Literatura:

(1) B. Stroustrup, "Język C++"
(2) H. Schildt, "Programowanie C++"
(3) J. Gręboszz, Symfonia C++, t. II, I

C++ rozszerza możliwości języka C pod kątem:

1) bardziej rygorystycznej kontroli typów (nie wszystko co jest poprawne w C jest poprawne w C++);

2) uzupełnień (nieobiektowych):

   – mniej restrykcyjny porządek definicji i instrukcji (np. deklaracja zmiennych nie musi się znajdować na początku programu)

   – wprowadzenie operatora zakresu,

   – nowy typ //referencja do//;

   – możliwość przeciążania identyfikatorów,

   – funkcje ze zmienną liczbą parametrów;

3) programowania obiektowego (możliwość tworzenia klas obiektów, które określają zarówno zestaw danych, jak i operacje wykonywane na tych danych)

Przykład.

```
#include <iostream>
using namespace std;// przestrzeń nazw
main()
{int n;
 float x;
 cout<<"Podaj liczbę naturalną i liczbę rzeczywistą"<<endl;
       //albo << "\n"; przejście do nowej linii
 cin>>n>>x;
 cout<<n<<"+"<<x<<"="<<n+x;
 system("pause"); //w DevC++
}
```

cin - standardowy strumień wejściowy // ⇔ scanf()
cout - standardowy strumień wyjściowy //⇔ printf()

Przykład. Klasa dla liczby zespolonej.

```
class zespolona
{  private:
   float re, im;  //pola składowe

   public:
   zespolona ();        //konstruktor bezparametrowy
   zespolona (float x, float y); // konstruktor
                                 // z dwoma parametrami

   float modul();
   float argument();
   zespolona dodaj (zespolona);
   void wczytaj();
   void wypisz();
};

zespolona::zespolona() // konstruktor
{re = 0;               // bezparametrowy
 im = 0;
}
```

```
zespolona::zespolona(float x, float y)
{re = x;          //konstruktor z dwoma
 im = y;          //parametrami
}
float zespolona::modul()
{return sqrt(re*re+im*im);   //trzeba dołączyć bibliotekę
                             //matematyczną
}
void zespolona::wczytaj()
{cout<<"Podaj część rzeczywistą i urojoną"<<endl;
 cin>>re>>im;
}
void zespolona::wypisz()
{cout<<re<<"+i"<<im;   //2+i3, 2+i-3 :(
}



main ()
{ zespolona z1, z2(1,2), z3;
  z1.wczytaj();
  cout<<"Moduł z1 wynosi "<<z1.modul();
  cout<<"Moduł z2 wynosi "<<z2.modul();// 2.23...

  cout<<z1;      //źle, bo cout nie działa dla typu
                 //zespolona ale działa tylko dla typów
                 //wbudowanych (int, float, char, ...)

  z3=z1+z2; //źle, bo operator + działa na typach int,
              // float, ...

  z3=z1.dodaj(z2);
  z3.wypisz();       //ale można też prościej
                     // (z1.dodaj(z2)).wypisz();

 }
```

Definiowanie klas

```
class nazwa_klasy
{[modyfikator_praw_dostępu:]

 deklaracje_komponentów;
 ⋮
 [modyfikator_praw_dostępu:]

 deklaracje_komponentów;
};
```

Rodzaje komponentów (składowych):

- pola danych (pola składowe) //np. re, im w klasie zespolona
- funkcje składowe (funkcje i operacje na danych składowych)

Pod względem praw dostępu komponenty dzielimy na:

1) prywatne (modyfikator **private**) - dostęp jest możliwy tylko dla innych komponentów klasy oraz tzw. funkcji zaprzyjaźnionych;

```
Przykład. main ()
   { ...
     cout<<z1.re; //nie zadziała bo funkcja main()
                  //nie ma dostępu do prywatnych pól
                  //składowych klasy zespolona
   }
```

2) publiczne (modyfikator **public**) - ogólnie dostępne (nawet przez inne funkcje niezwiązane z tą klasą);

3) chronione/zabezpieczone (modyfikator **protected**) - dostępne dla funkcji składowych oraz dla klas wywodzących się od tej klasy (pochodnych tej klasy, związane z dziedziczeniem).

Uwaga

Zakres widzialności komponentów obejmuje cały blok klasy (bez względu na miejsce deklaracji w klasie).

Dopóki w definicji klasy nie pojawi się żadna z etykiet (public, protected) to przez domniemanie wszystkie komponenty mają dostęp private.

Klasa może być zadeklarowana:

- na zewnątrz wszystkich funkcji programu (tzw. klasa zewnętrzna) - jest ona widoczna we wszystkich plikach programu;
- wewnątrz jakiejś funkcji (tzw. klasa lokalna) - jej zakres widoczności nie wykracza poza zasięg tej funkcji;
- wewnątrz innej klasy (tzw. klasa zagnieżdżona) - jej zakres widoczności nie wykracza poza zakres klasy zewnętrznej.

Definicja funkcji składowej występująca poza ciałem klasy:

```
typ_zwracanej_wartości  nazwa_klasy::nazwa_funkcji(lista_arg)
```

↑
operator zakresu

Dwuargumentowy operator zakresu :: (podwójny dwukropek) poprzedzony nazwą klasy określa, że funkcja występująca po :: jest składową tejże klasy.

```
np. float zespolona::modul()
   {return sqrt(re*re+im*im);}
```

Definiowanie (deklarowanie) obiektów:

```
nazwa_klasy lista_obiektów;

np.  zespolona z;
     zespolona z1, z2(1,2);
     zespolona tab[10];
     zespolona *wsk;
```

Użycie w programie funkcji składowych dla obiektów:

```
zm_obiekt.nazwa_funkcji_składowej(lista_parametrów)
```

Przykłady:

```
zespolona z, z1(1,2), z3;
z.wczytaj();
z.wypisz();
(z.dodaj(z1)).wypisz();
z3 = z.dodaj(z1);
```

Przeładowanie (przeciążanie) nazw funkcji

Można w programie umieścić więcej niż jedną funkcję o tej samej nazwie, pod warunkiem, że te funkcje będą różniły się argumentami (ich liczbą lub typem argumentów).

Przykład.

```
zespolona pomnoz(zespolona z)
      // mnożenie dwóch liczb zespolonych

zespolona pomnoz(float x)
      // mnożenie liczby zespolonej
      // przez liczbę rzeczywistą
```

Uwaga. Typ zwracany przez funkcję **NIE** jest brany pod uwagę (przy sprawdzaniu czy te funkcje się różnią).

(Np. mając już funkcję **zespolona** pomnoz(float) nie można dodać funkcji
float pomnoz(float)).

Przeładowanie nazw funkcji dotyczy nie tylko funkcji składowych, ale i pozostałych funkcji.

Przykład.

```
void sortowanie(int n, int A[100]);
void sortowanie(int A, int A[100], int l, int p);
```

Obie funkcje mogą wystąpić w jednym programie.

Funkcje z parametrami domniemanymi (funkcje z różną liczbą argumentów)

Przykład.

```
zespolona::zespolona()
{re = 0; im = 0}

zespolona::zespolona(float x, float y)
{re = x; im = y}
```

Chcielibyśmy, aby zespolona z(5) spowodowała, że z = 5 + i0.

Powyższe dwa konstruktory można zastąpić jednym konstruktorem:

```
zespolona::zespolona(float x=0, float y=0)
{re = x; im = y}

Wtedy  zespolona z;       //z = 0 + i0;
       zespolona z(1,2);  //z = 1 + 2i;
       zespolona z(5);    //z = 5 + i0;
```

Funkcje z argumentami domniemanymi można wywołać z mniejszą liczbą parametrów niż wynikałoby to z deklaracji funkcji.

Brakujące w wywołaniu argumenty otrzymują wartości domniemane.

Jeżeli wartość argumentu zostanie podana, to przesłania ona wartość domniemaną.

Parametry domniemane muszą stać na końcu listy parametrów.

```
np. zespolona::zespolona (float x=0, float y);   //ŹLE
    zespolona::zespolona (float x, float y=0);
//dobrze ale gorzej niż
    zespolona::zespolona (float x=0, float y=0);
```

Zdefiniowanie obiektu jest w rzeczywistości wywołaniem specjalnej funkcji - **konstruktora**.
Konstruktory możemy definiować sami, a jeśli tego nie uczynimy, to kompilator doda tzw. **konstruktor domyślny**:

```
nazwa_klasy() {};
```

Jeśli nawet nie zdefiniowalibyśmy żadnego konstruktora dla klasy zespolona, to i tak ta klasa będzie działać. Po zadeklarowaniu obiektu:

```
zespolona z;
```

obiekt z istnieje (czyli ma przydzieloną pamięć na re i im ale te wartości są nieokreślone (losowe)).

Ale gdyby w klasie został zadeklarowany tylko konstruktor

```
zespolona(float x, float y);
```

to po zadeklarowaniu obiektu:

```
zespolona z;
```

kompilator zgłosi błąd. bo nie ma konstruktora bezparametrowego, a domyślny (bezparametrowy) nie został utworzony (bo jest inny konstruktor z dwoma parametrami).

Funkcje operatorowe (przeciążanie operatorów)

Jest to szczególny przypadek przeładowywania nazw funkcji, w którym dodajemy nowe znaczenia operatorom już występującym w języku C++.
Chodzi o to, aby można było np. napisać z1+z2 zamiast z1.dodaj(z2). W tym wypadku znany operator dodawania + (wykorzystywany dla typów int, float, ...) zyskuje dodatkowe nowe znaczenie, a mianowicie dodawania liczb zespolonych).

Funkcje operatorowe można tworzyć dla większości operatorów języka C++ pod warunkiem, że choć jeden z argumentów jest typu obiektowego lub funkcja operatorowa jest częścią klasy.

Prototyp funkcji operatorowej będącej składową klasy:

```
typ_wyniku operator @ (void) ← funkcja jednoargumentowa
                   ↑
             symbol operatora


typ_wyniku operator @ (typ_argu arg1) ← funkcja
                   ↑                    dwuargumentowa
             symbol operatora
```

Przykład.

```
class zespolona
{
 ⋮
zespolona operator*(zespolona);
zespolona operator*(float)   //z1*5 ⇔ z1.operator*(5)
};                           //ale nie 5*z1!



zespolona zespolona::operator*(zespolona z2)
{zespolona z1; //zmienna pomocnicza
 z1.re = re*z2.re-im*z2.im;
 z1.im = re*z2.im+im*z2.re;
 return z1;
}

zespolona zespolona::operator*(float x)
 {zespolona z1;
  z1.re = re*x;
  z1.im = im*x;
  return z1;
 }
```

Jeśli funkcja operatorowa nie jest funkcją składową klasy, wtedy jej deklaracja jest następująca:

```
typ_wyniku operator@(typ_argumentu_1) ← funkcja
                                          jednoargumentowa

typ_wyniku operator@(typ_arg1 arg1, typ_arg2 arg2) ←
                                   funkcja dwuargumentowa



class wektor3
{float x,y,z;

 public:
 wektor3();
 wektor3(float, float, float);
 wektor3 operator*(wektor3);   //iloczyn wektorowy
 wektor3 operator*(float);     //mnożenie przez skalar
 float operator*(wektor3);     //iloczyn skalarny ŹLE
 float operator^(wektor3);     //iloczyn skalarny
};
```

Uwagi o przeładowywaniu operatorów

(1) Przeładować można prawie każdy operator, tzn.: `+ , - , * , / , % , ^ , & , ~ , ! , = , < , > ,...,` new, delete, (),[], ale nie można `.,.*,::,?:`

(2) Nie można wymyślać swoich operatorów i ich przeciążać.

(3) Operator jest przeciążony względem klasy, w której został zadeklarowany, ale nie traci żadnego ze swoich dotychczasowych znaczeń.

(4) Nie można zmienić składni wyrażenia dla danego operatora, priorytetu operatora i reguł łączności.

(5) Przeciążone operatory nie mogą mieć argumentów domniemanych.

```
np.
zespolona operator^(int n=2) //z1=z^; - źle
zespolona potega(int n=2) // dobrze
```

(6) Funkcje operatorowe nie są dziedziczone (z wyjątkiem operatora =).

(7) Funkcja operatorowa, która jest funkcją składową klasy wymaga, aby obiekt stojący po lewej stronie operatora @ był obiektem jej klasy.

(8) Jest kilka operatorów, które automatycznie są generowane dla każdej klasy: = , `& , , , new , delete.`

(9) Aktywowanie funkcji operatorowej na rzecz danego obiektu odbywa się zgodnie ze schematem składniowym dla danego operatora w C++.

W przypadku operatorów dwuargumentowych wyrażenie pojawiające się po lewej stronie operatora jest obiektem, na rzecz którego funkcja operatorowa jest aktywowana.

Funkcje zaprzyjaźnione

Funkcje zaprzyjaźnione z klasą mają dostęp do wszystkich (również prywatnych) komponentów klasy, chociaż same do tej klasy (jako funkcje składowe) nie należą.

Definicja funkcji zaprzyjaźnionej z klasą:

1) Jeśli funkcja nie jest funkcją składową innej klasy, to umieszczamy w definicji klasy:

```
friend typ_zwracany nazwa_funkcji (argumenty_funkcji);
```

2) Jeśli zaś funkcja byłaby składową klasy B, to umieszczamy w definicji klasy A:

```
friend typ_zwracany B::nazwa_funkcji(argumenty_funkcji);
```

Przykład.

```
zespolona operator*(float x, zespolona z)
{ zespolona z1;
  z1.re = x*z.re;
  z1.im = x*z.im;
  return z1;
}
```

Wtedy w main() można napisać np.:

```
z2=5*z1;
```

Aby powyższa funkcja mogła poprawnie działać musi zostać zaprzyjaźnioną z klasą zespolona tzn.

```
class zespolona
{
  ⋮

 friend zespolona operator*(float, zespolona);
};
```

Zaprzyjaźnianie klas

Można zaprzyjaźnić całą klasę A z inną klasą B

```
friend class nazwa_klasy;
```

Przykład.

```
class A
{
⋮
friend class B;
⋮
}

class B
{
⋮
}
```

/* Klasa A mówi, że klasa B jest jej przyjacielem i że B może w niej korzystać ze wszystkich komponentów czyli wszystkie funkcje składowe klasy B mają dostęp do wszystkich pól składowych i funkcji składowych klasy A. Uwaga. Relacja zaprzyjaźniania nie jest symetryczna. */

Operatory **new** i **delete**

new typ_danej - operator new przydziela (alokuje) obszar pamięci potrzebny do przechowywania obiektu typu "typ-danej"i zwraca wskaźnik do początku tego obszaru (new ⇔ malloc).
Jeśli nie uda się przydzielić tej pamięci, to zwraca wskaźnik pusty NULL.

delete wskaźnik - zwalnia obszar pamięci wskazywanej przez ten "wskaźnik"(delete ⇔ free)

Przykład.

```
int *wsk, *tab;
wsk = new int;
tab = new int[10]
tab[0] = 5;          //*tab = 5;
tab[2] = 7;          //*(tab+2) = 7;
delete wsk;
delete [] tab;
```

Przykład.

```
class wektor
{int n;
 float *wsk;
 public:
        wektor(){};
        wektor(int);

 friend wektor operator*(float, wektor);
 friend float operator *(wektor, wektor);
};

wektor::wektor(int m)
{ n=m;
  wsp = new float[n]; //0,...,n-1
}



wektor operator*(float x, wektor v)
{ wektor u(v.n);
  for(i=0; i<u.n; i++)
    u.wsp[i]=x*v.wsp[i];
  return u;
}
```

```
wektor operator*(float x, wektor v)
{ wektor u(v.n);
  float *pom1, *pom2;
  pom1=u.wsp; //pom1 = &u.wsp[0];
  pom2=v.wsp;
  while(pom2<= &v.wsp[v.n-1];
  { *pom1 = (*pom2)*x;
    pom1++; pom2++;
  }
  return u;
}



float operator*(wektor u, wektor v)
{ float wart=0;
  if(u.n != v.n) return 0; //umowa
  else
  { for(int i=0; i<u.n; i++)
    { wart=wart+(*(u.wsp+i))*(*(v.wsp+i));
    }
    return wart;
  }
}
```

Przykład.

```
class punkt;
class wektor
{
 ⋮
 friend punkt operator+(punkt, wektor);
};

class punkt
{ int n;
  float *wsp;

  ⋮
  friend punkt operator+(punkt, wektor);
};



punkt operator+(punkt P, wektor v)
{ if (P.n != v.n) return P; // umowa
  else
  { punkt Q(P.n);
    for(int i=0; i<P.n; i++)
      Q.wsp[i]=P.wsp[i]+v.wsp[i];
    return Q;
   }
 }
```

Zmienna **this**

Zmienna **this** jest dostępna w każdej funkcji składowej klasy (zdefiniowana niejawnie przez kompilator), która jest wskaźnikiem na obiekt na rzecz którego funkcja została aktywowana.

Przykład.

```
class wektor
{
  ⋮
  wektor zwieksz(float k);
};
```

```
wektor wektor::zwieksz (float k)
{ for(int i=0; i<n; i++)
    wsp[i]=wsp[i]*k;
  return (*this);
}
```

II wersja

```
wektor wektor::zwieksz (float k)
{ wektor v(n);
  for(int i=0; i<n; i++)
    v.wsp[i]=k*(this→wsp[i]);
 //albo prościej  v.wsp[i]=k*wsp[i]);
  return v;
}
```

Destruktor

Likwidowaniem obiektu zajmuje się dekonstruktor (który może być zdefiniowany w ciele klasy albo zostanie dodany automatycznie (wtedy ~nazwa_klasy() {};)).

Definicja destruktora

```
~nazwa_klasy()
{
 ⋮
}
```

Destruktor, tak jak konstruktor, nie zwraca żadnej wartości.
Przed definicją destruktora nie można wpisać nawet void.
Destruktor NIE ma żadnych parametrów.
Destruktor może być tylko jeden.

```
class zespolona
{ float re, im;
  ⋮
};
```

W klasie zespolona destruktor nie jest konieczny, bo destruktor domyślny zwolni pamięć zajmowaną przez re i im.

```
class wektor
{ int n;
float *wsp;
  ⋮
~wektor();
};
```

```
wektor::wektor(int m)
{  ⋮
  wsp = new int[m];
}
```

Gdybyśmy nie mieli destruktora, destruktor domyślny zwolniłby tylko miejsce zajmowane przez zmienną n i przez wskaźnik wsp ale nie zwolniłby pamięci zajmowanej przez całą tablicę.

```
wektor::~wektor()
{ delete [] wsp;
}
```

Referencje jako argumenty funkcji

Przykład 1.

```
void zwieksz2razy(int a) //przekazywanie parametru
{ a=2*a }                //przez wartość

main()
{ int x =1;
zwieksz2razy(x);
cout<<x;  //wypisze 1
}
```

Przykład 2.

```
void zwieksz2razytablice(int tab[100], int n)
{ for(int i=0; i<n; i++)
    tab[i] = tab[i]*2;
}

main()
{ int A[100],n;
  Wczytaj(A,n);
  zwieksz2razytablice(A,n);
  Wypisz(A,n);
}
```

Przykład 3.

```
void zwieksz2razy(int &a)
{ a=2*a;
}

main()
{ int x=1;
  zwieksz2razy(x);
  cout<<x;  //wypisze 2
}
```

Deklaracja referencji

```
typ_zmiennej &nazwa_zmiennej;
```

Przekazując do funkcji argument przez referencję unikamy kopiowania jego wartości.
W rzeczywistości przekazywany jest adres tego argumentu.
Wszystkie operacje wewnątrz funkcji dotyczące referencji do tego argumentu dotyczą samego argumentu.
Przekazana referencja do argumentu jest synonimem tego argumentu.
Czyli tak jak w poprzednim przykładzie **a** jest synonimem **x**.

Operacje wejścia/wyjścia

Uruchomienie programu w C++ powoduje automatyczne otwarcie czterech strumieni (predefiniowanych):

cin - standardowy strumień wejścia (zdefiniowany w klasie istream, związany z klawiaturą)

cout - standardowy strumień wyjścia (zdefiniowany w klasie ostream, związany z monitorem)

cerr - standardowy strumień błędów (zdefiniowany w klasie ostream)
powiązany z ekranem strumień niebuforowany (każdy komunikat o błędzie pojawia się od razu na ekranie)

clog - standardowy strumień błędów - buforowany

W pliku **iostream** zdefiniowane są klasy **istream** i **ostream**.

W klasie istream zdefiniowany jest operator > > odczytujący dane ze strumienia cin (z klawiatury).

W klasie ostream zdefiniowany jest operator < < zapisujący dane do strumienia cout (na ekranie).

Formatowane funkcje wejścia/wyjścia

→ SZEROKOŚĆ POLA

cout.width(k), $k \in \mathbb{N}$ - określa na ilu miejscach należy wypisać daną liczbę

Przykład.

```
main()
{ int x=10, y=25;
   ⋮
  cout.width(5);   //wypisze _ _ _ 10
  cout<<x<<endl;   //          25
  cout<<y;         //cout.width() dotyczy tylko
                   //pierwszego wywołania cout
```

Przykład

```
char S[10];
cin.width(sizeof(S));
cin>>S;
```

Gdyby wpisać PROBABILISTYKA, to S=PROBABILIS.

→ PRECYZJA

cout. precision (k) - określa, że liczba zmiennoprzecinkowa będzie wypisywana z dokładnością do k miejsc po przecinku (kropce)

Przykład.

```
float x = 10.257;
cout.precision(2);
cout<<x;            //10.26
```

Operatory << i >> są operatorowymi funkcjami składowymi klas strumieniowych; ich lewym argumentem jest referencja do odpowiedniego strumienia.

Operatory te zwracają referencje do odpowiednich strumieni, dlatego operatory te nie mogą być przeciążone na rzecz innych klas. W celu ich przeładowywania musimy zastosować mechanizm ich zaprzyjaźniania.

Przykład.

```
zespolona z;
z.wczytaj();    //⇔ cin>>z.
```

Gdyby to przeładowanie było zrealizowane jako funkcja składowa w klasie zespolona, to musiałoby to wyglądać tak: z >> cin.

Trzeba to zrobić tak:

```
class zespolona
{
  ⋮                                          musi być
  ⋮                                                  ↘
 friend istream & operator >> (istream &, zespolona &);
 friend ostream & operator << (ostream &, zespolona );
};

istream & operator >>(istream &we, zespolona &z)
{cout<<"Podaj czesc rzeczywista i urojona liczby zespolonej: ";
 we>>z.re>>z.im;
 return we;
}
```

Przykład.

```
cin>>z1;
cin>>z2>>z3;

ostream & operator<<(ostream &wy, zespolona z)
{ wy<<z.re<<"+ i"<<z.im;
  return wy;
}
```

Przykład.

```
//jeśli z1 = 1 + i2
cout<<z1;  //1 + i2

//jeśli z2 = 1 − i2
cout<<z2;  //1 + i−2
```

Nieformatowane operacje wejścia/wyjścia

Wejście (wczytywanie)

- istream & get(**char** &znak)
  - funkcja wczytuje jeden znak ze strumienia i umieszcza go w podanej zmiennej znak
    Przykład

```
char c,d;
cin.get(c);
cin.get(c).get(d);
```

- istream & get(**char** *gdzie, **int** dlugosc, **char** ogran = '\n')
  - funkcja wczytuje do tablicy znakowej o nazwie `gdzie` co najwyżej `dlugosc` znaków (chyba, że wcześniej pojawi się znak ogran) i jako ostatni znak zostanie wpisany znak `NULL`
    Przykład

```
char nazwisko[20];       //char *nazwisko;
cin.get(nazwisko, 20); //nazwisko może być co
                       //najwyżej 19 literowe
```

- istream & getline(**char** *gdzie, **int** dlugosc, **char** ogran = '\n')
  - jw. z tym, że nie dopisuje znaku NULL na końcu

Wczytywanie binarne

- istream & read(**char** *gdzie, **int** ile)

Wyjście (wypisywanie)

- ostream & put(**char** znak)
  - wstawia jeden znak do strumienia wyjściowego
    Przykład.

```
char c='a';
cout.put(c);//na ekranie wypisze a
```

Wypisywanie binarne

- ostream & write(**char** *skad, **int** ile)
  Przykład.

```
char nazwisko[11] = "SZYDŁO";
cout.write(nazwisko, 6);
```

Operacje wejścia/wyjścia na plikach

```
#include <fstream>
```

W tej bibliotece zdefiniowane są 3 klasy "plików":

ofstream (output file stream) - plik do zapisu

ifstream (input file stream) - plik do odczytu

fstream (file stream) - plik do zapisu i odczytu

Te klasy są pochodnymi klas, odpowiednio, ostream, istream i iostream, więc można korzystać ze wszystkich funkcji działających na tamtych klasach.

Otwieranie pliku

```
void open (char *nazwa, int tryb=..., int zabezpieczenie)
                                        bardzo rzadko używane
```

Przykład.

```
ifstream f;
  f.open("C:baza.txt") //otwarty do czytania
                       //w trybie tekstowym

ofstream f1;
  f1.open("C:baza1.txt") //otwarty do wpisywania
                         //w trybie tekstowym

fstream f2;
f2.open("C:baza2.txt", ios::in) //do czytania
                       ios::out) //do zapisu
                       ios::app) //do dopisywania
                                 //na końcu pliku
                       ios::ate) //at the end -
                                 //otwórz i ustaw
                                 //się na końcu pliku
                       ios::nocreate) //otwórz,
                                   //jeśli plik istnieje
                       ios::noreplace) //otwórz,
                                  //jeśli plik nie istnieje
                       ios::trunc) //otwórz,
             jeśli plik istnieje i skasuj starą zawartość
                       ios::binary)
//otwórz w trybie binarnym (bo domyślnie otwiera w trybie tekstowym)
```

Te parametry trybu (ios::in, ...) można łączyć po kilka jednocześnie operatorem | (lub).

```
np. f.open("C:baza2.txt", ios::in | ios::nocreate);
```

Zamykanie pliku

```
nazwa_zmiennej_plikowej.close();

np. f1.close();
```

Wczytanie z pliku

```
get(...); getline(...); read(...); >>
np. ifstream f.open("...");
    char * nazwisko;
    f>>nazwisko;
```

Wpisywanie do pliku

```
put(...); write(...); <<
```

Funkcje pomocnicze

- **int** eof() - zwraca niezerową wartość, jeśli przy operacji czytania napotkany został znak końca pliku
- istream & seekg (**long** streampos, seekdir = ios::beg) - ustawia wskaźnik czytania na bajt odległy od początku pliku o streampos bajtów

```
..., seekdir = ios::end) //... od końca pliku ...
..., seekdir = ios::cur) //... od bieżącej pozycji w pliku...
```

- ostream & seekp (**long** streampos, ...) jw. tylko dotyczy wskaźnika pisania
- streampos tellg () - zwraca miejsce, w którym znajdujemy się w pliku do czytania
- streampos tellp () - zwraca miejsce, w którym znajdujemy się w pliku do pisania

Przykład.
W zbiorze slownik.txt zapisano plik, w którym w każdym wierszu znajduje się jedno słowo (co najwyżej 30-literowe). Napisz program w C++, który utworzy nowy zbiór palindromy.txt, w którym zapisze wszystkie słowa ze slownik.txt, które są palindromami.
(funkcję palindrom(...) pisaliśmy w II sem.)

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

main()
{int palindrom (char slowo[31]); //lub char *slowo;
 char slowo[31];
 ifstream f1;
 ofstream f2;
 f1.open("slownik.txt");
 f2.open("palindromy.txt");

 while(!f1.eof())
 { f1>>slowo;
   if(palindrom(slowo)==1)
      f2<<slowo<<endl;
 }

 f1.close();
 f2.close();
   }
```

Przykład 2.
Napisz funkcję o nazwie przestawienie, której parametrami będą 2 słowa, co najwyżej 30-literowe.
Funkcja powinna zwrócić wartość 1, gdy przez przestawienie 2 różnych liter w jednym słowie da się otrzymać drugie słowo oraz zwrócić 0 w przeciwnym wypadku.
W zbiorze slownik.txt (i jego kopii slownik2.txt) zapisano plik, w którym w każdym wierszu znajduje się słowo co najwyżej 30-literowe.
Napisz program w C++, który utworzy nowy zbiór "przestawianki.txt", w którym zapisze wszystkie pary słów ze zbioru slownik.txt, dla których funkcja przestawienie zwróci 1. Każda para słów powinna być wpisana w jednym wierszu.

```
int przestawienie(char slowo1[31], char slowo2[31])
   {int d1, d2;
    int nr1, nr2;
    int licznik; //licznik różnic
    d1 = strlen(slowo1);
    d2 = strlen(slowo2);

    if(d1!=d2) return 0;
```

```
    for(int i=0, i<d1;i++)
      {if slowo1[i]!=slowo2[i])
        {licznik++;
         if(licznik==1) nr1=i;
         else nr2=i;
                }
         }
        if(licznik!=2) return 0;
if((slowo1[nr1]==sowo2[nr2] && (slowo1p[nr2]==slowo2[nr1]))
          return 1;
        else return 0;
   }

   main ()
   {char slowo1[31], slowo2[31];
    ifstream f1, f2;
    ofstream f3;

    f1.open("slownik.txt");
    f3.open("przestawianki.txt");

    while(!f1.eof())
     {f1>>slowo1;
      f2.open("slownik2.txt");

       while(!f2.eof())
        {f2>>slowo2;
if((przestawienie(slowo1,slowo2)==1) && strcmp(slowo1, slowo2)<0)
                    //slowo1 jest wcześniej niż slowo2
         f3<<slowo1<<" "<<slowo2<<endl;
        }
      f2.close();
     }
    f1.close();
    f3.close();
  }
```

Pola i funkcje statyczne w klasie

Pola statyczne to pola wspólne dla wszystkich obiektów danej klasy (w pamięci występuje tylko jeden obszar dla pola statycznego niezależnie od liczby obiektów).

Deklaracja składnika pola statycznego:

```
static typ_zmiennej nazwa_pola_statycznego;
```

Wystąpienie deklaracji takiej zmiennej nie jest równoważne jego definicji, która powinna pojawić się poza ciałem klasy.

Przykład.

```
class wektor
{int n;
 float *wsp;
 public:
 static int liczba_wektorow;
  ⋮
}
```

```
main ()
{int wektor::liczba_wektorow; //definicja zmiennej
                              //liczba_wektorow
 wektor v;
 v.liczba_wektorow=1;

 /* albo można nawet:
 int  wektor::liczba_wektorow=0;
(gdy nie jest zadeklarowany żaden obiekt typu wektor) */
```

Zastosowanie pól statycznych

- Gdy obiekty chcą się porozumiewać między sobą.
- Gdy wszystkie mają tę samą cechę, która może się zmieniać.
- Gdy zliczamy liczbę elementów danej klasy.

W tym ostatnim przypadku w konstruktorze możemy wtedy dodać:

```
wektor::wektor()
{...
 liczba_wektorow++;
}
```

Do modyfikacji i odczytu pól statycznych najlepiej użyć **statycznych funkcji składowych**, przy czym pola statyczne powinny być prywatne.

Funkcje statyczne to takie funkcje, które możemy wywołać nawet gdy nie istnieje żaden obiekt danej klasy. Funkcje te nie mogą się odwoływać do niestatycznych składowych klasy bez jawnego wskazania obiektu (bo nie są aktywowane na rzecz żadnego obiektu).

Przykład.

```
class wektor
{private:
 ⋮
 static int liczba_wektorow;

 public:
 static void ustal_l_wektorow(int n) {liczba_wektorow=n;}
 static int odczytaj_l_wektorow(void) {return liczba_wektorow;}
 ⋮
};

   main ()
   { int wektor::liczba_wektorow;
     wektor::ustal_l_wektorow(0);
     cout<<wektor::odczytaj_l_wektorow(); //wypisze 0
     wektor u,v;
     cout<<v.odczytaj_l_wektorow(); //wypisze 2
      ⋮
   }
```

Fundamentalną własnością programowania obiektowego jest możliwość wykorzystywania istniejących klas do tworzenia nowych klas. W tym celu wyróżniamy dwie podstawowe możliwości:

1) kompozycja - obiekt staje się częścią (polem) innego obiektu
2) dziedziczenie - rozszerzenie obiektu klasy bazowej poprzez stworzenie obiektu klasy pochodnej (rozszerzenie własności i funkcjonalności obiektu).

Klasa jako składowa klasy

Mając zdefiniowane klasy możemy budować klasy złożone z tych wcześniej zdefiniowanych klas.

Przykład.

```
class silnik
{
 ⋮
};

class podwozie
{
 ⋮
};

class nadwozie
{int kolor;
 ⋮
};

class samochod
{silnik nazwa_silnika;
 podwozie nazwa_podwozia;
 nadwozie nazwa_nadwozia;
  ⋮
};
```

Przykład.

```
class wektor  //w ℝ^2
{float x,y;
 public:
 wektor(){};
 wektor(float, float);
 void wczytaj();
 void wypisz();
};

wektor::wektor(float a, float b)
{x=a; y=b;}

void wektor::wczytaj()
{cout<<"Podaj współrzędne wektora: ";
 cin>>x>>y;
}

void wektor::wypisz()
{cout<<"["<<x<<","<<y<<"]"; //[1,2]
}
```

```
class punkt   //w R^2
{float x,y;
 public:
 punkt(){};
 punkt(float, float);
 void wczytaj();
 void wypisz();
};

class wektorzaczepiony
{wektor v;
 punkt x;
 public:
 wektorzaczepiony(){};
 wektorzaczepiony(wektor, punkt);
 void wczytaj();
 void wypisz();
};

wektorzaczepiony::wektorzaczepiony(wektor u, punkt y)
{v=u;
 x=y;
}

void wektorzaczepiony::wczytaj()
{v.wczytaj();
 x.wczytaj();
}

void wektorzaczepiony::wypisz()
{cout<<"wektor ";
 v.wypisz();
 cout<<" zaczepiony w punkcie ";//wektor [1,2] zaczepiony
 x.wypisz();                    //w punkcie (3,1)
}
```

Dziedziczenie

Mechanizm dziedziczenia polega na przyjmowaniu własności jednej klasy (**bazowej**) przez inną klasę (**pochodną**). W skład klasy pochodnej mogą wchodzić wybrane pola klasy bazowej uzupełnione w klasie pochodnej o nowe pola danych i nowe funkcje składowe.

Mechanizm ten pozwala tworzyć ciągi coraz bardziej rozbudowanych klas przejmujących własności innych klas.

Definicja

```
class nazwa_klasy_pochodnej : modyfikator_praw_dostępu
    nazwa_kl_bazowej
{
 ⋮
};
```

Jeżeli w nagłówku klasy pochodnej użyto modyfikatora dostępu **public** to mamy do czynienia z tzw. dziedziczeniem publicznym, jeśli zaś użyto słowa **private**, to jest to tzw. dziedziczenie prywatne (rzadko używane).

W przypadku dziedziczenia publicznego wszystkie składowe publiczne klasy bazowej stają się jednocześnie składowymi publicznymi klasy pochodnej. Użytkownik klasy pochodnej będzie mógł korzystać wtedy ze składowych publicznych

klasy bazowej.

Prywatne składowe klasy bazowej **nie** są dostępne dla funkcji składowych klasy pochodnej (o ile nie są zaprzyjaźnione). Ale najczęściej mamy dostęp do nich poprzez publiczne funkcje składowe klasy bazowej.

Można również w klasie bazowej zadeklarować, że pewne składowe są chronione, tzn. **protected**, zamiast prywatne. Wtedy składowe chronione są uważane za publiczne dla funkcji składowych klasy pochodnej (i prywatne dla użytkownika klasy pochodnej.)

Nawet jeżeli jakaś składowa (pole składowe lub funkcja składowa) klasy bazowej została predefiniowana w klasie pochodnej (tzn. nazwa pola lub funkcji składowej w klasie pochodnej jest taka sama jak w klasie bazowej) to i tak funkcje składowe i użytkownik klasy pochodnej mogą w dalszym ciągu korzystać z niej. Dostęp do predefiniowanej składowej umożliwia operator zakresu (::).

Przykład.

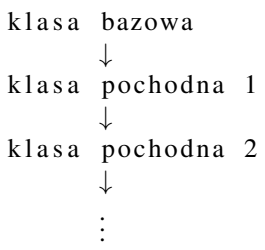
```
class punkt
{protected: float x,y;
 public:
 punkt(){};
 punkt(float, float);
 void wczytaj();
 void wypisz();
};

class punkt_kolorowy : public punkt
{int kolor;
 public:
 punkt_kolorowy(){};
 punkt_kolorowy(float, float, int);
 void wczytaj();
 void wypisz();
};

punkt_kolorowy::punkt_kolorowy(float a,float b,int n): punkt(a,b) //!
 {kolor=n;
 }

void punkt_kolorowy::wczytaj()
{punkt::wczytaj();
 count<<"Podaj kolor punktu: ";
 cin>>kolor;
}

void punkt_kolorowy::wypisz()
{punkt::wypisz();
 cout<<"kolor="<<kolor; //np. (1,5) kolor=3
}
```

Możliwe schematy dziedziczenia

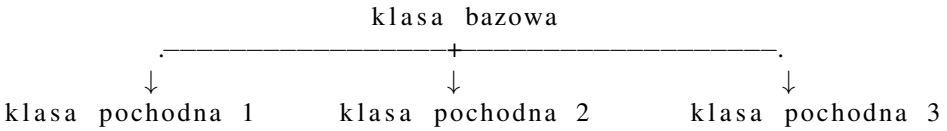
1) proste

```
klasa bazowa
      ↓
klasa pochodna
```

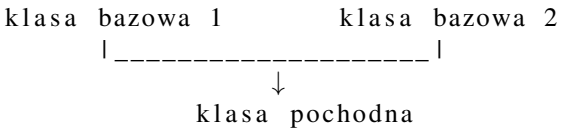
2) sekwencyjne (wielopokoleniowe)



3) wielostronne (wiele klas dziedziczy jedną klasę bazową)



4) wielobazowe (klasa pochodna dziedziczy po wielu klasach bazowych)



Przykład dziedziczenia wielobazowego.

```
class punkt
{
 ⋮
};
class wektor
{
 ⋮
};
class wektorzaczepiony2: public wektor, public punkt
{
  ⋮
 public:
 wektorzaczepiony2(){};
 wektorzaczepiony2(float a, float b, float c, float d);
 void wczytaj();
 void wypisz();
  ⋮
};

wektorzaczepiony2::wektorzaczepiony2(float a, float b,
     float c, float d):  wektor(a,b), punkt(c,d)
{
}

wektorzaczepiony2::wczytaj()
{wektor::wczytaj();
 punkt::wczytaj();
}

wektorzaczepiony2::wypisz()
{wektor::wypisz();
 punkt::wypisz();     //np. [1,2](3,4)
}
```

Struktury danych

Tworzymy je w celu lepszego gromadzenia i szybszego dostępu do danych.
Strukturami danych są m.in. tablice i struktury (struct).

Stosy i kolejki

Są to struktury danych, w których operacje wstaw i usuń jednoznacznie określają miejsce wstawienia i usunięcia elementu.

Stos - struktura danych typu LIFO (last in, first out)

Kolejka - struktura danych typu FIFO (first in, first out)

Stos

Wstawiamy element na szczyt stosu, usuwamy element również ze szczytu stosu.

Implementacja tablicowa stosu

```
class stos
{int szczyt;
 typ_klucza tab[ROZMIAR];
 public:
 stos();
 int pusty();
 void wstaw(typ_klucza x);
 typ_klucza zwroc();  //zwraca wartość na szczycie
 void usun();
};

stos::stos()
{szczyt=-1;
}
int stos::pusty()
{if(szczyt==-1) return 1;
 else return 0;
}

void stos::wstaw(typ_klucza x)
{if(szczyt < ROZMIAR-1)
 {szczyt++;
  tab[szczyt]=x; //tab[++szczyt]=x;
 }
}

typ_klucza stos::zwroc()
{if(pusty()==0)
   return tab[szczyt];
}

void stos::usun()
 {if(pusty()==0)
  {szczyt--;
  }
 }
```

Implementacja wskaźnikowa stosu

```
struct el_stosu
{typ_klucza klucz;
 el_stosu *nast;
};

class stos
{el_stosu *szczyt;
 public:
 stos();
 int pusty();
 void wstaw(typ_klucza x);
 void usun();
 typ_klucza zwroc();
 ~stos();
};

stos::stos()
{szczyt = NULL;
}

int stos::pusty()
{if(szczyt == NULL) return 1;
 else return 0;
}

void stos::wstaw(typ_klucza x)
{el_stosu *pom;
 pom = new el_stosu;
 if(pom != NULL)  //udało się zarezerwować
    {pom->klucz=x;
     pom->nast=szczyt;
     szczyt=pom;
    }
}

typ_klucza stos::zwroc()
{if(pusty()==0)
  return szczyt->klucz;
}

void stos::usun()
{if(pusty()==0)
  {el_stosu *pom;
   pom=szczyt;
   szczyt=szczyt->nast;
   delete pom;
  }
}

stos::~stos()
{while(pusty()==0)
  usun();
}
```

Kolejka

Wstawiamy element na koniec kolejki, usuwamy element z początku kolejki.

Implementacja wskaźnikowa kolejki

```
struct el_kolejki
{typ_klucza klucz;
 el_kolejki *nast;
};

class kolejka
{el_kolejki *poczatek, *koniec;
 public:
 kolejka();
 int pusta();
 void wstaw(typ_klucza x);
 void usun();
 typ_klucza zwroc();    //zwraca z początku
 ~kolejka();
};

kolejka::kolejka()
{poczatek=koniec=NULL;
}

int kolejka::pusta()
{if(poczatek == NULL) return 1;
 else return 0;
}

void kolejka::wstaw(typ_klucza x)
{el_kolejki *pom;
 pom = new el_kolejki;
 if(pom != NULL)
   {pom->klucz=x;
    if(pusta() == 1)
      poczatek=koniec=pom;
    else
     {koniec->nast=pom;
      koniec=pom;
     }
   }
}

typ_klucza kolejka::zwroc()
 {if(pusta()==0)
   return poczatek->klucz;
 }

 void kolejka::usun()
 {if(pusta()==0)
   {el_kolejki *pom;
    pom=poczatek;
    if(poczatek == koniec)
      poczatek=koniec=NULL;
    else                        //poczatek ≠ koniec
      poczatek=poczatek->nast;
```

```
      delete pom;
    }
  }

  kolejka::~kolejka()
  {while(pusta() == 0)
    usun();
  }
```

Implementacja tablicowa (cykliczna) kolejki

```
  class kolejka
  {int poczatek, koniec;
   typ_klucza tab[ROZMIAR];
   public:
   kolejka();
   int pusta();
   void wstaw(typ_klucza x);
   void usun();
   typ_klucza zwroc();
  };
```

Kolejka jest pusta $\Leftrightarrow$ początek = koniec.
Kolejka jest pełna $\Leftrightarrow$ początek = koniec + 1 (mod ROZMIAR).
Początek wskazuje pierwszy element w kolejce, a koniec wskazuje komórkę za ostatnim elementem w kolejce.
W kolejce możemy mieć maksymalnie ROZMIAR-1 elementów.

```
  kolejka::kolejka()
  {poczatek=koniec=0;
  }

 int kolejka::pusta()
  {if(poczatek==koniec) return 1;
   else return 0;
  }

  void kolejka::wstaw(typ_klucza x)
  {if(koniec+1)%ROZMIAR != poczatek)
    {tab[koniec]=x;
     koniec=(koniec+1)%ROZMIAR;
    }
  }

  void kolejka::usun()
  {if(pusta()==0)
    {poczatek=(poczatek+1)%ROZMIAR;
    }
  }

  typ_klucza kolejka::zwroc()
  {if(pusta()==0)
     return tab[poczatek];
  }
```

Lista (jednokierunkowa) → ciąg elementów tego samego rodzaju

Implementacja wskaźnikowa listy

```
struct el_listy
{typ_klucza klucz;
 el_listy *nast;
};

class lista
{el_listy *glowa;
 public:
 lista();
 int pusta();
 void wstaw(el_listy *poz, typ_klucza x);
 void wypisz_liste();
 el_listy *wyszukaj(typ_klucza);
 typ_klucza zwroc(el_listy *wsk);
 void usun(el_listy *poz);
 el_listy *nastepnik(el_listy *wsk);
 el_listy *poprzednik(el_listy *wsk);
 ~lista();
};

lista::lista()
{glowa=NULL;
}

int lista::pusta()
{if(glowa == NULL) return 1;
 else return 0;
}
```

Pozycją elementu na liście (jednokierunkowej) nazywamy wskaźnik do elementu, który poprzedza ten element w liście.
Jeśli element jest pierwszy w liście, to jego pozycją jest wskaźnik pusty NULL.

```
void lista::wstaw(el_listy *poz, typ_klucza x)
{el_listy *pom;
 pom = new el_listy;
 if(pom != NULL)
  {pom->klucz=x;

   if(poz == NULL)      //wstawianie na początku
      {pom->nast=glowa;
       glowa=pom;
      }
    else               //poz != NULL
      {pom->nast=poz->nast;
       poz->nast=pom;
      }
  }
}

typ_klucza lista::zwroc(el_listy *wsk)
{if(pusta()==0 && wsk != NULL)
  return wsk->klucz;
}
```

```
void lista::wypisz_liste()
{el_listy *pom;
 pom=glowa;
 while(pom!=NULL)
  {cout<<pom->klucz<<", ";
   pom=pom->nast;
  }
}

el_listy *wyszukaj(typ_klucza x)//zwracamy pozycję elementu
{if(pusta()==1) return NULL;     //z kluczem równym x;
 else
  {if(glowa->klucz == x) return NULL;
   else
    {el_listy *pom;
     pom=glowa;
     while(pom->nast!=NULL)     //zwracamy tylko pozycję
      {if(pom->nast->klucz == x) //pierwszego el. z x, nieistotne,
        return pom;              //czy występuje więcej x
       pom=pom->nast;
      }
     return pom;                 //jeśli x nie występuje
    }                            //zwracamy wskaźnik do
  }                              // ostatniego elementu
}

void lista::usun(el_listy *poz)
{if(pusta()==0)
  {if(poz == NULL)
    {el_listy *pom;
     pom=glowa;
     glowa=glowa->nast;
     delete pom;
    }
   else
    {pom=poz->nast;
     poz->nast=pom->nast;
     delete pom;
    }
  }
}

el_listy *lista::nastepnik(el_listy *wsk)
{if(wsk == NULL) return glowa;
 else
   return(wsk->nast);
}

el_listy *lista::poprzednik(el_listy *wsk)
{if(pusta() == 1) return NULL;
 else
  {if(wsk!=NULL)
    {el_listy *pom;
     if(wsk == glowa) return NULL;
     else
       {pom=glowa;
        while(pom->nast!=wsk)
          pom=pom->nast;
        return pom;
       }
```

```
        }
      }
    }

    lista::~lista()
    {while(pusta()==0)
      usun(NULL);
    }
```

Implementacja tablicowa listy jednokierunkowej

```
  class lista
   {int  n; //indeks ostatniego elementu
    typ_klucza tab[ROZMIAR];
    public:
    lista();
    int pusta();
    int pelna();
    void wstaw(int indeks, typ_klucza x);
    void wypisz_liste();
    int wyszukaj(typ_klucza);
    typ_klucza zwroc(int indeks);
    void usun(int indeks);
    int nastepnik(int indeks);
    int poprzednik(int indeks);
   };

   lista::lista()
   {n=-1;
   }

   int lista::pusta()
   {if(n == -1) return 1;
    else return 0;
   }

   int pelna()
   {if (n == ROZMIAR-1) return 1;
    else return 0;
   }

void lista::wstaw(int indeks, typ_klucza x)
{ if (pelna()==0)
   { for(int i=n; i>=indeks; i--)
       tab[i+1]=tab[i];

     tab[indeks]=x;
      n++;
    }
}

void lista::wypisz_liste()
{ for(int i=0; i<=n; i++)
    cout <<tab[i]<<", ";
}

int lista::wyszukaj(typ_klucza x)
{ for(int i=0; i<=n;i++)
    if(tab[i]==x) return i;
  return -1;
}
```

```
typ_klucza lista::zwroc(int indeks)
{if (pusta()==0 && indeks!=-1)
  return tab[indeks];
}

void lista::usun(int indeks)
{ if (pusta()==0 && indeks!=-1)
   { for(int i=indeks; i<n; i++)
       tab[i]=tab[i+1];

     n--;
   }
}

int lista::nastepnik(int indeks)
{if (indeks <ROZMIAR-1) return indeks+1;
  else return -1;
}

int lista::poprzednik(int indeks)
{ if (indeks != -1) return indeks-1;
   else return -1;
}
```

Porównanie efektywności

	Implementacja tablicowa	Implementacja wskaźnikowa
wstaw(...)	O(n)	O(1)
usun(...)	O(n)	O(1)
wyszukaj(...)	O(n)	O(n)
nastepnik(...)	O(1)	O(1)
poprzednik(...)	O(1)	O(n)
wyszukaj(...) w posortowanej tablicy lub liście	$O(\log_2 n)$	O(n)

Lista dwukierunkowa

```
struct ellisty2
{typ_klucza klucz;
 ellisty2 *nast, *poprz;
};

class lista2
{ellisty2 *glowa, *ogon;
 public:
 lista2();
 int pusta();
 typ_klucza zwroc(ellisty2*);
 void wypisz_od_glowy();
 void wypisz_od_ogona();
 ellisty2 *wyszukaj(typ_klucza);
 ellisty2 *nastepnik(ellisty2*);
 ellisty2 *poprzednik(ellisty2*);
 void wstaw(ellisty2*, typ_klucza);
 void usun(ellisty2*);
 ~lista2();
};
```

```
lista2::lista2()
{glowa=ogon=NULL;
}

int lista2::pusta()
{if(glowa == NULL) return 1;
 else return 0;
}

typ_klucza lista2::zwroc(ellisty2 *wsk)
{if(wsk!=NULL)
  return wsk->klucz;
}

void lista2::wypisz_od_glowy()
{ellisty2 *pom;
 if(pusta() == 0)
  {pom=glowa;
   while(pom!=ogon)
    {cout<<pom->klucz<<", ";
     pom=pom->nast;
    }
   cout<<pom->klucz;
  }
}

void lista2::wypisz_od_ogona()
{ellisty2 *pom;
 if(pusta()==0)
  {pom=ogon;
   while(pom!=glowa)
    {cout<<pom->klucz<<", ";
     pom=pom->poprz;
    }
   cout<<pom->klucz;
  }
}

ellisty2* lista2::wyszukaj(typ_klucza x)
{ellisty2 *pom;
 pom=glowa;
 while(pom!=ogon)
   {if(pom->klucz == x) return pom;
    pom=pom->nast;
   }
 if (ogon->klucz == x) return pom;
 return NULL; //nie znaleziono x
}

ellisty2* lista2::poprzednik(ellisty2 *wsk)
{if(wsk!=glowa)
  return wsk->poprz;
 else
  return NULL;
}

ellisty2* lista2::nastepnik(ellisty2 *wsk)
{if(wsk!=ogon)
  return wsk->poprz;
 else
  return NULL;
}
```

```
void lista2::wstaw(ellisty2 *wsk, typ_klucza x)
   //wsk jest wskaźnikiem do elementu poprzedzającego
   //element, który zostanie wstawiony.
   {ellisty2 *pom;
    pom=new ellisty2;
    if(pom!=NULL)
      {pom->klucz=x;
       if(wsk == NULL) //wstawiamy na poczatek
        {pom->nast=glowa;
         glowa->poprz=pom;
         glowa=pom;
         //glowa->poprz=NULL;
        }
       else
     {if(wsk == ogon) //wstawiamy na koniec
           {ogon->nast=pom;
            pom->poprz=ogon;
            ogon=pom;
            //ogon->nast=NULL;
           }
          else
           {pom->nast=wsk->nast;
            wsk->nast=pom;
            pom->nast->poprz=pom;
            pom->poprz=wsk;
           }
        }
      }
    }
```

Jeśli zadbamy o to, żeby pierwszy element w liście miał pole poprz równe NULL oraz ostatmi element miał pole nast równe NULL, to niektóre funkcje można zrobić odrobinę prościej.

```
ellisty2* lista2::wyszukaj2(typ_klucza x)
   {ellisty2 *pom;
    pom=glowa;
    while(pom!=NULL)
      {if(pom->klucz == x) return pom;
       pom=pom->nast;
      }
    return pom;    //return NULL;
   }

void lista2::usun(ellisty2 *wsk) //wsk wskazuje element
    {                                  //do usunięcia
     if (wsk!=NULL)
      {
       if (glowa!=ogon) //jest więcej niż 1 element w liście
       {
       if(wsk == glowa)
        {glowa=glowa->nast;
         }
        else
         {if(wsk == ogon)
          {ogon=ogon->poprz;
          }
          else
           {wsk->poprz->nast=wsk->nast;
            wsk->nast->poprz=wsk->poprz;
           }
```

```
        }

       }
       else //jest 1 element w liście
            // wsk=glowa=ogon
        {glowa=ogon=NULL;}

      delete wsk;
      }
    }

void lista2::usun2(ellisty2 *wsk) //wsk wskazuje element
    {                                  //do usunięcia
     if (wsk!=NULL)
      {
       if(wsk == glowa)
        {glowa=glowa->nast;
         glowa->poprz=NULL;
         }
        else
         {if(wsk == ogon)
          {ogon=ogon->poprz;
           ogon->nast=NULL;
          }
          else
           {wsk->poprz->nast=wsk->nast;
            wsk->nast->poprz=wsk->poprz;
           }
         }
       delete wsk;
      }
    }
```

Zad. Napisz funkcję sortowania listy jednokierunkowej. Zastosuj metodę sortowania przez proste wybieranie. Aby uprościć funkcję zamiany elementów zastosuj listę z "pustą głową".

Lista jednokierunkowa z pustą głową

```
   struct ellisty
   {typ_klucza klucz;
    ellisty *nast;
   };

   class lista1
   {ellisty *glowa;
    public:
    lista1();
  void  sortowanie();
    ...
   };

lista1::lista1()
   { glowa=new ellisty;
    if (glowa!=NULL)
        glowa->nast=NULL;
   }
```

Przypomnienie sortowania przez proste wybieranie w wersji tablicowej.

```
for (int i=0; i<n; i++)
 { min=A[i];
   k=i;
   for(int j=i+1; j<n; j++)
    if (A[j]<min)
      {min=A[j];
       k=j;
      }
  zamień elementy A[i] z A[k];
 }

void  lista1::sortowanie()
  { ellisty *pomi, *pomj, *pomk;
    typ_klucza min;
    pomi=glowa;
    while(pomi->nast!=NULL)
     {
      min=pomi->nast->klucz;
      pomk=pomi;
      pomj=pomi->nast;
      while(pomj->nast!=NULL)
       {
        if(pomj->nast->klucz<min)
        {min=pomj->nast->klucz;
         pomk=pomj;
        }
        pomj=pomj->nast;
       }
      zamien1(pomi, pomk);
      //lub zamien2 (pomi,pomk);
      pomi=pomi->nast;
      }
  }

void lista1::zamien1(ellisty *poz1, ellisty *poz2)
{typ_klucza z;
 z=poz1->nast->klucz;
 poz1-nast->klucz=poz2->nast->klucz;
 poz2->nast->klucz=z;
}

void lista1::zamien2(ellisty *poz1, ellisty *poz2)
{
if(poz1!=poz2)
 {if(poz1->nast==poz2)
   {ellisty *pom3;
    pom3=poz2->nast;
    poz1->nast=pom3;
    poz2->nast=pom3->nast;
    pom3->nast=poz2;
    }
   else
   { ellisty *pom3, *pom4;
     pom3=poz1->nast;
     pom4=poz2->nast->nast;

     poz1->nast=poz2->nast;
     poz2->nast->nast=pom3->nast;
```

```
        poz2->nast=pom3;
        pom3->nast=pom4;
      }
  }
}
```

Zad. dom. Napisz dwie funkcje sortowania listy dwukierunkowej. Zastosuj metody sortowania przez proste wstawiania i bąbelkowe. Aby uprościć funkcję wstawiania (bądź zamiany) elementów zastosuj listę dwukierunkową z "pustą głową"i z "pustym ogonem".