Przykład – klasa biblioteczna `string`

- C++ pozwala na dwie reprezentacje napisów
- napisy w stylu języka C
  - napisy w stylu języka C++

Napisy w stylu języka C

- Ciąg znaków zakończony znakiem `'\0'` , przechowywany w tablicy znakowej.
- Deklarowania napisu w stylu języka C:

```
char s[]="witam"; // tablica będzie miała rozmiar 5+1
char *s1="tak też można"; // inicjalizacja wskaźnikiem do napisu
```

- Przykład funkcji działającej na napisach:

```
int napis_dlugosc(const char *nap)
{
  int dlugosc=0;
  if (nap)
    while (*dlugosc++)
      ++dlugosc;
  return dlugosc;
}
```

- Korzystanie ze standardowych funkcji na napisach w stylu C:

  `#include <cstring>` (stanowi on odpowiednik pliku `<string.h>` języka C).

- Przykłady często używanych funkcji z biblioteki `<cstring>`:

```
// obliczenie długości napisu
int strlen(const char* );  // bez znaku końca napisu

// porównanie dwóch napisów
int strcmp(const char*, const char* );
int strncmp(const char*, const char*, int); // porównuj tylko n znaków

// skopiowanie drugiego napisu do pierwszego
char* strcpy(char*, const char*);
char* strncpy(char*, const char*, int); //kopiuj tylko n znaków

// dołączenie drugiego napisu do pierwszego
char* strcat(char*, const char*);
char* strncat(char*, const char*, int); //dołącz tylko n znaków
```

- Przypisywanie napisów: tylko za pomocą kopiowania

```
char tekst[100];
strcpy(tekst,p)  // kopiuj z p do tekst, łącznie ze znakiem końca napisu
              // zagrożenie: może spowodować przepełnienie tablicy tekst
strncpy(tekst,p,100);  // kopiuj z p do tekst 100 znaków
              // zagrożenie: pominięcie znaku końca napisu
```

- Inne użyteczne biblioteki:

  `<cstdlib>` (odpowiednik `<stdlib.h>`) - konwersje napisów do typów liczbowych, np. `atoi()`

  `<cctype>` (odpowiednik `<ctype.h>`) - klasyfikacja znaków, np. `isupper()`

Napisy w stylu języka C++ - klasa `string`

- Napis w stylu języka C++ to obiekt typu `string`:

  ```
  string s("witam");
  ```

  Przechowywany w obiekcie napis jest ciągiem znaków o zmiennej długości.
- Korzystanie z typu `string` wymaga dołączenia pliku nagłówkowego:

  `#include <string>` i przestrzeni nazw `std`.

Przykłady korzystania

- Klasa `string` posiada wiele konstruktorów. Pozwalają one inicjować obiekty w różny sposób.

  ```
  string s1; // konstruktor domyślny - tworzy napis pusty
  string s2("witam"); // napis z wartością początkową
  string s3="od tego zaczynamy"; // to samo co wyżej
  string s4(s3); // można inicjować jeden napis drugim
                // ponieważ klasa posiada konstruktor kopiujący -
  string s7(10,'\n'); // napis ma 10 znaków nowego wiersza
  ```

- Obliczenie długości napisu - metoda `length()` lub `size()`:

  ```
  cout << "Napis ma " << s2.length() << " znaków\n";
  cout << "Napis ma " << s2.size()   << " znaków\n";
  ```

- Przypisywanie napisów

  ```
  string s1,s2;
  s1="w klasie string tak można";
  s2=s1;
  ```

- Zachowany jest dostęp do pojedynczych znaków (pozycje napisu są numerowane od 0 do `length()-1`)

  ```
  string s("wsisiz.edu.pl");
  int ile=s.length();
  for (int i=0; i<ile; ++i)
    if (s[i]=='.')
        s[i]='_';
  ```

- Sprawdzenie, czy napis jest pusty

  ```
  if (!s2.size()) ...  // za pomocą sprawdzenia długości napisu
  ```

  lub

  ```
  if (s2.empty()) ...  // empty() zwraca true, jeśli napis nie ma znaków
  ```

- Łączenie napisów (konkatenacja) jest realizowane za pomocą operatora + :

  ```
  string s1("Adam "),s2("Kowalski "),s3;
  s3=s1+s2; // w s3 jest napis "Adam Kowalski "
  ```

- Do napisu można dołączać znak:

  ```
  string s1("Adam"),s2("Kowalski"),s3;
  s3=s1+' '+s2; // w s3 jest napis "Adam Kowalski"
  ```

- Napisy można porównywać z napisami

  ```
  if (s2 == s3) ... // operator == jest przeciążony
  ```

  Można używać operatorów ==, !=, >, <, >= i <=

- Napisy można wyświetlać tak, jak typy podstawowe (przeciążony operator <<):

```
string s("Adam");
cout << s << endl;
```

- Napisy można wczytywać (przeciążony operator >> oraz funkcja `getline()`):

```
string s1, s2;

cin >> s1; // do  znaku spacji

getline(cin,s2); // do znaku nowej linii: jest on wczytywany,
                 // ale nie zachowywany

getline(cin,s2,'\n'); // do znaku nowej linii: jest on wczytywany,
                 // ale nie zachowywany, można zdefiniować inny
                 // ogranicznik tekstu można zmienić na inny
```

- Przykład: Wyświetlanie pliku tekstowego wierszami

```
// Wersja A: nazwa pliku podana w programie
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream we("plik.txt"); // Plik, którego zawartość wyświetlamy
  string wiersz;
  while(getline(we, wiersz,'\n')) {
    cout << wiersz;  // Brak końca wiersza!
    cin.get();
  }
}

// Wersja B: nazwa pliku podawana podczas wykonywania programu
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  string n_pliku;
  cout << "Podaj nazwe pliku: ";
  cin >> n_pliku;
  ifstream we(n_pliku.c_str()); // konwersja stringu na napis w stylu C
  string wiersz;
  while(getline(we, wiersz,'\n')) {
    cout << wiersz;  // Brak końca wiersza!
    cin.get();
  }
}
```

- Przykład: Kopiowanie jednego pliku do drugiego wiersz po wierszu

```
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream we("plikwe.txt");  // Plik, którego zawartość kopiujemy
  ofstream wy("plikwy.txt"); // Plik do którego kopiujemy
  string wiersz;
  while(getline(we, wiersz, '\n')) // Funkcja usuwa znak nowego wiersza
    wy << wiersz << "\n";      //  - musi dodać go z powrotem w pliku wy
}
```

- Przykład: Zliczanie wystąpień określonego słowa w pliku

```
// Wersja A - nazwa pliku podana w programie
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream f("plik.txt");
  int licznik = 0;
  string slowo;
  while (f >> slowo)
    ++licznik;
  cout << "Liczba slow = " << licznik << endl;
}

// Wersja B - nazwa pliku podana w wierszu wywołania programu
#include <iostream>
#include <string>
using namespace std;

int main() {
  int licznik = 0;
  string slowo;
  while (cin >> slowo)
    ++licznik;
  cout << " Liczba slow = " << licznik << endl;
}
```

- Przykład: wyznaczenie liczby wystąpień określonego słowa w pliku. Wywołanie programu:

```
liczslowa slowo plik.txt
```

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {

  if (argc < 3) {
    cerr << "uzycie: LiczSlowa slowo plik\n";
    return -1;
  }
  string wzorzec(argv[1]);
  ifstream plik(argv[2]);

  long licznik = 0;
  string slowo;
  while (plik >> slowo)
    if (slowo == wzorzec )
      ++licznik;
    cout << '"' << wzorzec << "\" wystepuje "
         << licznik << " razy\n";
}
```

- Przykład: Funkcja sprawdza, czy w wierszu znajduje się podany napis. Jeśli napis zostanie znaleziony, zwracana jest jego pozycja w stosunku do początku wiersza, jeśli nie - zwracane jest -1.

```
// W stylu języka C

int wierszZawiera(char napis[], char wiersz[]) {
  if ( napis == NULL || napis[0] == '\0')   // napis pusty
    return -1;
  int j, k;
  for (int i = 0 ; wiersz[i] != '\0' ; i++) {
    if ( wiersz[i] != napis[0] )   // pierwsze znaki są różne
      continue;                    // zacznij od następnego znaku wiersza
     // porównuj pozostałe znaki
     // dopóki nie napotkasz pierwszych różnych lub końca napisu
     for ( j = i + 1, k = 1;
           wiersz[j]==napis[k] && napis[k] != '\0';
           j++, k++ )
     { }
     if ( napis[k] == '\0')       // napotkano koniec napisu
       return i;                  // znaleziono
     else if (wiersz[j] == '\0')  // napotkano koniec wiersza
       return -1;                 // nie znaleziono
    }
    return -1;                    // nie znaleziono
}


// W stylu języka C++ - korzysta z klasy string


int napisZawiera1(string napis, string wiersz) {
  if ( napis.empty()) return -1;   // napis pusty
  int j, k;
  int ndl=napis.size();             // długość napisu
  int wdl=wiersz.size();            // długość wiersza
  for (int i = 0 ; i<wdl ; i++) {
    if ( wiersz[i] != napis[0] )   // pierwsze znaki są różne
      continue;                    // zacznij od następnego znaku wiersza
     // porównuj pozostałe znaki
     // dopóki nie napotkasz pierwszego różnego lub końca napisu
     for ( j = i + 1, k = 1;
           k < ndl && wiersz[j]==napis[k]  ;
           j++, k++ )
     { }
     if ( k == ndl)       // napotkano koniec napisu
       return i;           // znaleziono
     else if (j == wdl)  // napotkano koniec wiersza
       return -1;         // nie znaleziono
    }
    return -1;          // nie znaleziono
}
```

Zaprzyjaźnianie

- Funkcja składowa danej klasy ma dostęp do wszystkich składowych prywatnych dowolnego obiektu tej samej klasy.
- Do składowych prywatnych jakiegoś obiektu nie ma dostępu funkcja innej klasy ani funkcja zewnętrzna (niezależna).
- Funkcja *zaprzyjaźniona* - taka, która mimo, że jest składowa ma dostęp do składowych prywatnych klasy. Może to być:
  - niezależna funkcja zewnętrzna, zaprzyjaźniona z klasą
  - funkcja składowa (metoda) jednej klasy, zaprzyjaźniona z inną klasą,

- **Przykład 1**: Pokazuje sposób definiowania funkcji zaprzyjaźnionej

Trzeba przekazać dwa argumenty, bo funkcja zaprzyjaźniona nie jest funkcją składową i nie otrzymuje wskaźnika `this`.

```
#include <iostream>
using namespace std;

class Punkt
{ int x, y ;
public :
  Punkt (int xx=0, int yy=0)
  { x=xx ; y=yy ; }
  friend bool Pzyjaciel(const Punkt &, const Punkt &) ; // funkcja
                                                         // zaprzyjaźniona
};

// funkcja zaprzyjaźniona ma dostęp do składowych prywatnych
// mimo, że nie jest funkcją składową klasy
// Możemy np. sprawdzić w ten sposób, czy dwa punkty są sobie równe
bool Przyjaciel (const Punkt &p, const Punkt &q)
{ if ((p.x == q.x) && (p.y == q.y)) return true ;
  else return false ;
}

int main()
{
  Punkt a(1,0), b(1), c ;
  if ( (a,b)) cout << "a rowny b \n" ;
  else cout << "a i b sa rozne\n" ;
  if (Przyjaciel(a,c)) cout << "a rowny c \n" ;
  else cout << "a i c sa rozne \n" ;
  return 0;
}
```

- **Przykład 2**: mamy klasę `Wektor` i klasę `Macierz`, dane w tych klasach są prywatne. Chcemy zdefiniować funkcję mnożenia macierzy przez wektor.

- Wersja A: wykorzystywana jest *niezależna* funkcja zaprzyjaźniona

```
#include <iostream>
using namespace std;
class Macierz ;

class Wektor
{
  double v[3] ;
public :
  Wektor (double v1=0, double v2=0, double v3=0)
  { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }
  friend Wektor produkt(Macierz, Wektor);
  void Drukuj ()
  { int i ;
    for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
    cout << "\n" ;
  }
} ;

class Macierz
{
  double m[3][3];
public :
  Macierz (double t[3][3])
  { int i ; int j ;
    for (i=0 ; i<3 ; i++)
      for (j=0 ; j<3 ; j++)
        m[i][j] = t[i][j] ;
    }
  friend Wektor produkt(Macierz, Wektor);
} ;

// niezależna funkcja
// ta funkcja jest zaprzyjaźniona z klasą Macierz i Wektor
Wektor produkt (Macierz a, Wektor x)
{
  int i, j ;
  double suma ;
  Wektor wynik ;
  for (i=0 ; i<3 ; i++)
  {
    for (j=0, suma=0 ; j<3 ; j++)
      suma += a.m[i] [j] * x.v[j] ;
    wynik.v[i] = suma ;
  }
  return wynik ;
}

int main()
{
  Wektor w (1,2,3) ;
  Wektor wynik;
  double tb [3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} } ;
  Macierz a =  tb  ;
  wynik = produkt(a, w) ;
  wynik.Drukuj () ;
}
```

- Wersja B: wykorzystywana jest funkcja zaprzyjaźniona będąca składową innej klasy

```
#include <iostream>
using namespace std;

class Wektor ;

class Macierz
{
  double m[3][3] ;
public :
  Macierz (double t[3][3])
  {
    int i ; int j ;
    for (i=0 ; i<3 ; i++)
      for (j=0 ; j<3 ; j++)
         m[i] [j] = t[i] [j] ;
  }
  Wektor produkt (Wektor);
};

class Wektor
{
  double v[3] ;
public :
  Wektor (double v1=0, double v2=0, double v3=0)
  { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }
  friend Wektor Macierz::produkt (Wektor);
  void Drukuj()
  { int i ;
    for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
    cout << "\n" ;
  }
} ;

// Ta metoda jest zaprzyjaźniona z klasą Wektor
Wektor Macierz::produkt (Wektor x)
{
  int i, j ;
  double suma ;
  Wektor wynik ;
  for (i=0 ; i<3 ; i++)
  {
    for (j=0, suma=0 ; j<3 ; j++)
      suma += m[i] [j] * x.v[j] ;
    wynik.v[i] = suma ;
  }
  return wynik ;
}

main()
{  Wektor w (1,2,3) ;
   Wektor wynik  ;
   double tb [3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} } ;
   Macierz a =  tb  ;
   wynik = a.produkt (w) ;
   wynik.Drukuj() ;
}
```

Operatory klas

- W C++ można definiować *własne wersje operatorów* przeznaczone dla argumentów będących obiektami klas. Czynność ta nazywana jest *przeciążaniem operatora*.
- Postać funkcji operatora:

```
typ nazwa_klasy::operatorop(lista_argumentów)
{
  // definicja operacji względem klasy
}
```

- Funkcja określająca nowe działanie operatora może być zdefiniowana w postaci:
  - funkcji składowej klasy

```
class Punkt
{  int x, y ;
public :
   ...
   Punkt operator+(Punkt&);
} ;
Punkt Punkt::operator+(Punkt& b)
{  Punkt p ;
   p.x = x + b.x ;
   p.y = y + b.y ;
   return p ;
}
```

W `main()`:

```
  Punkt a(1,2), b(2,5), c ;
  c = a+b;    // skrót dla: c = a.operator+(b);
  c = a+b+c ; // skrót dla: c = (a.operator+(b)).operator+(c);
```

  - niezależnej funkcji (najczęściej zaprzyjaźnionej z klasą)

```
class Punkt
{  int x, y ;
public :
   ...
   friend Punkt operator+ (Punkt, Punkt);
} ;
Punkt operator+(Punkt a, Punkt b)
{  Punkt p ;
   p.x = a.x + b.x ;
   p.y = a.y + b.y ;
   return p ;
}
```

W `main()`:

```
  Punkt a(1,2), b(2,5), c ;
  c = a+b;    // skrót dla: c = operator+(a,b)
  c = a+b+c ; // skrót dla: c = operator+(operator+(a,b),c)
```

Przeciążanie operatora +

- Przykład: dodawanie ułamków
  Chcemy uzupełnić klasę `Ulamek` o dodawanie ułamków.

```
1/2 + 1/3 = 5/6
1/6 + 2/6 = 1/2
```

```
class Ulamek
{ int l;                    // licznik
  int m;                    // mianownik
  int nwp(int p, int q);  // największy wspólny podzielnik
public:
  Ulamek(int a=0, int b=1)    // Konstruktor
  { int q=nwp(a,b);
    if(b < 0) q = -q;       // mianownik ma być zawsze dodatni
    l = a/q;
    m = b/q;
  }
  int  ZwrocLicznik()const { return l; }
  int  ZwrocMian()const { return m;}
};
```

- Wersja 1 – „naiwna"

```
void Ulamek::DodajUlamki(Ulamek a, Ulamek b)
{
  l=a.l*b.m+b.l*a.m;
  m=a.m*b.m;
}

int main ()
{
  Ulamek  f1(1,2), f2(1,3), f3, f4(1,6),f5(1,3),f6;
  cout << "Test dodawania ułamków\n";
  cout << f1.ZwrocLicznik() << '/' << f1.ZwrocMian() << '+'
       << f2.ZwrocLicznik() << '/' << f2.ZwrocMian() << '=';
  f3.DodajUlamki(f1,f2);
  cout << f3.ZwrocLicznik() << '/' << f3.ZwrocMian()<< endl;

  cout << f4.ZwrocLicznik() << '/' << f4.ZwrocMian() << '+'
       << f5.ZwrocLicznik() << '/' << f5.ZwrocMian() << '=';
  f6.DodajUlamki(f4,f5);
  cout << f6.ZwrocLicznik() << '/' << f6.ZwrocMian()<< endl;

  return 0;
}
```

Wyniki:

```
1/2+1/3=5/6
1/6+1/3=9/18   // Ułamek nie został skrócony
```

- Wersja 2. Poprawka: uzupełnienie o skracanie ułamka powstałego w wyniku dodawania

```
Ulamek Ulamek::DodajUlamki(Ulamek b)
{
  long rl, rm
  rl=l*b.m+b.l*m;
  rm=m*b.m;
  return Ulamek(rl,rm);  //tutaj tworzony obiekt tymczasowy
}

int main () {
  Ulamek  f1(1,2), f2(1,3), f3, f4(1,6),f5(1,3),f6;
  ...
  f3=f1.DodajUlamki(f2);
  ...
  return 0;
}
```

Wyniki:

```
1/2+1/3=5/6
1/6+1/3=1/2   // Ułamek został skrócony
```

- Wersja 3: Zabezpieczenie i zwiększenie wydajności:

```
Ulamek Ulamek::DodajUlamki(const Ulamek &b)
{
  long rl, rm
  rl=l*b.m+b.l*m;
  rm=m*b.m;
  return Ulamek(rl,rm);
}
```

- Wersja 4. Poprawka: uproszczenie zapisu dodawania, czyli przeciążenie operatora +

```
class Ulamek
{
  ...
public:
  ...
  Ulamek operator+(const Ulamek &f);
};

Ulamek Ulamek::operator+(const Ulamek &b)
{
  long rl, rm;
  rl=l*b.m+b.l*m;
  rm=m*b.m;
  return Ulamek(rl,rm);
}

int main ()
{
  Ulamek  f1(1,2), f2(1,3), f3, f4(1,6),f5(1,3),f6;
  ...
  f3=f1+f2; // skrót dla f3=f1.operator+(f2)
  ...
  return 0;
}
```

Ograniczenia przeciążania operatorów

- Można przeciążać tylko operatory istniejące w C++, z wyjątkiem operatorów
  `.` (wybór składowej)
  `.*` (wybór składowej za pomocą wskaźnika do składowej)
  `::` (rezolucja zasięgu)
  `?:` (wybór)
  `sizeof` (rozmiar reprezentacji zmiennej)
- Nie można tworzyć nowych symboli operatorów z operatorów istniejących.
- Nie można zmieniać definicji operatorów dla typów wbudowanych.
- Nie można zmieniać pierwszeństwa operatorów, liczby argumentów, na których działają i łączności operatora (lewo lub prawostronnej).
- Operator może być zdefiniowany jako metoda (funkcje składowe klasy) lub jako funkcja zewnętrzna. Niektóre operatory można definiować tylko jako metody; należą do nich operatory **`= [] () ->`**
- Funkcja operatora, której pierwszym argumentem jest typ podstawowy lub obiekt innej klasy *nie może być metodą*, musi być zdefiniowana jako funkcja zewnętrzna.

- Przykład: Fakt, że zdefiniowano dla klasy operator + oraz = nie oznacza, że zdefiniowano operator += .

```
Ulamek Ulamek::operator += (const Ulamek &b)
{
// możemy wykorzystać istniejący operator
// a += b  =>  a = a + b  operator + jest już zdefiniowany
  *this = *this + b;
  return *this;
}
```

- Przykład: Fakt, że wymuszamy to, żeby mianownik był zawsze dodatni upraszcza implementację.

```
bool Ulamek::operator<(const Ulamek &b)
{
// wystarczy pomnożyć ponieważ mianownik zawsze jest dodatni
  return l * b.m < b.l * m;
}
```

Metoda czy funkcja zewnętrzna?

Przykład 1 – dodawanie liczby do ułamka

- Czy przy definicji operatora **+** dla klasy `Ulamek` z poprzedniego przykładu możemy wykonać działania?

```
Ulamek a(1,2),b(1,3),c;
c = a + 2;
c = 2 + b;
```

- Dla `c = a + 2` potrzebujemy funkcji:

```
class Ulamek
{ int l;
  int m;
  ...
public:
  ...
  Ulamek operator+(int n);  // funkcja składowa
  ...
};
...
Ulamek Ulamek::operator+(int)
{ return Ulamek(l+m*n, m);}
...
main() {
...
c=a+2; // równoważne c=a.operator+(2)
}
```

Może to być funkcja składowa klasy, ponieważ pierwszym argumentem jest obiekt klasy.

- Dla `c = 2 + a` potrzebujemy funkcji:

```
class Ulamek
{ int l;
  int m;
...
public:
  ...
  friend Ulamek operator+(int, Ulamek );  // funkcja zaprzyjaźniona
  ...
};
...
Ulamek operator+(int n, Ulamek f)
{  return Ulamek(n*f.m+f.l, f.m); }
...
main() {
...
c=2+a; // równoważne c=operator+(2,a)
}
```

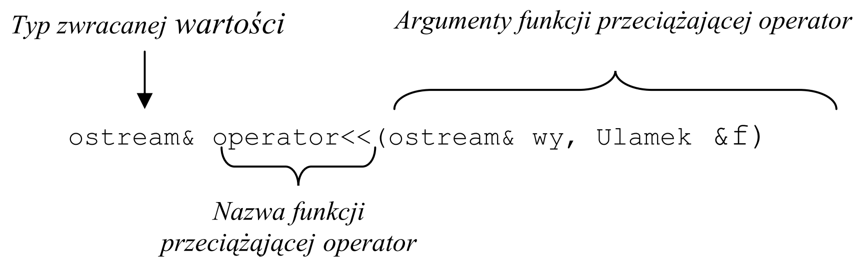
Nie może to być funkcja składowa klasy, ponieważ pierwszym argumentem jest typ wbudowany `int`.

Przykład 2 – drukowanie ułamka

- Co chcemy uzyskać? Uproszczenie drukowania ułamka czyli przeciążenie operatora <<

```
int main ()
{
  Ulamek  f1(1,6), f2(2,6), f3;
  cout << "TEST KLASY Ulamek\n";
  cout << f1 << '+' << f2 << '=' << (f1+f2) << endl;
  return 0;
}
```

- Jak to zrobić? Musi to być funkcja zewnętrzna.



- Rozwiązanie 1: dwie funkcje: metoda zapisująca ułamek do strumienia wyjściowego, funkcja zewnętrzna wykorzystująca to

```
void Ulamek::drukuj(ostream& wy)
{
    wy << l << '/' << m;
}

ostream& operator<<(ostream& wy, Ulamek &f)
{
  f.drukuj(wy);
  return wy;
}
```

- Rozwiązanie 2: zaprzyjaźnienie funkcji wyprowadzającej

```
class Ulamek
{
  ...
public:
  ...
  friend ostream& operator << (ostream &wy, const Ulamek &f);
};
ostream& operator<<(ostream& wy, const Ulamek &f)
{
  wy << f.l << "/" <<f.m;
  return wy;
}
```

Przeciążanie operatora `[]`

- Jest to operator dwuargumentowy, drugim argumentem jest wartość indeksu.
- Funkcja operatora `[]` nie może być funkcją zaprzyjaźnioną.
- Funkcja operatora `[]` musi zwracać referencję jeśli operator ten może występować zarówno po prawej jak i po lewej stronie operatora przypisania (musimy dobrać taki typ, aby wartość elementu po lewej stronie przypisania można było zmieniać).
- Funkcję operatora `[]` można uzupełnić o sprawdzanie przekroczenia dopuszczalnego zakresu indeksów.

- Przykład:

  - Chcemy uzyskać dla klasy `wektor` możliwość odwoływania się do elementów składowych obiektów tej klasy za pomocą notacji: `obiekt[i]`.
  - Musimy przeciążyć operator `[]`. Wtedy zamiast pisać `obiekt.operator[](i)` można będzie pisać `obiekt[i]`.

```
#include <iostream>
using namespace std;
class wektor
{
   int lelem ;
   int * W ;
 public :
   wektor (int n) { W = new int [lelem=n] ; }
   ~wektor () {delete [] W ;}
   int & operator [] (int) ; // przeciążenie operatora []
} ;

int & wektor::operator [] (int i)
{ return W[i] ; }

// przykład użycia
int main()
{  int i ;
   wektor a(3), b(3), c(3) ;
   for (i=0 ; i<3 ; i++) {a[i] = i ; b[i] = 2*i ; }
   for (i=0 ; i<3 ; i++) c[i] = a[i]+b[i] ;
   for (i=0 ; i<3 ; i++) cout << c[i] << " " ;
   return 0'
}
```

Przeciążanie operatora ++

- Jest to operator jednoargumentowy. Może być zapisany w notacji przedrostkowej (`++a`) lub przyrostkowej (`x++`).
- Musimy przeciążyć dwa operatory. Rozróżnienie następuje poprzez deklarację dodatkowego, nieużywanego parametru:

```
class X {
...
public:
typ operator++(); // wersja przedrostkowa
typ operator++(int); // wersja przyrostkowa
};
```

- Przykład: Chcemy uzyskać dla klasy `Ulamek` możliwość zwiększania ułamka o 1.

```
// operator przedrostkowy przeciąża się w zwykły sposób
// działanie operatora: zwiększ i pobierz
Ulamek operator++()
{
  l +=m;   // równoważne this->l += this->m
  return *this;
}

//operator przyrostkowy wymaga wprowadzenia sztucznego argumentu int,
// którego się w funkcji nie używa
// działanie operatora: pobierz i zwiększ
Ulamek operator++(int)
{
  l +=m;                  // nowa wartość
  return Ulamek(l-m,m);   // zwróć starą wartość
}
```

- Chcemy uzyskać dla klasy Punkt możliwość zwiększania współrzędnych punktu o 1.

```
// operator przedrostkowy
Punkt operator++()
{
  x++; y++;        // zwiększ
  return *this;    // pobierz nową wartość
}

//operator przyrostkowy
Punkt operator++(int)
{
  Punkt p = *this;        // stara wartość
  x++; y++;               // nowa wartość
  return p;               // zwróć starą wartość
}
```