

Pojęcie klasy, zwracanie przez referencję, przeciążanie operatorów strumieniowych

Bogdan Kreczmer

bogdan.kreczmer@pwr.wroc.pl

Zakład Podstaw Cybernetyki i Robotyki
Instytut Informatyki, Automatyki i Robotyki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Copyright©2017 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>

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 Zapis operacji na strumieniu standardowym
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 Zapis operacji na strumieniu standardowym
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Typ obiektowy

Typ obiektowy ???

Typ obiektowy

Typ obiektowy jest pojęciem, tzn. jest koncepcją lub ideą, którą stosujemy do obiektów występujących w naszej świadomości.

Typ obiektowy

Typ obiektowy jest pojęciem, tzn. jest koncepcją lub ideą, którą stosujemy do obiektów występujących w naszej świadomości.

Typ obiektowy jest typem obiektu.

Typ obiektowy

Typ obiektowy jest pojęciem, tzn. jest koncepcją lub ideą, którą stosujemy do obiektów występujących w naszej świadomości.

Typ obiektowy jest typem obiektu.

Przykład *typów obiektowych*:

- robot przemysłowy,
- pojazd,
- idealny człowiek,
- wektor,
- równanie liniowe.

Typ obiektowy

Własności:

- *Typ obiektowy* jest też pojęciem.
- Do pojedynczego obiektu może stosować się wiele pojęć (*typów obiektowych*).
- *Typy obiektowe* mogą tworzyć hierarchię od bardzo ogólnych pojęć do pojęć szczegółowych.

Typ obiektowy

Własności:

- *Typ obiektowy jest też pojęciem.*
- Do pojedynczego obiektu może stosować się wiele pojęć (*typów obiektowych*).
- *Typy obiektowe mogą tworzyć hierarchię od bardzo ogólnych pojęć do pojęć szczegółowych.*

Typ obiektowy

Własności:

- *Typ obiektowy* jest też pojęciem.
- Do pojedynczego obiektu może stosować się wiele pojęć (*typów obiektowych*).
- *Typy obiektowe* mogą tworzyć hierarchię od bardzo ogólnych pojęć do pojęć szczegółowych.

Typ obiektowy

Własności:

- *Typ obiektowy* jest też pojęciem.
- Do pojedynczego obiektu może stosować się wiele pojęć (*typów obiektowych*).
- *Typy obiektowe* mogą tworzyć hierarchię od bardzo ogólnych pojęć do pojęć szczegółowych.

Przykład hierarchii



Czym jest klasa

Klasa jest typem obiektowym

Co cechuje podejście obiektowe

Cechą charakterystyczną podejścia obiektowego i języków obiektowych jest ściśle powiązanie operacji z danymi, na których są one wykonywane.

Co cechuje podejście obiektowe

Cechą charakterystyczną podejścia obiektowego i języków obiektowych jest ściśle powiązanie operacji z danymi, na których są one wykonywane.

Takie powiązanie może być również realizowane na poziomie języka **C**.

Co cechuje podejście obiektowe

Cechą charakterystyczną podejścia obiektowego i języków obiektowych jest ściśle powiązanie operacji z danymi, na których są one wykonywane.

Takie powiązanie może być również realizowane na poziomie języka **C**.

*W języku **C** można również programować obiektowo!!!*

Metody

Aby lepiej odzwierciedlić powiązanie operacji z danymi wprowadza się pojęcie **metody**.

Metody

Aby lepiej odzwierciedlić powiązanie operacji z danymi wprowadza się pojęcie **metody**.

Metoda jest specyfikacją sposobu wykonania sekwencji operacji.

Metody

Aby lepiej odzwierciedlić powiązanie operacji z danymi wprowadza się pojęcie **metody**.

Metoda jest specyfikacją sposobu wykonania sekwencji operacji.

Na poziomie języka programowania metodę należy rozumieć jako rodzaj funkcji, która związana jest z obiektami danej klasy.

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- Analiza
- Projektowanie
- Konstrukcja

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza**
- Projektowanie
- Konstrukcja

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza**
- **Projektowanie**
- **Konstrukcja**

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza**
- **Projektowanie**
- **Konstrukcja**

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza** – jest odwzorowaniem rzeczywistego świata na jego model koncepcyjny
- **Projektowanie**
- **Konstrukcja**

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza** – jest odwzorowaniem rzeczywistego świata na jego model koncepcyjny
- **Projektowanie** – jest odwzorowaniem modelu koncepcyjnego na model implementacji.
- **Konstrukcja**

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza** – jest odwzorowaniem rzeczywistego świata na jego model koncepcyjny
- **Projektowanie** – jest odwzorowaniem modelu koncepcyjnego na model implementacji.
- **Konstrukcja** – jest odwzorowaniem modelu implementacji na działający system.

Skąd się biorą typy

Przejściu od sformułowania problemu do jego rozwiązania w postaci działające systemu towarzyszą zwykle trzy fazy:

- **Analiza** – jest odwzorowaniem rzeczywistego świata na jego model koncepcyjny
- **Projektowanie** – jest odwzorowaniem modelu koncepcyjnego na model implementacji.
- **Konstrukcja** – jest odwzorowaniem modelu implementacji na działający system.

Etapy te nie muszą przebiegać sekwencyjnie. Wszystko zależy od przyjętych technik i strategii rozwiązywania danego problemu.

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 Zapis operacji na strumieniu standardowym
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Łączność działania – porządek wykonywania operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
{
    Symbol y = a, x = b, wynik;
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + b + a = ?$

Łączność działania – porządek wykonywania operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
{
    Symbol y = a, x = b, wynik;
    wynik = y + x + y;
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + b + a = ?$

Łączność działania – porządek wykonywania operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
{
    Symbol y = a, x = b, wynik;
```

```
wynik = (y + x) + y;
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + b + a = ?$

Łączność działania – porządek wykonywania operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
```

```
{
```

```
    Symbol y = a, x = b, wynik;
```

```
    wynik = (y + x) + y;
```

```
    wynik = operator+ (operator+ (y ,x), y );
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + b + a = ?$

Łączność działania – porządek wykonywania operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
```

```
{
```

```
    Symbol y = a, x = b, wynik;
```

```
wynik = (y + x) + y;
```

```
wynik = operator+ (operator+ (y ,x), y );
```

```
wynik = Dodaj(Dodaj(x, y), y);
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

a + b + a = ?

Jawna zmiana porządku wykonywanych operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
{
    Symbol y = a, x = b, wynik;
```

```
wynik = y + (x + y);
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + (b + a) = ?$

Jawna zmiana porządku wykonywanych operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
```

```
{
```

```
    Symbol y = a, x = b, wynik;
```

```
    wynik = y + (x + y);
```

```
    wynik = operator+ (y, operator+ (x ,y) );
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + (b + a) = ?$

Jawna zmiana porządku wykonywanych operacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol Dodaj( Symbol x, Symbol y );
```

```
Symbol operator + ( Symbol x, Symbol y );
```

```
...
```

```
int main( )
```

```
{
```

```
    Symbol y = a, x = b, wynik;
```

```
    wynik = y + (x + y);
```

```
    wynik = operator + (y, operator + (x, y) );
```

```
    wynik = Dodaj(y , Dodaj(x, y) );
```

```
}
```

Tabliczka działania

+	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

$a + (b + a) = ?$

Łączność działania

Niech \circ będzie operatorem jakiegoś działania. Działanie \circ w zbiorze S jest łączne jeśli

$$\forall a, b, c \in S, \quad a \circ b \circ c = (a \circ b) \circ c = a \circ (b \circ c).$$

Rodzaje łączność

Niech \circ będzie operatorem jakiegoś działania. Działanie \circ w zbiorze S jest łączne jeśli

$$\forall a, b, c \in S, \quad a \circ b \circ c = (a \circ b) \circ c = a \circ (b \circ c).$$

W przypadku działań, które nie są łączne ustalona jest konwencja wykonywania operacji, dla zapisu bez nawiasów. W sensie tej konwencji działanie może być lewostronnie łączne lub prawostronnie łączne.

Rodzaje łączność

Niech \circ będzie operatorem jakiegoś działania. Działanie \circ w zbiorze S jest łączne jeśli

$$\forall a, b, c \in S, \quad a \circ b \circ c = (a \circ b) \circ c = a \circ (b \circ c).$$

W przypadku działań, które nie są łączne ustalona jest konwencja wykonywania operacji, dla zapisu bez nawiasów. W sensie tej konwencji działanie może być lewostronnie łączne lub prawostronnie łączne.

Działanie jest lewostronnie łączne gdy:

$$a \circ b \circ c = (a \circ b) \circ c.$$

Działanie jest prawostronnie łączne gdy:

$$a \circ b \circ c = a \circ (b \circ c).$$

Rodzaje łączność

Niech \circ będzie operatorem jakiegoś działania. Działanie \circ w zbiorze S jest łączne jeśli

$$\forall a, b, c \in S, \quad a \circ b \circ c = (a \circ b) \circ c = a \circ (b \circ c).$$

W przypadku działań, które nie są łączne ustalona jest konwencja wykonywania operacji, dla zapisu bez nawiasów. W sensie tej konwencji działanie może być lewostronnie łączne lub prawostronnie łączne.

Przykład działania lewostronnie łącznego:

$$4 - 1 - 2 = (4 - 1) - 2.$$

Przykład działania prawostronnie łącznego:

$$4^{2^3} = 4^{(2^3)}.$$

Łączność operatorów

W przypadkach języków programowania porządek wykonywania operacji jest zawsze ustalony niezależnie od tego czy działanie jest łączne, czy też nie.

W tym sensie mówimy, że operator jest lewostronnie łączny lub prawostronnie, gdy zapis ciągu danej operacji bez nawiasów wykonywany jest odpowiednio od lewej strony do prawej lub od prawej do lewej.

Łączność operatorów

W przypadkach języków programowania porządek wykonywania operacji jest zawsze ustalony niezależnie od tego czy działanie jest łączne, czy też nie.

W tym sensie mówimy, że operator jest lewostronnie łączny lub prawostronnie, gdy zapis ciągu danej operacji bez nawiasów wykonywany jest odpowiednio od lewej strony do prawej lub od prawej do lewej.

Łączność operatorów

W przypadków języków programowania porządek wykonywania operacji jest zawsze ustalony niezależnie od tego czy działanie jest łączne, czy też nie.

W tym sensie mówimy, że operator jest lewostronnie łączny lub prawostronnie, gdy zapis ciągu danej operacji bez nawiasów wykonywany jest odpowiednio od lewej strony do prawej lub od prawej do lewej.

Przykład operatora lewostronnie łącznego:

$$\text{suma} = x + y + z \quad \longrightarrow \quad \text{suma} = ((x + y) + z).$$

Łączność operatorów

W przypadków języków programowania porządek wykonywania operacji jest zawsze ustalony niezależnie od tego czy działanie jest łączne, czy też nie.

W tym sensie mówimy, że operator jest lewostronnie łączny lub prawostronnie, gdy zapis ciągu danej operacji bez nawiasów wykonywany jest odpowiednio od lewej strony do prawej lub od prawej do lewej.

Przykład operatora lewostronnie łącznego:

$$\text{suma} = x + y + z \quad \longrightarrow \quad \text{suma} = ((x + y) + z).$$

Przykład operatora prawostronnie łącznego:

$$x = y = z = 5 \quad \longrightarrow \quad (x = (y = (z = 5))).$$

Operatory arytmetyczne i binarne

Operatory lewostronnie łączne (dwuargumentowe)

+	-	*	/	%
	&	^	>>	<<

Operatory prawostronnie łączne (dwuargumentowe)

=				
+=	--	*=	/=	%=
=	&=	^=	>>=	<<=

Operatory relacyjne i logiczne

Operatory lewostronnie łączne (dwuargumentowe)

< > == !=

<= >=

Operatory prawostronnie łączne (jednoargumentowe)

! (negacja logiczne)

Indeksowanie	<i>wskaznik</i> [<i>wyrażenie</i>]
Wywołanie funkcji	<i>wyrażenie</i> (<i>lista_wyrażeń</i>)
Przyrostkowa inkrementacja	<i>l-wartość</i> ++
Przyrostkowa dekrementacja	<i>l-wartość</i> --
Przedrostkowa inkrementacja	++ <i>l-wartość</i>
Przedrostkowa dekrementacja	-- <i>l-wartość</i>
Negacja bitowa	~ <i>wyrażenie</i>
Negacja logiczna	! <i>wyrażenie</i>
Minus jednoargumentowy	- <i>wyrażenie</i>
Plus jednoargumentowy	+ <i>wyrażenie</i>
Mnożenie	<i>wyrażenie</i> * <i>wyrażenie</i>
Dzielenie	<i>wyrażenie</i> / <i>wyrażenie</i>
Modulo	<i>wyrażenie</i> % <i>wyrażenie</i>
Dodawanie	<i>wyrażenie</i> + <i>wyrażenie</i>
Odejmowanie	<i>wyrażenie</i> - <i>wyrażenie</i>
Przesuwanie w lewo	<i>wyrażenie</i> << <i>wyrażenie</i>
Przesuwanie w prawo	<i>wyrażenie</i> >> <i>wyrażenie</i>
Mniejszy	<i>wyrażenie</i> < <i>wyrażenie</i>
Mniejszy lub równy	<i>wyrażenie</i> <= <i>wyrażenie</i>
Większy	<i>wyrażenie</i> > <i>wyrażenie</i>
Większy lub równy	<i>wyrażenie</i> >= <i>wyrażenie</i>



Operatory zawarte w tej samej części tabeli mają ten sam priorytet.



Równy	<i>wyrażenie == wyrażenie</i>
Nierówny	<i>wyrażenie != wyrażenie</i>
Koniunkcja bitowa	<i>wyrażenie & wyrażenie</i>
Różnica symetryczna	<i>wyrażenie ^ wyrażenie</i>
Alternatywa bitowa	<i>wyrażenie — wyrażenie</i>
Iloczyn logiczny	<i>wyrażenie && wyrażenie</i>
Suma logiczna	<i>wyrażenie —— wyrażenie</i>
Wyrażenie warunkowe	<i>wyrażenie ? wyr. : wyr.</i>
Proste przypisanie	<i>l-wartość = wyrażenie</i>
Przemnóż i przypisz	<i>l-wartość *= wyrażenie</i>
Podziel i przypisz	<i>l-wartość /= wyrażenie</i>
Weź modulo i przypisz	<i>l-wartość %= wyrażenie</i>
Dodaj i przypisz	<i>l-wartość += wyrażenie</i>
Odejmij i przypisz	<i>l-wartość -= wyrażenie</i>
Przesuń w lewo i przypisz	<i>l-wartość <<= wyr.</i>
Przesuń w prawo i przypisz	<i>l-wartość >>= wyr.</i>
Koniunkcja bitowa i przypisz	<i>l-wartość &= wyrażenie</i>
Alternatywa bitowa i przypisz	<i>l-wartość ^= wyrażenie</i>
Różnica bitowa i przypisz	<i>l-wartość ^= wyrażenie</i>

Operatory zawarte w tej samej części tabeli mają ten sam priorytet.

Dodatkowe oznaczenia operatorów

<code>&&</code>	→	<code>and</code>
<code> </code>	→	<code>or</code>
<code>!</code>	→	<code>not</code>
<code>!=</code>	→	<code>not_eq</code>
<code>&</code>	→	<code>bitand</code>
<code>&=</code>	→	<code>and_eq</code>
<code> </code>	→	<code>bitor</code>
<code> =</code>	→	<code>or_eq</code>
<code>^</code>	→	<code>xor</code>
<code>^=</code>	→	<code>xor_eq</code>
<code>~</code>	→	<code>compl</code>

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 Zapis operacji na strumieniu standardowym
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
```

```
}
```

```
int main( )
{
```

```
}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
```

```
}
```

```
int main( )
{
```

```
}
```

Zdefiniujemy funkcję, która zwiększa wartość zmiennej o jeden i zwraca kopie wcześniejszej wartości przed dokonaniem inkrementacji.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
}

int main( )
{

}

}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
}

int main( )
{

}

}
```

Korzystając z funkcji możemy bezpośrednio w liście parametrów posłużyć się stałą liczbą, zaś wynik przypisać zmiennej.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int & Wart)
{
    int Kopia = Wart;
    Wart += 1;
}
```

```
int main( )
{

}
}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
    *wWart += 1;
}
```

```
int main( )
{

}
}
```

Dokonujemy modyfikacji wartości zmiennej udostępnianej w pierwszym przypadku przez referencję, w drugim zaś przez wskaźnik.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{

}
}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
    *wWart += 1;
    return Kopia;
}
```

```
int main( )
{

}
}
```

Zwracamy wartość parametru sprzed momentu dokonania jej zmiany.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int & Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
    *wWart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

}
```

Definiujemy przykładowe dwie zmienne i jedną z nich inicjalizujemy.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int & Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(Liczba);

}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
    *wWart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(&Liczba);

}
```

Wywołujemy funkcję. W przypadku przekazywania parametru przez wskaźnik musimy jawnie zażądać udostępnienia adresu zmiennej. W przypadku referencji operacja ta jest dla nas *przezroczysta*.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
    *wWart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Dodaj_1_Kopia(&Liczba);
    cout << Wynik << Liczba << endl;
}
```

Co się wyświetli? Czy w obu przypadkach wyświetli się to samo?

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int & Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int Dodaj_1_Kopia(int * const wWart)
{
    int Kopia = *wWart;
    *wWart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Dodaj_1_Kopia(&Liczba);
    cout << Wynik << Liczba << endl;
}
```

Czy te funkcje czegoś nie przypominają?

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}

int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

Czy czasem nie kojarzy się to z jakimś operatorem? To szczególnie dobrze jest widoczne, gdy spojrzymy na konstrukcję z referencją.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}

int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Liczba++;
    cout << Wynik << Liczba << endl;
}
```

Schemat działania funkcji `Dodaj_1_Kopia` jest identyczne ze schematem działania operatora post-inkrementacji.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}

int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Liczba++++;
    cout << Wynik << Liczba << endl;
}
```

A jaki będzie wynik podwójnej post-inkrementacji.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}

int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = (Liczba++)++;
    cout << Wynik << Liczba << endl;
}
```

Dla lepszego zrozumienia tego wyrażenia możemy wprowadzić równoważny zapis pokazujący naturalny porządek operacji.

Jak zapisać to za pomocą wywołań funkcji Dodaj_1_Kopia?

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}

int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Kopia(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = (Liczba++)++;
    cout << Wynik << Liczba << endl;
}
```

Dla większej prostoty skróćmy nazwę funkcji z Dodaj_1_Kopia do ...

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int D1k(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1k(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = (Liczba++)++;
    cout << Wynik << Liczba << endl;
}
```

Dla większej prostoty skróćmy nazwę funkcji z `Dodaj_1_Kopia` do `D1k`.

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int D1k(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1k(D1k(Liczba));
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = (Liczba++)++;
    cout << Wynik << Liczba << endl;
}
```

Zapis wywołań operatora post-inkrementacji pokazuje jak to zrobić w przypadku wywołań funkcji D1k. Ale czy oba zapisy są na pewno poprawne?

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int D1k(int &Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1k(D1k(Liczba));
    cout << Wynik << Liczba << endl;
}
```

d1k.cc: In function 'int main()':
d1k.cc:14:error: invalid initialization of non-const reference of
type 'int&' from a temporary of type 'int'
d1k.cc:4: error: in passing argument 1 of 'int D1k(int&)'

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = (Liczba++)++;
    cout << Wynik << Liczba << endl;
}
```

inc.cc: In function 'int main()':
inc.cc:14:error: 7: error: lvalue required as increment operand

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int D1k(int & Wart)
{
    int Kopia = Wart;
    Wart += 1;
    return Kopia;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1k(D1k(Liczba));
    cout << Wynik << Liczba << endl;
}
```

**Natura tych błędów jest identyczna.
W obu przypadkach zwracana jest wartość, zamiast referencji do zmiennej.**

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = (Liczba++)++;
    cout << Wynik << Liczba << endl;
}
```

d1k.cc: In function 'int main()':
d1k.cc:14:error: invalid initialization of non-const reference of type 'int&' from a temporary of type 'int'
d1k.cc:4: error: in passing argument 1 of 'int D1k(int&)'

inc.cc: In function 'int main()':
inc.cc:14:error: 7: error: lvalue required as increment operand

Czym jest *l*- i *p*-wartość

Pojęcie *l*-wartości i *p*-wartości związane jest z operacją przypisania.

$$l\text{-wartość} = p\text{-wartość}$$

l-wartość – nazwana pojedyncza zmienna reprezentująca obszar pamięci przeznaczony do przechowywania zmiennej.

p-wartość – stała lub zmienna lub wyrażenie zwracające wartość.

Czym jest *l*- i *p*-wartość

Pojęcie *l*-wartości i *p*-wartości związane jest z operacją przypisania.

$$l\text{-wartość} = p\text{-wartość}$$

l-wartość – nazwana pojedyncza zmienna reprezentująca obszar pamięci przeznaczony do przechowywania zmiennej.

p-wartość – stała lub zmienna lub wyrażenie zwracające wartość.

Czym jest *l*- i *p*-wartość

Pojęcie *l*-wartości i *p*-wartości związane jest z operacją przypisania.

$$l\text{-wartość} = p\text{-wartość}$$

l-wartość – nazwana pojedyncza zmienna reprezentująca obszar pamięci przeznaczony do przechowywania zmiennej.

p-wartość – stała lub zmienna lub wyrażenie zwracające wartość.

Zwracanie referencji

Spróbujmy rozwiązać problem, który uniemożliwił jednoczesną modyfikację zmiennej i kaskadowe wywołanie tej samej funkcji.

Jako przykład zdefiniujemy funkcję, która zwiększa wartość zmiennej o jeden i zwraca referencję do tej samej zmiennej. W przypadku rozwiązania bazującego na wskaźnikach, musi zwrócić wskaźnik do tej zmiennej.

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}
```


Zwracanie referencji

```
int& Dodaj_1_Oryginal(int & Arg)
{

}

```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}

```

```
int* Dodaj_1_Oryginal(int * const wArg)
{

}

```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}

```

Parameter, tak jak wcześniej, przekazywany jest przez referencję. Spowoduje to, że zmiany jego wartości będą *widoczne* na zewnątrz. Wersja wskaźnikowa ma analogiczną konstrukcję.

Zwracanie referencji

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}
```

```
int* Dodaj_1_Oryginal(int * const wArg)
{
    *wArg += 1;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}
```

Dokonyjemy modyfikacji wartości zmiennej udostępnianej w pierwszym przypadku przez referencję, w drugim zaś przez wskaźnik.

Zwracanie referencji

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}
```

```
int* Dodaj_1_Oryginal(int * const wArg)
{
    *wArg += 1;
    return wArg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    ...

    cout << Wynik << Liczba << endl;
}
```

Choć w wersji z referencją zapis jest taki sam jak poprzednio, to jednak tym razem nie zwracana jest wartość, a referencja do parametru. Pozwala to na dostęp do niego i bezpośrednio wykorzystanie go w dalszych operacjach.

Zwracanie referencji

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Dodaj_1_Oryginal(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int* Dodaj_1_Oryginal(int * const wArg)
{
    *wArg += 1;
    return wArg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = *Dodaj_1_Oryginal(&Liczba);
    cout << Wynik << Liczba << endl;
}
```

Wynik działania obu wersji operacji jest identyczny.

Zwracanie przez referencję

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Oryginal(Liczba);
    cout << Wynik << Liczba << endl;
}
```

Czy czasem nie kojarzy się to z jakimś operatorem? To szczególnie dobrze jest widoczne, gdy spojrzymy na konstrukcję z referencją.

Zwracanie przez referencję

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Oryginal(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = ++Liczba;
    cout << Wynik << Liczba << endl;
}
```

Schemat działania funkcji `Dodaj_1_Oryginal` jest identyczne ze schematem działania operatora pre-inkrementacji.

Zwracanie przez referencję

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Oryginal(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = ++++Liczba;
    cout << Wynik << Liczba << endl;
}
```

Czy tym razem podwójna pre-inkrementacja będzie poprawnym zapisem?

Zwracanie przez referencję

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;

    Wynik = Dodaj_1_Oryginal(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = ++(++Liczba);
    cout << Wynik << Liczba << endl;
}
```

Ponownie dla lepszego zrozumienia tego wyrażenia możemy wprowadzić równoważny zapis pokazujący naturalny porządek operacji.

Jak zapisać to za pomocą wywołań funkcji Dodaj_1_Oryginal?

Zwracanie przez referencję

```
int& Dodaj_1_Oryginal(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = Dodaj_1_Oryginal(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = ++(++Liczba);
    cout << Wynik << Liczba << endl;
}
```

Dla większej prostoty skróćmy nazwę funkcji z `Dodaj_1_Oryginal` do ...

Zwracanie przez referencję

```
int& D1o(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1o(Liczba);
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = ++(++Liczba);
    cout << Wynik << Liczba << endl;
}
```

Dla większej prostoty skróćmy nazwę funkcji z Dodaj_1_Oryginal do D1o.

Zwracanie przez referencję

```
int& D1o(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1o(D1o(Liczba));
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = ++(++Liczba);
    cout << Wynik << Liczba << endl;
}
```

Zapis wywołań operatora pre-inkrementacji pokazuje jak to zrobić w przypadku wywołań funkcji D1o. Ale czy oba zapisy są na pewno poprawne?

Zwracanie przez referencję

```
int& D1o(int & Arg)
{
    Arg += 1;
    return Arg;
}
```

Tym razem wszystko jest dobrze dzięki temu, że zwracana jest referencja do zmiennej, a nie tylko jej wartość. Daje to możliwość dalszego dostępu do tej zmiennej.

Uwaga: Tak jak nie można zwracać wskaźników do zmiennych lokalnych funkcji, tak też nie można zwracać do nich referencji

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = D1o(D1o(Liczba));
    cout << Wynik << Liczba << endl;
}
```

```
int main( )
{
    int Liczba=2, Wynik;
    Wynik = ++(++Liczba);
    cout << Wynik << Liczba << endl;
}
```

Przykład definicji przeciążenia operatora pre-inkrementacji

```
enum Symbol { e, a, b };
```

```
Symbol& operator ++ (Symbol& Arg)  
{  
    Symbol TabSymboli[ ] = { e, a, b, e };  
    int Indeks = Arg;  
    Arg = TabSymboli[++Indeks];  
    return Arg;  
}
```

```
int main( )  
{  
    Symbol ZmSym = a;  
    ++++ZmSym;  
}
```

Przykład definicji przeciążenia operatora pre-inkrementacji

```
enum Symbol { e, a, b };
```

```
Symbol& operator ++ (Symbol& Arg)
```

```
{
    Symbol TabSymboli[ ] = { e, a, b, e };
    int Indeks = Arg;
    Arg = TabSymboli[++Indeks];
    return Arg;
}
```

```
int main( )
{
    Symbol ZmSym = a;
    ++++ZmSym;
}
```

Jako parametr przeciążenia tego operatora powinna zostać przekazana referencja do zmiennej. Ona też powinna zostać zwrócona przez polecenie return .

Uwaga: Zarówno ten, jak i każdy inny operator, nie można przeciążać gdy parametr lub parametry (w przypadku operatorów wieloargumentowych) są tylko typu wbudowanego (np. **int**, **char**, itd.).

Przykład definicji przeciążenia operatora pre-inkrementacji

```
enum Symbol { e, a, b };
```

```
Symbol& operator ++ (Symbol& Arg)
```

```
{
    Symbol TabSymboli[] = { e, a, b, e }, *wTab = TabSymboli;
    return Arg = *(++(wTab+Arg));
}
```

Można to samo zapisać w sposób bardziej zwarty.
Dlaczego możliwy jest taki zapis? Czy występują tutaj jakieś niejawne rzutowania?

```
int main( )
```

```
{
    Symbol ZmSym = a;
    ++++ZmSym;
}
```

Przykład definicji przeciążenia operatora post-inkrementacji

```
enum Symbol { e, a, b };  
...  
Symbol operator ++ (Symbol& Arg, int)  
{  
    Symbol Kopia = Arg;  
    ++Arg;  
    return Kopia;  
}  
  
int main( )  
{  
    Symbol ZmSym = a;  
    ( ++ZmSym ) ++;  
}
```


Przykład definicji przeciążenia operatora post-inkrementacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol operator ++ (Symbol& Arg, int)
```

```
{
```

```
    Symbol Kopia = Arg;
```

```
    ++Arg;
```

```
    return Kopia;
```

```
}
```

```
int main( )
```

```
{
```

```
    Symbol ZmSym = a;
```

```
    (++)ZmSym)++;
```

```
}
```

Podobnie jak w poprzednim przykładzie, parametrem przeciążenia operatora post-inkrementacji powinna być referencja do zmiennej. Tym razem jednak przeciążenie to powinno zwrócić wcześniejszą wartość, a nie referencję.

W przedstawionej definicji założono, że w tym samym module znajduje się definicja przeciążenia operatora pre-inkrementacji. Ten sposób definiowania następnego przeciążenia poprawia czytelność samej definicji i zmniejsza prawdopodobieństwo popełnienia błędu.

Przykład użycia przeciążenia operatora inkrementacji

```
enum Symbol { e, a, b };  
...  
Symbol operator - (Symbol Arg)  
{  
    for (Symbol Arg2 = a; Arg2 != e; ++Arg2) {  
        if (Arg + Arg2 == e) return Arg2;  
    }  
    return e;  
}  
  
int main( )  
{  
    Symbol ZmSym = -a;  
    ZmSym = -ZmSym;  
}
```

Przykład użycia przeciążenia operatora inkrementacji

```
enum Symbol { e, a, b };
```

```
...
```

```
Symbol operator - (Symbol Arg)
```

```
{
    for (Symbol Arg2 = a; Arg2 != e; ++Arg2) {
        if (Arg + Arg2 == e) return Arg2;
    }
    return e;
}
```

```
int main( )
```

```
{
    Symbol ZmSym = -a;
    ZmSym = -ZmSym;
}
```

Definicja przeciążenia operatora jednoargumentowego '-' umożliwia wyznaczenie elementu odwrotnego do danego symbolu względem operacji dodawania (operacja dodawania musi być wcześniej zdefiniowana, podobnie jak operator pre-inkrementacji).

Można to zrobić prościej *ręcznie* wyznaczając te elementy. Następnie można je bezpośrednio indeksować.

Przykład użycia przeciążenia operatora inkrementacji

```
enum Symbol { e, a, b };
```

```
Symbol operator – (Symbol Arg)
```

```
{
```

```
    Symbol TabElemOdwr[ ] = { e, b, a };
```

```
    return TabElemOdwr[Arg];
```

```
}
```

Tak wygląda maksymalnie uproszczona definicja przeciążenia operatora „-”.

```
int main( )
```

```
{
```

```
    Symbol ZmSym = -a;
```

```
    ZmSym = -ZmSym;
```

```
}
```

Przykład użycia przeciążenia operatora inkrementacji

```
enum Symbol { e, a, b };
```

```
Symbol operator - (Symbol Arg)
```

```
{
```

```
    Symbol TabElemOdwr[ ] = { e, b, a };
```

```
    return TabElemOdwr[Arg];
```

```
}
```

```
Symbol operator + (Symbol Arg)
```

```
{
```

```
    return Arg;
```

```
}
```

```
int main( )
```

```
{
```

```
    Symbol ZmSym = +a;
```

```
    ZmSym = -ZmSym;
```

```
}
```

Definicja jednoargumentowego operatora „+”
jest jeszcze prostsza.

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 Zapis operacji na strumieniu standardowym
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Problem

Zadanie

Należy napisać funkcję, która umożliwia wczytanie liczb poprzez wywołanie *łańcuchowe*.

Problem

Zadanie

Należy napisać funkcję, która umożliwia wczytanie liczb poprzez wywołanie *łańcuchowe*.

```
int main( )  
{  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm, Num1 ), Num2 );  
}
```


Próba rozwiązania

```
int main( )  
{  
  
    int    Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
};
```

```
int main( )  
{  
    IStream IStrm = { stdin };  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
};
```

```
int main( )  
{  
    IStream IStrm = { stdin };  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
};  
  
IStream &ReadNum(IStream &Strm, int &Num)  
{  
  
}  
  
int main( )  
{  
    IStream IStrm = { stdin };  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
};  
  
IStream &ReadNum(IStream &Strm, int &Num)  
{  
  
    return Strm;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin };  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
};  
  
IStream &ReadNum(IStream &Strm, int &Num)  
{  
    fscanf(Strm.pCStrm,"%i",&Num);  
    return Strm;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin };  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
    bool Fail;  
};  
  
IStream &ReadNum(IStream &Strm, int &Num)  
{  
    Strm.Fail = (fscanf(Strm.pCStrm,"%i",&Num) != 1);  
    return Strm;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin, false };  
    int Num1, Num2;  
  
    ReadNum( ReadNum( IStrm , Num1), Num2 );  
}
```

Próba rozwiązania

```

struct IStream {
    FILE *pCStrm;
    bool Fail;
};

IStream &ReadNum(IStream &Strm, int &Num)
{
    Strm.Fail = (fscanf(Strm.pCStrm,"%i",&Num) != 1);
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    int Num1, Num2;

    ReadNum( ReadNum( IStrm , Num1), Num2 );
    if ( !IStrm.Fail ) return 0;
    cerr << "Bład!!!" << endl;

    ...
}

```


Próba rozwiązania

```

struct IStream {
    FILE *pCStrm;
    bool Fail;
};

...

IStream &ReadChar(IStream &Strm, char &Ch)
{
    Strm.Fail = (fscanf(Strm.pCStrm,"%c",&Ch) != 1);
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    int Num1, Num2;

    ReadNum( ReadNum( IStrm , Num1), Num2 );
    if ( !IStrm.Fail ) return 0;
    cerr << "Bład!!!" << endl;

    ...
}

```

Próba rozwiązania

```

struct IStream {
    FILE *pCStrm;
    bool Fail;
};

...

IStream &ReadChar(IStream &Strm, char &Ch)
{
    Strm.Fail = (fscanf(Strm.pCStrm,"%c",&Ch) != 1);
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    int Num1, Num2;
    char Oper;

    ReadChar( ReadNum( IStrm , Num1), Oper );
    if ( !IStrm.Fail ) return 0;
    cerr << "Bład!!!" << endl;

    ...
}

```

Próba rozwiązania

```

struct IStream {
    FILE *pCStrm;
    bool Fail;
};

...

IStream &ReadChar(IStream &Strm, char &Ch)
{
    if (Strm.Fail) return Strm;
    Strm.Fail = (fscanf(Strm.pCStrm,"%c",&Ch) != 1);
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    int Num1, Num2;
    char Oper;

    ReadChar( ReadNum( IStrm , Num1), Oper );
    if ( !IStrm.Fail ) return 0;
    cerr << "Blad!!!" << endl;

    ...
}

```

Próba rozwiązania

```

struct IStream {
    FILE *pCStrm;
    bool Fail;
};

...

IStream &ReadChar(IStream &Strm, char &Ch)
{
    if (Strm.Fail) return Strm;
    Strm.Fail = (fscanf(Strm.pCStrm,"%c",&Ch) != 1);
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    int Num1, Num2;
    char Oper;

    ReadChar( ReadNum( IStrm , Num1), Oper );
    if ( !IStrm.Fail ) return 0;
    ReadChar( IStrm, Oper );

    ...
}

```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
    bool Fail;  
};  
  
    . . .  
void Clear(IStream &Strm )  
{  
    Strm.Fail = false;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin, false };  
    int      Num1, Num2;  
    char     Oper;  
    ReadChar( ReadNum( IStrm , Num1), Oper );  
    if ( !IStrm.Fail ) return 0;  
    ReadChar( IStrm, Oper );  
    . . .  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
    bool Fail;  
};  
  
    . . .  
void Clear(IStream &Strm )  
{  
    Strm.Fail = false;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin, false };  
    int      Num1, Num2;  
    char     Oper;  
    ReadChar( ReadNum( IStrm , Num1), Oper );  
    if ( !IStrm.Fail ) return 0;  
    Clear( IStrm );  
    ReadChar( IStrm, Oper );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
    bool Fail;  
};  
  
    . . .  
void Clear(IStream &Strm )  
{  
    Strm.Fail = false;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin, false };  
    Symbol Sym;  
    char Oper;  
    ReadChar( ReadSymbol( IStrm , Sym), Oper );  
    if ( !IStrm.Fail ) return 0;  
    Clear( IStrm );  
    ReadChar( IStrm, Oper );  
}
```

Próba rozwiązania

```
struct IStream {  
    FILE *pCStrm;  
    bool Fail;  
};  
  
    . . .  
void Clear(IStream &Strm )  
{  
    Strm.Fail = false;  
}  
  
int main( )  
{  
    IStream IStrm = { stdin, false };  
    Symbol Sym;  
    char Oper;  
    ReadChar( ReadSymbol( IStrm , Sym), Oper );  
    if ( !IStrm.Fail ) return 0;  
    Clear( IStrm );  
    ReadChar( IStrm, Oper );  
}
```


Próba rozwiązania

...

```
IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
{
```

```
}
```

```
int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    ReadChar( ReadSymbol( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    ReadChar( IStrm, Oper );
}
```

Próba rozwiązania

...

```
IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
{
```

```
    if (Strm.Fail) return Strm;
```

```
    return Strm;
}
```

```
int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    ReadChar( ReadSymbol( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    ReadChar( IStrm, Oper );
}
```

Próba rozwiązania

...

```
IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
```

```
{
    char Ch = 'x';
```

```
    if (Strm.Fail) return Strm;
```

```
    ReadChar(Strm,Ch);
```

```
    return Strm;
}
```

```
int main( )
```

```
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
```

```
    char Oper;
```

```
    ReadChar( ReadSymbol( IStrm , Sym), Oper );
```

```
    if ( !IStrm.Fail ) return 0;
```

```
    Clear( IStrm );
```

```
    ReadChar( IStrm, Oper );
```

```
}
```

Próba rozwiązania

...

```
IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
```

```
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
```

```
    if (Strm.Fail) return Strm;
    ReadChar(Strm,Ch);
    if ((pSymCh = strchr(SymChars,Ch)))
```

```
        return Strm;
}
```

```
int main( )
```

```
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    ReadChar( ReadSymbol( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    ReadChar( IStrm, Oper );
```

Próba rozwiązania

...

```

IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };

    if (Strm.Fail) return Strm;
    ReadChar(Strm,Ch);
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh–SymChars]; }

    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    ReadChar( ReadSymbol( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    ReadChar( IStrm, Oper );
}

```

Próba rozwiązania

...

```

IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };

    if (Strm.Fail) return Strm;
    ReadChar(Strm,Ch);
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    ReadChar( ReadSymbol( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    ReadChar( IStrm, Oper );
}

```

Próba rozwiązania

```

...
IStream& ReadNum(IStream& Strm, int& Num)
{
    ...
}

IStream& ReadChar(IStream& Strm, char& Ch)
{
    ...
}

IStream& ReadSymbol(IStream& Strm, Symbol& Sym)
{
    ...
}

int main( )
{
    IStream  IStrm = { stdin, false };
    Symbol   Sym;
    char     Oper;

    ReadChar( ReadSymbol( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    ReadChar( IStrm, Oper );
}

```

Próba rozwiązania

```

...
IStream& Read(IStream& Strm, int& Num)
{
    ...
}

IStream& Read(IStream& Strm, char& Ch)
{
    ...
}

IStream& Read(IStream& Strm, Symbol& Sym)
{
    ...
}

int main( )
{
    IStream  IStrm = { stdin, false };
    Symbol  Sym;
    char    Oper;

    Read( Read( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    Read( IStrm, Oper );
}

```


Próba rozwiązania

...

```

IStream& Read(IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };

    if (Strm.Fail) return Strm;
    Read(Strm,Ch);
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    Read( Read( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    Read( IStrm, Oper );
}

```

Próba rozwiązania

...

```

IStream& operator >> (IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };

    if (Strm.Fail) return Strm;
    operator >> (Strm,Ch);
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    operator >> ( operator >> ( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    operator >> ( IStrm, Oper );
}

```

Próba rozwiązania

...

```
IStream& operator >> (IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.Fail) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}
```

```
int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;
    operator >> ( operator >> ( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    operator >> ( IStrm, Oper );
}
```

Próba rozwiązania

...

```

IStream& operator >> (IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.Fail) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;
    operator >> ( operator >> ( IStrm , Sym), Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    IStrm >> Oper;
}

```

Próba rozwiązania

...

```

IStream& operator >> (IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };

    if (Strm.Fail) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;

    operator >> ( IStrm >> Sym, Oper );
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    IStrm >> Oper;
}

```

Próba rozwiązania

...

```

IStream& operator >> (IStream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.Fail) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { ungetc(Ch,Strm.pCStrm); Strm.Fail = true; }
    return Strm;
}

int main( )
{
    IStream IStrm = { stdin, false };
    Symbol Sym;
    char Oper;
    IStrm >> Sym >> Oper;
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    IStrm >> Oper;
}

```

Próba rozwiązania

```

...
istream& operator >> (istream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.fail( )) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { Strm.unget( ); Strm.setstate(ios::failbit); }
    return Strm;
}

int main( )
{

    Symbol Sym;
    char Oper;
    IStrm >> Sym >> Oper;
    if ( !IStrm.Fail ) return 0;
    Clear( IStrm );
    IStrm >> Oper;
}

```

Próba rozwiązania

```

...
istream& operator >> (istream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.fail( )) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh–SymChars]; }
    else { Strm.unget( ); Strm.setstate(ios::failbit); }
    return Strm;
}

int main( )
{

    Symbol Sym;
    char Oper;
    cin >> Sym >> Oper;
    if ( !IStream.Fail ) return 0;
    Clear( IStream );
    IStream >> Oper;
}

```


Próba rozwiązania

...

```

istream& operator >> (istream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.fail( )) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh–SymChars]; }
    else { Strm.unget( ); Strm.setstate(ios::failbit); }
    return Strm;
}

int main( )
{

    Symbol Sym;
    char Oper;
    cin >> Sym >> Oper;
    if ( !cin.fail( ) ) return 0;
    Clear( IStrm );
    IStrm >> Oper;
}

```

Próba rozwiązania

...

```

istream& operator >> (istream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.fail( )) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh–SymChars]; }
    else { Strm.unget( ); Strm.setstate(ios::failbit); }
    return Strm;
}

int main( )
{

    Symbol Sym;
    char Oper;
    cin >> Sym >> Oper;
    if ( !cin.fail( ) ) return 0;
    cin.clear( );
    IStrm >> Oper;
}

```

Próba rozwiązania

...

```

istream& operator >> (istream& Strm, Symbol& Sym)
{
    char Ch = 'x';
    char const *SymChars = "eabcd", *pSymCh;
    Symbol SymTab[ ] = {e, a, b, c, d };
    if (Strm.fail( )) return Strm;
    Strm >> Ch;
    if ((pSymCh = strchr(SymChars,Ch))) { Sym = SymTab[pSymCh-SymChars]; }
    else { Strm.unget( ); Strm.setstate(ios::failbit); }
    return Strm;
}

int main( )
{

    Symbol Sym;
    char Oper;
    cin >> Sym >> Oper;
    if ( !cin.fail( ) ) return 0;
    cin.clear( );
    cin >> Oper;
}

```

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 Zapis operacji na strumieniu standardowym
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Lewostronna łączność operatora

```
cout << "Napis";
```

Operacje wejścia/wyjścia są zdefiniowane w oparciu o standardowe mechanizmy dostępne na poziomie języka *C++*. W tym sensie biblioteka standardowa nie wykorzystuje żadnych dodatkowych mechanizmów, które byłyby spoza definicji języka, tak jak to ma miejsce np. w *Pascalu*.

Lewostronna łączność operatora

```
cout << "Napis";
```

```
operator << (cout, "Napis");
```

Operacje wejścia/wyjścia są zdefiniowane w oparciu o standardowe mechanizmy dostępne na poziomie języka *C++*. W tym sensie biblioteka standardowa nie wykorzystuje żadnych dodatkowych mechanizmów, które byłyby spoza definicji języka, tak jak to ma miejsce np. w *Pascalu*.

Lewostronna łączność operatora

```
cout << "Napis";
```

```
operator << (cout, "Napis");
```

```
cout << "Napis" << endl;
```

Operacje wejścia/wyjścia są zdefiniowane w oparciu o standardowe mechanizmy dostępne na poziomie języka *C++*. W tym sensie biblioteka standardowa nie wykorzystuje żadnych dodatkowych mechanizmów, które byłyby spoza definicji języka, tak jak to ma miejsce np. w *Pascalu*.

Lewostronna łączność operatora

```
cout << "Napis";
```

```
operator << (cout, "Napis");
```

```
cout << "Napis" << endl;
```

```
operator << ( operator << (cout,"Napis"), endl);
```

Operacje wejścia/wyjścia są zdefiniowane w oparciu o standardowe mechanizmy dostępne na poziomie języka *C++*. W tym sensie biblioteka standardowa nie wykorzystuje żadnych dodatkowych mechanizmów, które byłyby spoza definicji języka, tak jak to ma miejsce np. w *Pascalu*.

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 **Zapis operacji na strumieniu standardowym**
 - Połączenie kaskadowe operacji zapisu
 - **Przeciążenie operatora zapisu do strumienia standardowego**
 - Przeciążenie operatora czytania ze strumienia standardowego
 - Rozpoznawanie i reakcja na błąd czytania

Wyświetlanie z wykorzystaniem funkcji

...

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

```
void Wyswietl( Element x)  
{  
    const char *Nazwa = "abc";  
    cout << Nazwa[x];  
}
```

```
int main( )  
{  
    Element x = a;  
  
    cout << "Wartosc zmiennej x: ";  
    Wyswietl( x );  
    cout << endl;  
}
```

Wyświetlanie z wykorzystaniem funkcji

...

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

```
void Wyswietl( Element x)  
{  
    const char *Nazwa = "abc";  
    cout << Nazwa[x];  
}
```

```
int main( )  
{  
    Element x = a;  
  
    cout << "Wartosc zmiennej x: ";  
    Wyswietl( x );  
    cout << endl;  
}
```

Tworzenie specjalizowanej funkcji do wyświetlania wartości zmiennej danego typu jest podejściem właściwym dla języka C (gdyż nie ma tam innej możliwości). Oczywiście wadą takiego podejścia jest brak elastyczności.

Przeciążenie operacji dla strumienia wyjściowego

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

```
ostream & operator << ( ostream & StrmWy, Element x )  
{  
    StrmWy << "abc"[x];  
    return StrmWy;  
}
```

```
int main( )  
{  
    Element Zm = a;  
    cout << "Wartosc zmiennej z: " << Zm << endl;  
}
```

Przeciążenie operacji dla strumienia wyjściowego

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

```
ostream & operator << ( ostream & StrmWy, Element x )
{
    StrmWy << "abc" [x];
    return StrmWy;
}
```

```
int main( )
{
    Element Zm = a;
    cout << "Wartosc zmiennej z: " << Zm << endl;
}
```

Przy definiowaniu przeciążenia muszą być spełnione dwa warunki:

1. Pierwszy parametr musi być klasy **ostream** i musi on być przekazywany przez referencję.
2. Przeciążenie operatora musi zwracać referencję do pierwszego parametru klasy **ostream**.

Przeciążenie operacji dla strumienia wyjściowego

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

```
ostream & operator << ( ostream & StrmWy, Element x )
{
    return StrmWy << "abc" [x];
}
```

```
int main( )
{
    Element Zm = a;
    cout << "Wartosc zmiennej z: " << Zm << endl;
}
```

Dzięki temu, że przeciążenie operatora << zwraca zawsze referencję do obiektu **ostream**, możliwy jest znacznie bardziej zwarty zapis.

Operacje wyświetlania dla dowolnego typu wyliczeniowego

```
enum Stan { Uspienie, Czuwanie, Aktywny };
```

```
ostream & operator << ( ostream & StrmWy, Stan x )
{
    const char *Nazwa[ ] = { "Uspienie", "Czuwanie", "Aktywny" };
    return StrmWy << Nazwa[x];
}
```

```
int main( )
{
    Stan Zm = Czuwanie;
    cout << "Wartosc zmiennej z: " << Zm << endl;
}
```

Tłumaczenie wartości stanu na nazwę można łatwo zrealizować wykorzystując tablicę napisów.

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 **Zapis operacji na strumieniu standardowym**
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - **Przeciążenie operatora czytania ze strumienia standardowego**
 - Rozpoznawanie i reakcja na błąd czytania

Przeciążenie operacji dla strumienia wejściowego

enum *Element* { a, b, c };

```
int main( )  
{  
    Element Zm;  
    cin >> Zm  
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )  
{
```

```
    ...
```

```
}
```

```
int main( )  
{  
    Element Zm;  
    cin >> Zm  
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )  
{
```

Przy definiowaniu przeciążenia muszą być spełnione trzy warunki:

- 1 Pierwszy parametr musi być klasy **istream** i musi on być przekazywany przez referencję.
- 2 Przeciążenie operatora musi zwracać referencję do pierwszego parametru klasy **istream**.
- 3 Drugi parametr musi być przekazany przez referencję.

```
}  
int main( )  
{  
    Element Zm;  
    cin >> Zm  
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{  
    Element TabSymboli[ ] = { a, b, c };  
    char      ZnakSymbolu;
```

```
}
```

```
int main( )
```

```
{  
    Element Zm;
```

```
    cin >> Zm
```

```
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
  Element TabSymboli[ ] = { a, b, c };
```

```
  char      ZnakSymbolu;
```

```
  StrmWe >> ZnakSymbolu;
```

```
}
```

```
int main( )
```

```
{
```

```
  Element Zm;
```

```
  cin >> Zm
```

```
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
    Element TabSymboli[ ] = { a, b, c };
    char      ZnakSymbolu;
```

```
    StrmWe >> ZnakSymbolu;
```

```
    if ( strchr("abc", ZnakSymbolu) ) {WczytSym = TabSymboli[ZnakSymbolu-'a'];}
```

```
        else { StrmWe.unget(); StrmWe.setstate(ios::failbit); }
```

```
}
```

```
int main( )
```

```
{
```

```
    Element Zm;
```

```
    cin >> Zm
```

```
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
    Element TabSymboli[ ] = { a, b, c };
    char      ZnakSymbolu;

    StrmWe >> ZnakSymbolu;
    if ( strchr("abc", ZnakSymbolu) ) {WczytSym = TabSymboli[ZnakSymbolu-'a'];}
        else { StrmWe.unget(); StrmWe.setstate(ios::failbit); }
    return StrmWe;
}
```

```
int main( )
```

```
{
    Element Zm;
    cin >> Zm
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
    Element TabSymboli[ ] = { a, b, c };
    char      ZnakSymbolu;

    StrmWe >> ZnakSymbolu;
    if ( strchr("abc", ZnakSymbolu) ) {WczytSym = TabSymboli[ZnakSymbolu-'a'];}
        else { StrmWe.unget(); StrmWe.setstate(ios::failbit); }
    return StrmWe;
}
```

```
int main( )
```

```
{
    Element Zm;
    cin >> Zm
}
```


Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
    Element TabSymboli[ ] = { a, b, c };
    char      ZnakSymbolu;

    StrmWe >> ZnakSymbolu;
    if ( strchr("abc", ZnakSymbolu) ) {WczytSym = TabSymboli[ZnakSymbolu-'a'];}
        else { StrmWe.unget(); StrmWe.setstate(ios::failbit); }
    return StrmWe;
}
```

```
int main( )
```

```
{
    Element Zm;
    cin >> Zm
}
```

Problem

Mając zdefiniowane przeciążenie jak wczytać z wejścia standardowego zapis operacji:

a+c

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
    Element TabSymboli[ ] = { a, b, c };
    char      ZnakSymbolu;
```

```
    ...
```

```
}
int main( )
{
    Element Arg1, Arg2;
    char      Oper;
    cin >> Arg1 >> Oper >> Arg2
}
```

Rozwiązanie (baaaardzo proste)

a+c

Procedura staje się bardzo prosta. Wczytujemy pierwszy argument jako symbol, znak operatora, a następnie drugi argument także jako symbol.

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
    Element TabSymboli[ ] = { a, b, c };
    char      ZnakSymbolu;
```

To nie koniec problemów :(

...

Jak sprawdzić, że operacja czytania nie powiodła się?

```
}
int main( )
{
    Element Arg1, Arg2;
    char      Oper;
    cin >> Arg1 >> Oper >> Arg2
}
```

Przeciążenie operacji dla strumienia wejściowego

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

```
istream & operator >> ( istream & StrmWe, Element & WczytSym )
```

```
{
```

```
    ...
```

```
}
```

Rozwiązanie (nie takie trudne)

Należy w tym celu sprawdzić jaki wynik zwraca metoda `fail` dla obiektu `cin`. Metoda ta może zwrócić wartość `true` (gdy operacja powiodła się) lub `false` (w przypadku przeciwnym).

```
int main( )
```

```
{
```

```
    Element Arg1, Arg2;
```

```
    char Oper;
```

```
    cin >> Arg1 >> Oper >> Arg2
```

```
    if ( cin.fail( ) ) { cerr << "Bład" << endl; }
```

```
    ...
```

```
}
```

Spis treści

- 1 Typu obiektowego
 - Typy obiektowe, hierarchie typów
- 2 Przeciążanie operatorów
 - Łączność operatorów i porządek wykonywania operacji
- 3 Wykorzystanie referencji
 - Przekazywanie parametrów przez referencję i zwracanie referencji
 - Operacje czytania
- 4 **Zapis operacji na strumieniu standardowym**
 - Połączenie kaskadowe operacji zapisu
 - Przeciążenie operatora zapisu do strumienia standardowego
 - Przeciążenie operatora czytania ze strumienia standardowego
 - **Rozpoznawanie i reakcja na błąd czytania**

Jak radzić sobie z błędami czytania

```
int main( )  
{  
    int Arg;
```

Proste zadanie

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

- 1 wykryć błędne wprowadzenie liczby,
- 2 poinformować użytkownika o błędzie,
- 3 zignorować wprowadzony napis,
- 4 ponownie podjąć próbę czytania,
- 5 i tak aż do skutku.

Jak radzić sobie z błędami czytania

```
int main( )  
{  
    int Arg;  
    cin >> Arg;
```

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

- 1 wykryć błędne wprowadzenie liczby,
- 2 poinformować użytkownika o błędzie,
- 3 zignorować wprowadzony napis,
- 4 ponownie podjąć próbę czytania,
- 5 i tak aż do skutku.

Rozwiązanie

Najpierw musimy podjąć próbę wczytania liczby.

Jak radzić sobie z błędami czytania

```
int main( )
{
    int Arg;
    cin >> Arg;
    while ( cin.fail( ) ) {

    }
}
```

Program ma wczytać liczbę. *Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis.* Program musi:

- 1 *wykryć błędne wprowadzenie liczby,*
- 2 *poinformować użytkownika o błędzie,*
- 3 *zignorować wprowadzony napis,*
- 4 *ponownie podjąć próbę czytania,*
- 5 *i tak aż do skutku.*

Rozwiązanie

Następnie należy sprawdzić czy operacja się powiodła. Jeżeli nie (metoda `fail` zwróci wartość `true`), to należy rozpocząć pętlę, w której próby wczytania będą podejmowane, aż do pozytywnego skutku.

Jak radzić sobie z błędami czytania

```
int main( )
{
    int Arg;
    cin >> Arg;
    while ( cin.fail( ) ) {
        cout << "Bład. Wpisz ponownie." << endl;
    }
}
```

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

- 1 wykryć błędne wprowadzenie liczby,
- 2 **poinformować użytkownika o błędzie**,
- 3 zignorować wprowadzony napis,
- 4 ponownie podjąć próbę czytania,
- 5 i tak aż do skutku.

Rozwiązanie

Najpierw musimy użytkownika poinformować o błędzie.

Jak radzić sobie z błędami czytania

```
int main( )
{
    int Arg;
    cin >> Arg;
    while ( cin.fail( ) ) {
        cout << "Bład. Wpisz ponownie." << endl;
        cin.clear( );
    }
}
```

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

- 1 wykryć błędne wprowadzenie liczby,
- 2 poinformować użytkownika o błędzie,
- 3 zignorować wprowadzony napis,
- 4 ponownie podjąć próbę czytania,
- 5 i tak aż do skutku.

Rozwiązanie

Błąd wykonania operacji w prowadzi strumień w stan *fail*. To powoduje, że od tego momentu nie można wykonać jakiegokolwiek operacji wczytania (będą one ignorowane).

Aby wyprowadzić strumień z tego stanu i przywrócić mu możliwość wykonywania dalszych operacji, należy wywołać metodę `clear`.

Jak radzić sobie z błędami czytania

```
int main( )
{
    int Arg;
    cin >> Arg;
    while ( cin.fail( ) ) {
        cout << "Bład. Wpisz ponownie." << endl;
        cin.clear( );
        cin.ignore(10000, '\n');
    }
}
```

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

- 1 wykryć błędne wprowadzenie liczby,
- 2 poinformować użytkownika o błędzie,
- 3 **zignorować wprowadzony napis**,
- 4 ponownie podjąć próbę czytania,
- 5 i tak aż do skutku.

Rozwiązanie

Teraz możemy zignorować cały napis, którą błędnie wprowadził użytkownik. Będąc bardziej precyzyjnym, metoda `ignore` ignoruje 10000 znaków, chyba, że wcześniej napotka znak przejścia do nowej linii (użytkownik wprowadza go naciskając klawisz Enter). Wszystko będzie dobrze działać pod warunkiem, że użytkownik w jednej linii nie wprowadzi więcej niż 10000 znaków. Można śmiało założyć, że prędzej się zmęczy i da spokój ;)

Jak radzić sobie z błędami czytania

```
int main( )
{
    int Arg;
    cin >> Arg;
    while ( cin.fail( ) ) {
        cout << "Bład. Wpisz ponownie." << endl;
        cin.clear( );
        cin.ignore(10000, '\n');
        cin >> Arg;
    }
}
```

Program ma wczytać liczbę. Użytkownik może jednak wprowadzić błędnie zamiast liczby, jakiś napis. Program musi:

- 1 wykryć błędne wprowadzenie liczby,
- 2 poinformować użytkownika o błędzie,
- 3 zignorować wprowadzony napis,
- 4 **ponownie podjąć próbę czytania,**
- 5 i tak aż do skutku.

Rozwiązanie

Wykonujemy ponowną próbę wczytania liczby.

Koniec prezentacji
Dziękuję za uwagę