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©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`

Spis treści

Spis treści

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

maszyna

↑

pojazd

↑

samochód

↑

Ferrante V Concept

Czym jest klasa

Klasa jest typem obiektowym

Co cechuje podejście obiektowe

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

Co cechuje podejście obiektowe

Cechą charakterystyczną podejścia obiektowego i języków obiektowych jest ścisłe 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 ścisłe 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

Łączność działania – porządek wykonywania operacji

```
struct   LZespolona  { double re, im; };
 ...
LZespolona  Dodaj( LZespolona  Skl1, LZespolona  Skl2 );
LZespolona  operator + ( LZespolona  Skl1, LZespolona  Skl2 );
 ...

int  main( )
{
     LZespolona     y, x, wynik;




 }
```

Łączność działania – porządek wykonywania operacji

```
struct  LZespolona { double re, im; };
 ...
LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 );
LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 );
 ...

int main( )
{
    LZespolona    y, x, wynik;
    wynik = y + x + y;


 }
```

Łączność działania – porządek wykonywania operacji

```
struct  LZespolona { double re, im; };
 ...
LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 );
LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 );
 ...

int main( )
{
    LZespolona    y, x, wynik;
    wynik = (y + x) + y;


 }
```

Łączność działania – porządek wykonywania operacji

```
struct  LZespolona { double re, im; };
 ...
LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 );
LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 );
 ...

int main( )
{
    LZespolona    y, x, wynik;
    wynik = (y + x) + y;
    wynik = operator+ (operator+ (y ,x), y );

 }
```

Łączność działania – porządek wykonywania operacji

```
struct  LZespolona { double re, im; };
 ...
LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 );
LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 );
 ...

int main( )
{
    LZespolona    y, x, wynik;

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

Jawna zmiana porządku wykonywanych operacji

```
struct  LZespolona  { double re, im; };
 ...
LZespolona  Dodaj( LZespolona  Skl1, LZespolona  Skl2 );
LZespolona  operator + ( LZespolona  Skl1, LZespolona  Skl2 );
 ...

int  main( )
{
     LZespolona     y = a, x = b, wynik;
     wynik = y + (x + y);


 }
```

Jawna zmiana porządku wykonywanych operacji

```
struct  LZespolona { double re, im; };
 ...
LZespolona Dodaj( LZespolona Skl1, LZespolona Skl2 );
LZespolona operator + ( LZespolona Skl1, LZespolona Skl2 );
 ...

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

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

 }
```

Jawna zmiana porządku wykonywanych operacji

```
struct  LZespolona  { double re, im; };
 ...
LZespolona  Dodaj( LZespolona  Skl1, LZespolona  Skl2 );
LZespolona  operator + ( LZespolona  Skl1, LZespolona  Skl2 );
 ...

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

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

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

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

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

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

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 lewostronne łącznie gdy:

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

Działanie jest prawostronne łącznie 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

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

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 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.

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.

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:

`suma = x + y + z` $\longrightarrow$ `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:

```
suma = x + y + z   ⟶   suma = ((x + y) + z).
```

Przykład operatora prawostronnie łącznego:

```
x = y = z = 5   ⟶   (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	*wskaźnik* **[** *wyrażenie* **]**
Wywołanie funkcji	*wyrażenie* **(** *lista_wyrażeń* **)**
Przyrostkowa inkrementacja	*l-wartość* ++
Przyrostkowa dekrementacja	*l-wartość* −−
Przedrostkowa inkrementacja	++ *l-wartość*
Przedrostkowa dekrementacja	−− *l-wartość*
Negacja bitowa	∼*wyrażenie*
Negacja logiczna	**!** *wyrażenie*
Minus jednoargumentowy	− *wyrażenie*
Plus jednoargumentowy	+ *wyrażenie*
Mnożenie	*wyrażenie* ∗ *wyrażenie*
Dzielenie	*wyrażenie* / *wyrażenie*
Modulo	*wyrażenie* **%** *wyrażenie*
Dodawanie	*wyrażenie* + *wyrażenie*
Odejmowanie	*wyrażenie* − *wyrażenie*
Przesuwanie w lewo	*wyrażenie* << *wyrażenie*
Przesuwanie w prawo	*wyrażenie* >> *wyrażenie*
Mniejszy	*wyrażenie* < *wyrażenie*
Mniejszy lub równy	*wyrażenie* <= *wyrażenie*
Większy	*wyrażenie* > *wyrażenie*
Większy lub równy	*wyrażenie* >= *wyrażenie*

⇓

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

⇓

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

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

Dodatkowe oznaczenia operatorów

&&	$\longrightarrow$	and
||	$\longrightarrow$	or
!	$\longrightarrow$	not
!=	$\longrightarrow$	not_eq
&	$\longrightarrow$	bitand
&=	$\longrightarrow$	and_eq
|	$\longrightarrow$	bitor
|=	$\longrightarrow$	or_eq
^	$\longrightarrow$	xor
^=	$\longrightarrow$	xor_eq
~	$\longrightarrow$	compl

Spis treści

Przekazywanie parametrów przez referencję – ciąg dalszy

```
int Dodaj_1_Kopia(int  & Wart)
{




}

int main( )
{




}
```

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




}

int  main( )
{




}
```

Zdefiniujmy 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 liczbową, 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 przekazwania 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 przypadków 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.

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

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.

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

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.

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

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 zdefiniujmy 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;
}
```

Dokonujemy 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 bezpośrednie 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-inkrementacji 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;
}
```

Spis treści

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 << "Blad!!!" << 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 << "Blad!!!" << 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 << "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;
    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   {  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   {  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   {  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   {  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   {  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   {  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   {  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.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.setstate(ios::failbit); }
   return  Strm;
}

int  main( )
{

   Symbol    Sym;
   char      Oper;
   cin >> 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.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.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.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

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

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). Oczywistą 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;
}
```

Definicja tego przeciążenia musi spełniać 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 typu wyliczeniowego Operator

```
enum   Operator { Op_Dodaj, Op_Odejmij };

ostream & operator  << ( ostream & StrmWy, Operator Op )
{
   const char ZnakOp[ ] = "+-";
   return   StrmWy << ZnakOp[Op];
}

int  main( )
{
   Operator   Op = Op_Dodaj;
   cout <<  "Wartosc zmiennej Op: " << Op << endl;
}
```

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

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

Przeciążenie operacji dla strumienia wejściowego

```
enum  Operator { Op_Dodaj, Op_Odejmij, };
```

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

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator  >> ( istream & StrmWe, Operator & WczytSym )
{

                        . . .



}

int  main( )
{
    Operator   Zm;

    cin >>  Zm
}
```

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
}

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

Definicja tego przeciążenia musi spełniać 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.

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;









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

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

   StrmWe >> CzytSymOp;




}
int  main( )
{
    Operator   Zm;

    cin >>  Zm
}
```

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

   StrmWe >> CzytSymOp;
   if  ( (wSymOp = strchr(SymOp,CzytSymOp)) == nullptr) { StrmWe.setstate(ios::failbit); }
         else  {WczytSym = TabTypOp[wSymOp - CzytSymOp];}

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

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

   StrmWe >> CzytSymOp;
   if  ( (wSymOp = strchr(SymOp,CzytSymOp)) == nullptr) { StrmWe.setstate(ios::failbit); }
         else  {WczytSym = TabTypOp[wSymOp - CzytSymOp];}
   return  StrmWe;
}
int  main( )
{
    Operator   Zm;
    cin >>  Zm
}
```

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

   StrmWe >> CzytSymOp;
   if  ( (wSymOp = strchr(SymOp,CzytSymOp)) == nullptr) { StrmWe.setstate(ios::failbit); }
         else  {WczytSym = TabTypOp[wSymOp - CzytSymOp];}
   return  StrmWe;
}
int  main( )
{
    Operator   Zm;
    cin >>  Zm
}
```

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

   StrmWe >> CzytSymOp;
   if  ( (wSymOp = strchr(SymOp,CzytSymOp)) == nullptr) { StrmWe.setstate(ios::failbit); }
         else  {WczytSym = TabTypOp[wSymOp - CzytSymOp];}
   return  StrmWe;
}
int  main( )
{
    Operator   Zm;
    cin >>  Zm
}
```

Problem

Mając zdefiniowane przeciążenie jak wczytać z wejścia standardowego zapis operacji:

$$(2+3i)+(4-2i)$$

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

                . . .

}
int  main( )
{
   Operator   Oper;
   LZespolona  Arg1, Arg2;

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

Rozwiązanie (baaaardzo proste)

$$(2+3i)+(4-2i)$$

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

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & WczytSym )
{
   Operator     TabTypOp[ ] = { Op_Dodaj, Op_Odejmij,};
   const  char  TabSymOp[ ] = "+-",  *wSymOp;
   char         CzytSymOp;

                . . .
}
int  main( )
{
   Operator   Oper;
   LZespolona  Arg1, Arg2;

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

To nie koniec problemów :(

Jak sprawdzić, że operacja czytania nie powiodła się?

Przeciążenie operacji dla strumienia wejściowego

```
enum   Operator { Op_Dodaj, Op_Odejmij, };

istream & operator >> ( istream & StrmWe, Operator & 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( )
{
   Operator   Oper;
   LZespolona  Arg1, Arg2;

   cin >>  Arg1 >>  Oper >>  Arg2
   if  ( cin.fail( ) ) { cerr << "Blad" << endl; }

   . . .
}
```

Spis treści

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Dla części urojonej wymuszone zostaje uwidocznienie znaku liczby.

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Wyświetlona wartość:
2+5i

Dla części urojonej wymuszone zostaje uwidocznienie znaku liczby.

Operacje zapisu do pliku

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona   LZesp;
     ofstream          PlikWy;

     PlikWy.open("nowy_test.txt");
     if (!PlikWy.is_open()) return 1;
     LZesp.re = 2;    LZesp.im = 5;

     PlikWy <<  "Liczba zespolona: " << LZesp << endl;
     if (PlikWy.fail())  cerr << "Blad operacji zapisu\n";
     PlikWy.close();
}
```

W analogiczny sposób możemy daną liczbę zapisać do pliku.
Pamiętać jedynie należy aby dołączyć plik nagłówkowy `fstream`.

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Wyświetlona wartość:
2+5i

Czy w tym przypadku jedynym możliwym sposobem przekazania parametru jest przekazanie go przez wartość?

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, const LiczbaZespolona &Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Wyświetlona wartość:
2+5i

Można go również przekazać przez referencję do obiektu stałego. W tym konkretnym przypadku jest to bardziej właściwe.

Operacje czytania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

istream & operator >>  ( istream & StrmWy, LiczbaZespolona & Lz )
{
     ...
}

int  main( )
{
     LiczbaZespolona LZesp;
     cin >>  LZesp;
}
```

Natomiast dla tego przypadku przekazywanie parametru przez referencję jest **konieczne**. Musi to być jednak referencja do obiektu modyfikowalnego.

Operacje czytania dla własnej struktury

```
struct  LiczbaZespolona { float  re, im; };

istream & operator >>  ( istream & StrmWe, LiczbaZespolona & Lz )
{
      StrmWe >>  Lz.re;
      StrmWe >>  Lz.im;
      StrmWe.ignore( );
      return  StrmWe;
}

int  main( )
{
      ...
}
```

Wczytywana liczba:
2+5i

Prezentowana implementacja czytania liczby zespolonej jest bardzo uproszczona (brak reakcji na błędy). Realizacja czytania: wczytane zostają dwie kolejne liczby, następnie zostaje pominięty pierwszy znak znajdujący się za drugą liczbą.

Operacje czytania dla własnej struktury

```
struct  LiczbaZespolona { float  re, im; };

istream & operator >>  ( istream & StrmWe, LiczbaZespolona & Lz )
{
     StrmWe >>  Lz.re;
     StrmWe >>  Lz.im;
     StrmWe.ignore( );
     return  StrmWe;
}

int  main( )
{
     LiczbaZespolona  LZesp;
     cin >>  LZesp;
     if ( cin.fail() ) {
        cerr << "Blad formatu liczby zespolonej" <<endl;
     }
}
```

Wczytywana liczba:
Napis+5i

Operacje czytania dla własnej struktury

```
struct  LiczbaZespolona { float  re, im; };

istream & operator >>  ( istream & StrmWe, LiczbaZespolona & Lz )
{
     StrmWe >>  Lz.re;
     StrmWe >>  Lz.im;
     StrmWe.ignore( );
     return  StrmWe;
}

int  main( )
{
     LiczbaZespolona  LZesp;
     cin >>  LZesp;
     if ( cin.fail() ) {
        cerr << "Blad formatu liczby zespolonej" <<endl;
     }
}
```

Jeśli ta operacja nie powiedzie się, to następne również.

Wczytywana liczba:
Napis+5i

Operacje czytania dla własnej struktury

```
struct  LiczbaZespolona { float  re, im; };

istream & operator >>  ( istream & StrmWe, LiczbaZespolona & Lz )
{
     StrmWe >>  Lz.re;
     StrmWe >>  Lz.im;
     StrmWe.ignore( );
     return  StrmWe;
}

int  main( )
{
     LiczbaZespolona  LZesp;
     cin >>  LZesp;
     if ( cin.fail() ) {
        cerr << "Blad formatu liczby zespolonej" <<endl;
        cin.clear( );
     }
}
```

Jeśli ta operacja nie powiedzie się, to następne również.

Wczytywana liczba:
Napis+5i

Aby móc ewentualnie czytać dalej.

Operacje czytania dla własnej struktury

```
struct  LiczbaZespolona { float  re, im; };

istream & operator >>  ( istream & StrmWe, LiczbaZespolona & Lz )
{
     StrmWe >>  Lz.re >>  Lz.im;
     StrmWe.ignore( );
     return  StrmWe;
}


int  main( )
{
     LiczbaZespolona  LZesp;
     cin >>  LZesp;
     if ( cin.fail() ) {
        cerr << "Blad formatu liczby zespolonej" <<endl;
        cin.clear( );
     }
}
```

Bardziej zwarty zapis, taki sam efekt końcowy.

Wczytywana liczba:
Napis+5i

Koniec prezentacji
Dziękuję za uwagę