

# Podstawy programowania 2

dr inż. Piotr Szwed  
Katedra Informatyki Stosowanej  
C2, pok. 403

e-mail: [pszwed@agh.edu.pl](mailto:pszwed@agh.edu.pl)

<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 25.04.2021

# 7. Operatory rzutowania i RTTI

# Operatory rzutowania w C++

Operatory rzutowania muszą być stosowane, jeżeli zachodzi konieczność zmiany typu zmiennych lub stałych, a równocześnie kompilator nie jest w stanie dokonać konwersji typów w sposób automatyczny.

Konwersje **automatyczne** dotyczą przede wszystkim:

- konwersji pomiędzy typami całkowitoliczbowymi
- konwersji typów całkowitoliczbowych do zmiennoprzecinkowych
- konwersji wskaźników lub referencji klas potomnych do klas bazowych.
- konwersji wskaźników lub referencji bez modyfikatora `const` do wskaźników (referencji) typu `const`.

W innych przypadkach konwersja typów musi być dokonana jawnie przez skorzystanie z operatorów rzutowania.

# Operatory rzutowania w C++

Składnia podstawowych operatorów rzutowania w C/C++ jest następująca:

(TYPE)expression [C i C++]

lub

TYPE(expression) [tylko C++]

```
void moveto(int x,int y);
void lineto(int x,int y);

void drawCircle(double cx,double cy, double radius){
    for(double angle=0;angle<=2*M_PI;angle+=M_PI/32){
        int x=int(cx+cos(angle)*radius);
        int y=int(cy+sin(angle)*radius);
        if(angle==0)moveto(x,y);
        else lineto(x,y);
    }
}
```

# Operatory rzutowania w C++

Twórcy języka C++ uznali, że rzutowanie jest operacją potencjalnie niebezpieczną – i dlatego warto oznaczyć miejsca, w których następuje rzutowanie specjalnymi operatorami o zmienionej składni. Operatory te są także bardziej elastyczne, między innymi możliwe jest sprawdzanie poprawności rzutowania.

Nowe operatory w C++ to

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`

# Operator `static_cast`

Składnia:

```
static_cast<TYPE>(expr)
```

TYPE – symbol docelowego typu

expr – wyrażenie lub stała

Operator konwertuje wyrażenie `expr` do typu `TYPE`. Możliwość poprawnego zastosowania operatora jest sprawdzana *statycznie* – w trakcie kompilacji. Operator jest przeznaczony do konwersji pomiędzy typami wbudowanymi, wskaźnikami do klas należącej do wspólnej hierarchii oraz wskaźnikami typu `void*`.

## Przykład 1

```
int*ptr=static_cast<int*>( malloc(20*sizeof(int)) );  
double x=2.7;  
int ix=static_cast<int>( x ); // obetnie .7  
cout<<ix<<endl; //wypisze 2
```

# Operator static\_cast

## Przykład 2

```
class Base{};
class Derived:public Base {};

int main(){
    Derived d;
    Base*pb=&d;
    Derived*pd=static_cast<Derived*>( pb );
}
```

## Przykład 3

```
int main()
{
    Base*ptrToBase=new Base();
    int *ptrToInt=static_cast<int*>(ptrToBase);
}
```

```
error: invalid static_cast from type 'Base*' to type 'int*'
int *ptrToInt=static_cast<int*>(ptrToBase);
                        ^
```

# Operator `const_cast`

Operator pozwala na odrzucenie modyfikatora `const` lub `volatile`. Analogiczny efekt można *bezwiednie* uzyskać stosując standardowy operator rzutowania.

Usuwanie modyfikatora `const` jest potencjalnie niebezpieczne, ponieważ wyłącza on standardowy mechanizm sprawdzania prawa do modyfikacji danych. Z tego powodu taką zmianę lepiej jest świadomie oznaczyć specjalnym operatorem.

Składnia:

```
const_cast<TYPE>(expr)
```



# Operator const\_cast

```
void read(char*file){
    cout<<"forget to declare \'const char*file\'"<<endl;
    ifstream ifs(file);
    // ...
}

int main(){
    //volatile // 1
    const int x = 5;
    int*px=&x; // 2 błąd kompilacji
    int*cc_px=const_cast<int*>(&x); // 3
    *cc_px=7;
    cout<<"x="<<x<<" *cc_px="<<*cc_px<<endl;
    read("plik.txt"); // 4 błąd kompilacji
    read(const_cast<char*>("plik.txt")); //5
}
```

## Bez volatile

x=5 \*cc\_px=7

forget to declare 'const char\*file'

## Z volatile

x=7 \*cc\_px=7

forget to declare 'const char\*file'

Instrukcje //2 i //4 nie zostaną skompilowane. Poprawne rzutowanie (usunięcie const) w instrukcjach //3 i //4. Bez volatile: kompilator zakłada, że wartość x nie zmienia się od inicjalizacji. Z volatile: wartość x jest odczytywana podczas każdego dostępu

# Operator reinterpret\_cast

Operator jest najbardziej niebezpieczny z całej grupy i potencjalnie może być źródłem błędów i pojawiających się przy próbach przeniesienia programu na inne platformy.

Zmienia on *interpretację* danych traktując je jak sekwencje bitów należących do zupełnie innego typu danych.

Składnia:

```
reinterpret_cast<TYPE>(expr)
```

Jego użycie wymaga wiedzy o rozmiarach danych i sposobie ich rozmieszczania przez kompilator.

# Operator reinterpret\_cast

```
struct coord {double x,y,z};

void print(const struct coord*c){
    printf("[%g,%g,%g]",c->x,c->y,c->z);
}

int main(){
    struct coord c={3,2,1};
    print(&c);
    double*cd=reinterpret_cast<double*>(&c); //(1)
    cd[0]=1;
    cd[1]=2;
    cd[2]=3;
    struct coord* pc= reinterpret_cast<struct coord*>(cd); //(2)
    print(pc);
}
```

Wynik :  
[3,2,1][1,2,3]

Pierwsze wywołanie operatora //(1) rzutuje wskaźnik do struktury struct coord na tablicę typu double\*. Modyfikacja danych następuje wewnątrz tablicy. W instrukcji //(2) 3-elementowa tablica cd jest z powrotem rzutowana na wskaźnik do struktury.

# Operator reinterpret\_cast

```
int main(){
    char b[4];
    b[0]=1;
    b[1]=2;
    b[2]=3;
    b[3]=4;
    int*pInt=reinterpret_cast<int*>(b);
    cout<<*pInt<<" ?= "<<
        b[0]+256*b[1]+256*256*b[2]+256*256*256*b[3]<<endl;
}
```

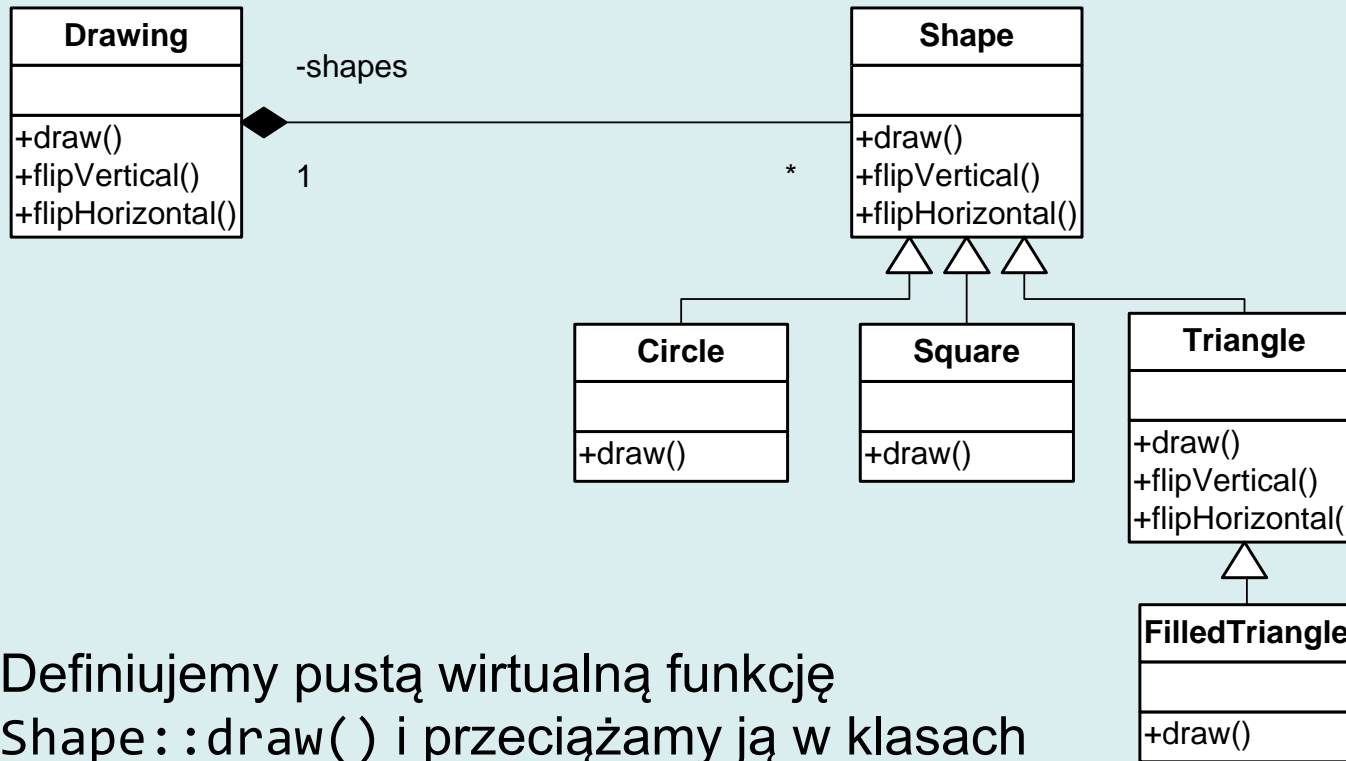
67305985 ?= 67305985

Poprzez rzutowanie pamięć 4 znaków w tablicy b interpretowana jest, jak liczba typu int.

# Downcasting

- Termin *downcasting* oznacza *rzutowanie w dół*, czyli konwersję wskaźnika lub referencji do typu bazowego w hierarchii klas do typu potomnego. Określenie „w dół” wynika z tego, że zazwyczaj na diagramach struktur dziedziczenia klasy potomne są umieszczane poniżej klas bazowych.
- Rzutowanie w dół powinno być zjawiskiem występującym stosunkowo rzadko w dobrze skonstruowanej hierarchii klas.
- Zazwyczaj pragnąc dodać jakąś funkcjonalność do istniejącej hierarchii, dodajemy wirtualną metodę w klasie bazowej i przeciążamy ją w klasach pochodnych.

# Przykład 1



1. Definiujemy pustą wirtualną funkcję `Shape::draw()` i przeciążamy ją w klasach `Circle`, `Square`, `Triangle`, `FilledTriangle`.
2. Definiujemy pustą wirtualną funkcję `Shape::flipVertical()` i przeciążamy ją w klasie `Triangle`. (Dla klas `Circle` i `Square` zastosowanie metody nie ma sensu, klasa `FilledTriangle` odziedziczy implementację po `Triangle`).

# Przykład 1

## 3. Implementujemy metody klasy Drawing

```
Drawing::draw(){
    for (each shape in shapes )shape.draw()
}
Drawing::flipVertical(){
    for (each shape in shapes )shape.flipVertical()
}
```

### Zalety

- Naturalne w językach obiektowych wykorzystanie polimorfizmu.

### Wady

- Zazwyczaj tak prowadzone projektowanie obiektowe prowadzi do bardzo szerokiego interfejsu klasy bazowej stojącej u szczytu hierarchii. Duża grupa klas dziedziczy puste implementacje.
- W przypadku gotowej hierarchii klas dostarczonej przez zewnętrznego producenta oprogramowania w postaci biblioteki nie jest możliwe rozszerzanie interfejsu klasy bazowej.

## Przykład 2

Organizacja zapisu rysunku w sytuacji, kiedy brak jest odpowiednich funkcji w hierarchii klas.

```
writeXX(stream, Circle&);  
writeXX(stream, Square&);  
writeXX(stream, Triangle&);  
  
Drawing::saveAsXX(stream)  
{  
    for (each shape in shapes) {  
        if (shape is Circle)  
            writeXX(stream, (Circle&)shape)  
        if (shape is Square)  
            writeXX(stream, (Square&)shape)  
        // itd.  
    }  
}
```

Kluczowym elementem jest określenie typu obiektu, kiedy wiemy jedynie, że należy do klasy bazowej: `if (shape is Circle)`



# RTTI – Run Time Type Information

RTTI jest mechanizmem, który pozwala na określenie rzeczywistego typu obiektu wskazywanego przez wskaźnik lub referencję typu bazowego.

Możliwe są tu dwie implementacje:

- jawne wprowadzenie do klas kodu odpowiedzialnego za RTTI
- obsługa automatyczna oparta na wykorzystaniu informacji o typie zapisanych w VTABLE

# Przykład manualnej implementacji RTTI

## Wirtualna funkcja zwracająca typ

Jest to jedno z najczęściej spotykanych rozwiązań.

1. Definiujemy zbiór stałych, np.: CIRCLE, SQUARE, TRIANGLE.
2. W klasie bazowej definiujemy czystą wirtualną funkcję

```
virtual int getType()const=0;
```

i przeciążamy ją w klasach potomnych:

```
int Circle::getType()const{return CIRCLE;}
```

```
int Triangle::getType()const{return TRIANGLE;}
```

# Automatyczna implementacja RTTI

Automatyczna obsługa RTTI pojawiła się w kompilatorach stosunkowo późno, dlatego nie jest jeszcze powszechnie stosowana – najczęściej w bibliotekach można spotkać jawne implementacje RTTI połączone ze statycznymi operatorami rzutowania.

Przesłanką za wprowadzeniem automatycznej obsługi RTTI na poziomie języka była powszechność tego typu mechanizmów w różnych bibliotekach (i równocześnie brak kompatybilności pomiędzy rozwiązaniami różnych producentów).

Włączenie obsługi RTTI wiąże się z dodatkowym narzutem czasowym i niewielkim wzrostem objętości wygenerowanego kodu.

Wbudowana obsługa RTTI jest dostępna za pośrednictwem dwóch mechanizmów:

- operatora `dynamic_cast`
- operatora `typeid`

# Operator `dynamic_cast`

Operator `dynamic_cast` jest przeznaczony do bezpiecznego rzutowania w dół hierarchii dziedziczenia (*ang. downcasting*) lub do rzutowania wskaźnika do typu `void*`.

Składnia:

```
dynamic_cast<TYPE>(expr),
```

gdzie `TYPE` jest typem wskaźnikowym lub referencją;

wyrażenie `expr` musi być również typem wskaźnikowym lub referencją.

Jeżeli `TYPE` jest typem **wskaźnikowym**, operator zwraca wskaźnik typu `TYPE`:

- Niezerowy, jeżeli rzutowanie zakończy się powodzeniem
- Zerowy (`null`), jeżeli rzutowanie nie jest możliwe

Jeżeli `TYPE` jest typem **referencyjnym** operator generuje wyjątek `std::bad_cast` w przypadku niepowodzenia.

Aby zastosować ten operator wyrażenie `expr` musi być typu polimorficznego (czyli musi istnieć dla niego `VTABLE`)

Czasem konieczne jest również jawne włączenie opcji generacji RTTI<sub>20</sub> w kompilatorze

# Przykład 1

```
class Shape{
public:
    virtual ~Shape(){} // typ polimorficzny
};

class Circle:public Shape {};
class Square:public Shape {};
class Triangle:public Shape {};
class FilledTriangle:public Triangle {};

void whoAmI(Shape*shape){
    if( dynamic_cast<Circle*>(shape)!=0 )
        cout<<"Circle ";
    if( dynamic_cast<Square*>(shape)!=0 )
        cout<<"Square ";
    if( dynamic_cast<Triangle*>(shape)!=0 )
        cout<<"Triangle ";
    if( dynamic_cast<FilledTriangle*>(shape)!=0 )
        cout<<"FilledTriangle ";
}
```

# Przykład 1

```
int main(){
    Shape*table[4];
    table[0]=new Circle();
    table[1]=new Square();
    table[2]=new Triangle();
    table[3]=new FilledTriangle();
    for(int i=0;i<4;i++){
        cout<<"I am a ";
        whoAmI(table[i]);
        cout<<endl;
    }
    for(int i=0;i<4;i++)delete table[i];
}
```

I am a Circle  
I am a Square  
I am a Triangle  
I am a Triangle FilledTriangle

## Przykład 2

```
class A{ public:
    virtual void printName(){
        cout<< "A"<<endl;
    }
};
class B : public A{ public:
    virtual void printName(){
        cout<< "B"<<endl;
    }
};
class C : public A{ public:
    virtual void printName(){
        cout<< "C"<<endl;
    }
};
```

```
int main(){
    C c;
    A&ra=c;
    try {
        C &rc = dynamic_cast<C &>(ra);
        rc.printName();
        B &rb = dynamic_cast<B &>(ra);
        rb.printName();
    }
    catch(std::bad_cast e){
        cout<<"bad cast"<<endl;
    }
}
```

Użycie operatora w przypadku referencji może powodować generację wyjątku.

Operator lepiej sprawdza się w przypadku wskaźników.

C  
bad cast

# Operator typeid

Operator typeid pozwala na określenie typu obiektu w trakcie wykonania programu.

Składnia:

```
typeid(TYPE)
```

lub

```
typeid(expr)
```

Operator typeid zwraca stałą referencję typu `const type_info &` do obiektu klasy `type_info`. Obiekty te są generowane automatycznie przez kompilator dla każdej klasy po włączeniu opcji użycia RTTI.

Argument TYPE powinien być nazwą klasy lub (dla kompatybilności) typem wbudowanym.

Typem wyrażenia expr powinna być:

- referencja
- wskaźnik, na którym zastosowano operator dereferencji `*`.  
*Bez dereferencji rezultatem działania operatora byłaby informacja o typie wskaźnikowym, a nie wskazywanym obiekcie.*



# Operator typeid

Typ wyrażenia powinien być typem polimorficznym (zawierającym co najmniej jedną funkcję wirtualną) . W przeciwnym wypadku rezultatem będzie informacja o typie wyrażenia, a nie wskazywanym obiekcie.

Klasa `type_info` zdefiniowana jest w pliku nagłówkowym `<typeinfo.h>` lub `<typeinfo>` jako:

```
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:
    ...
};
```

- Operatory `==` i `!=` umożliwiają porównanie typów.
- Funkcja `name()` zwraca nazwę typu obiektu, natomiast `raw_name()` wewnętrzną dekorowaną nazwę tworzoną przez kompilator.

# Operator typeid

```
void writeIfCircle(ostream&os, Shape&shape)
{
    if( typeid(shape) != typeid(Circle) )return;
    Circle&circle=static_cast<Circle&>(shape);
    // write circle here
}

void whoAmI(Shape*shape)
{
    cout<<typeid(*shape).name();
    std::string s = typeid(*shape).name();
    cout<<s<<endl;
}
// dla obiektu typu FilledTriangle wypisze:
// wyłącznie 'FilledTriangle'
```

# dynamic\_cast i typeid

- Oba operatory zachowują się poprawnie dla typów polimorficznych. Dla innych typów program może nie zostać skompilowany lub ich działanie może być nieoczekiwane.
- Operator typeid pozwala wyłącznie na określenie rzeczywistego typu, natomiast nie pozwala na stwierdzenie, czy należy on również do klasy umieszczonej w środku hierarchii.
- Operator dynamic\_cast pozwala rzutowanie na poziom pośredni.
- Oba operatory nie pozwalają na określenie typu lub rzutowanie wskaźnika typu void\*. Wskaźnik void\* nie ma informacji o typie.
- Operator typeid działa również dla typów wbudowanych.

# dynamic\_cast i typeid

- Jeżeli argumentami operatorów są referencje i nie jest możliwe określenie typu lub przeprowadzenie bezpiecznego rzutowania – oba operatory generują wyjątki. Powinny być one przechwytywane. Alternatywą jest wołanie wyłącznie operatora `dynamic_cast` dla typów wskaźnikowych.
- Techniki programowania wykorzystujące rzutowanie w dół oparte na RTTI powinny być stosowane z umiarem, ponieważ prowadzą do kodu wykonywanego warunkowo (wewnątrz instrukcji `if-else` lub `switch-case`). Tego typu kod jest trudny w konserwacji i kłopotliwy w przypadku rozszerzeń. Bardzo często może być on wyeliminowany przez użycie funkcji polimorficznych.