy konwersji z których oba wykorzystują konwersje użytkownika, lecz tylko jeden oprócz tego konwersje standardowe, są traktowane jako równie dobre
- Konwersje za pomocą konstruktora i funkcji konwertującej są traktowane jako równie dobre

# Ilustracja problemu niejednoznaczności

```
class Circle
{
public:
    .
    .
    operator TwoCircles() const;
};

Circle::operator TwoCircles() const
{
    return TwoCircles(radius, 0);
}

class TwoCircles
{
public:
    TwoCircles(const Circle& right):
    circle1(right),
    circle2(0)
    {
    }
.
.
};
```

# Ilustracja problemu niejednoznaczności - cd

```
void DummyFunction(const TwoCircles& t)
{
}

Circle first(0);
Circle second(5);
TwoCircles tc(second);
DummyFunction(second);
```

# Część 3

Dziedziczenie

# Po co?

- Analogia do świata rzeczywistego
- Możliwość opierania typów bardziej złożonych na prostszych
- Wielożywalność kodu
- Możliwość traktowania obiektów typów pochodnych jak obiektu typu podstawowego
- Możliwość wywołania właściwej funkcji w zależności od typu obiektu, mimo braku znajomości tego typu w momencie kompilacji (późne wiązanie)

# Zasady

- Klasa pochodna uzyskuje dostęp do składników public i protected klasy podstawowej
- Klasa pochodna może określić jakie ograniczenia dostępu będą miały te składniki
- Klasa pochodna może zdefiniować nowe składniki
- Klasa pochodna może ponownie zdefiniować składniki istniejące już w klasie podstawowej
- Niektóre z funkcji składowych klasy podstawowej nie są dziedziczone: konstruktory, destruktor, operator przypisania

# Przykład dziedziczenia

```
class ColorfulCircle : public Circle
{
public:
    ColorfulCircle(float radius_ = 0, unsigned long color_ = 0x00000000):
    Circle(radius_),
    color(color_)
    {
    }
    unsigned long GetColor()
    {
        return color;
    }
private:
    unsigned long color;
};


ColorfulCircle cc(12, 0x0000FF00);
cout << cc.Area() << ' ' << cc.GetColor() << endl;
```

# Konwersje przy dziedziczeniu

- Wskaźnik lub referencja do klasy pochodnej może być niejawnie przekształcony do publicznie odziedziczonej klasy bazowej
- W drugą stronę – tylko jawnie, jest to niebezpieczne

```
NextCircle nc;
circlePtr = &nc;
NextCircle* nextCirclePtr = (NextCircle*)circlePtr;
```

# Inny przykład - wstęp

```
class Array
{
    public:
    .
    .
    Array(int size_ = 1):
    size(size_),
    arrayPtr(new int[size])
    {
        for (int i = 0; i < size; i++)
        {
            arrayPtr[i] = 0;
        }
    }
    .
    .
};

double Average(const Array ar);
double Average(const Array* arPtr);
```

# Inny przykład

```
class AdvancedArray : public Array
{
public:
    AdvancedArray(int size):Array(size), good(true){}
    int& operator[](int index)    {
        if (good && index < GetSize()){
            return Array::operator [](index);
        }
        good = false;
        return dummy;
    }
    int operator[](int index) const{
        if (good && index < GetSize()){
            return Array::operator [](index);
        }
        good = false;
        return 0;
    }
    bool GetGood(){
        if (good)  {
            return true;
        }
        good = true;
        return false;
    }
private:
    bool good;
    int dummy;
};
```

# Inny przykład - cd

```
AdvancedArray aar0;
AdvancedArray aar1(10);
AdvancedArray aar2(aar1);
AdvancedArray aar3(10);
Average(aar1);
Average(&aar1);
aar1[2] = 4;
cout << aar1.Get(2);
aar3 = aar1;
aar3 = aar1 + aar2;
```

# Dygresja – mutable, operator++ dla typu bool

```
class AdvancedArray : public Array
{
public:
    AdvancedArray(int size):Array(size), good(true){}
    int& operator[](int index)    {
        if (good && index < GetSize()){
            return Array::operator [](index);
        }
        good = false;
        return dummy;
    }
    int operator[](int index) const{
        if (good && index < GetSize()){
            return Array::operator [](index);
        }
        good = false;
        return 0;
    }
    bool GetGood(){
        return good++;
    }
private:
    mutable bool good;
    int dummy;
};
```

# Kolejność konstruowania i niszczenia obiektów

- Najpierw konstruowany jest obiekt klasy bazowej
- Następnie obiekty zawarte w klasie (składniki)
- Na końcu sam obiekt danej klasy

- Składniki konstruowane są w kolejności wystąpienia w definicji klasy

- Destrukcja wręcz przeciwnie

# Zły przykład z kolejnością konstrukcji

```
class Test
{
public:
    Test():
    c(0),
    cc(0, 0)
    {}
    ColorfulCircle cc;
    Circle c;
};
```

# Konstruktory i operator przypisania przy dziedziczeniu

- W konstruktorach i operatorze przypisania możemy posłużyć się ich odpowiednikami w klasie bazowej

```
AdvancedArray(int size):
Array(size),
good(true)
{}
AdvancedArray(const AdvancedArray& right):
Array(right)
{
}
const AdvancedArray& operator=(const AdvancedArray& right)
{
    (*this).Array::operator =(right);
    return *this;
}
```

# Inny przykład

```
typedef unsigned int uint;
class Vehicle
{
public:
    Vehicle(uint maxSpeed):
    maxSpeed(maxSpeed){
    }
    void SetSpeed(uint newSpeed){
        if ((speed = newSpeed) > maxSpeed){
            speed = maxSpeed;
        }
    }
    uint GetSpeed() const{
        return speed;
    }
    uint GetMass()const{
        return 0;
    }
protected:
private:
    uint speed;
    const uint maxSpeed;
};
```

# Inny przykład cd

```
class Bike : public Vehicle
{
public:
    Bike():
    Vehicle(40){
    }
    uint GetMass() const{
        return 15;
    }
};

class Car : public Vehicle
{
public:
    Car(uint mass, uint maxSpeed):
    Vehicle(maxSpeed),
    mass(mass){
    }
    uint GetMass() const{
        return mass;
    }
private:
    uint mass;
};
```

```
class Truck : public Car
{
public:
    Truck(uint mass, uint maxSpeed):
    Car(mass, maxSpeed),
    load(0){
    }
    void SetLoad(uint newLoad){
        load = newLoad;
    }
    uint GetMass() const
    {
        return Car::GetMass() + load;
    }
private:
    uint load;
};
```

# Inny przykład cd

```
Vehicle v1(10); Bike b1;
Car c1(1000, 150), c2(1200, 220);
Truck t1(8000, 80), t2(5000, 120);
t1.SetLoad(8000); v1.SetSpeed(5); b1.SetSpeed(40); c1.SetSpeed(200);
t1.SetSpeed(20);
cout << v1.GetMass() << ' ' << b1.GetMass() << ' ' <<
   c1.GetMass() << ' ' << t1.GetMass() << endl;
cout << v1.GetSpeed() << ' ' << b1.GetSpeed() << ' ' <<
   c1.GetSpeed() << ' ' << t1.GetSpeed() << endl;
```

0 15 1000 16000
5 40 150 20

```
Vehicle* vPtr = &v1;
Bike* bPtr = &b1;
Car* cPtr = &c1;
Truck* tPtr = &t1;
cout << vPtr->GetMass() << ' ' << bPtr->GetMass() << ' ' <<
   cPtr->GetMass() << ' ' << tPtr->GetMass() << endl;
cout << vPtr->GetSpeed() << ' ' << bPtr->GetSpeed() << ' ' <<
   cPtr->GetSpeed() << ' ' << tPtr->GetSpeed() << endl;
```

0 15 1000 16000
5 40 150 20

```
vPtr = &b1;
cout << vPtr->GetMass() << endl << vPtr->GetSpeed() << endl;
cPtr = &t1;
cout << cPtr->GetMass() << endl << cPtr->GetSpeed() << endl;
Car& cRef = t1;
cout << cRef.GetMass() << endl << cRef.GetSpeed() << endl;
```

0
40
8000
20
8000
20

# Kiedy to może się przydać...

```
int CompareMass(const Vehicle& v1, const Vehicle& v2){
    if (v1.GetMass() > v2.GetMass()){
        return 1;
    }
    if (v1.GetMass() < v2.GetMass()){
        return -1;
    }
    return 0;
}
```

```
Vehicle* vPtrArr[10];
vPtrArr[0] = &b1;
vPtrArr[1] = &c1;
for (int i = 2; i < 10; i++){
    vPtrArr[i] = new Truck(1000 * i, 150 - 10 * i);
}
for (int i = 0; i < 10; i++){
    cout << vPtrArr[i]->GetMass() << ' ';
}
```

# Jak to zrobić

```
typedef unsigned int uint;
class Vehicle
{
public:
    Vehicle(uint maxSpeed):
    maxSpeed(maxSpeed){
    }
    void SetSpeed(uint newSpeed){
        if ((speed = newSpeed) > maxSpeed){
            speed = maxSpeed;
        }
    }
    uint GetSpeed() const{
        return speed;
    }
    virtual uint GetMass()const{
        return 0;
    }
protected:
private:
    uint speed;
    const uint maxSpeed;
};
```

# Jaki wynik

```
Vehicle v1(10); Bike b1;
Car c1(1000, 150), c2(1200, 220);
Truck t1(8000, 80), t2(5000, 120);
t1.SetLoad(8000); v1.SetSpeed(5); b1.SetSpeed(40); c1.SetSpeed(200);
t1.SetSpeed(20);
cout << v1.GetMass() << ' ' << b1.GetMass() << ' ' <<
   c1.GetMass() << ' ' << t1.GetMass() << endl;
cout << v1.GetSpeed() << ' ' << b1.GetSpeed() << ' ' <<
   c1.GetSpeed() << ' ' << t1.GetSpeed() << endl;
```

0 15 1000 16000
5 40 150 20

```
Vehicle* vPtr = &v1;
Bike* bPtr = &b1;
Car* cPtr = &c1;
Truck* tPtr = &t1;
cout << vPtr->GetMass() << ' ' << bPtr->GetMass() << ' ' <<
   cPtr->GetMass() << ' ' << tPtr->GetMass() << endl;
cout << vPtr->GetSpeed() << ' ' << bPtr->GetSpeed() << ' ' <<
   cPtr->GetSpeed() << ' ' << tPtr->GetSpeed() << endl;
```

0 15 1000 16000
5 40 150 20

```
vPtr = &b1;
cout << vPtr->GetMass() << endl << vPtr->GetSpeed() << endl;
cPtr = &t1;
cout << cPtr->GetMass() << endl << cPtr->GetSpeed() << endl;
Car& cRef = t1;
cout << cRef.GetMass() << endl << cRef.GetSpeed() << endl;
```

15
40
16000
20
16000
20

# Wirtualne destruktory

```
class Truck : public Car
{
public:
    Truck(uint mass, uint maxSpeed, uint load = 0, void* loadPtr = NULL):
    Car(mass, maxSpeed),
    load(load),
    loadPtr(loadPtr){
    }
    void SetLoad(uint newLoad, void* newLoadPtr = NULL){
        load = newLoad;
        delete loadPtr;
        loadPtr = newLoadPtr;
    }
    uint GetMass() const{
        return Car::GetMass() + load;
    }
    ~Truck(){
        delete loadPtr;
    }
private:
    uint load;
    void* loadPtr;

};
```

# Wirtualne destruktory cd

```
Circle* circlePtr1 = new Circle();
Circle* circlePtr2 = new Circle();
t1.SetLoad(1000, circlePtr1);
Vehicle* vPtr = new Truck(3000, 120, 2000, circlePtr2);
delete vPtr;
```

Jakie destruktory zostaną wywołane?

# Wirtualne destruktory cd

```
class Vehicle
{
public:
    .
    .
    virtual ~Vehicle(){
    }
    .
    .
};
```

# Funkcje czysto wirtualne i klasy abstrakcyjne

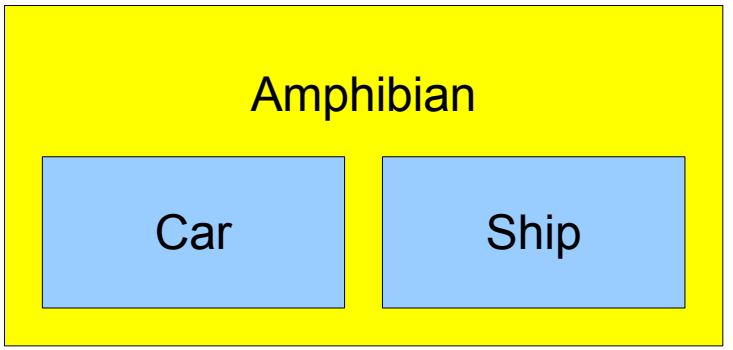
```
class Vehicle
{
public:
    Vehicle(uint maxSpeed):
    maxSpeed(maxSpeed){
    }
    void SetSpeed(uint newSpeed){
        if ((speed = newSpeed) > maxSpeed){
            speed = maxSpeed;
        }
    }
    uint GetSpeed() const{
        return speed;
    }
    virtual uint GetMass()const = 0;
protected:
private:
    uint speed;
    const uint maxSpeed;
};
```

# Dziedziczenie wielokrotne

```
class Ship
{
public:
    Ship(float mass, uint length):
    mass(mass),
    length(length),
    anchorUp(false){
    }
    virtual float GetMass() const{
        return mass;
    }
    virtual uint GetLength() const{
        return length;
    }
    void Anchor(bool up){
        anchorUp = up;
    }
private:
    float mass;
    uint length;
    bool anchorUp;
};
```

# Dziedziczenie wielokrotne - cd

```
class Amphibian : public Car, public Ship
{
public:
    Amphibian(uint mass, uint maxSpeed, uint length):
    Car(mass, maxSpeed),
    Ship(0, length){
    }
};
```



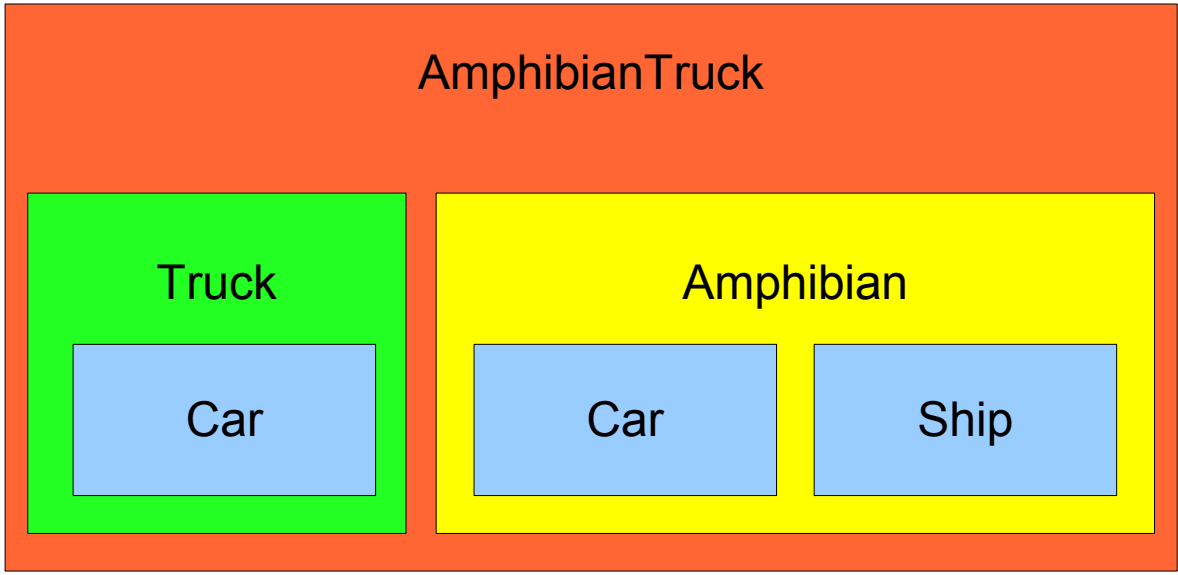

```
Amphibian am(5000, 90, 6);
am.GetMass();
am.Car::GetMass();

class Amphibian : public Car, public Ship
{
public:
    .
    .
    virtual uint GetAmphibianMass() const{
        return Car::GetMass();
};
```

# Dziedziczenie wielokrotne - cd

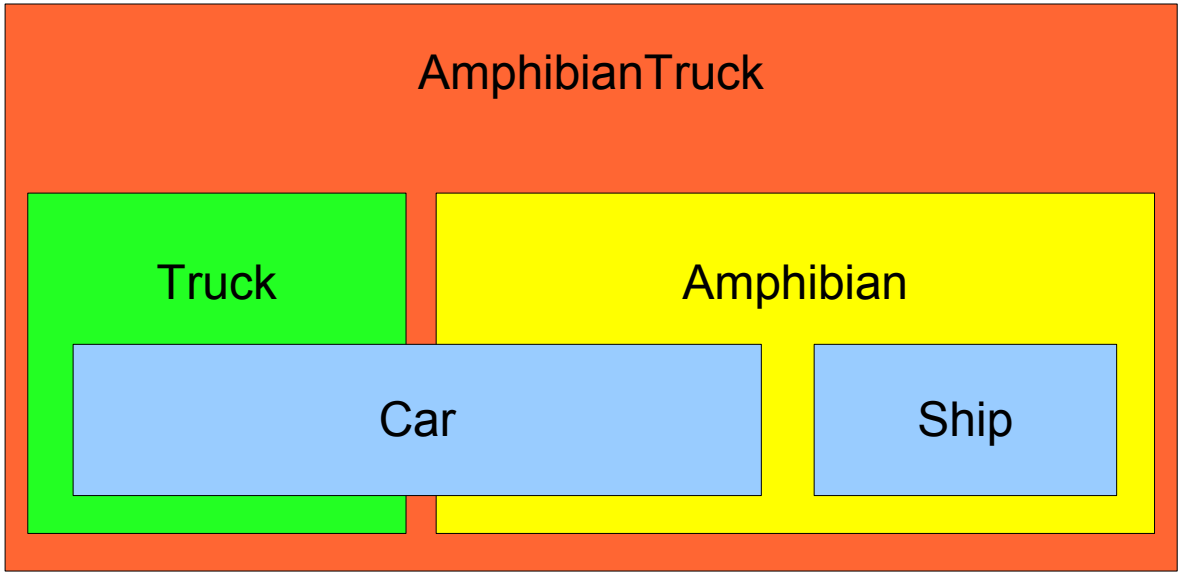
```
class AmphibianTruck : public Amphibian, public Truck
{
public:
    AmphibianTruck(uint mass, uint maxSpeed, uint length, uint load = 0,
        void* loadPtr = NULL):
    Amphibian(0, 0, length),
    Truck(mass, maxSpeed, load, loadPtr){
    }
    uint TestMass()
    {
        return Car::GetMass();
    }
};

AmphibianTruck amt(8000, 60, 8);
amt.TestMass();
```



# Wirtualne klasy podstawowe

```
class AmphibianTruck : public Amphibian, public Truck
{
public:
    AmphibianTruck(uint mass, uint maxSpeed, uint length, uint load = 0,
        void* loadPtr = NULL):
    Amphibian(mass, 0, length),
    Truck(mass, maxSpeed, load, loadPtr),
    Car(mass, maxSpeed){
    }
    uint TestMass()
    {
        return Car::GetMass();
    }
};
```



# Dygresja: składowe statyczne

- Pola są wspólne dla wszystkich obiektów danej klasy
- Są dostępne nawet gdy nie istnieje żaden obiekt klasy
- Można się do nich odnosić jak do innych składników albo operatorem ::
- Funkcja statyczna może odwoływać się tylko do składowych statycznych

# Przykład składowych statycznych

```
class Vehicle
{
public:
    Vehicle(uint maxSpeed):
    maxSpeed(maxSpeed){
        ++counter;
    }
    .
    .
    .
    virtual ~Vehicle(){
        --counter;
    }
    static uint GetCount(){
        return counter;
    }
protected:
private:
    uint speed;
    const uint maxSpeed;
    static uint counter;
};

uint Vehicle::counter = 0;
```

# Typy wyliczeniowe

- Umożliwiają utworzenie zamkniętej listy wartości
- Każda wartość jest nazwana
- Wartości są typu int
- Można jawnie wyspecyfikować jaka wartość odpowiada danej nazwie; domyślnie pierwsza to 0, każda następna większa o 1
- Można niejawnie konwertować do int
- Polepszają czytelność programu i zwiększają szansę wykrycia błędu

# Przykład typów wyliczeniowych

```
enum propulsion
{
    wind,
    sail = 0,
    steam,
    diesel
};
```

```
class Ship
{
public:
    Ship(float mass, uint length,
        propulsion prop = diesel):
    mass(mass),
    length(length),
    anchorUp(false),
    prop(prop){
    }
    .
    .
private:
    float mass;
    uint length;
    bool anchorUp;
    propulsion prop;
};

Ship sailShip(150, 35, sail);
```

# Alternatywne rozwiązanie

```
class Ship
{
public:
    enum propulsion
    {
        wind,
        sail = 0,
        steam,
        diesel
    };
    Ship(float mass, uint length, propulsion prop = diesel):
    mass(mass),
    length(length),
    anchorUp(false),
    prop(prop){
    }
    .
    .
private:
    float mass;
    uint length;
    bool anchorUp;
    propulsion prop;
};

Ship sailShip(150, 35, Ship::sail);
```

# Część 4

Wyjątki

# Zalety i wady wyjątków

- Zalety:
  - Wygodniejsze podejście niż w klasycznych metodach (np. przy użyciu wartości zwracanej): przejrzysty zapis, możliwość wygodnego przekazywania szczegółowych informacji, wykorzystanie hierarchii klas, obsługa błędów w konstruktorach itp.
  - Większa “presja” na prawidłową obsługę błędów
- Wady:
  - Możliwość wystąpienia nieprawidłowości w pewnych przypadkach
  - Większe wykorzystanie zasobów

# Przykład bez wyjątków

```
class AdvancedArray : public Array
{
public:
    AdvancedArray(int size):Array(size), good(true){}
    int& operator[](int index)    {
        if (good && index < GetSize()){
            return Array::operator [](index);
        }
        good = false;
        return dummy;
    }
    int operator[](int index) const{
        if (good && index < GetSize()){
            return Array::operator [](index);
        }
        good = false;
        return 0;
    }
    bool GetGood(){
        if (good)  {
            return true;
        }
        good = true;
        return false;
    }
private:
    bool good;
    int dummy;
};
```

# Konstrukcja try-catch, instrukcja throw

```
try
{
    .
    .
    throw  obj;
}
catch(type1 t1)
{
}
catch(type2& t2)
{
    throw;
}
catch(...)
{
}
```

# Przykład z wyjątkami

```
struct IndexException
{
    IndexException(int requested, int max):
    requestedIndex(requested),
    maxIndex(max)
    {
    }
    int requestedIndex;
    int maxIndex;
};
```

# Przykład z wyjątkami cd

```
class AdvancedArray : public Array
{
public:
    AdvancedArray(int size):Array(size){}
    int& operator[](int index){
        if (index < GetSize()) {
            return Array::operator [](index);
        }
        throw IndexException(index, GetSize() - 1);
    }
    int operator[](int index) const {
        if (index < GetSize()){
            return Array::operator [](index);
        }
        throw IndexException(index, GetSize() - 1);
    }
};
```

# Przykład z wyjątkami cd

```
AdvancedArray smallArray(10);
try
{
    smallArray[2] = 9;
    smallArray[10] = 7;
    smallArray[0] = -9;
}
catch(const IndexException& e)
{
    cout << "Index out of range, requested: "
       << e.requestedIndex << ", max: " << e.maxIndex << endl;
}
catch(...)
{
    cout << "Unknown error" << endl;
}
```

Index out of range, requested: 10, max: 9

# Dalsza rozbudowa przykładu

```
void CopyArrayElements(AdvancedArray& to, const AdvancedArray& from,
    int start, int count)
{
    try
    {
        for (int i = start; i < start + count; i++)
        {
            to[i] = from[i];
        }
    }
    catch(IndexException& e)
    {
        e.requestedIndex = start + count;
        e.maxIndex = to.GetSize() < from.GetSize() ?
            to.GetSize() : from.GetSize();
        throw;
    }
}
```

# Dalsza rozbudowa przykładu cd

```
AdvancedArray bigArray(100);
try
{
    CopyArrayElements(smallArray, bigArray, 5, 20);
}
catch(const IndexException& e)
{
    cout << "Index out of range, requested: " << e.requestedIndex
       << ", max: " << e.maxIndex << endl;
}
catch(...)
{
    cout << "Unknown error" << endl;
}
```

Index out of range, requested: 25, max: 10

# Kiedy wyjątek nie jest obsłużony?

- Brak pasującej sekcji catch
- Kolejny wyjątek po rzuceniu i przed złapaniem pierwszego
- Wyjątek podczas czyszczenia stosu

# Wyjątek po rzuceniu

```
struct IndexException{
    IndexException(int requested, int max):
    requestedIndex(requested),
    maxIndex(max){
    }
    IndexException(const IndexException& right){
        throw 1;
    }
    int requestedIndex;
    int maxIndex;
};

AdvancedArray smallArray(10);
try
{
    smallArray[2] = 9;
    smallArray[10] = 7;
    smallArray[0] = -9;
}
catch(IndexException e)
{
    cout << "Index out of range, requested: "  << e.requestedIndex
        << ", max: " << e.maxIndex << endl;
}
```

# Wyjątek przy czyszczeniu stosu

```
struct NastyClass
{
    ~NastyClass()
    {
        throw 1;
    }
};

AdvancedArray smallArray(10);
try
{
    NastyClass nc;
    smallArray[2] = 9;
    smallArray[10] = 7;
    smallArray[0] = -9;
}
catch(const IndexException& e)
{
    cout << "Index out of range, requested: " << e.requestedIndex
        << ", max: " << e.maxIndex << endl;
}
```

# Wyjątki a zwalnianie zasobów

```
class ExceptionTest
{
public:
    void WorkWithFile(const char* fname)
    {
        FILE* fPtr = fopen(fname, "w");
        writeToFile(fPtr, "too long text....");
        fclose(fPtr);
    }
};
try
{
    ExceptionTest et;
    et.WorkWithFile("testfile");
}
catch(...)
{
    cout << “Error.” << endl;
}
```

# Wyjątki a zwalnianie zasobów cd

```
class ExceptionTest
{
public:
    void WorkWithFile(const char* fname)
    {
        FILE* fPtr = fopen(fname, "w");
        writeToFile(fPtr, "too long text....");
        fclose(fPtr);
    }
    void WorkWithFileSafe(const char* fname)
    {
        FILE* fPtr = fopen(fname, "w");
        try
        {
            writeToFile(fPtr, "too long text....");
            fclose(fPtr);
        }
        catch(...)
        {
            fclose(fPtr);
            throw;
        }
    }
};
```

# Wyjątki a zwalnianie zasobów cd

```
class FILEWrapper
{
public:
    FILEWrapper(const char* fname, const char* faccess):
    fPtr(fopen(fname, faccess)){
    }
    operator FILE*() {
        return fPtr;
    }
    ~FILEWrapper(){
        fclose(fPtr);
    }
private:
    FILE* fPtr;
};


class ExceptionTest
{
public:
    void WorkWithFileSmart(const char* fname)
    {
        FILEWrapper f(fname, "w");
        writeToFile(f, "too long text....");
    }
};
```

# Wyjątki w konstruktorze

```
class ExceptionTest
{
public:
    ExceptionTest(const char* file1, const char* file2):
    fPtr1(fopen(file1, "w")),
    fPtr2(fopen(file2, "w")){
        if (fPtr1 == NULL || fPtr2 == NULL){
            throw FileException();
        }
    }
    ~ExceptionTest() {
        fclose(fPtr1);
        fclose(fPtr2);
    }
private:
    FILE* fPtr1;
    FILE* fPtr2;

};
try{
    ExceptionTest et2("file1", "file2");
}
catch(...){
    cout << "Error." << endl;
}
```

# Wyjątki w konstruktorze cd

```
class ExceptionTest
{
public:
    ExceptionTest(const char* file1, const char* file2):
    f1(file1, "w"),
    f2(file2, "w"){
        if (f1 == NULL || f2 == NULL) {
            throw FileException();
        }
    }
private:
    FILEWrapper f1;
    FILEWrapper f2;
};
try{
    ExceptionTest et2("file1", "file2");
}
catch(...){
    cout << "Error." << endl;
}
```

# Specyfikacja wyjątków funkcji

```
void function1(int arg) throw(ex1, ex2);
void function2(int arg) throw();
void function2(int arg) throw(...);
void function3(int arg);
```

# unexpected(), terminate() i abort()

- unexpected() jest wołana gdy funkcja rzuci wyjątek którego nie zadeklarowała. Domyślnie woła terminate()
- terminate() jest wołana gdy wyjątek nie został złapany lub podczas czyszczenia stosu zostanie zgłoszony kolejny wyjątek. Domyślnie woła abort()
- Domyślne zachowanie można zmienić poprzez set_unexpected i set_terminate
- Nie wszystkie implementacje obsługują tę funkcjonalność

# Część V

Wzorce

# Po co to?

- Możliwość przeprowadzenia przez programistę jednej implementacji dla wielu typów (nawet nieistniejących w momencie pisania)
- Skrócenie kodu źródłowego
- Szczególnie przydatne do klas typu “pojemnik”: wektorów, list itp.

- Problemem mogą być różnice i niedokładności w implementacji wzorców w kompilatorach

# Podstawy

- Można tworzyć wzorce klas oraz wzorce funkcji globalnych
- Parametrami dla wzorca są **typy** (określane słowem **class** lub **typename**, mogą być to również typy wbudowane) lub wartości
- Dookreślenie (specjalizacja, *instantiation*) wzorca dla danego typu jest przeprowadzana w miarę potrzeby (również poszczególne funkcje) i powoduje wygenerowanie implementacji specyficznej dla danego typu – nie ma oszczędności w kodzie wynikowym.

# Prymitywny wzorzec

```
template<class T, int size>
class FixedArray
{
public:
    T& operator[](int index)
    {
        if (index < size)
        {
            return storage[index];
        }
        throw IndexException(index, size - 1);
    }
private:
    T storage[size];
};

FixedArray<Circle, 100> cf100;
cout << cf100[15].Area() << endl;
```

# Bardziej typowy wzorzec

```
template<class T>
class VariableArray
{
public:
    VariableArray(int size):
    size(size),
    storagePtr(new T[size]){
    }
    ~VariableArray() {
        delete[] storagePtr;
    }
    T& operator[](int index){
        if (index < size){
            return storagePtr[index];
        }
        throw IndexException(index, size - 1);
    }
private:
    int size;
    T* storagePtr;
};

VariableArray<Circle> cv100(100);
cout << cv100[15].Area() << endl;
```

# Funkcje składowe definiowane poza klasą

```
template<class T>
class VariableArray
{
public:
    VariableArray(int size);
    ~VariableArray();
    T& operator[](int index);
private:
    int size;
    T* storagePtr;
};
template<class T>VariableArray<T>::VariableArray(int size):
size(size),
storagePtr(new T[size]){
}
template<class T>VariableArray<T>::~VariableArray(){
    delete[] storagePtr;
}
template<class T>T& VariableArray<T>::operator[](int index){
    if (index < size)   {
        return storagePtr[index];
    }
    throw IndexException(index, size - 1);
}
```

# Wzorce funkcji

```
template<class T> void Swap(T& left, T& right)
{
    T tmp = left;
    left = right;
    right = tmp;
}

int val1 = 1, val2 = 2;
Circle obj1(12), obj2(45);
Swap(val1, val2);
Swap(obj1, obj2);
Swap(obj1, val2);
cout << val1 << ' ' << val2 << endl;
```

# Jawne dookreślenie

- Przydatne np. w momencie tworzenia biblioteki. Normalnie nieużyte funkcje/klasy nie będą wygenerowane
- Można dookreślić całą klasę, wybrane funkcje składowe bądź też funkcję globalną

```
template class VariableArray<ColorfulCircle>;
template int& VariableArray<int>::operator[](int);
template void Swap<float>(float& left, float& right);
```

# Część VI

## Biblioteka STL

# Cechy

- Oparta o wzorce
- Zawiera kolekcje, strumienie i łańcuchy tekstowe oraz wsparcie dla alokacji pamięci, obsługi wyjątków, programowania algorytmów, lokalizacji itp.

# Kolekcje (wybrane)

- vector
- list
- deque
- map

# vector

- Najbardziej zbliżony do typowej tablicy
- Zapewnia stały czas swobodnego dostępu
- Dopisywanie na końcu jest szybkie jeśli nie jest wymagane wydłużenie kolekcji
- Dopisywanie w środku wymaga przesunięcia części kolekcji i jest czasochłonne

- Zdefiniowany w nagłówku <vector>

# vector – funkcje składowe

- vector( );
- explicit vector(size_type _Count);
- size_type capacity( ) const;
- void clear( );
- bool empty( ) const;
- void push_back(const Type& _Val);
- void pop_back( );
- void resize(size_type _Newsize);
- void reserve(size_type _Count);
- size_type size( ) const;
- reference at(size_type _Pos);
- const_reference at(size_type _Pos) const;
- reference operator[](size_type _Pos);
- const_reference operator[](size_type _Pos) const;

# vector - przykład

```
vector<Circle> vc(10);
cout << "Liczba elementow: " << vc.size() << ", zaalokowane: "
   << vc.capacity() << endl;
cout << vc[5].Area() << endl;
vc.reserve(100);
cout << "Liczba elementow: " << vc.size() << ", zaalokowane: "
   << vc.capacity() << endl;
try{
   cout << vc[15].Area() << endl;
   cout << vc.at(15) << endl;
}
catch(out_of_range& e){
   cout << e.what() << endl;
}
catch(...){
   cout << "Blad!" << endl;
}
```

```
Liczba elementow: 10, zaalokowane: 10
0
Liczba elementow: 10, zaalokowane: 100
5.85217e+017
invalid vector<T> subscript
```

# vector – przykład cd

```
vc.push_back(Circle(15));
cout << "Liczba elementow: " << vc.size() << ", zaalokowane: "
   << vc.capacity() << endl;
cout << vc[10].Area() << endl;
vector<Circle> vc2 = vc;
cout << vc2[10].Area() << endl;
vc.clear();
cout << "Liczba elementow: " << vc.size() << ", zaalokowane: "
   << vc.capacity() << endl;
cout << "Liczba elementow: " << vc2.size() << ", zaalokowane: "
   << vc2.capacity() << endl;
try{
   cout << vc[10].Area() << endl;
}
catch(out_of_range& e){
   cout << e.what() << endl;
}
catch(...){
   cout << "Blad!" << endl;
}
```

```
Liczba elementow: 11, zaalokowane: 100
706.858
706.858
Liczba elementow: 0, zaalokowane: 0
Liczba elementow: 11, zaalokowane: 11
Blad!
```

# Iteratory

- Rodzaj inteligentnego wskaźnika umożliwiającego poruszanie się po elementach kolekcji
- Szeroko stosowane w kolekcjach STL, w wielu przypadkach konieczne narzędzie do pracy z kolekcją
- W zależności od klasy z jaką współpracują mają różne możliwości (np. jedno lub dwukierunkowość, sekwencyjność lub swobodny dostęp, zapis lub odczyt)

# vector a iteratory

- const_iterator begin( ) const;
- iterator begin( );
- iterator end( );
- const_iterator end( ) const;
- iterator insert(iterator _Where, const Type& _Val);

# vector a iteratory - przykład

```
vector<Circle>::iterator it = vc2.begin();
while (cout << it->Area() << ' ', ++it != vc2.end());
cout << endl;
vc2.insert(it, Circle(20));
it = vc2.end();
while (--it, cout << it->Area() << ' ', it != vc2.begin());
cout << endl;
it = vc2.begin();
it += 10;
cout << it->Area() << endl;
```

0 0 0 0 0 0 0 0 0 0 706.858
1256.64 706.858 0 0 0 0 0 0 0 0 0
706.858

# list

- lista łączona dwukierunkowo
- szybkie dodawanie i usuwanie elementów
- poruszanie się po elementach kolekcji – tylko sekwencyjne
- nie ma potrzeby realokacji przy dodawaniu elementów
- przy dodawaniu elementów iteratory zachowują ważność, przy usuwaniu – wszystkie z wyjątkiem wskazującego na element usuwany (w wektorze iteratory stają się nieważne przy realokacji, przy wstawieniu bez realokacji – za punktem wstawienia)
- nagłówek <list>

# list – funkcje składowe

- list( );
- explicit list(size_type _Count);
- const_iterator begin( ) const;
- iterator begin( );
- const_iterator end( ) const;
- iterator end( );
- iterator erase(iterator _First, iterator _Last);
- void push_back(const Type& _Val);
- void push_front(const Type& _Val);
- void clear( );

# list - przykład

```
list<Circle> lc(10);
cout << "Liczba elementow: " << lc.size() << endl;
lc.push_back(Circle(15));
cout << "Liczba elementow: " << lc.size() << endl;
list<Circle> lc2 = lc;
lc.clear();
cout << "Liczba elementow: " << lc.size() << endl;
cout << "Liczba elementow: " << lc2.size() << endl;
list<Circle>::iterator it = lc2.begin();
while (cout << it->Area() << ' ', ++it != lc2.end());
cout << endl;
lc2.insert(it, Circle(20));
it = lc2.end();
while (--it, cout << it->Area() << ' ', it != lc2.begin());
cout << endl;
it = lc2.begin();
it++;
lc2.insert(it, Circle(30));
*it = Circle(40);
it = lc2.begin();
while (cout << it->Area() << ' ', ++it != lc2.end());
cout << endl;
```

```
Liczba elementow: 10
Liczba elementow: 11
Liczba elementow: 0
Liczba elementow: 11
0 0 0 0 0 0 0 0 0 0 706.858
1256.64 706.858 0 0 0 0 0 0 0 0 0 0
0 2827.43 5026.54 0 0 0 0 0 0 0 706.858 1256.64
```

# deque

- Kolekcja podobna do vector, ale umożliwiająca szybkie dopisywanie i usuwanie elementów na początku

# map

- Kolekcja zawierająca uporządkowane pary klucz-wartość
- Klucz musi być unikatowy, wartość nie
- Klucz umożliwia sortowanie kolekcji, co zapewnia szybki dostęp
- Wartość można zmieniać bezpośrednio, klucze tylko dodawać i usuwać
- typedef pair<const Key, Type> value_type;
- Nagłówek <map>

# map - funkcje

- map( );
- const_iterator begin( ) const;
- iterator begin( );
- const_iterator end( ) const;
- iterator end( );
- void clear( );
- size_type count(const Key& _Key) const;
- size_type size( ) const;
- iterator find(const Key& _Key);
- const_iterator find(const Key& _Key) const;
- pair <iterator, bool> insert(const value_type& _Val);
- Type& operator[](const Key& _Key);

# map - przykład

```
map<int, Person> hotel;
hotel[101] = Person("Kowalski");
hotel[201] = Person("Karwowska");
hotel[102] = Person("Malinowska");
hotel[202] = Person("Nowak");
int x = 201;
cout << "Pokoi w hotelu: " << hotel.size() << ", w tym o numerze "
   << x << ": " << hotel.count(x) << endl;
map<int, Person>::iterator it = hotel.begin();
while (cout << "W pokoju " << it->first << " jest " << it->second
   << ". ", ++it != hotel.end());
cout << endl;
cout << "W pokoju " << x << " jest " << hotel.find(x)->second << endl;
hotel[x] = "Bond, James Bond";
cout << "W pokoju " << x << " jest " << hotel.find(x)->second << endl;
```

Pokoi w hotelu: 4, w tym o numerze 201: 1
W pokoju 101 jest Kowalski. W pokoju 102 jest Malinowska.
W pokoju 201 jest Karwowska. W pokoju 202 jest Nowak.
W pokoju 201 jest Karwowska
W pokoju 201 jest Bond, James Bond

# string

- Umożliwia wygodne użycie łańcuchów (nie tylko tekstowych)
- Zawiera funkcje i operatory typowe dla takiego zastosowania
- Pojemnik nazywa się basic_string, string jest jego specjalizacją dla typu char
- Nagłówek <string>

# string - funkcje

- basic_string()
- basic_string(const value_type* _Ptr)
- const value_type *c_str( ) const;
- size_type capacity( ) const;
- int compare(const basic_string& _Str) const;
- size_type copy(value_type* _Ptr, size_type _Count, size_type _Off = 0) const;
- size_type find(const value_type* _Ptr, size_type _Off = 0) const;
- basic_string& insert(size_type _P0, const value_type* _Ptr);
- basic_string& replace(size_type _Pos1, size_type _Num1, const value_type* _Ptr);
- basic_string substr(size_type _Off = 0, size_type _Count = npos) const;
- begin, end, size, clear etc.

# string - operatory

- basic_string& operator+=(const value_type* _Ptr);
- basic_string& operator=(const value_type* _Ptr);
- const_reference operator[](size_type _Off) const;
- reference operator[](size_type _Off);

# string - przykład

```
string sample = "Ala ma kota";
char* copy = strdup(sample.c_str());
free(copy);
sample += " i psa";
cout << sample << endl;
cout << sample.substr(7, 4) << endl;
cout << "Pierwsze \'m\' na pozycji: " << sample.find("m") << endl;
sample.insert(sample.find("kota"), "rybki, ");
cout << sample << endl;
string action = "zjadla";
sample.replace(sample.find("ma"), 2, action);
cout << sample << endl;
```

Ala ma kota i psa
kota
Pierwsze 'm' na pozycji: 4
Ala ma rybki, kota i psa
Ala zjadla rybki, kota i psa

# Strumienie

- Strumień można traktować jako rodzaj pliku
- Dzięki zastosowaniu dziedziczenia można tak samo traktować operacje na różnych strumieniach (np. powiązanych z plikiem, konsolą, pamięcią, drukarką itp.)
- Można pisać własne klasy – strumienie, obsługujące inne urządzenia
- Współpraca ze strumieniem odbywa się głównie za pomocą operatorów << i >> oraz tzw. manipulatorów – obiektów przekazywanych do strumienia tak jak dane
- Do współpracy z konsolą wystarczy włączyć nagłówek <iostream>

# Strumienie – cd

- Bazami są klasy ios_base i basic_ios
- Zawierają istotne funkcje, związane ze stanem strumienia  formatowaniem
- Zawarte w nagłówku <ios>, tam również znajdują się manipulatory

# basic_ios

- bool bad( ) const;
- bool fail( ) const;
- bool eof( ) const;
- bool good( ) const;
- void clear();
- operator void *( ) const;
- bool operator!( ) const;

# ios_base

- streamsize precision(streamsize _Prec);
- streamsize width(streamsize _Wide);

# Manipulatory

- ios_base& dec(ios_base& _Str);
- ios_base& hex(ios_base& _Str);
- ios_base& fixed(ios_base& _Str);
- ios_base& scientific(ios_base& _Str);
- ios_base& left(ios_base& _Str);
- ios_base& right(ios_base& _Str);
- ios_base& skipws(ios_base& _Str);
- ios_base& noskipws(ios_base& _Str);
- ios_base& boolalpha(ios_base& _Str);
- ios_base& noboolalpha(ios_base& _Str);

# Strumienie wyjściowe

- klasa ostream. Predefiniowane strumienie (obiekty):
  - cout
  - cerr
  - clog
- klasa ofstream. Nagłówek <fstream>
- klasa ostrstream. Nagłówek <strstream>
- klasa fstream. Nagłówek <fstream>
- klasa strstream. Nagłówek <strstream>

# ofstream – funkcje

- ofstream();
- explicit ofstream(const char *_Filename, ios_base::openmode _Mode = ios_base::out);
- void close( );
- void open(const char *_Filename, ios_base::openmode _Mode = ios_base::out);
- ostream& flush( );
- basic_ostream& put(char_type _Ch);
- basic_ostream& seekp(pos_type _Pos);
- basic_ostream& seekp(off_type _Off, ios_base::seekdir _Way);
- pos_type tellp( );
- basic_ostream& write(const char_type *_Str, streamsize _Count);

# ofstream – operator <<

- Przeciążony m.in. dla typów: bool, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double

# ofstream – przykład

```
bool StreamWriter(ostream& o, const char* animal)
{
    o << "Dzisiejsze zwierze Ali to " << animal;
    return o.good();
}
ofstream os("TestFile.txt");
cout << boolalpha << "Ofstream OK: " << os.good() << endl;
StreamWriter(cout, "pies");
StreamWriter(os, "pies");
cout << endl;
int number = 15;
cout << "Dec: " << number << ", Hex: " << hex << number << endl;
cout << "A teraz: " << number << endl;
cout.width(20);
cout << right << "A teraz: " << number << endl;
cout << left << scientific << dec << "Zmiennoprzecinkowe wykladnicze: "
    << 3.14 << endl;
cout << "Pozycja w strumieniu: " << os.tellp() << endl;
os.seekp(-4, ios_base::cur);
cout << "Pozycja w strumieniu: " << os.tellp() << endl;
os << "czarna pantera";
```

Ofstream OK: true
Dzisiejsze zwierze Ali to pies
Dec: 15, Hex: f
A teraz: f
A teraz: f
Zmiennoprzecinkowe wykladnicze: 3.140000e+000
Pozycja w strumieniu: 30
Pozycja w strumieniu: 26

Dzisiejsze zwierze Ali to czarna pantera

# Strumienie wejściowe

- klasa istream. Predefiniowane strumienie (obiekty):
  - cin
- klasa ifstream. Nagłówek <fstream>
- klasa istrstream. Nagłówek <strstream>
- klasa fstream. Nagłówek <fstream>
- klasa strstream. Nagłówek <strstream>

# ifstream – przykład

```
ifstream is("TestFile.txt");
int testNumber;
is >> testNumber;
cout << boolalpha << "Ifstream OK: " << is.good() << endl;
is.clear();
cout << boolalpha << "Ifstream OK: " << is.good() << endl;
char buf[128];
while(is.good())
{
    is >> buf;
    cout << buf << endl;
}
```

```
Ifstream OK: false
Ifstream OK: true
Dzisiejsze
zwierze
Ali
to
czarna
pantera
```

# Część VII

## Operatory rzutowania i RTTI

# Operatory rzutowania w C++

- static_cast
- const_cast
- reinterpret_cast
- dynamic_cast

- Składnia: operator<typ>(wyrażenie)

# static_cast

- Umożliwia rzutowanie pomiędzy typami powiązanymi ze sobą (np. w ramach hierarchii dziedziczenia)
- Może być niebezpieczny (np. przy rzutowaniu wskaźnika do klasy bazowej na wskaźnik do klasy pochodnej)

# static_cast – przykład

```
struct Base{
    Base() : baseNumber(5){}
    int baseNumber;
};

struct Derived : public Base{
    Derived() : derivedNumber(10){}
    int derivedNumber;
};

int i1 = 7;
Base b1;
Derived d1;
float f1 = static_cast<float>(i1);
float* fPtr;
Base* bPtr;
Derived* dPtr;
bPtr = &d1;
// fPtr = static_cast<float*>(bPtr);
fPtr = (float*)bPtr;
dPtr = static_cast<Derived*>(bPtr);
cout << dPtr->derivedNumber << endl;
bPtr = &b1;
dPtr = static_cast<Derived*>(bPtr);
cout << dPtr->derivedNumber << endl;
```

10
-858993460

# const_cast

- Służy do pozbycia się modyfikatora const lub volatile
- Typ przed i po rzutowaniu musi być ten sam (z wyjątkiem j.w.)
- Nie można nim “zdjąć” stałości ze stałej

# const_cast – przykład

```
void TestFun(Base& b){}

const Base* cbPtr = new Base;
//bPtr = static_cast<Base*>(cbPtr);
bPtr = const_cast<Base*>(cbPtr);
const Base cb;
//TestFun(const_cast<Base>(cb));
```

# reinterpret_cast

- Umożliwia rzutowanie dowolnego typu wskaźnika na dowolny typ wskaźnika (z wyjątkiem const)
- Umożliwia rzutowanie wskaźnika na typ całkowity i odwrotnie

```
int i1 = 7;
Derived* dPtr;

dPtr = reinterpret_cast<Derived*>(&i1);
cout << dPtr->derivedNumber << endl;
```

# dynamic_cast

- Wykonuje rzutowanie na podstawie informacji o typie obiektu uzyskanej **podczas wykonywania programu**
- Działa tylko dla wskaźników i referencji
- Obiekt rzutowany musi być typu posiadającego funkcję wirtualną
- Jeśli nie można wykonać bezpiecznego rzutowania, rzutowanie wskaźnika daje NULL, a rzutowanie referencji wyjątek bad_cast
- Stanowi dodatkowe obciążenie
- Często wymaga niestandardowych ustawień kompilatora

# dynamic_cast – przykład 1

```
Base b1;
Derived d1;
Base* bPtr;
Derived* dPtr;
bPtr = &d1;
dPtr = dynamic_cast<Derived*>(bPtr);
((dPtr == NULL) ? cout << "Nie ten typ" : cout << dPtr->derivedNumber)
   << endl;
bPtr = &b1;
dPtr = dynamic_cast<Derived*>(bPtr);
((dPtr == NULL) ? cout << "Nie ten typ" : cout << dPtr->derivedNumber)
   << endl;
```

10
Nie ten typ

# dynamic_cast i interfejsy

```
struct Sizeable{
   virtual void Resize(float factor) = 0;
};

struct Movable{
   virtual void MoveX(int distance) = 0;
   virtual void MoveY(int distance) = 0;
};

struct Item{
   virtual string GetState() const = 0;
};
```

# dynamic_cast i interfejsy – cd

```
class Chair : public Item, public Movable{
public:
    Chair() : posX(0), posY(0){}
    virtual void MoveX(int distance){
        posX += distance;
    }
    virtual void MoveY(int distance){
        posY += distance;
    }
    virtual string GetState() const
    {
        ostringstream s;
        s << "Krzeslo jest na pozycji " << posX << ", " << posY;
        return s.str();
    }
private:
    int posX;
    int posY;
};
```

# dynamic_cast i interfejsy – cd

```
class Ball : public Item, public Movable{
public:
    Ball() : posX(0), posY(0){}
    virtual void MoveX(int distance){
        posX += int(10 * distance);
        posY += int(rand() * distance / RAND_MAX);
    }
    virtual void MoveY(int distance){
        posY += int(10 * distance);
        posX += int(rand() * distance / RAND_MAX);
    }
    virtual string GetState() const
    {
        ostringstream s;
        s << "Pilka jest na pozycji " << posX << ", " << posY;
        return s.str();
    }
private:
    int posX;
    int posY;
};
```

# dynamic_cast i interfejsy – cd

```
class Bed : public Item, public Sizeable{
public:
    Bed() : big(false){}
    virtual void Resize(float factor){
        if (factor > 0){
            big = true;
        }else if (factor < 0){
            big = false;
        }
    }
    virtual string GetState() const
    {
        ostringstream s;
        s << "Lozko jest " << (big ? "rozlozone" : "zlozone");
        return s.str();
    }

private:
    bool big;
};
```

# dynamic_cast i interfejsy – cd

```
class Baloon : public Item, public Movable, public Sizeable{
public:
    Baloon() : posX(0), posY(0), size(1){}
    virtual void MoveX(int distance){
        posX += int(0.5 * distance);
        posY += int(0.5 * rand() * distance / RAND_MAX);
    }
    virtual void MoveY(int distance){
        posY += int(0.5 * distance);
        posX += int(0.5 * rand() * distance / RAND_MAX);
    }
    virtual void Resize(float factor){
        size *= factor;
        if (size > 100){size = 0;}
    }
    virtual string GetState() const  {
        ostringstream s;
        s << "Balon jest na pozycji " << posX << ", " << posY;
        if (size){s << " i ma rozmiar " << size;}else{
            s << " i jest pekniety";}
        return s.str();
    }

private:
    int posX;
    int posY;
    float size;
};
```

# dynamic_cast i interfejsy -- cd

```
void ShowItems(Item* array[], int count)
{
    cout << "Stan obiektow:" << endl;
    for (int i = 0; i < count; i++)
    {
        cout << array[i]->GetState() << endl;
    }
}
```

# dynamic_cast i interfejsy

```
srand(0);
int count = 6;
Item** stuff = new Item*[count];
stuff[0] = new Chair();
stuff[1] = new Chair();
stuff[2] = new Ball();
stuff[3] = new Ball();
stuff[4] = new Bed();
stuff[5] = new Baloon();
ShowItems(stuff, count);
cout << "Przesuwamy co sie da o 10, 20" << endl;
for (int i = 0; i < count; i++){
    Movable* mPtr =
        dynamic_cast<Movable*>(stuff[i]);
    if (mPtr != NULL){
        mPtr->MoveX(10);
        mPtr->MoveY(20);
    }
}
ShowItems(stuff, count);
cout << "Powiekszamy co sie da o 10 razy" << endl;
for (int i = 0; i < count; i++){
    Sizeable* sPtr = dynamic_cast<Sizeable*>(stuff[i]);
    if (sPtr != NULL){
        sPtr->Resize(10);
    }
}
ShowItems(stuff, count);
```

```
Stan obiektow:
Krzeslo jest na pozycji 0, 0
Krzeslo jest na pozycji 0, 0
Pilka jest na pozycji 0, 0
Pilka jest na pozycji 0, 0
Lozko jest zlozone
Balon jest na pozycji 0, 0 i ma rozmiar 1
Przesuwamy co sie da o 10, 20
Stan obiektow:
Krzeslo jest na pozycji 10, 20
Krzeslo jest na pozycji 10, 20
Pilka jest na pozycji 104, 200
Pilka jest na pozycji 101, 206
Lozko jest zlozone
Balon jest na pozycji 8, 11 i ma rozmiar 1
Powiekszamy co sie da o 10 razy
Stan obiektow:
Krzeslo jest na pozycji 10, 20
Krzeslo jest na pozycji 10, 20
Pilka jest na pozycji 104, 200
Pilka jest na pozycji 101, 206
Lozko jest rozlozone
Balon jest na pozycji 8, 11 i ma rozmiar 10
```

# operator typeid

- Umożliwia sprawdzenie typu obiektu podczas wykonywania programu
- Umożliwia też uzyskanie takich samych informacji o typie
- Jeśli informacja statyczna nie jest wystarczająca do określenia typu, sprawdzana jest informacja dynamiczna
- Dynamiczne sprawdzenie będzie działało poprawnie w przypadku typów posiadających funkcję wirtualną (w przeciwnym wypadku przeprowadzone będzie tylko sprawdzenie statyczne)
- Zwraca klasę type_info

# klasa type_info

```
class type_info {
public:
   virtual ~type_info();
   int operator==(const type_info& rhs) const;
   int operator!=(const type_info& rhs) const;
   int before(const type_info& rhs) const;
   const char* name() const;
   const char* raw_name() const;
private:
   ...
};
```

# typeinfo -- przykład

```
cout << "int: " << typeid(int).name() << ", " << typeid(int).raw_name()
    << endl;
cout << "Bed: " << typeid(Bed).name() << ", " << typeid(Bed).raw_name()
    << endl;
for (int i = 0; i < count; i++){
    cout << typeid(*stuff[i]).name() << endl;
}
Circle* circlePtr = new ColorfulCircle();
cout << typeid(*circlePtr).name() << endl;
```

int: int, .H
Bed: class Bed, .?AVBed@@
class Chair
class Chair
class Ball
class Ball
class Bed
class Baloon
class Circle