

WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

Programowanie zaawansowane FM i NI

Wykład 7-8

Klasy, konstruktory, operatory, header file

Krzysztof Piasecki

Semestr letni roku akad. 2025-26


krzysztof.piasecki@fuw.edu.pl

- **Class (klasa)** – to przepis na dany obiekt
(podobnie jak: typ jest przepisem na zmienną)

  Klasa ma **pola** i **funkcje składowe** („**metody**”).

  Pole: to deklaracja zmiennej,
  którą będzie posiadał każdy obiekt tej klasy.

  Metoda: funkcja, którą można „wywołać na obiekcie”
  tej klasy. Metoda ma dostęp do pól i innych metod.

  Uwaga: ciało klasy kończymy przez `;`

- **Prywatność** pól i metod.

  Słowo **`public:`** pola/metody poniżej - widać poza klasą
  Słowo **`private:`** tych pól/metod nie widać poza klasą

  Domyślnie wszystko jest **`private`**. Gdy podmienimy
  `class` na **`struct`**, domyślnie wszystko będzie **`public`**.

```
25 int main () {
26   Complex a = {3, 4}, b {2, -5};
27   Complex c = add (a,b);
28   cout << b.re << endl;
29   cout << "|a| = " << a.module () << endl;
30   c.print ();
31 }
```

[Link]

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  class Complex {
6   public:
7    double re, im;
8
9    double module () {
10     return sqrt (re*re + im*im) ;
11   }
12
13   void print () {
14     cout << '[' << re << ' ' << im <<"]\n";
15   }
16 };
17
18 Complex add (Complex& a, Complex& b) {
19   Complex sum;
20   sum.re = a.re + b.re;
21   sum.im = a.im + b.im;
22   return sum;
23 }
```

Odtąd możemy używać obiekty tej klasy

- **Gdy pola są prywatne,** zwykle tworzy się metody:

  setter , przez który do pola wpisujemy wartość
  getter , przez który obiekt zwraca wartość pola

  Argument(y) settera nie są polami klasy.
  Te zmienne żyją tylko podczas wywołania settera.

⊙ Deklarując pole, można je zainicjować

  Można od razu zadeklarować obiekt klasy,
  wpisując jego nazwę przed kończącym ;

⊙ **Rozdzielenie prototypu metody i jej implementacji**:

  Prototyp zostawiamy w ciele klasy.
  W implementacji, w nagłówku metody jej nazwę
  poprzedzamy przez `nazwa_klasy::`

[Link]

```
5  class Complex {
6    double re = 0, im = 0;
7  public:
8    void set (double, double);
9    double getRe () { return re; }
10   double getIm () { return im; }
11   double module () ;
12   void print () ;
13 } C ;
14
15 void Complex::set (double R, double I) {
16   re = R; im = I;
17 }
18
19 double Complex::module () {
20   return sqrt (re*re + im*im) ;
21 }
22
23 void Complex::print () {
24   cout << '[' << re << ' ' << im <<"]\n";
25 }
```

```
28 int main () {
29   C.print ();
30   Complex a, b;
31   a.set (3, 4);
32   b.set ( a.getRe() , a.getIm() ) ;
33   cout << "|a| = " << a.module () << endl;
34   b.print ();
35 }
```

- **Konstruktor klasy** – to metoda oferująca użytkownikowi sposób inicjalizacji obiektu podczas jego deklaracji. W jego nagłówku jest nazwa klasy. Konstruktor nic nie zwraca.

⊙ **Konstruktor domyślny (default)** – taki, który wywołuje się bez podania argumentów. Np. wtedy:

```
Complex a1 ;
Complex a2 () ;
```

⊙ **Konstruktory 1- , 2-, N-argumentowe**

⊕ **Konstruktor kopiujący (copy)** – wywoływany, gdy w linii deklaracji jest przypisanie. Np. wtedy:

```
Complex d = c;
Complex e ( b ) ;
```

W nagłówku musi mieć `const...&`

Można stworzyć obiekt tymczasowy (w *rvalue*) , wywołując konstruktor.

⊙ **Destruktor** – określa, co ma się stać przed likwidacją obiektu.

⊙ Kompilator sam utworzy konstr. domyślny, kopiujący i destruktor, jeśli ich nie wpiszemy. Ale gdy wpiszemy dowolny konstruktor, to konstruktory domyślne nie będą już dostawiane.

[Link]

```
5  class Complex {
6    float re, im;
7  public:
8    Complex ()         { re = 0; im = 0; }
9    Complex (float x) { re = x; im = 0; }
10   Complex (float x, float y) ;
11   Complex (const Complex& Z) { re = Z.re ; im = Z.im; }
12   ~Complex ()        { cout << "Destruktor.\n"; }
13
14   Complex add (Complex& ) ;
15 //...
```

```
25 Complex::Complex (float x, float y) {
26   re = x ;  im = y ;
27 }
28
29 Complex Complex::add (Complex& b) {
30   return Complex (re + b.re , im + b.im) ;
31 }
32 //...
```

```
43 int main () {
44   Complex a ;              a.print ();
45   Complex b ( -5 ) ;       b.print ();
46   Complex c (3, 4) ;       c.print ();
47   Complex d = c ;          d.print ();
48   Complex e ( b ) ;        e.print ();
49   Complex f = b.add(c)  ; f.print ();
50   Complex g ( b.add(c) ); g.print ();
51 }
```

- **Lista inicjalizacyjna konstruktora** : specjalna składnia w nagłówku konstruktora, ułatwiająca inicjalizację pól.

```
24 Complex::Complex () : re (0.) , im (0.) {
25   cout << "[0-arg] ";
26 }
27
28 Complex::Complex (double x) : re (x) , im (0.) {
29   cout << "[1-arg] ";
30 }
31
32 Complex::Complex (double x, double y) : re (x) , im (y) {
33   cout << "[2-arg] ";
34 }
35
36 Complex::Complex (const Complex& Z) : re (Z.re) , im (Z.im) {
37   cout << "[copy] ";
38 }
39
40 Complex Complex::add (const Complex& b) {
41   return Complex (re + b.re , im + b.im) ;
42 }
43 //...
```

[Link]

```
int main () {
  Complex a ;              a.print ();
  Complex b ( -5 ) ;       b.print ();
  Complex c (3, 4) ;       c.print ();
  Complex d = c ;          d.print ();
  Complex e ( b ) ;        e.print ();
  Complex f = b.add(c)  ; f.print ();
  Complex g ( b.add(c) ); g.print ();
}
```

```
[0-arg] [0 0]
[1-arg] [-5 0]
[2-arg] [3 4]
[copy] [3 4]
[copy] [-5 0]
[2-arg] [-2 4]
[2-arg] [-2 4]
[x] [x] [x] [x] [x] [x] [x]
```

- Nb. zauważmy, że choć metoda add zwraca kopię obiektu, to podczas wywołania konstruktor kopiujący się nie zgłasza. To nie pomyłka. Jest to tzw. **copy elision** (**ominięcie kopiowania**).
  Copy elision wykonują kompilatory celem optymalizacji szybkości, w sytuacjach „nieszkodliwych”.

  Tu: owa niedoszła kopia stałaby się zmienną tymczasową w *rvalue* . Po przypisaniu do obiektu f , zniknęłaby. Zamiast tego, kompilator od razu do f przypisuje obiekt zwrócony z metody add .

- **Operatory w klasie**. To specjalne metody, definiujące działania na obiektach i pomiędzy obiektami.

Np. dodanie 2 obiektów `Complex` w ten sposób: `c = a.add(b)` jest nieporęczne. Lepiej móc tak: `a + b` .
Trzeba by zakodować `operator+`. Zadziała on na **a** jako **obiekcie rodzimym**, a **b** będzie **argumentem wejścia**.

Dwa podejścia przy kodowaniu, pod potrzebę użycia:

① Przy `c = a + b` zauważmy, że ani obiekt `a`, ani `b` nie mogą zostać zmienione.
Wynik działania jest nowym obiektem i ma być zwrócony (tu: przypisany do `c`).

② Przy `c += 5` zauważmy, że obiekt `c` ma zostać zmieniony (dodane do niego `5`).
Można tu zwrócić obiekt `c`, aby umożliwić ciągi działań, np.: `c += 5 -= 4 ;`

Ad ① . Można tak:

```
Complex Complex::operator+ (Complex& C2) {
  Complex temp ( re + C2.re , im + C2.im) ;
  return temp;
}
```

albo krócej – tak:

```
Complex Complex::operator+ (Complex& C2) {
  return Complex ( re + C2.re , im + C2.im) ;
}
```

Ad ② . Można np. tak:

```
Complex& Complex::operator+= (double k) {
  re += k ; im += k ;
  return *this ;
}
```

- Słowo kluczowe **this** zwraca adres obiektu, na którym wykonuje się metoda. ***this** oznacza „ten obiekt”.

- **Operatory w klasie** – cd.

Odwrócenie liczby to przykład operatora bezargumentowego.
Np. dla: `b = -a ;` możemy zakodować:

```
Complex Complex::operator- ()  {
  return Complex (-re, -im) ;  }
```

⊙ **Operator przypisania =** działa na obiekcie w *lvalue* i przypisuje mu obiekt z *rvalue* .
Uwaga: nie mylmy go z konstruktorem kopiującym, który się wykonuje przy deklaracji nowego obiektu.

Np.:

```
Complex& Complex::operator= (const Complex& z2) {
  re = z2.re;
  im = z2.im;
  return *this;                                  }
```

⊙ Operator **preinkrementacji**: (By móc napisać `++a`) . Inkrementujemy wartość i zwracamy nasz obiekt:

```
Complex& Complex::operator++ () {
  re += 1. ;   im += 1. ;
  return *this ;                }
```

⊙ **Postinkrementacja**: problem, bo `operator++` już jest użyty. Aby go obejść, twórcy C++ narzucają nagłówek:

```
Complex Complex::operator++ (int) {
  Complex myCopy ( *this );
  ++( *this ) ;
  return myCopy;                  }
```

Uwaga: słowo `int` jest tylko wybiegiem. Operatora używamy „jak zwykle”, czyli np.: `b = a++ ;`

- **Operatory w klasie** – cd.

Chcąc utworzyć operator umożliwiający `k * Z` (k = `double`, Z = `Complex`), napotykamy na problem: argumentem po lewej stronie nie jest obiekt klasy `Complex`, a liczba.
Trzeba wtedy zakodować operator poza klasą, jako funkcję dwóch argumentów. Np. tak:

```
Complex operator* (double k, Complex& Z)    {
  return Complex ( k * Z.re , k * Z.im );   }
```

Kłopot w tym, że funkcje poza klasą `Complex` nie mają dostępu do jej pól prywatnych (`re` i `im`).

⊙ **Kwalifikator dostępu `friend`**

Jest to klucz dostępu, który klasa daje konkretnej funkcji zewnętrznej (może też dawać innej klasie).
Podajemy go w środku naszej klasy i wygląda on tak:

```
friend Complex operator* (double k, Complex& Z);
```

czyli po `friend` piszemy prototyp funkcji, której klasa ma udostępnić wszystkie pola.

⊙ Kod demonstracyjny ze wszystkimi powyższymi operatorami jest tu.

Wiele operatorów w C++ użytkownik może przeciążyć i napisać własne (np. pełna lista tu, omówienie też tu).
Niektórych operatorów nie wolno przeciążać (np. lista tu).

- **operator< , komparatory , sort**

Można klasie K zakodować **operatory relacyjne** (porównania). Powinny zwracać `true/false`. W klasie są one 1-argumentowe, a poza nią 2-argumentowe.

Jeżeli obiekty klasy K zestawiamy w tablicy, to możemy chcieć tablicę posortować.

Musi być jednak reguła sortowania, ujęta w funkcję zwaną **komparatorem**. Działa ona na 2 obiektach klasy K i zwraca `bool`. **`operator<`** jest takim komparatorem.

Szablon **sort** z **<algorithm>** posortuje tablicę elementów klasy K. Domyślnie wymaga, aby w klasie K był `operator<`.

Alternatywnie, możemy napisać **własny komparator** i podać go w trzecim argumencie wejścia `sort`.

[Link]

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

struct Person {
  string Name;
  bool operator< (Person& P2) { return (Name < P2.Name); }
} P1, P2, P3;

bool ReverseComp (Person& P1, Person& P2) {
  return (P2.Name < P1.Name) ;
}

int main () {
  P1.Name = "Ola"; P2.Name = "Jan"; P3.Name = "Ed";
  Person MyTable[] = { P1, P2, P3 };
  for (auto& p : MyTable) cout << p.Name <<'\t';
  cout << endl;

  sort (MyTable, MyTable + 3);
  for (auto& p : MyTable) cout << p.Name <<'\t';
  cout << endl;

  sort (MyTable, MyTable + 3, ReverseComp );
  for (auto& p : MyTable) cout << p.Name <<'\t';
  cout << endl;
}
```

- **Podział kodu na pliki nagłówka (.h) i implementacyjne (.cpp)** :

  Często wyjmuje się z kodu fragment poświęcony klasie. Następnie dzieli się go na 2 :
  część z definicjami i **część z implementacją**, a każdą część wstawia do odrębnego pliku.

  Dzięki temu, kod opisujący klasę można załączać przez `#include` w wielu innych kodach (plikach).

  Podział na pliki poświęcone klasom może też ułatwiać debugowanie złożonych kodów.

  ① **Plik nagłówka (header file)** jest rodzajem spisu treści. Ma rozszerzenie `.h` lub `.hpp` .
  Wstawiamy tu deklarację klasy z jej ciałem.
  W ciele wstawiamy pola i prototypy metod (też konstruktorów i operatorów).
  Również metody jednolinijkowe, definicje stałych itd. Słowem – skrótowce.

  ② W **pliku implementacyjnym** klasy (rozszerzenia `.C`, `.cpp`, `.cxx`, `.cc`)
  wstawiamy pełne implementacje metod (w tym konstruktorów i operatorów).

  ③ Kod z funkcją `main` osiada w osobnym **pliku klienta** (też `.C`, `.cpp`, `.cxx`, `.cc`) .
  Tu przez `#include` załączamy plik nagłówka klasy, którą chcemy użyć.

- **Następna strona:**

  przykład podziału jednolitego kodu z klasą `Complex` na 3 powyższe pliki.

- Całość w jednym pliku

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

class Complex {
  float re, im;
public:
  Complex (float, float);
  float getRe () { return re; }
  float getIm () { return im; }
  Complex operator+ (Complex& );
  void print ();
};

Complex::Complex (float X, float Y)
       : re(X), im(Y) {}

Complex Complex::operator+ (Complex& Z)   {
  return Complex (re + Z.re, im + Z.im) ; }

void Complex::print ()                    {
  cout << '[' << re << ' ' << im << "]\n"; }

int main () {
  Complex Z1 (3., 4.) , Z2 (-1. , -2.);
  Complex Z3 = Z1 + Z2;
  Z3.print ();
}
```

[Link]

Plik nagłówkowy (**complex.h**)

```
class Complex {
  float re, im;
public:
  Complex (float, float);
  float getRe () { return re; }
  float getIm () { return im; }
  Complex operator+ (Complex& );
  void print ();
};
```

Wersja uproszczona (bez *include guard*)

Plik implementacji (**complex.C**)

```
#include "complex.h"
#include <iostream>
using std::cout;

Complex::Complex (float X, float Y)
       : re(X), im(Y) {}

Complex Complex::operator+ (Complex& Z)   {
  return Complex (re + Z.re, im + Z.im) ; }

void Complex::print ()                    {
  cout << '[' << re << ' ' << im << "]\n"; }
```

```
#include "complex.h"
#include <iostream>
using namespace std;

int main () {
  Complex Z1 (3., 4.) , Z2 (-1. , -2.);
  Complex Z3 = Z1 + Z2;
  Z3.print ();
}
```

Plik klienta (**main.C**)

Pliki nagłówka i implementacyjne: include guard

⊙ Preprocesor, analizując nieco bardziej złożony kod, może kilkukrotnie napotkać na `#include`, przez co załączy ten sam plik nagłówka klasy. Ale w C++ nie można czegoś zdefiniować 2×, Więc skutkiem będzie błąd kompilacji.

Aby to uniemożliwić, stosuje się "**include guard**". W pliku nagłówka obejmujemy kod dyrektywami:

```
#ifndef __MojaKlasa__
#define __MojaKlasa__
...
#endif
```

Dzięki temu preprocesor wklei plik do kodu tylko 1×.

```
1  #ifndef __Complex__
2  #define __Complex__
3
4  class Complex {
5    float re, im;
6  public:
7    Complex (float, float);
8    float getRe () { return re; }
9    float getIm () { return im; }
10   Complex operator+ (Complex& );
11   void print ();
12 };
13
14 #endif
```

⊙ W pliku implementacyjnym klasy załączamy jej plik nagłówka:

```
#include "TwojaKlasa.h"
```

Nie stosujemy tu `<...>`, a `"..."`.
Konwencja: Nawiasy `<...>` służą załączaniu bibliotek fabrycznych.

Pliki nagłówka i implementacyjne c.d.

- W każdym z plików **załącz** przez `#include` te biblioteki, których polecenia są jawnie użyte w pliku.

  Jeśli plik nagłówka Twojej klasy A używa innej klasy B, to trzeba w nim załączyć plik nagłówka klasy B.

  Gdy w Twoim pliku A załączasz plik nagłówka klasy B, a w nim (w B) są załączone nagłówki C, to w Twoim pliku A nie musisz ponownie załączać nagłówków C.

- **Przestrzeń nazw** w plikach Twojej klasy

  Możesz załączyć całą przestrzeń (choć warto tego unikać) :

```
using namespace std;
```

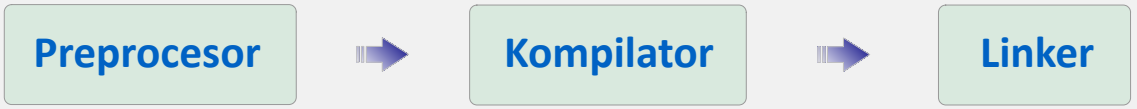
  Bezpiecznej jest załączyć tylko elementy użyte w pliku, np.:

```
using std::cout, std::endl, std::string ;
```

- Kod klienta z założenia jest „końcówką” łańcucha, więc w nim możesz załączyć całą przestrzeń nazw.

Pliki nagłówka i implementacyjne – kompilacja

Przypomnijmy, że **budowanie** ("**building**") pliku wykonywalnego ma 3 etapy:



⊙ Mamy projekt z klasami A, B, … (pliki `A.h`, `A.C`, `B.h`, `B.C`, …) i z plikiem `klient.C` zaw. funkcję `main`.

▻ Jeśli chcemy **jedynie wytworzyć plik wykonywalny (aplikację)**, to piszemy:

```
g++ {opcje kompilacji}  A.C  B.C  ...C  klient.C  -o {nazwa_aplikacji}
```

Kompilacja tych plików będzie zachodzić **osobno** (to tzw. odrębne **jednostki translacyjne**).

Nb. Jeżeli nasz plik nagłówka jest w innej ścieżce, np.: `/mypath/myheader.h`, to musimy ją wskazać:

```
g++ -I/mypath {opcje}  A.C B.C ...C klient.C  -o {nazwa_aplikacji}
```

{Linux/MacOS} Ścieżkę z plikami .h można też podać, dodając ją do zmiennej środowiskowej CPATH.

- **Pliki nagłówka i implementacyjne – kompilacja**

▷ Alternatywnie, **kody klas** można wpierw **skompilować** (do **plików obiektowych,** rozszerzenie **.o**) …

```
g++ -c  {opcje kompilacji}  A.C B.C
```

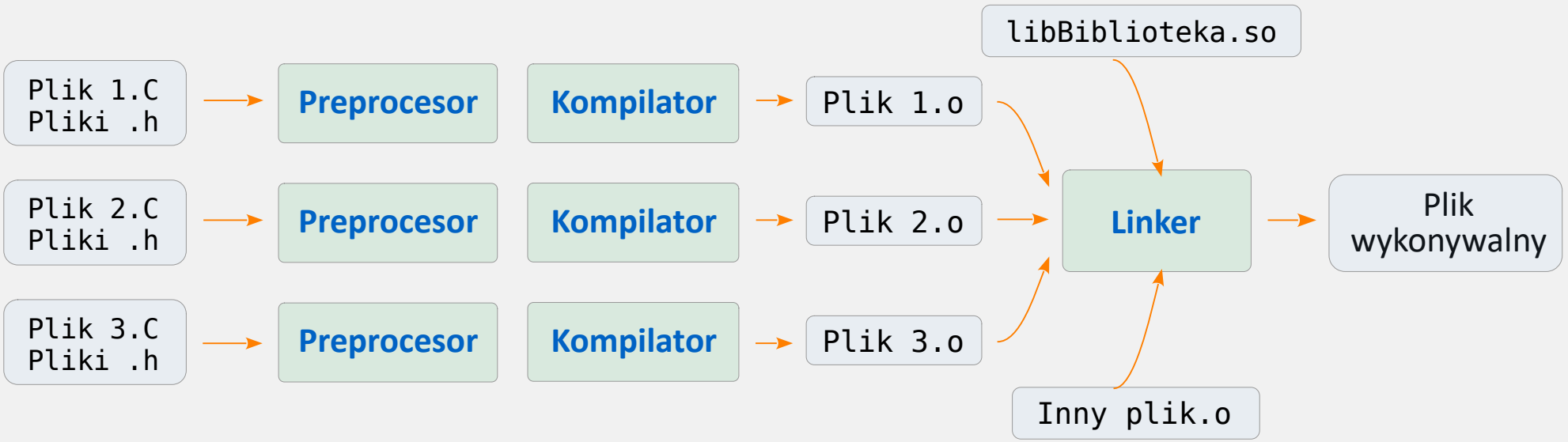
… a potem **skompilować kod klienta** i **zlinkować** wszystko do **aplikacji**:

```
g++  {opcje linkowania}  klient.C  A.o B.o ...  -o {nazwa_aplikacji}
```

Przy czym, jeśli korzystamy z zewnętrznej biblioteki {Linux: *libBiblioteka.so* } , dodajemy do linkowania też, np.:

```
g++  {opcje linkowania}  klient.C  A.o B.o ...  -lBiblioteka  -o {nazwa_aplikacji}
```

⊙ To oznacza, że w zasadzie schemat budowania wygląda tak:



[Link]

- **Klasa a `const`.** Omówimy tu trzy aspekty.

① jeżeli pole ma nie podlegać zmianom,
to jego deklarację poprzedzamy przez `const`.

Możemy zainicjować to pole tylko:
w deklaracji lub w liście inicjalizacyjnej konstruktora.

② jeżeli metoda ma nie zmieniać pól (**metoda stała**),
to na końcu nagłówka dodajemy `const`.

③ gdy zadeklarujemy obiekt klasy z modyfikatorem `const`,
to wolno na nim wykonywać **wyłącznie**
metody stałe (z `const` w nagłówku, jak w punkcie ②).

```
1  #include <iostream>
2  using namespace std;
3
4  struct MyType {
5    const int x = 5;
6    MyType (int temp) : x (temp) { }
7    void print () const {
8      cout << x << endl;
9    }
10 };
11
12 int main () {
13   const MyType A (-5);
14   MyType B (123);
15   A.print ();
16   B.print ();
17
18   MyType* Bptr = &B ;
19   Bptr->print ();
20   MyType* Cptr = new MyType (-123);
21   Cptr->print ();
22 }
```

- **Wskaźnik na obiekt klasy**

Tworzymy go wprost: `MyType* Bptr ;`
Można mu przypisać adres istniejącego obiektu.
Albo rezultat alokacji dynamicznej.

Jest specjalny operator do działań na polach i metodach obiektu, którego adres trzyma wskaźnik: `->`

```
Bptr -> print()
```

działa jak: `(*Bptr).print()`

- **Klasa z tablicą alokowaną dynamicznie**

Najważniejsze:

musimy napisać **destruktor** ,
kasujący wpis w tablicy alokacji.
Bez tego będą wycieki pamięci.

⊙ Do inicjowania przydatny jest
`initializer_list<typ>` .
Jest to pojemnik na zbiór elementów,
czyli na { . . . } .
Wymaga załączenia swojej biblioteki.

Można po nim iterować w pętli zakresowej.
Niestety, nie ma operatora [] .
Ale zna swój rozmiar: metoda `size()` .

Uwaga: elementy `initializer_list`
są stałe (`const`) .
Dlatego tu musi być `const`.

Uwaga: ten kod nie ma zabezpieczenia
na wypadek pustego zbioru.

[Link]

```
1  #include <iostream>
2  #include <initializer_list>
3  using namespace std;
4
5  struct Data {
6    int* Tab ;
7    int size;
8    Data ( initializer_list<int> IL ) ;
9    ~Data () { delete[] Tab; }
10   void print ();
11 };
12
13 Data::Data (initializer_list<int> IL) : size (IL.size()) {
14   Tab = new int [ size ] ;
15   int index = 0;
16   for (const int& elem : IL ) Tab[index++ ] = elem;
17 }
18
19 void Data::print () {
20   for (int i = 0; i < size; i++) cout << Tab[i] << ' ';
21   cout << endl;
22 }
23
24 int main () {
25   Data X = {6, 5, 4}  ;  X.print() ;
26   Data Y ( {3, 2, 1} );  Y.print() ;
27 }
```

Klasa z referencją w polu

[Link]

Najważniejsze:

Na etapie inicjalizacji obiektu, referencja musi zostać spięta z daną.

Należy napisać konstruktor, który wystawiając argument (najlepiej – też referencję), zepnie go z naszą referencją.

Konstruktor kopiujący też musi spinać obiekt z naszą referencją.

Nb. może być sytuacja, w której klasa ma inne pola (nie-referencje), a potrzeba konstruktora, który inicjuje tylko te „inne" pola. Propozycja rozwiązania: przez alokację dynamiczną.

```
1  #include <iostream>
2  using namespace std;
3
4  struct DataWrapper {
5    double& Xref;
6
7    DataWrapper (double& x) : Xref (x) { }
8
9    DataWrapper (const DataWrapper& wzorzec)
10         : Xref (wzorzec.Xref) { }
11 };
12
13 int main () {
14   double x = 12.345;
15   cout << x << '\t' << &x << endl;
16
17   DataWrapper DW ( x );
18   cout << DW.Xref << '\t' << &DW.Xref << endl;
19
20   DataWrapper DWcopy = DW;
21   cout << DWcopy.Xref << '\t' << &DWcopy.Xref << endl;
22 }
```

Konwersja z klasy i na klasę

- Rzutowanie **z obiektu o typie T** na naszą klasę wykona dla nas konstruktor przyjmujący 1 argument, o typie T.

- Rzutowanie z obiektu naszej klasy **na typ `Ytyp`** : piszemy taki operator

```
operator Ytyp () { return ...; }
```

Uwaga:

przed przypisaniem do F, kompilator wykona konwersją niejawną (implicit) konstruktorem `Complex (double)`

[Link]

```
5  struct Complex {
6    double re, im;
7    Complex (double X)           : re (X)        { }
8    Complex (double X, double Y) : re (X), im(Y) { }
9    double module ()  { return sqrt (re*re + im*im) ; }
10   operator double() { return module () ; }
11   void print () {
12     cout << '[' << re << ' ' << im << "]\t";
13   }
14 } ;
15
16 int main () {
17   Complex C (3, 4);
18   double mod ( C );
19   cout << mod << '\t' << (double) C << '\t'
20        << double (C) << '\t'
21        << static_cast<double> (C) << endl;
22
23   Complex D ( 3. )          ;  D.print () ;
24   Complex E = Complex( 3. );  E.print () ;
25   Complex F = 3.            ;  F.print () ;
26   Complex G = (Complex) 3. ;  G.print () ;
27   Complex H = static_cast<Complex> ( 3. );
28   H.print () ;
29 }
```

Znaczenie `static` w klasie

Pola i metody statyczne „żyją” nawet wtedy, gdy nie utworzyliśmy obiektu.

Poza klasą `S` odwołujemy się do pola statycznego `x` przez `S::x`, a metodę statyczną `M` wołamy przez `S::M`.

Pole statyczne pamięta swoją wartość. O ile nie jest `const`, to można ją zmienić w dowolnej chwili.

⊙ Wszystkie **pola statyczne trzeba zainicjować**. Pytanie, w którym miejscu kodu?

- w ciele klasy wolno tylko dla takich liczb:

```
static const int  X ;
static constexpr typ_liczbowy  Y;
```

(`constexpr` to rozszerzenie `const`. C nie pozwala na `static const` inne, niż `int`)

- resztę inicjujemy poza klasą, pamiętając o przedrostku `nazwa_klasy::`

[Link]

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

struct S {
  static double count ;
  S ()   { count++; }
  static double getCount () { return count; }

  static const     int    MyInt    = 123;
  static constexpr double MyDouble= 123.456;
  static constexpr double Max      = DBL_MAX ;
  static const     string Version ;
};

double       S::count   = 0. ;
const string S::Version = "Wersja Klasy = 1";

int main () {
  cout << S::MyInt << '\t' << S::MyDouble << '\t'
       << S::Max   << '\t' << S::Version  << endl;
  for (int i = 0; i < 3; i++) {
    S a;
    cout << S::count <<'\t'<< S::getCount () <<endl;
  }
}
```

Słowa `explicit`, `default` i `delete` w klasie

[Link]

Gdy komputer konstruuje obiekt,
wolno mu 1× niejawnie przekonwertować
typ argumentu. Ale możemy tego nie chcieć.

- Można zablokować możliwość podania przez użytkownika argumentu danego typu:

```
Klasa ( typ_blokowany ) = delete;
```

- Można zakazać konstruktorowi, aby był wołany niejawnie:

```
explicit NazwaKlasy ( typ ) ...
```

choć nie zablokuje to tej drogi:

```
Typ obiekt = Typ ( … );
```

Wtedy też kopiowanie wykona się tylko tak:

```
Typ obiekt ( wzorzec );
```

a tak nie: `Typ obiekt = wzorzec ;`

- Możemy nakazać kompilatorowi utworzenie „fabrycznych” konstruktorów domyślnych, np.:

```
NazwaKlasy ( ) = default;
```

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

struct S {
  int idata = 123;

  S ()           = default ;
  explicit S (const S& S0) : idata (S0.idata) {}
  explicit S ( int itmp )  : idata (itmp) {
                             cout << "[ctor-int] "; }
  S ( double )   = delete  ;
};

int main () {
  S a          ;  cout << a.idata << endl;
  S b ( -4 )   ;  cout << b.idata << endl;
//S c ( 1. )   ; // error (deleted constructor)
//S d = 55     ; // error (explicit S (int)   )
  S e = S(55) ;  cout << e.idata << endl;
//S f = e      ; // error (explicit copy ctor )
  S g (e)      ;  cout << g.idata << endl;
}
```