Metody wirtualne i abstrakcyjne, operatory rzutowania, przestrzenie nazw, wyjątki

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Niniejsza prezentacja została wykonana przy użyciu systemu składu LaTeX *oraz stylu* `beamer`, *którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:
`http://latex-beamer.sourceforge.net`

Plan prezentacji

Plan prezentacji

Rozmiar obiektów

```
struct KlasaBezWMetody { //.....
        int _PoleInt;
        int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda1 { //.....
        int _PoleInt;
        virtual int Wartosc( ) const { return 1; }
}; //.....

struct KlasaZWirMetoda2 { //.....
        int _PoleInt;
        virtual int Wartosc1( ) const { return 1; }
        virtual int Wartosc2( ) const { return 2; }
}; //.....

int main( )
{
      cout << sizeof(int ) << endl;
      cout << sizeof(KlasaBezWMetody) << endl;
      cout << sizeof(KlasaZWirMetoda1) << endl;
      cout << sizeof(KlasaZWirMetoda2) << endl;
}
```

Rozmiar obiektów

```
struct KlasaBezWMetody { //.....
        int _PoleInt;
        int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda1 { //.....
        int _PoleInt;
        virtual int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda2 { //.....
        int _PoleInt;
        virtual int Wartosc1( ) const { return 1; }
        virtual int Wartosc2( ) const { return 2; }
}; //.....

int main( )
{
      cout << sizeof(int ) << endl;
      cout << sizeof(KlasaBezWMetody) << endl;
      cout << sizeof(KlasaZWirMetoda1) << endl;
      cout << sizeof(KlasaZWirMetoda2) << endl;
}
```

Wynik działania:

```
4
4
8
8
```

Rozmiar obiektów

```
struct KlasaBezWMetody { //.....
        int _PoleInt;
        int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda1 { //.....
        int _PoleInt;
        virtual int Wartosc( ) const { return 1; }
}; //.....

struct KlasaZWirMetoda2 { //.....
        int _PoleInt;
        virtual int Wartosc1( ) const { return 1; }
        virtual int Wartosc2( ) const { return 2; }
}; //.....

int main( )
{
      cout << sizeof(int ) << endl;
      cout << sizeof(KlasaBezWMetody) << endl;
      cout << sizeof(KlasaZWirMetoda1) << endl;
      cout << sizeof(KlasaZWirMetoda2) << endl;
}
```

Wynik działania:

```
4
4
8
8
```

Rozmiar obiektów

```
struct KlasaBezWMetody { //.....
        int _PoleInt;
        int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda1 { //.....
        int _PoleInt;
        virtual int Wartosc( ) const { return 1; }
}; //.....

struct KlasaZWirMetoda2 { //.....
        int _PoleInt;
        virtual int Wartosc1( ) const { return 1; }
        virtual int Wartosc2( ) const { return 2; }
}; //.....

int main( )
{
      cout << sizeof(int ) << endl;
      cout << sizeof(KlasaBezWMetody) << endl;
      cout << sizeof(KlasaZWirMetoda1) << endl;
      cout << sizeof(KlasaZWirMetoda2) << endl;
}
```

Wynik działania:

```
4
4
8
8
```

Rozmiar obiektów

```
struct KlasaBezWMetody { //.....
        int _PoleInt;
        int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda1 { //.....
        int _PoleInt;
        virtual int Wartosc( ) const { return 1; }
}; //.....

struct KlasaZWirMetoda2 { //.....
        int _PoleInt;
        virtual int Wartosc1( ) const { return 1; }
        virtual int Wartosc2( ) const { return 2; }
}; //.....

int main( )
{
      cout << sizeof(int ) << endl;
      cout << sizeof(KlasaBezWMetody) << endl;
      cout << sizeof(KlasaZWirMetoda1) << endl;
      cout << sizeof(KlasaZWirMetoda2) << endl;
}
```

Wynik działania:

```
4
4
8
8
```

Rozmiar obiektów

```
struct KlasaBezWMetody { //.....
        int _PoleInt;
        int Wartosc( ) const { return 1; }
}; //.....

 struct KlasaZWirMetoda1 { //.....
        int _PoleInt;
        virtual int Wartosc( ) const { return 1; }
}; //.....

struct KlasaZWirMetoda2 { //.....
        int _PoleInt;
        virtual int Wartosc1( ) const { return 1; }
        virtual int Wartosc2( ) const { return 2; }
}; //.....

int main( )
{
      cout << sizeof(int ) << endl;
      cout << sizeof(KlasaBezWMetody) << endl;
      cout << sizeof(KlasaZWirMetoda1) << endl;
      cout << sizeof(KlasaZWirMetoda2) << endl;
}
```

Wynik działania:

```
4
4
8
8
```

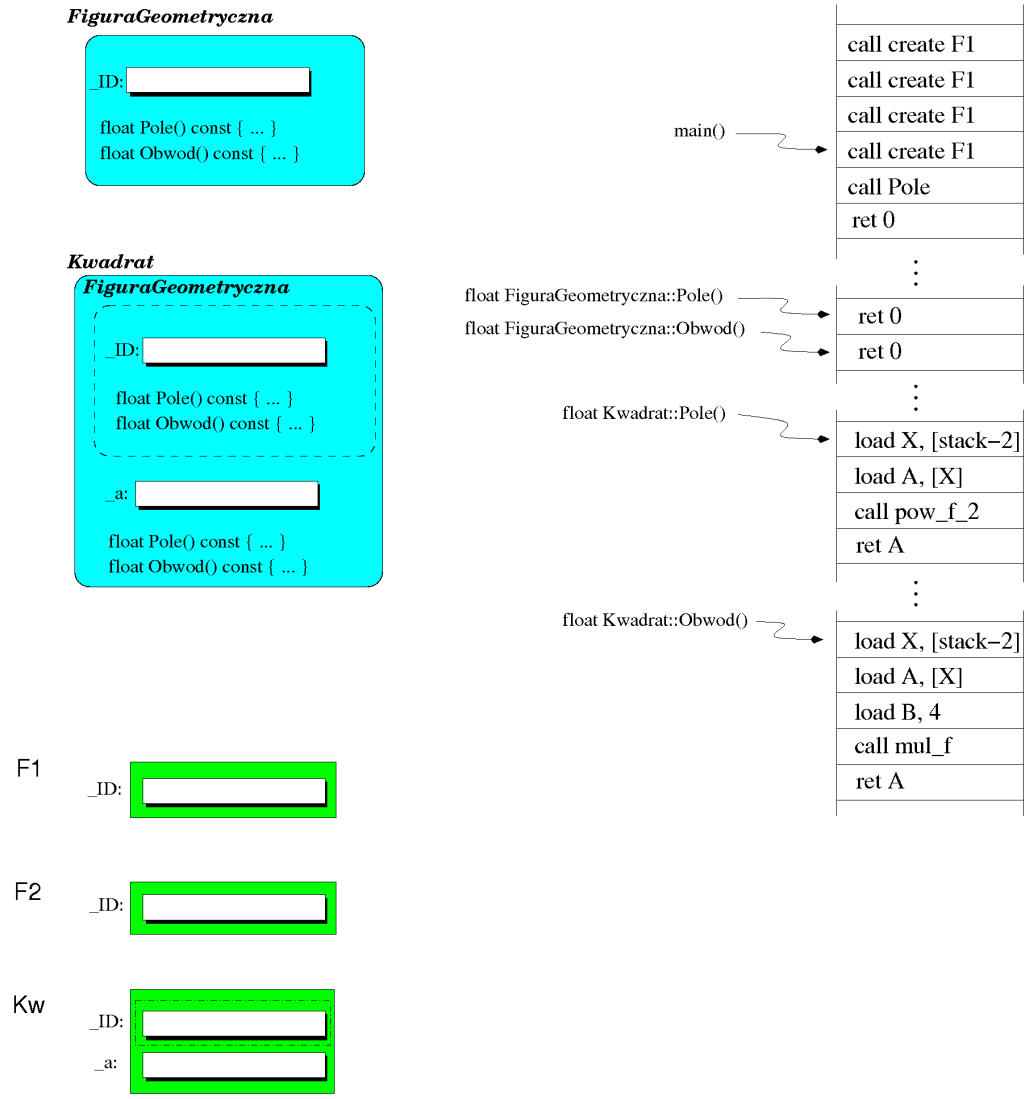
Rozmiar obiektów

```
class FiguraGeometryczna {
  public:
    int   _ID;
    float Pole() const { return 0; }
    float Obwod() const { return 0; }
};


class Kwadrat: public FiguraGeometryczna {
  public:
    float   _a;
    float Pole() const { return _a*_a; }
    float Obwod() const { return 4*_a; }
};


int main()
{
  FiguraGeometryczna  F1, F2;
  Kwadrat   Kw;

  F1.Pole();
}
```



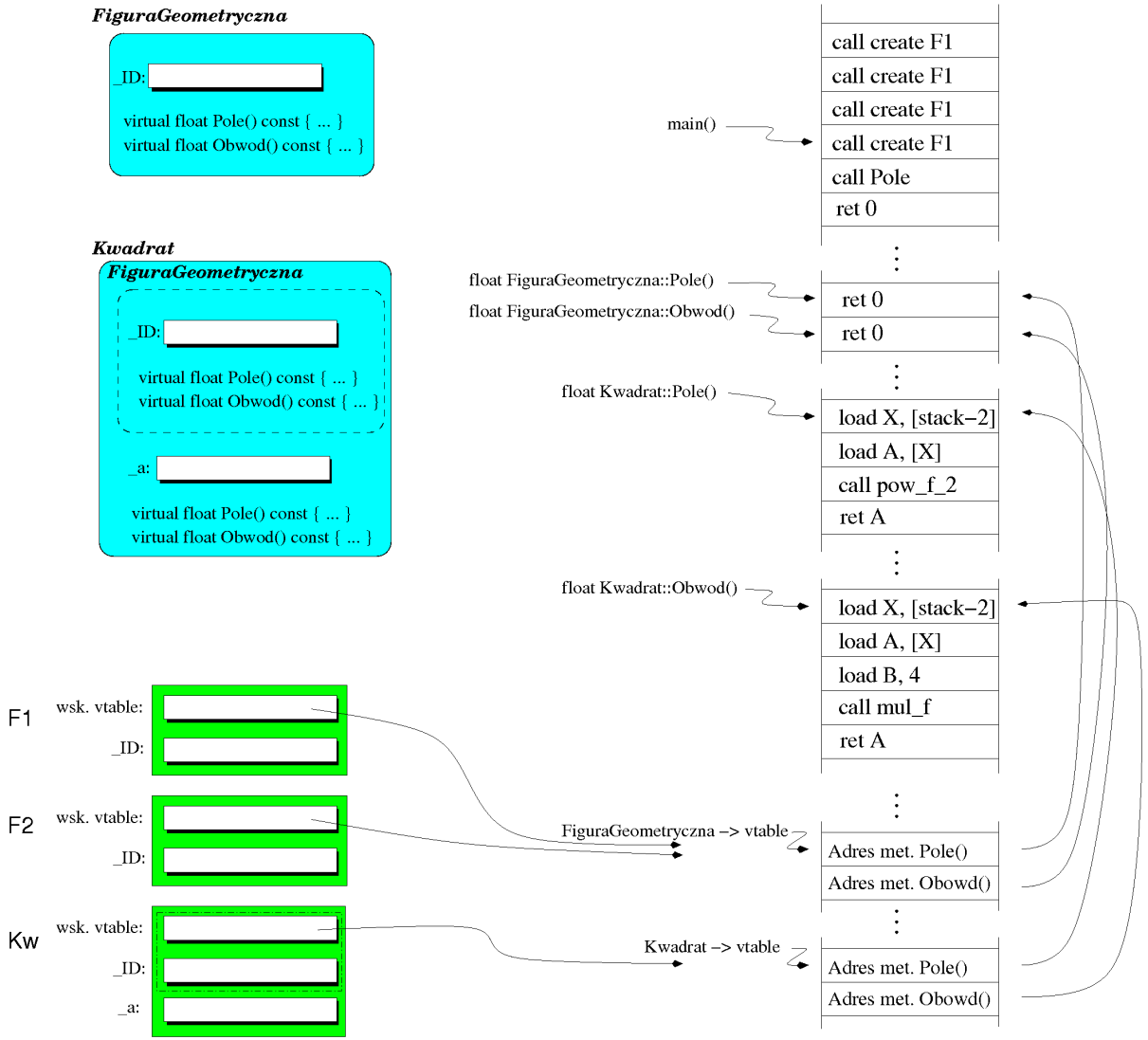
Rozmiar obiektów

```
class FiguraGeometryczna {
  public:
    int   _ID;
    virtual float Pole() const { return 0; }
    virtual float Obwod() const { return 0; }
};


class Kwadrat: public FiguraGeometryczna {
  public:
    float   _a;
    virtual float Pole() const { return _a*_a; }
    virtual float Obwod() const { return 4*_a; }
};


int main()
{
  FiguraGeometryczna  F1, F2;
  Kwadrat   Kw;

  F1.Pole();
}
```



Plan prezentacji

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const { return   0; }
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna   ObFig;
    Kwadrat              ObKwa;



}
```

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const = 0;
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna   ObFig;
    Kwadrat              ObKwa;



}
```

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const = 0;
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna   ObFig;
    Kwadrat              ObKwa;



}
```

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const = 0;
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna   ObFig;
    Kwadrat              ObKwa;



}
```

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const = 0;
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna  ObFig;
    Kwadrat             ObKwa;



}
```

Zdefiniowanie metody abstrakcyjnej w danej klasie uniemożliwia samodzielne istnienie obiektu tej klasy, gdyż jedna z metod nie ma kodu. Klasa nie jest więc w pełni zdefiniowana.

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const = 0;
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna  ObFig;
    Kwadrat             ObKwa;

    FiguraGeometryczna  *wsk_ObFig = &ObKwa;

}
```

Zdefiniowanie metody abstrakcyjnej w danej klasie uniemożliwia samodzielne istnienie obiektu tej klasy, gdyż jedna z metod nie ma kodu. Klasa nie jest więc w pełni zdefiniowana.
Możliwie jest jednak posługiwanie się wskaźnikami lub referencjami na obiekty tej klasy, które są składnikami innych obiektów. W klasach tych obiektów musi istnieć kod dla wszystkich metod abstrakcyjnych klasy bazowej.

Definiowanie metod abstrakcyjnych

```
struct FiguraGeometryczna { //.....
    ...
    virtual   float  Pole( ) const = 0;
}; //.....

struct Kwadrat: public  FiguraGeometryczna{ //.....
    ...
    float  _DlugoscBoku;
    virtual   float  Pole( ) const  { return   _DlugoscBoku*_DlugoscBoku; }
    ...
}; //.....

int  main( )
{
    FiguraGeometryczna  ObFig;
    Kwadrat             ObKwa;

    FiguraGeometryczna  *wsk_ObFig = &ObKwa;
    FiguraGeometryczna  &ref_ObFig = ObKwa;
}
```

Zdefiniowanie metody abstrakcyjnej w danej klasie uniemożliwia samodzielne istnienie obiektu tej klasy, gdyż jedna z metod nie ma kodu. Klasa nie jest więc w pełni zdefiniowana.
Możliwie jest jednak posługiwanie się wskaźnikami lub referencjami na obiekty tej klasy, które są składnikami innych obiektów. W klasach tych obiektów musi istnieć kod dla wszystkich metod abstrakcyjnych klasy bazowej.

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //.....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna  ObFig;
    Wielobok            ObWielobok;
    Kwadrat             ObKwadrat;
}
```

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //.....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna  ObFig;
    Wielobok            ObWielobok;
    Kwadrat             ObKwadrat;
}
```

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //.....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna ObFig;
    Wielobok           ObWielobok;
    Kwadrat            ObKwadrat;
}
```

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //.....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna ObFig;
    Wielobok           ObWielobok;
    Kwadrat            ObKwadrat;
}
```

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //.....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna ObFig;
    Wielobok           ObWielobok;
    Kwadrat            ObKwadrat;
}
```

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //.....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna ObFig;
    Wielobok           ObWielobok;
    Kwadrat            ObKwadrat;
}
```

Problem pełnej zgodność nazw metod i ich parametrów

```
struct FiguraGeometryczna { //.....
    virtual double Pole( ) const = 0;
}; //.....

struct Wielobok: FiguraGeometryczna { //....
    double Pole( ) { ... }
}; //.....

struct Kwadrat: Wielobok { //.....
    double Pole( ) const { ... }
}; //.....

int main( )
{
    FiguraGeometryczna ObFig;
    Wielobok           ObWielobok;
    Kwadrat            ObKwadrat;
}
```

Jeżeli w klasie pochodnej nie zostanie zdefiniowany kod dla dziedziczonej metody abstrakcyjnej, to klasa ta staje się również klasą abstrakcyjną

Plan prezentacji

Sposób na korzystanie z funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.....
    virtual float Pole( ) const = 0;
}; //.....

struct Kwadrat: public FiguraGeometryczna{ //.....
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.....

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Sposób na korzystanie z funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.....
    virtual float Pole( ) const = 0;
}; //.....

struct Kwadrat: public FiguraGeometryczna{ //.....
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.....

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Sposób na korzystanie z funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.....
    virtual float Pole( ) const = 0;
}; //.....

struct Kwadrat: public FiguraGeometryczna{ //.....
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.....

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Sposób na korzystanie z funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.....
    virtual float Pole( ) const = 0;
}; //.....

struct Kwadrat: public FiguraGeometryczna{ //.....
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.....

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Sposób na korzystanie z funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.....
    virtual float Pole( ) const = 0;
}; //.....

struct Kwadrat: public FiguraGeometryczna{ //.....
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.....

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Wynik działania:

```
Pole: 4
```

Sposób na korzystanie z funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.....
    virtual float Pole( ) const = 0;
}; //.....

struct Kwadrat: public FiguraGeometryczna{ //.....
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.....

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Wynik działania:

```
Pole: 4
```

Możliwość korzystania ze wskaźników i referencji do klasy abstrakcyjnej pozwala na definiowanie dla niej funkcji lub metod, które będą wykorzystywane dla obiektów klasy pochodnej.

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Wynik działania:

```
1
2
```

Własny identyfikator typu

```
struct FiguraGeometryczna { //.....
        ...
    virtual int ID( ) const = 0;
}; //.....

struct Kwadrat: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 1; }
}; //.....

struct Okrag: FiguraGeometryczna { //.....
        ...
    int ID( ) const { return 2; }
}; //.....

int main( )
{
    FiguraGeometryczna *wFigA = new Kwadrat;
    FiguraGeometryczna *wFigB = new Okrag;

    cout << wFigA->ID( ) << endl;
    cout << wFigB->ID( ) << endl;
}
```

Metody wirtualne można wykorzystać do konstrukcji własnego identyfikatora typu. Wymaga to wykorzystanie jednej wspólnej klasy bazowej dla danego zbioru klas pochodnych.

Wynik działania:

```
1
2
```

Podsumowanie

- Metody abstrakcyjne pozwalają na tworzenie prototypu interfejsu klasy bazowej, który przejmowany jest przez klasę pochodną i w niej jest definiowany.
  Pozwala to stworzyć prototypy funkcjonalności klasy bazowej.

Plan prezentacji

Lista operatorów rzutowania

const_cast – rzutowanie usuwające modyfikatory **const** oraz **volatile**. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań bezpiecznych.

static_cast – używany do zdefiniowanych przez użytkownika, standardowych lub niejawnych konwersji typów. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań w zasadzie bezpiecznych.

reinterpret_cast – jest najbardziej niebezpiecznym rzutowaniem. Wykonuje on konwersję między wskaźnikami oraz wskaźnikami i liczbami. Źle przeprowadzone rzutowanie może być źródłem błędów trudnych do wykrycia.

dynamic_cast – obsługuje tylko obiekty klas polimorficznych. Zapewnia realizację rzutowania "w górę". Rzutowanie to należy do rzutowań bezpiecznych.

Lista operatorów rzutowania

const_cast – rzutowanie usuwające modyfikatory **const** oraz **volatile**. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań bezpiecznych.

static_cast – używany do zdefiniowanych przez użytkownika, standardowych lub niejawnych konwersji typów. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań w zasadzie bezpiecznych.

reinterpret_cast – jest najbardziej niebezpiecznym rzutowaniem. Wykonuje on konwersję między wskaźnikami oraz wskaźnikami i liczbami. Źle przeprowadzone rzutowanie może być źródłem błędów trudnych do wykrycia.

dynamic_cast – obsługuje tylko obiekty klas polimorficznych. Zapewnia realizację rzutowania "w górę". Rzutowanie to należy do rzutowań bezpiecznych.

Operator const_cast

```
int  main( )
{
     const  int  ZmStala = 5;
     int         ZmZmien;


}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;
    int       ZmZmien;

    ZmZmien = ...;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;
    int       ZmZmien;

    ZmZmien = const_cast<int>(ZmStala);
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;
    int       ZmZmien;

    ZmZmien = ZmStala;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;
    int       ZmZmien;

    ZmZmien = 10;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;


    ZmStala = 10;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;


    ZmStala = 10;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;


    const_cast <int>(ZmStala) = 10;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;

    const_cast <int>(ZmStala) = 10;
}
```

Operator const_cast

```
int main( )
{
    const int ZmStala = 5;


    const_cast <int &>(ZmStala) = 10;
}
```

Operator const_cast

```
int  main( )
{
     volatile  int  ZmUlotna = 5;
     int            ZmZmien = 1;

     ;
     ...
}
```

Operator const_cast

```
int main( )
{
    volatile int ZmUlotna = 5;
    int          ZmZmien = 1;

    ZmZmien = 10;
    ...
}
```

Operator const_cast

```
int main( )
{
    volatile int ZmUlotna = 5;
    int          ZmZmien = 1;

    const_cast<volatile int&>(ZmZmien) = 10;
    ...
}
```

Operator const_cast

```
int main( )
{
    const char* sNapis = "lodka";



}
```

Operator const_cast

```
int main( )
{
    const char* sNapis = "lodka";


    sNapis[0] = 'w';
}
```

Operator const_cast

```
int main( )
{
    const char* sNapis = "lodka";


    sNapis[0] = 'w';
}
```

Operator const_cast

```
int main( )
{
    const char* sNapis = "lodka";


    const_cast<char*>(sNapis)[0] = 'w';
}
```

Operator const_cast

```
int main( )
{
    const char* sNapis = "lodka";


    const_cast<char*>(sNapis)[0] = 'w';
}
```

Operator const_cast

```
int main( )
{
    const char* sNapis[ ] = "lodka";


    const_cast<char*>(sNapis)[0] = 'w';
}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:

        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   LZespolona    Z;


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:

        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:

        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:

        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z = LZespolona( );


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ) { re = im = 0; }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z = LZespolona( );


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ) { re = im = 0; }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ): re(0), im(0) {  }
        void Zmien(double r, double i) { re = r;  im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ): re(), im() {  }
        void Zmien(double r, double i) { re = r;  im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Obiekty stałe

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ) { }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Operator const_cast

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ): re(0), im(0) { }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;


}
```

Operator const_cast

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ): re(0), im(0) { }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;

   Z.Zmien(1,5);
}
```

Operator const_cast

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ): re(0), im(0) { }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;

   Z.Zmien(1,5);
}
```

Operator const_cast

```
class LZespolona {
        double re, im;
      public:
        LZespolona( ): re(0), im(0) { }
        void Zmien(double r, double i) { re = r; im = i; }
        ...
};

int main( )
{
   const LZespolona Z;

   const_cast<LZespolona&>(Z).Zmien(1,5);
}
```

Stałość obiektu

```
class ElemListy_Num {

    int _Numer;
  public:
    ElemListy_Num(int Num): _Numer(Num) { }
    int Numer( ) const { return _Numer; }


    ...
};

int main( )
{
  const ElemListy_Num *wElem1 = new ElemListy_Num(7);


}
```

Stałość obiektu

```
class ElemListy_Num {
      ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }


      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);


}
```

Stałość obiektu

```
class ElemListy_Num {
      ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }


      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);
   const  ElemListy_Num     *wElem2 = new  ElemListy_Num(8);


}
```

Stałość obiektu

```
class ElemListy_Num {
      ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }
      void  DolaczNastepny( const ElemListy_Num *wNast )
                                          { _wNastepny = wNast; }
      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);
   const  ElemListy_Num     *wElem2 = new  ElemListy_Num(8);


}
```

Stałość obiektu

```
class ElemListy_Num {
      ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }
      void  DolaczNastepny( const ElemListy_Num *wNast )
                                          { _wNastepny = wNast; }
      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);
   const  ElemListy_Num     *wElem2 = new  ElemListy_Num(8);

   wElem1->DolaczNastepny(wElem2);
}
```

Stałość obiektu

```
class ElemListy_Num {
      ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }
      void  DolaczNastepny( const ElemListy_Num *wNast )
                                          { _wNastepny = wNast; }
      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);
   const  ElemListy_Num     *wElem2 = new  ElemListy_Num(8);

   wElem1->DolaczNastepny(wElem2);
}
```

Stałość obiektu

```
class ElemListy_Num {
      ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }
      void  DolaczNastepny( const ElemListy_Num *wNast ) const
                                          { _wNastepny = wNast; }
      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);
   const  ElemListy_Num     *wElem2 = new  ElemListy_Num(8);

   wElem1->DolaczNastepny(wElem2);
}
```

Stałość obiektu

```
class ElemListy_Num {
      mutable  ElemListy_Num* _wNastepny;
      int  _Numer;
   public :
      ElemListy_Num(int Num): _Numer(Num) { }
      int  Numer( ) const  { return   _Numer; }
      void  DolaczNastepny( const ElemListy_Num *wNast ) const
                                          { _wNastepny = wNast; }
      ...
};

int  main( )
{
   const  ElemListy_Num     *wElem1 = new  ElemListy_Num(7);
   const  ElemListy_Num     *wElem2 = new  ElemListy_Num(8);

   wElem1->DolaczNastepny(wElem2);
}
```

Lista operatorów rzutowania

const_cast – rzutowanie usuwające modyfikatory **const** oraz **volatile**. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań bezpiecznych.

static_cast – używany do zdefiniowanych przez użytkownika, standardowych lub niejawnych konwersji typów. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań w zasadzie bezpiecznych.

reinterpret_cast – jest najbardziej niebezpiecznym rzutowaniem. Wykonuje on konwersję między wskaźnikami oraz wskaźnikami i liczbami. Źle przeprowadzone rzutowanie może być źródłem błędów trudnych do wykrycia.

dynamic_cast – obsługuje tylko obiekty klas polimorficznych. Zapewnia realizację rzutowania "w górę". Rzutowanie to należy do rzutowań bezpiecznych.

Lista operatorów rzutowania

const_cast – rzutowanie usuwające modyfikatory **const** oraz **volatile**. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań bezpiecznych.

static_cast – używany do zdefiniowanych przez użytkownika, standardowych lub niejawnych konwersji typów. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań w zasadzie bezpiecznych.

reinterpret_cast – jest najbardziej niebezpiecznym rzutowaniem. Wykonuje on konwersję między wskaźnikami oraz wskaźnikami i liczbami. Źle przeprowadzone rzutowanie może być źródłem błędów trudnych do wykrycia.

dynamic_cast – obsługuje tylko obiekty klas polimorficznych. Zapewnia realizację rzutowania "w górę". Rzutowanie to należy do rzutowań bezpiecznych.

Lista operatorów rzutowania

const_cast – rzutowanie usuwające modyfikatory **const** oraz **volatile**. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań bezpiecznych.

static_cast – używany do zdefiniowanych przez użytkownika, standardowych lub niejawnych konwersji typów. Całość konwersji realizowana jest przez kompilator. Rzutowanie to należy do rzutowań w zasadzie bezpiecznych.

reinterpret_cast – jest najbardziej niebezpiecznym rzutowaniem. Wykonuje on konwersję między wskaźnikami oraz wskaźnikami i liczbami. Źle przeprowadzone rzutowanie może być źródłem błędów trudnych do wykrycia.

dynamic_cast – obsługuje tylko obiekty klas polimorficznych. Zapewnia realizację rzutowania "w górę". Rzutowanie to należy do rzutowań bezpiecznych.

Plan prezentacji

Zarys problemu

Należy stworzyć zbiór klas modelujących pojęcie prostej w przestrzeni i na płaszczyźnie. Ich metody powinny umożliwić w łatwy sposób określenie, czy dany punkt w przestrzeni lub na płaszczyźnie należy do danej prostej czy też nie.

Te same nazwy, różne definicje

`prosta-plaszczyzna.hh`

```
class Wektor {
      double  _x, _y;
      ...
};

class Prosta {
      Wektor  _WekNorm;
      double      _c;
      ...
};
```

`prosta-przestrzen.hh`

```
class Wektor {
      double  _x, _y, _z;
      ...
};

class Prosta {
      Wektor  _WekOsiowy;
      Wektor  _PnkProstej;
      ...
};
```

```
#include "prosta-plaszczyzna.hh"
#include "prosta-przestrzen.hh"

int main( )
{
    Prosta  Pr2;
    Prosta  Pr3;
    ...
}
```

Te same nazwy, różne definicje

prosta-plaszczyzna.hh

```
class Wektor {
      double  _x, _y;
      ...
};

class Prosta {
      Wektor  _WekNorm;
      double      _c;
      ...
};
```

prosta-przestrzen.hh

```
class Wektor {
      double  _x, _y, _z;
      ...
};

class Prosta {
      Wektor  _WekOsiowy;
      Wektor  _PnkProstej;
      ...
};
```

```
#include "prosta-plaszczyzna.hh"
#include "prosta-przestrzen.hh"

int main( )
{
    Prosta  Pr2;
    Prosta  Pr3;
    ...
}
```

Zroznicownie nazw

prosta2d.hh

```
class Wektor2D {
      double   _x, _y;
      ...
};

class Prosta2D {
      Wektor2D   _WekNorm;
      double     _c;
      ...
};
```

prosta3d.hh

```
class Wektor3D {
      double   _x, _y, _z;
      ...
};

class Prosta3D {
      Wektor3D   _WekOsiowy;
      Wektor3D   _PnkProstej;
      ...
};
```

```
#include "prosta2d.hh"
#include "prosta3d.hh"

int main( )
{
    Prosta2D  Pr2;
    Prosta3D  Pr3;
    ...
}
```

Zroznicownie nazw w stylu C

d2_prosta.hh

```
class D2_Wektor {
      double   _x, _y;
      ...
};

class D2_Prosta {
      D2_Wektor   _WekNorm;
      double      _c;
      ...
};
```

d3_prosta.hh

```
class D3_Wektor {
      double   _x, _y, _z;
      ...
};

class D3_Prosta {
      D3_Wektor   _WekOsiowy;
      D3_Wektor   _PnkProstej;
      ...
};
```

```
#include "d2_prosta.hh"
#include "d3_prosta.hh"

int main( )
{
    D2_Prosta  Pr2;
    D3_Prosta  Pr3;
    ...
}
```

Przestrzenie nazw

d2_prosta.hh

```
namespace D2 {
   class Wektor {
         double   _x, _y;
         ...
   };

   class Prosta {
         Wektor   _WekNorm;
         double        _c;
         ...
   };
}
```

d3_prosta.hh

```
namespace D3 {
   class Wektor {
         double   _x, _y, _z;
         ...
   };

   class Prosta {
         Wektor   _WekOsiowy;
         Wektor   _PnkProstej;
         ...
   };
}
```

```
#include "d2_prosta.hh"
#include "d3_prosta.hh"

int  main( )
{
    D2::Prosta  Pr2;
    D3::Prosta  Pr3;
    ...
}
```

Plan prezentacji

Blok przestrzeni nazw

```
int  ZmGlobalna;

namespace D3 {
      class Wektor {
            double   _x, _y, _z;
            ...
      };


      class Prosta {
            Wektor   _WekOsiowy;
            Wektor   _PnkProstej;
            ...
       };
}

int  main( )
{
       ZmGlobalna = 4;
       D3::Wektor   Wk;
       D3::Prosta    Pr;
}
```

W danej jednostce translacyjnej można stworzyć jednolity blok grupujący wszystkie definicje w wybranej przewstrzeni nazw.

Blok przestrzeni nazw

```
int  ZmGlobalna;

namespace D3 {
      class Wektor {
            double   _x, _y, _z;
            ...
      };
}
namespace D3 {
      class Prosta {
            Wektor   _WekOsiowy;
            Wektor   _PnkProstej;
            ...
       };
}

int  main( )
{
       ZmGlobalna = 4;
       D3::Wektor   Wk;
       D3::Prosta    Pr;
}
```

Tę samą przestrzeń nazw można jednak rozbić na dowolną liczbę bloków.

Ta sama przestrzeń nazw w różnych modułach

`d3_wektor.hh`

```
...
namespace D3 {
   class Wektor {
         double   _x, _y, _z;
         ...
   };
}
...
```

`d3_prosta.hh`

```
...
#include "d3_wektor.hh"

namespace D3 {
   class Prosta {
         Wektor   _WekOsiowy;
         Wektor   _PnkProstej;
         ...
   };
}
...
```

```
#include "d3_wektor.hh"
#include "d3_prosta.hh"

int main( )
{
    D3::Wektor   Wk;
    D3::Prosta   Pr;
    ...
}
```

Ta sama przestrzeń nazw może być *rozbita* na zbiór różnych jednostek translacyjnych (np. `3d_wektor.cpp` i `3d_prosta.cpp`) z własnymi plikami nagłówkowymi (np. `3d_wektor.hh` i `3d_prosta.hh`) zawierającymi definicje klas i zapowiedzi definicji metod.

Co dają przestrzenie nazw

- Definiowanie przestrzeni nazw pozwala rozwiązać konflikty nazw. Tym samym daje znacznie większą swobodę programiście.
- Definicja przestrzeni nazw może być „kontynuowana” w kilku jednostek translacyjnych.
- Jedną z ważniejszych konsekwencji wykorzystywania przestrzeni nazw jest możliwość *tworzenia modułów w sensie logicznym*, a nie tak jak to ma miejsce, np. w języku *C* tylko w sensie stricte fizycznym.

Co dają przestrzenie nazw

- Definiowanie przestrzeni nazw pozwala rozwiązać konflikty nazw. Tym samym daje znacznie większą swobodę programiście.
- Definicja przestrzeni nazw może być „kontynuowana” w kilku jednostek translacyjnych.
- Jedną z ważniejszych konsekwencji wykorzystywania przestrzeni nazw jest możliwość *tworzenia modułów w sensie logicznym*, a nie tak jak to ma miejsce, np. w języku *C* tylko w sensie stricte fizycznym.

Co dają przestrzenie nazw

- Definiowanie przestrzeni nazw pozwala rozwiązać konflikty nazw. Tym samym daje znacznie większą swobodę programiście.
- Definicja przestrzeni nazw może być „kontynuowana” w kilku jednostek translacyjnych.
- Jedną z ważniejszych konsekwencji wykorzystywania przestrzeni nazw jest możliwość *tworzenia modułów w sensie logicznym*, a nie tak jak to ma miejsce, np. w języku *C* tylko w sensie stricte fizycznym.

Dostęp do przesłoniętych zmiennych globalnych

```
int  Zm = 7;

void  Funkcja( )
{
     int  Zm = 55;
     ::Zm = Zm;
}

int  main( )
{
     cout <<  "Zm. Globalna: " <<  Zm <<  endl;
     Funkcja( );
     cout <<  "Zm. Globalna: " <<  Zm <<  endl;
}
```

Dostęp do przesłoniętych zmiennych globalnych

```
int Zm = 7;

void Funkcja( )
{
    int Zm = 55;
    ::Zm = Zm;
}

int main( )
{
    cout << "Zm. Globalna: " << Zm << endl;
    Funkcja( );
    cout << "Zm. Globalna: " << Zm << endl;
}
```

```
Wynik działania:
 Zm. Globalna: 7
 Zm. Globalna: 55
```

W nienazwanej przestrzeni nazw znajdują się wszystkie globalnie definiowane zmiennych, typy oraz funkcje. Zmienne te można zawsze "odsłonić" stosując nazwę kwalifikowaną.

Dostęp do przesłoniętych zmiennych globalnych

```
int Zm = 7;

void Funkcja( int Zm )
{
    ::Zm = Zm;
}

int main( )
{
    cout << "Zm. Globalna: " << Zm << endl;
    Funkcja( 10 );
    cout << "Zm. Globalna: " << Zm << endl;
}
```

```
Wynik działania:
 Zm. Globalna: 7
 Zm. Globalna: 10
```

W nienazwanej przestrzeni nazw znajdują się wszystkie globalnie definiowane zmiennych, typy oraz funkcje. Zmienne te można zawsze "odsłonić" stosując nazwę kwalifikowaną.

Dostęp do przesłoniętych typów

```
enum  Symbole { a, b, c};

void  Funkcja_LokalnyTyp( Symbole Param )
{     enum  Symbole { x, y, z };
      Symbole   Zm1 = x;
      ::Symbole   Zm2 = Param;
      ::Symbole   Zm3 = ::a;
}

int  main( )
{     Symbole   x = a;
      Funkcja_LokalnyTyp( x );
}
```

```
Wynik działania:
 Zm. Globalna: 7
 Zm. Globalna: 10
```

W nienazwanej przestrzeni nazw znajdują się wszystkie globalnie definiowane zmiennych, typy oraz funkcje. Zmienne te można zawsze "odsłonić" stosując nazwę kwalifikowaną.

Zagnieżdżanie przestrzeni nazw

```
int  ZmGlobalna;

namespace D3 {
      class Wektor {
            ...
      };
      namespace Bryly {
            class Szescian {
                  ...
            };
      }
}
```

```
int  main( )
{
    ZmGlobalna = 4;
    D3::Wektor          Wk;
    D3::Bryly::Szescian  Sz;
}
```

Przestrzenie nazw mogą być zagnieżdżane.

Dyrektywa using namespace

```
int  ZmGlobalna;

namespace D3 {
      class Wektor {
            ...
      };

      namespace Bryly {
            class Szescian {
                  ...
            };
      }
}

using namespace D3;

int  main( )
{
    ZmGlobalna = 4;
    Wektor            Wk;
    Bryly::Szescian   Sz;
}
```

Użycie dyrektyw **using namespace** powoduje przeniesienie wszystkich nazw z danej podrzestrzeni, do której się ona odnosi, do przestrzeni nazw, w której jest zastosowana.

W przedstawionym przykładzie nazwy z podprzestrzeni D3 zostały przeniesione do przestrzeni nienazwanej związanej z tradycyjnymi zmiennymi globalnymi.

Użycie dyrektywy **using namespace**

Dyrektywy **using namespace** nie należy używać w plikach nagłówkowych. Jej użycie tam jest przejawem bardzo złego stylu programowania.

Dyrektywa using namespace

```
int ZmGlobalna;

namespace D3 {
    class Wektor {
        ...
    };

    namespace Bryly {
        class Szescian {
            ...
        };
    }
}

using namespace D3::Bryly;

int main( )
{
    ZmGlobalna = 4;
    D3::Wektor      Wk;
    Szescian        Sz;
}
```

Dyrektywę **using namespace** możemy stosować selektywnie do wybranych podprzestrzeni.

W tym przykładzie została ona zastosowana do podprzestrzeni `D3::Bryly`. Do pozostałych elementów przestrzeni nazw `D3` musimy odwoływać się poprzez pełną nazwę kwalifikowaną.

Dyrektywa using namespace

```
namespace D3 {
    class Wektor {
        ...
    };

    namespace Bryly {
        class Szescian {
            ...
        };
        Szescian    SzesWzorcowy;
    }
    using namespace Bryly;
    Szescian    DrugiSzes;
}


int  main( )
{
   D3::DrugiSzes = D3::SzesWzorcowy;
}
```

Dyrektywę **using namespace** możemy stosować również w podprzestrzeniach. Jej działanie ograniczone jest tylko do danej podprzestrzeni.

W tym przykładzie dyrektywa **using namespace** powoduje przeniesienie wszystkich nazw z podprzestrzeni `Bryla` do przestrzeni nazw `D3`.

Dyrektywa using namespace

```
namespace D3 {
    class Wektor {
        ...
    };
    namespace Bryly {
        class Szescian {
            ...
        };
        Szescian    SzesWzorcowy;
    }

    Bryly::Szescian    DrugiSzes;
}
```

Bez tej dyrektywy nie *stracimy* podprzestrzeni `Bryly` i niezbędne jest wówczas odwoływanie się do jej poszczególnych elementów poprzez nazwy kwalifikowane.

```
int  main( )
{
   D3::DrugiSzes = D3::Bryly::SzesWzorcowy;
}
```

Deklaracja **using**

```
namespace D3 {
    class Wektor {
        ...
    };

    namespace Bryly {
        class Szescian {
            ...
        };
        Szescian    SzesWzorcowy;
    }

    Bryly::Szescian    DrugiSzes;
}
using D3::Bryly::SzesWzorcowy;

int  main( )
{
   D3::DrugiSzes = SzesWzorcowy;
}
```

Elementy danej podprzestrzeni nazw możemy przenosić selektywnie stosując deklarację **using**.

Dyrektywa **using namespace** – ograniczanie zakresu

```
namespace D3 {
    class Wektor {
        ...
    };
    namespace Bryly {
        class Szescian {
            ...
        };
        Szescian    SzesWzorcowy;
    }
    Bryly::Szescian    DrugiSzes;
}

int  main( )
{
   {
     using namespace D3;
     DrugiSzes = Bryly::SzesWzorcowy;
   }
   D3::DrugiSzes = D3::Bryly::SzesWzorcowy;
}
```

Ograniczenie użycia dyrektywy **using namespace** możemy zrealiwać poprzez jej wykorzystanie w bloku ciała funkcji. Ogranicza on wówczas ważność stosowania tej deklaracji do tego bloku.

Deklaracja **using** – ograniczanie zakresu

```
namespace D3 {
    class Wektor {
        ...
    };
    namespace Bryly {
        class Szescian {
            ...
        };
        Szescian    SzesWzorcowy;
    }
    Bryly::Szescian    DrugiSzes;
}

int  main( )
{
   {
     using D3::Bryly::SzescWzorcowy;
     D3::DrugiSzes = SzesWzorcowy;
   }
   D3::DrugiSzes = D3::Bryly::SzesWzorcowy;
}
```

Analogicznie można ograniczyć zakres ważności deklaracji **using**.

Użycie deklaacji **using**

W plikach nagłówkowych nie powinno używać się zarówno dyrektywy **using namespace** jak też deklaracji **using**.

Plan prezentacji

Tradycyjny sposób obsługi błędów

```
int Funkcja1()
{  . . .
   if  (Gdy_cos_jest_zle)  { errno = KOD_BLEDU;  return  -1; }
    . . .
   return  0;
}
. . .

int Funkcja()
{
   int err;
   if  (Funkcja1() < 0) {  /* Obsługa  błędu  */
      . . .
      if  (Jezeli_nie_mozemy_sobie_poradzic)  return  -1;
   . . .
   }
   if  ((err = Funkcja2()) < 0) {  /* Obsługa  błędu */  . . .  }
   . . .
   return  0;
}
```

Czym jest obsługa wyjątków

- Wyjątki w C++ są sposobem na oddzielenie wykrywania błędów od ich obsługi.
- Mechanizm obsługi wyjątków jest nielokalną strukturą sterującą, która korzysta ze *zwijania stosu*.

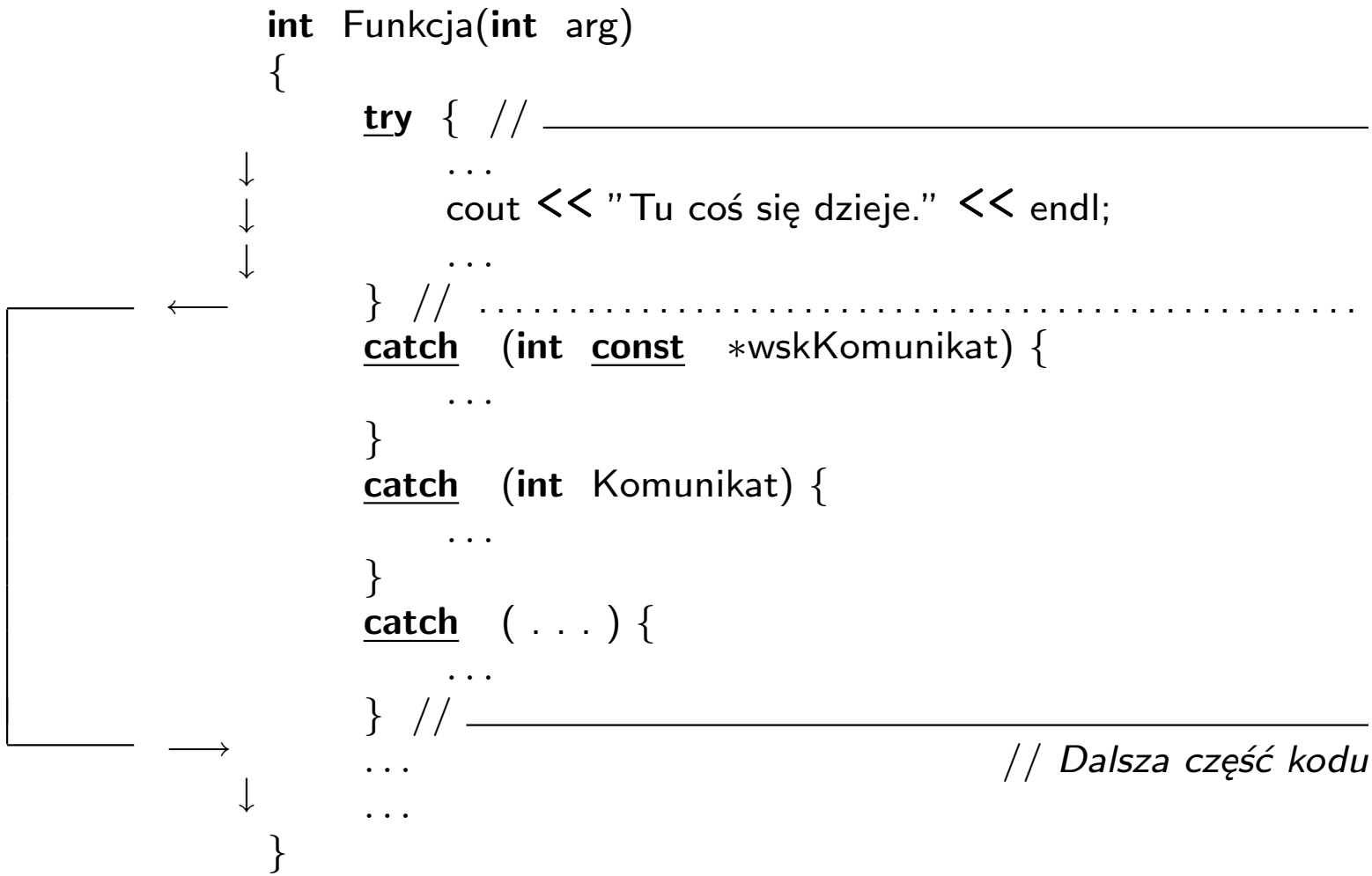
  Zwijaniem stosu nazywamy proces przeszukiwania stosu w celu znalezienia procedury obsługi wyjątku.
- Mechanizmy obsługi wyjątków służą do sygnalizowania i obsługi zdarzeń wyjątkowych. Jednak to programista musi zdecydować co to znaczy *wyjątkowy* w danym programie.
- Wyjątki często w sposób naturalny tworzą rodziny. Oznacza to, że dziedziczenie może być użyteczne w strukturalizacji wyjątków i ich obsłudze.

Plan prezentacji

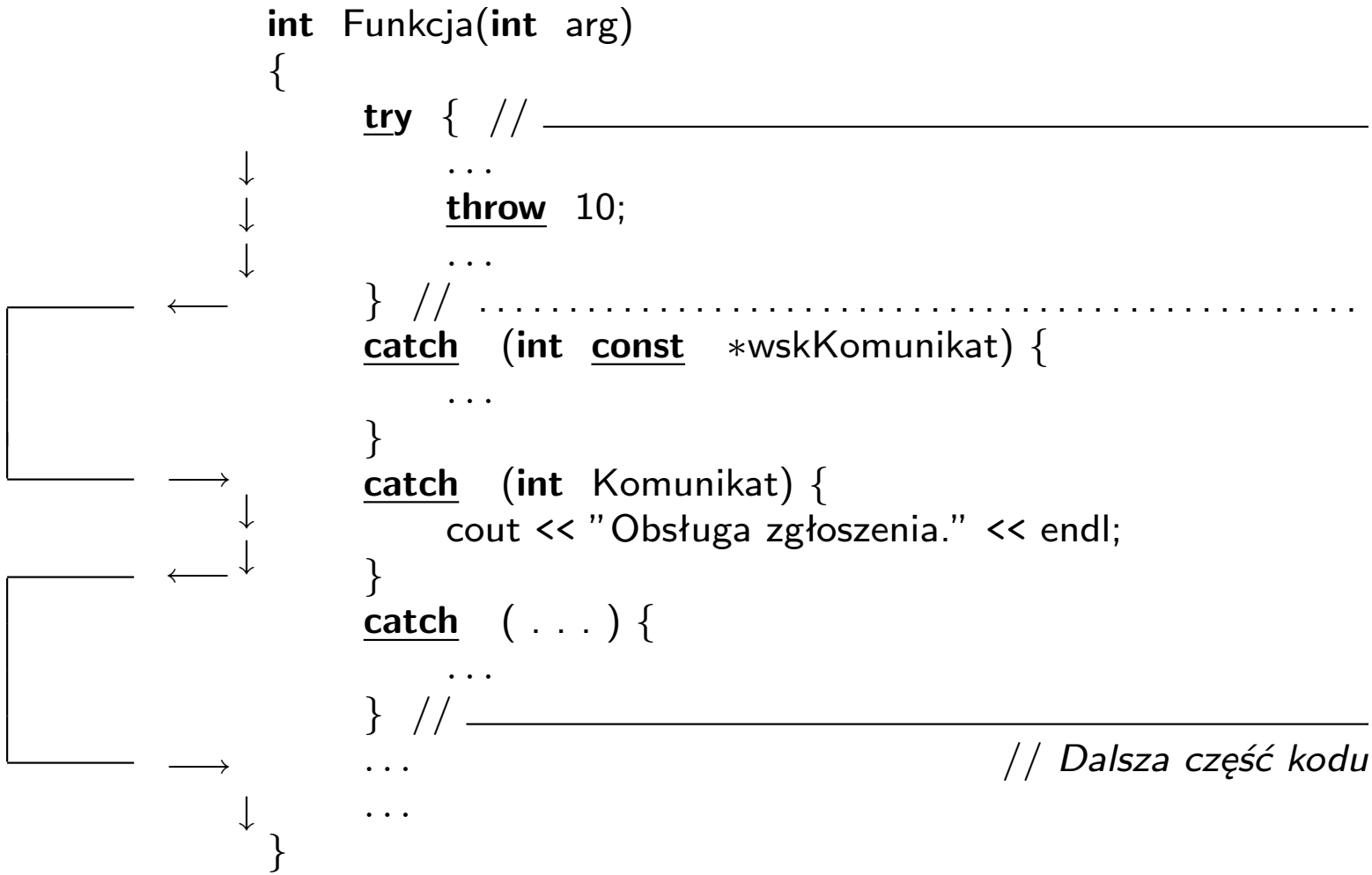
Obsługa wyjątków – trochę tradycyjnie

```
int Funkcja1( )
{
  ...
  if (Gdy_cos_jest_zle) throw KOD_BLEDU;   // ..... Tutaj zgłaszamy wyjątek ←
  ...
}
...
int Funkcja( )
{
  try { // ..... W tej sekcji zakładamy możliwość zgłoszeń wyjątki
    Funkcja1( );
    ...
  } // .....
  catch (int err) { // .. Tutaj rozpoczyna się przechwytywanie i obsługa wyjątków
    switch (err) {
      case KOD_BLEDU:     ...
                  if (Jezeli_nie_mozemy_sobie_poradzic) throw KOD_BLEDU;
                   ...
                  break ;
        ...
    }
  } // .....
  ...
```

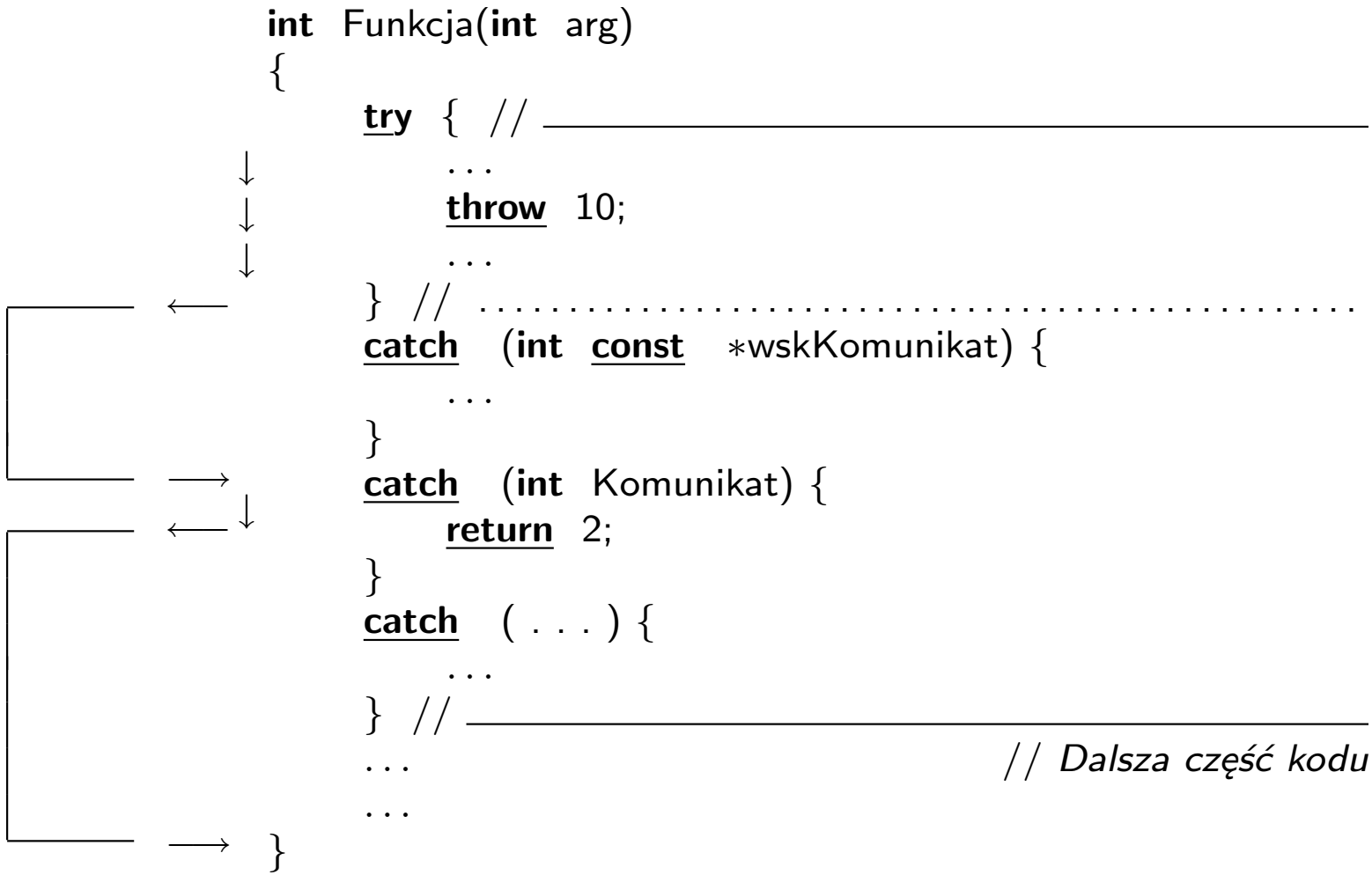
Przepływ sterowania – przypadek braku zgłoszenia wyjątku



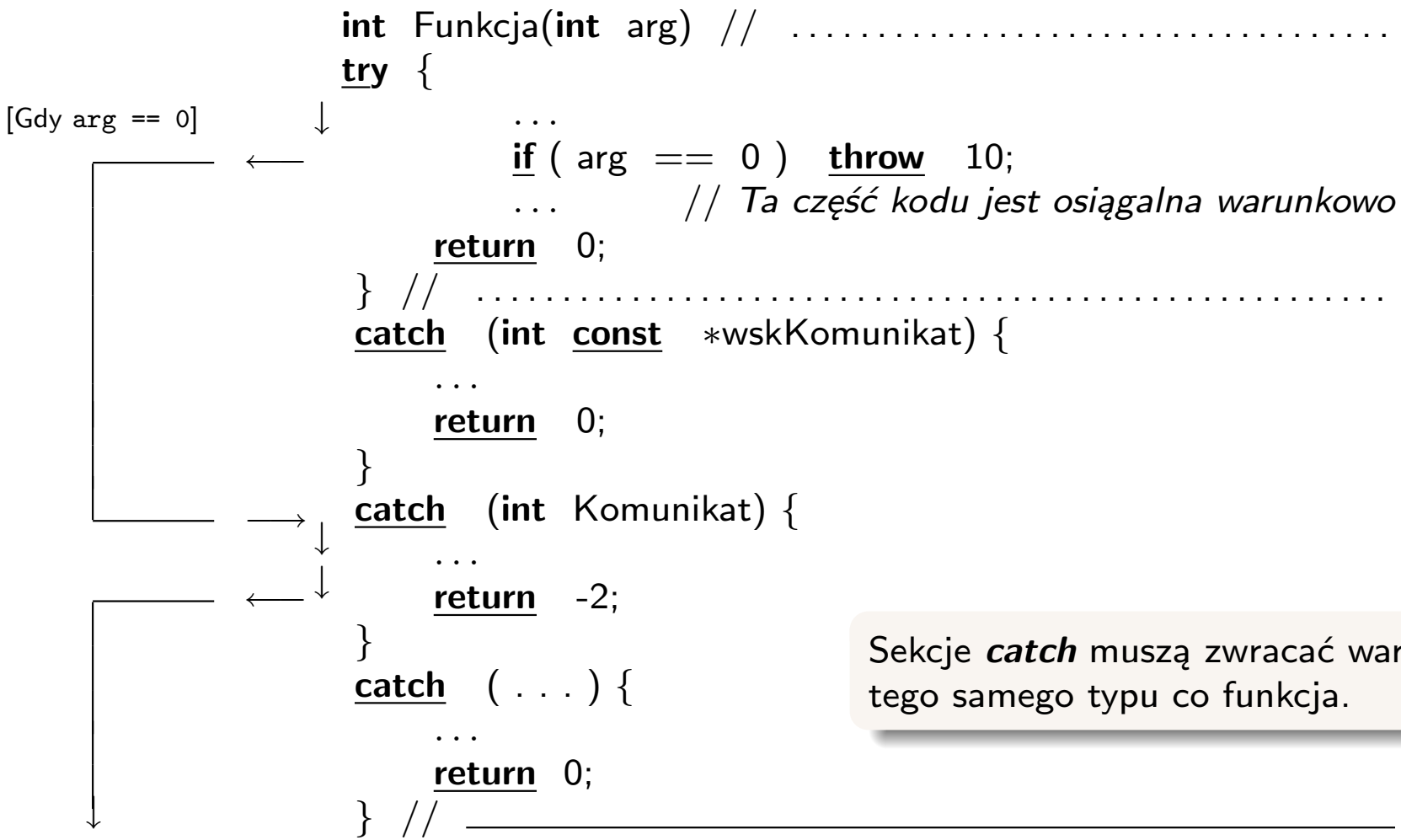
Przepływ sterowania – przypadek zgłoszenia wyjątku



Zgłoszenia wyjątku – wcześniejszy powrót z funkcji



Funkcja w sekcji **try**



Sekcje ***catch*** muszą zwracać wartość tego samego typu co funkcja.

Problem kopiowania zgłaszanej wartości

```
class Blad {
        Blad( Blad const  & ) { }
    public :
        int  _nr_bledu;
        Blad(int  nr = 0) { _nr_bledu = nr; };
};

void  Funkcja(int  arg)
{
    try  {
        ...
        throw  Blad(3);
        ...
    }
    catch  ( Blad  &Bl ) {
        cout <<"Nr Bledu: " << Bl._nr_bledu << endl;
    }
    ...
}
```

Kopiowanie obiektu jest niemożliwe (poza tą klasą)

Problem kopiowania zgłaszanej wartości

```
class Blad {
        Blad( Blad const & ) { }
    public :
        int _nr_bledu;
        Blad(int nr = 0) { _nr_bledu = nr; };
};

void Funkcja(int arg)
{
    try {
        ...
        throw Blad(3);
        ...
    }
    catch ( Blad &Bl ) {
        cout <<"Nr Bledu: " << Bl._nr_bledu << endl;
    }
    ...
}
```

Kopiowanie obiektu jest niemożliwe (poza tą klasą)

Zgłoszenie wyjątku powoduje zawsze przekopiowanie parametru polecenia *throw*.

Ominięcie problemu kopiowania obiektu

```
class Blad {
        Blad( Blad const & ) { }
    public :
        int _nr_bledu;
        Blad(int nr = 0) { _nr_bledu = nr; };
};

void Funkcja(int arg)
{
    try {
        ...
        throw new Blad(3);
        ...
    }
    catch ( const Blad* wB ) {
        cout <<"Nr Bledu: " << B->_nr_bledu << endl;
        delete wB;
    }
    ...
}
```

Kopiowanie obiektu jest niemożliwe (poza tą klasą)

W tym przypadku kopiowany jest jedynie wskaźnik.

Zwalnianie zasobów

```
void Funkcja(int arg)
{
    FILE *wPolecenie = popen("last -n 100","r");

    try {
        ...
        ...
    }
    catch ( ... ) {
        pclose(wProc);
        throw ;
    }
    pclose(wProc);
}
```

W przypadku zasobów, które nie mają charakteru lokalnego (np. dynamicznie przydzielana pamięć), należy pamiętać, aby zwalniać je w trakcie obsługi zgłoszonego wyjątku.

Wyjątki inaczej

```
#include <iostream>
#include <vector>
#include <stdexcept>      //Tu znajduje się definicje wyjątku std::out_of_range

int PrzegladajTablice( const std::vector<int> &Tablica )
{
    try {
        for (int ind = 0; ; ++ind) {
            if (Tablica.at(ind) / 2) std::cout << Tablica.at(++ind) << std::endl;
        }
    }
    catch (std::out_of_range) {
        return 0;
    }
    return 0;
}
```

Zgłoszenie wyjątku może być sposobem na przerwanie pętli w sytuacji, gdy bezpo"srednie sprawdzanie warunku jest kłopotliwe w zapisie i sprawdzenie to musiałoby występować w wielu miejscach.
Zgłoszenie wyjątku może być pomocne również w przypadkach konstrukcji niestrukturalnych, w których wykorzystywano instrukcję **goto**.

Konwersje typów w obsłudze wyjątków

```
class Wektor2f { //.....
    public :
        ...
      Wektor2f( );
}; //.....

class LZespolona { //.....
    public :
        ...
    LZespolona( );
    LZespolona(const Wektor2f &W);
}; //.....

int main()
{
    try { throw Wektor2f( ); }
    catch (LZespolona Z) {          // Czy wyjątek zgłoszony powyżej zostanie tu obsłużony?
        ...
    }
}
```

Przy obsłudze wyjątków domyślne konwersje nie są realizowane.

Plan prezentacji

Wyjątki standardowe

Standardowe wyjątki zgłaszane przez język		
Nazwa	Zgłaszane przez	Nagłówek
bad_alloc	`new`	<new>
bad_cast	`dynamic_cast`	<typeinfo>
bad_typeid	`typeid`	<typeinfo>
bad_exception	`specyfikacja wyjątku`	<exception>

Standardowe wyjątki zgłaszane przez bibliotekę		
Nazwa	Zgłaszane przez	Nagłówek
out_of_range	`at()`	<stdexcept>
	`bitset< >::operator []()`	<stdexcept>
invalid_argument	`konstruktor bitset`	<stdexcept>
overflow_error	`bitset< >::to_ulong()`	<stdexcept>
ios_base::failure	`ios base::clear()`	<ios>

Ustawianie handlera dla obsługi wyjątków

unexpected_handler **set_unexpected**(unexpected_handler fun) throw();

Wyjątki dla operacji wejścia/wyjścia

```
int main ( ) {
    ifstream Plik;
    int Liczba;

    Plik.exceptions( ifstream ::failbit | ifstream ::badbit );
    try {
        Plik.open("zbior_liczba.txt");

        while (!Plik.eof()) Plik >> Liczba;
    }
    catch (ifstream ::failure) {
        cout << "Wyjątek operacji otwarcia lub czytania";
    }

    return 0;
}
```

Koniec prezentacji
Dziękuję za uwagę