

Operatory rzutowania i konwerter

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Politechnika Wroclawska

Kurs: Programowanie obiektowe

Copyright©2018 Bogdan Kreczmer

Niniejszy dokument zawiera materiały do wykładu dotyczącego programowania obiektowego. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych prywatnych potrzeb i może on być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

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

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory **`static_cast`** i **`dynamic_cast`**
 - Operator **`reinterpret_cast`**
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

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

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

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. Źłe 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);
}

```

Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory **`static_cast`** i **`dynamic_cast`**
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

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

Przykład wykorzystania operatora `static_cast`

Dla własnych typów operator `static_cast` można wykorzystać do jawnego rzutowania *w górę*.

```
class ObiektGraficzny {
    ...
};

class Robot: public ObiektGraficzny {
    ...
};

int main()
{
    Robot          *wRob = new Robot();
    ObiektGraficzny *wObGraf = static_cast<ObiektGraficzny*>(wRob);
}
```

Przykład wykorzystania operatora `static_cast`

Uwaga: przy rzutowaniu obowiązują wszystkie reguły dostępu. Jeśli bazowa klasa dziedziczona jest jako prywatna, to rzutowanie staje się niemożliwe. Nie jest istotne wówczas, czy rzutowanie jest jawne, czy też nie.

```
class ObiektGraficzny {  
    ...  
};  
  
class Robot: private ObiektGraficzny {  
    ...  
};  
  
int main()  
{  
    Robot          *wRob = new Robot();  
    ObiektGraficzny *wObGraf = static_cast<ObiektGraficzny*>(wRob);  
}
```

Przykład wykorzystania operatora `static_cast`

Dla własnych typów operator `static_cast` można wykorzystać do jawnego rzutowania w dół. Ten typ rzutowania (tzn. w dół) można wykonać tylko jawnie.

```
class ObiektGraficzny {  
    ...  
};  
  
class Robot: public ObiektGraficzny {  
    ...  
};  
  
int main()  
{  
    ObiektGraficzny *wObGraf = new Robot();  
    Rob *wRob = static_cast<Rob*>(wObGraf);  
}
```

Przykład wykorzystania operatora `static_cast`

Źle wykonane rzutowanie *w dół* może być niebezpieczne i w dalszym działaniu programu może skutować błędem.

```
class ObiektGraficzny {  
    ...  
};  
  
class Robot: public ObiektGraficzny {  
    ...  
};  
  
int main()  
{  
    ObiektGraficzny *wObGraf = new ObiektGraficzny();  
    ObiektGraficzny *wNibyRobot = static_cast<Rob*>(wObGraf);  
}
```

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

Przykład wykorzystania operatora `dynamic_cast`

Operator `dynamic_cast`, podobnie jak operator `static_cast`, zapewnia jawne rzutowanie *w górę*. Jest jednak mniej efektywny.

```
class ObiektGraficzny {
    ...
    virtual ~ObiektGraficzny() {}
};

class Robot: public ObiektGraficzny {
    ...
    virtual ~Robot() {}
};

int main()
{
    Robot          *wRob = new Robot();
    ObiektGraficzny *wObGraf = dynamic_cast<ObiektGraficzny*>(wRob);
}
```

Przykład wykorzystania operatora `dynamic_cast`

Najistotniejszą różnicą między operatorem `dynamic_cast` i `static_cast` jest to, że ten pierwszy sprawdza, czy dane rzutowanie jest możliwe. Staje się to ważne w przypadku rzutowania *w dół*.

```
class ObiektGraficzny {
    ...
    virtual ~ObiektGraficzny() {}
};

class Robot: public ObiektGraficzny {
    ...
    virtual ~Robot() {}
};

int main()
{
    ObiektGraficzny *wObGraf = new Robot();
    Robot *wRob = dynamic_cast<Robot*>(wObGraf);
}
```

Przykład wykorzystania operatora `dynamic_cast`

Jeśli nie jest możliwe, jako wskaźnik zostanie zwrócony adres `nullptr`.

```
class ObiektGraficzny {
    ...
    virtual ~ObiektGraficzny() {}
};

class Robot: public ObiektGraficzny {
    ...
    virtual ~Robot() {}
};

int main()
{
    ObiektGraficzny *wObGraf = new ObiektGraficzny()
    Robot *wNibyRob = dynamic_cast<Robot*>(wObGraf);
}
```


Przykład wykorzystania operatora `dynamic_cast`

W przypadku błędnego rzutowania na referencję zgłaszany jest standardowy wyjątek `bad_cast`.

```
class ObiektGraficzny {  
    ...  
    virtual ~ObiektGraficzny() {}  
};  
  
class Robot: public ObiektGraficzny {  
    ...  
    virtual ~Robot() {}  
};  
  
int main()  
{  
    ObiektGraficzny    ObGraf;  
    Robot              &NibyRob = dynamic_cast<Robot*>(ObGraf);  
}
```

Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - **Operator `reinterpret_cast`**
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

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.
- `dynamic_cast`** – obsługuje tylko obiekty klas polimorficznych. Zapewnia realizację rzutowania *w górę*. Rzutowanie to należy do rzutowań bezpiecznych.
- `reinterpret_cast`** – jest najbardziej niebezpiecznym rzutowaniem. Wykonuje on konwersję między wskaźnikami oraz wskaźnikami i liczbami. Żle przeprowadzone rzutowanie może być źródłem błędów trudnych do wykrycia.

Przykład wykorzystania operatora `reinterpret_cast`

Ten operator pozwala w zasadzie na wszystko. W szczególności pozwala interpretować dowolny obiekt jako tablicę bajtów i dowolnie ją modyfikować wyłącznie ze wskaźnikami na tablice metod wirtualnych.

```
class ObiektGraficzny {
    ...
    virtual ~ObiektGraficzny() {}
};

class Robot: public ObiektGraficzny {
    ...
    virtual ~Robot() {}
};

int main()
{
    ObiektGraficzny *wObGraf = new ObiektGraficzny();
    int *wTab = reinterpret_cast<int*>(wObGraf);
}
```

Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

Wskaźniki dzielone – `static_pointer_cast`

Dla wskaźników dzielonych odpowiednikiem operatora `static_cast` jest szablon `static_pointer_cast`.

```
class ObiektGraf {  
    ...  
    virtual ~ObiektGraf() {}  
};  
  
class Robot: public ObiektGraf {  
    ...  
    virtual ~Robot() {}  
};  
  
int main()  
{  
    shared_ptr<ObiektGraf> WOb = make_shared<Robot>();  
    shared_ptr<Robot> WRob = static_pointer_cast<Robot>(wOb);  
}
```

Wskaźniki dzielone – `dynamic_pointer_cast`

Dla wskaźników dzielonych odpowiednikiem operatora `static_cast` jest szablon `static_pointer_cast`.

```
class ObiektGraf {  
    ...  
    virtual ~ObiektGraf() {}  
};  
  
class Robot: public ObiektGraf {  
    ...  
    virtual ~Robot() {}  
};  
  
int main()  
{  
    shared_ptr<ObiektGraf> WOb = make_shared<Robot>();  
    shared_ptr<Robot> WRob = dynamic_pointer_cast<Robot>(wOb);  
}
```

Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

Czym jest konwerter

Konwertery są definiowane w klasie jako metody bez parametrów. Nie są też poprzedzone deklaracją typu zwracanej wartości. Ich nazwa składa się ze słowa kluczowego **operator** oraz nazwy typu zwracanej wartości, na którą ma być przekonwertowany dany obiekt lub cokolwiek innego.

Przykład użycia konwertera do niejawnego konwersji typu

```
struct Wektor {
    double x, y;
};

struct LZespolona {
    double re, im;
    //----- Konwerter -----
    operator Wektor() { Wektor W; W.x = re; W.y = im; return W; }
};

int main()
{
    Wektor W;
    LZespolona Z;

    W = Z; // Tu niejawnie wywołuje się konwerter z klasy LZespolona
}
```

Czym jest konwerter

Modyfikator **const** i **volatile** w definicji konwertera pełnią taką samą rolę jak w przypadku metod. Odnoszą się więc do obiektu, dla którego wywołany zostaje konwerter/metoda.

Definicja konwertera z modyfikatorem const

```
struct Wektor {
    double x, y;
};

struct LZespolona {
    double re, im;
    //----- Konwerter -----
    operator Wektor() const { Wektor W; W.x = re; W.y = im; return W; }
};

int main()
{
    Wektor W;
    LZespolona Z;

    W = Z; // Tu niejawnie wywołuje się konwerter z klasy LZespolona
}
```

Zabronienie niejawnej konwersji

Począwszy od standardu C++11 dla konwerterów można zabronić jawnej konwersji wykorzystując słowo kluczowe **explicit**, tak jak to ma miejsce w przypadku konstruktorów.

Przykład użycia konwertera w jawnej konwersji typu

```
struct Wektor {
    double x, y;
};

struct LZespolona {
    double re, im;
    //----- Konwerter -----
    explicit operator Wektor() { Wektor W; W.x = re; W.y = im; return W; }
};

int main()
{
    Wektor W;
    LZespolona Z;

    W = static_cast<Wektor>(Z); // Tutaj jawnie wymuszone zostaje wywołanie konwertera.
}
```

Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

Konwersja poprzez konstruktor

Analogiczną konwersję można zrealizować za pomocą konstruktora jednoparametrowego.

Przykład niejawnego konwersji za pomocą konstruktora

```
struct LZespolona {
    double re, im;
};

struct Wektor {
    double x, y;
    Wektor(): x(), y() {}
    Wektor(LZespolona L): x(L.re), y(L.im) {}
};

int main()
{
    Wektor W;
    LZespolona Z;

    W = Z; // Tu niejawnie wywołuje się konstruktor z klasy Wektor
}
```

Konwersja poprzez konstruktor

Użycie słowa kluczowego **explicit** blokuje możliwość niejawnego wywołania konstruktora.

Przykład niejawnej konwersji za pomocą konstruktora

```
struct LZespolona {
    double re, im;
};

struct Wektor {
    double x, y;
    Wektor(): x(), y() {}
    explicit Wektor(LZespolona L): x(L.re), y(L.im) {}
};

int main()
{
    Wektor W;
    LZespolona Z;

    W = static_cast<Wektor>(Z); // Teraz wywołanie konstruktora zostało jawnie wymuszone
}
```

Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

Konwersja poprzez konstruktory i konwertery

Konstruktory

Konstruktory umożliwiają konwersję z dowolnego innego typu na typ obiektu, w którym są definiowane.

```
struct Wektor {  
    double x, y;  
  
    Wektor(Dowolny_Typ Ob) { ... } // Dowolny_Typ → Wektor  
};
```

Konwertery

Konwertery umożliwiają konwersję w kierunku przeciwnym, tzn. obiekt klasy, w której są zdefiniowane, na obiekt dowolnego innego typu. Może to być również jeden z wbudowanych typów, np. **int**.

```
struct Wektor {  
    double x, y;  
  
    operator Dowolny_Typ() { ... } // Wektor → Dowolny_Typ  
};
```


Plan prezentacji

- 1 Operatory rzutowania
 - Operator `static_cast`
 - Operatory `static_cast` i `dynamic_cast`
 - Operator `reinterpret_cast`
 - Operatory rzutowania dla wskaźników dzielonych
- 2 Konwertery
 - Konwertery – cechy i własności
 - Konstruktory jako konwertery
 - Konwertery vs konstruktory
 - Konwertery, konstruktory – wzajemne współistnienie
- 3 Konwertery w bibliotece strumieni

Konwersja konstruktor lub konwerter

```
class Wektor;  
  
struct LZespolona {  
    double re, im;  
    operator Wektor() const;  
};  
  
struct Wektor {  
    double x, y;  
    Wektor(): x(), y() {}  
    Wektor(LZespolona L): x(L.re), y(L.im) {}  
};  
  
LZespolona::operator Wektor() const  
{ Wektor W; W.x = re; W.y = im; return W; }  
  
int main()  
{  
    Wektor W;  
    LZespolona Z;  
  
    W = Z; // Konwersja bedzie zrealizowana  
          // za pomoca konstruktora.  
}
```

Gdy dopuszczalne są dwie konwersje, która z nich zostanie uruchomiona?

Konwersja konstruktor lub konwerter

Gdy dwa rodzaje konwersji są dopuszczalne pierwszeństwo ma konstruktor

```
class Wektor;  
  
struct LZespolona {  
    double re, im;  
    operator Wektor() const;  
};  
  
struct Wektor {  
    double x, y;  
    Wektor(): x(), y() {}  
    Wektor(LZespolona L): x(L.re), y(L.im) {}  
};  
  
LZespolona::operator Wektor() const  
{ Wektor W; W.x = re; W.y = im; return W; }  
  
int main()  
{  
    Wektor W;  
    LZespolona Z;  
  
    W = Z; // Konwersja będzie zrealizowana  
          // za pomoca konstruktora.  
}
```

W przypadku, gdy konwersja do danego typu może zajść na dwa sposoby, tzn. za pomocą konstruktora lub konwertera, pierwszeństwo ma konwersja z użyciem konstruktora.

Konwersja konstruktor lub konwerter

```
class Wektor;  
  
struct LZespolona {  
    double re, im;  
    operator Wektor() const;  
};  
  
struct Wektor {  
    double x, y;  
    Wektor(): x(), y() {}  
    explicit Wektor(LZespolona L): x(L.re), y(L.im) {}  
};  
  
LZespolona::operator Wektor() const  
{ Wektor W; W.x = re; W.y = im; return W; }  
  
int main()  
{  
    Wektor W;  
    LZespolona Z;  
  
    W = Z; // Konwersja bedzie zrealizowana  
          // za pomoca ...  
}
```

Jednak gdy niejawną konwersja z wykorzystaniem konstruktora staje się niedopuszczalna ...

Konwersja konstruktor lub konwerter

Teraz użyty zostanie konwerter

```
class Wektor;  
  
struct LZespolona {  
    double re, im;  
    operator Wektor() const;  
};  
  
struct Wektor {  
    double x, y;  
    Wektor(): x(), y() {}  
    explicit Wektor(LZespolona L): x(L.re), y(L.im) {}  
};  
  
LZespolona::operator Wektor() const  
{ Wektor W; W.x = re; W.y = im; return W; }  
  
int main()  
{  
    Wektor W;  
    LZespolona Z;  
  
    W = Z; // Konwersja będzie zrealizowana  
          // za pomoca ... .  
}
```

Jednak gdy niejawną konwersję z wykorzystaniem konstruktora staje się niedopuszczalna, wówczas realizowana jest ta, która jest wciąż dozwolona.

Konwersja konstruktor lub konwerter

```
class Wektor;  
  
struct LZespolona {  
    double re, im;  
    operator Wektor() const;  
};  
  
struct Wektor {  
    double x, y;  
    Wektor(): x(), y() {}  
    explicit Wektor(LZespolona L): x(L.re), y(L.im) {}  
};  
  
LZespolona::operator Wektor() const  
{ Wektor W; W.x = re; W.y = im; return W; }  
  
int main()  
{  
    Wektor W;  
    LZespolona Z;  
  
    W = static_cast<Wektor>(Z); // Konwersja bedzie  
                               // przez ...  
}
```

Jeżeli jawnie wymuszamy konwersję to oba sposoby konwersji są ponownie dopuszczalne. Tak więc konwersja zostanie dokonana za pomocą ...

Konwersja konstruktor lub konwerter

Teraz użyty zostanie konstruktor

```
class Wektor;  
  
struct LZespolona {  
    double re, im;  
    operator Wektor() const;  
};  
  
struct Wektor {  
    double x, y;  
    Wektor(): x(), y() {}  
    explicit Wektor(LZespolona L): x(L.re), y(L.im) {}  
};  
  
LZespolona::operator Wektor() const  
{ Wektor W; W.x = re; W.y = im; return W; }  
  
int main()  
{  
    Wektor W;  
    LZespolona Z;  
  
    W = static_cast<Wektor>(Z); // Konwersja bedzie  
                               // przez ... .  
}
```

Jeżeli jawnie wymuszamy konwersję to oba sposoby konwersji są ponownie dopuszczalne. Tak więc konwersja zostanie dokonana za pomocą konstruktora.

Konwerter dla `std::istream`

```
float d;  
do {  
    cin >> d;  
    if ( cin.fail( ) ) break ;  
    ...  
} while (true);
```

Co robi ta pętla

Chcemy wczytać zbiór liczb zmiennoprzecinkowych i przerwać ich czytanie w momencie, gdy operacja nie powiedzie się.

Zapis operacji odczytu z `cin` można znacznie uprościć. Jest to możliwe dzięki temu, że dla klasy `std::istream` przeciążenie operatora `>>` zwraca referencję do `std::istream` (obiekt zwraca referencję do samego siebie) oraz zdefiniowany jest odpowiedni operator konwersji.

Konwerter dla `std::istream`

```
float d;  
do {  
    cin >> d;  
    if ( cin.fail( ) ) break ;  
    ...  
} while (true);
```

⇒

```
float d;  
while (!(cin >> d).fail( ) ) {  
    ...  
}
```

Co robi ta pętla

Chcemy wczytać zbiór liczb zmiennoprzecinkowych i przerwać ich czytanie w momencie, gdy operacja nie powiedzie się.

Zapis operacji odczytu z `cin` można znacznie uprościć. Jest to możliwe dzięki temu, że dla klasy `std::istream` przeciążenie operatora `>>` zwraca referencję do `std::istream` (obiekt zwraca referencję do samego siebie) oraz zdefiniowany jest odpowiedni operator konwersji.

Konwerter dla `std::istream`

```
float d;  
do {  
    cin >> d;  
    if ( cin.fail( ) ) break ;  
    ...  
} while (true);
```



```
float d;  
while (!(cin >> d).fail( ) ) {  
    ...  
}
```



```
float d;  
while (cin >> d) {  
    ...  
}
```

Co robi ta pętla

Chcemy wczytać zbiór liczb zmiennoprzecinkowych i przerwać ich czytanie w momencie, gdy operacja nie powiedzie się.

Zapis operacji odczytu z `cin` można znacznie uprościć. Jest to możliwe dzięki temu, że dla klasy `std::istream` przeciążenie operatora `>>` zwraca referencję do `std::istream` (obiekt zwraca referencję do samego siebie) oraz zdefiniowany jest odpowiedni operator konwersji.

Przed C++11 i po

Dla klas strumieniowych (`std::istream`, `std::ostream`, `std::ifstream`, `std::ofstream`, `std::istringstream`, `std::ostringstream`, ...), zdefiniowane są operatory konwersji do typu skalarnego.

C++ – przed 2011

```
class ... {  
    ...  
    public:  
        operator void *() const; // (failbit || badbit) ? nullptr : coś_innego;  
};
```

C++11 – od 2011

```
class ... {  
    ...  
    public:  
        explicit operator bool() const; // !(failbit || badbit)  
};
```

Dlaczego operator void*() const

Sposób definicji konwertera w starszym standardzie wynikał z tego, że słowo kluczowe **explicit** nie można było stosować do konwerterów. Chąc lepiej zrozumieć problem, należy zbadać jakie zapisy były dopuszczalne, gdyby przyjąć konwersję do typu **bool** bez **explicit**.

C++ – przed 2011 – gdyby było tak

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

```
if (cin) { ... } // Prawidłowy
```

```
bool Fun() {
```

```
    ...
```

```
    return cin; // Prawidłowy
```

```
}
```

```
int Zm = cin + 5; // Prawidłowy!!!!!!!!!!!!!!!!!!!!
```

Dlaczego operator `void*()` `const`

Sposób definicji konwertera w starszym standardzie wynikał z tego, że słowo kluczowe **explicit** nie można było stosować do konwerterów. Wykorzystanie rzutowania na typ skalarny **`void*`** rozwiązywało problem niekontrolowanej konwersji do typów **`int`**, **`double`**, itp.

C++ – przed 2011 – gdyby było tak

```
class ... {  
    ...  
    public:  
        operator void*() const;  
};
```

```
if (cin) { ... } // Prawidłowy
```

```
bool Fun() {
```

```
    ...
```

```
    return cin; // Prawidłowy
```

```
}
```

```
int Zm = cin + 5; // Nieprawidłowy i tak musi być!
```

Co się zmienia, gdy explicit operator bool() const

Wprowadzenie słowa kluczowego explicit do definicji konwerterów rozwiązało problem niekontrolowanych konwersji do typu skalarnego w wyrażeniach. Konsekwencją tego jest również brak bezpośredniego użycia obiektów strumieniowych w połączeniu z instrukcją return.

C++11 – po 2011

```
class ... {  
    ...  
    public:  
        explicit operator bool() const;  
};
```

```
if (cin) { ... } // Prawidłowy
```

```
bool Fun() {  
    ...
```

```
    return cin; // Nieprawidłowy!  
}
```

```
int Zm = cin + 5; // Nieprawidłowy i tak musi być!
```

Co się zmienia, gdy explicit operator bool() const

W przypadku, gdy faktycznie chcemy zwrócić poprzez instrukcję return stan strumienia dostępny za pomocą konwertera do typu **bool**, możemy dokonać jawnej konwersji wykorzystując operator **static_cast**.

C++11 – po 2011

```
class ... {  
    ...  
    public:  
        explicit operator bool() const;  
};
```

```
if (cin) { ... } // Prawidłowy
```

```
bool Fun() {
```

```
    ...
```

```
    return static_cast<bool>(cin); // Prawidłowy!
```

```
}
```

```
int Zm = cin + 5; // Nieprawidłowy i tak musi być!
```

Koniec prezentacji
Dziękuję za uwagę