Jerzy Kisilewicz

Język C++

Programowanie obiektowe

Wydanie III

Oficyna Wydawnicza Politechniki Wrocławskiej
Wrocław 2005

Opiniodawca
Marian ADAMSKI

Opracowanie redakcyjne
Hanna BASAROWA

Projekt okładki
Dariusz GODLEWSKI

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 83-7085-891-0

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Zam. nr 582/2002.

Spis treści

1. Wprowadzenie

Książka jest przeznaczona dla osób znających standard języka C i programujących proceduralnie (nieobiektowo) w tym języku. Opisano te elementy języka obiektowego, które zostały zaimplementowane w wersji 3.1 kompilatora Borland C++. Zamieszczone przykłady zostały sprawdzone za pomocą tego właśnie kompilatora.

W trosce o niewielką objętość książki pominięto takie zagadnienia, jak: obsługa wyjątków (szczególnie przydatna do obsługi błędów), wykorzystanie klas kontenerowych, programowanie z użyciem pakietu Turbo Vision oraz tworzenie aplikacji windowsowych z użyciem Object Windows Library (OWL). Obsługę wyjątków oraz wykorzystanie klas kontenerowych szeroko opisano w książkach [12,13, 25]. Pakiet programowy OWL opisano w [1, 25, 28], natomiast proste przykłady użycia Turbo Vision w C++ zamieszczono w [21].

1.1. Rozszerzenia C++

W porównaniu ze standardem języka C, w języku C++ wprowadzono wiele zmian i rozszerzeń, takich jak: prototypy funkcji, operator zakresu, pojęcie klasy, wyrażenia strukturowe, instrukcje deklaracyjne, zmienne ustalone, zmienne referencyjne, funkcje przeciążone, argumenty domniemane, funkcje otwarte, operatory *new* i *delete*, obiektowe operacje wejścia i wyjścia.

W języku C literały (np. 'A' lub '\n') są stałymi typu *int*, natomiast w języku C++ są one stałymi typu *char*. Tak więc literały znaków o kodach większych od 127 mają wartości ujemne.

W języku C brak listy argumentów w nagłówku funkcji (np. *int main*( )) oznacza funkcję z nieokreśloną listą argumentów. Funkcję bezargumentową definiuje się z argumentem typu *void* (np *int getch*(*void*)). W języku C++ brak listy argumentów, tak samo jak argument typu *void* oznacza funkcję bezargumentową.

1.1.1. Prototypy funkcji

Prototypy są zaimplementowane w wielu kompilatorach nieobiektowych. Prototyp to nagłówek funkcji, w którym nazwy parametrów formalnych zastąpiono ich typami, a treść funkcji zastąpiono średnikiem.

Na przykład prototypami funkcji *sin*, *strchr*, *window* są:

```
double sin(double);
char *strchr(const char*, int);
void window(int, int, int, int);
```

Wywołanie funkcji powinno być poprzedzone jej definicją lub prototypem, aby kompilator mógł sprawdzić poprawność tego wywołania. Prototypy funkcji są konieczne, gdy wywoływana funkcja jest:

– dołączana z biblioteki własnej (lub kompilatora),
– dołączana wraz z innym plikiem półskompilowanym (*.obj),
– zdefiniowana w dalszej części programu.

Prototypy standardowych funkcji bibliotecznych są umieszczone w odpowiednich plikach nagłówkowych (np.: *conio.h*, *math.h*, *graphics.h*), które powinny być włączane do programu dyrektywą *#include*.

1.1.2. Instrukcje deklaracyjne

Deklaracje są traktowane jak instrukcje i nie muszą być grupowane na początku bloku. Deklaracje mogą być umieszczane między innymi instrukcjami z zastrzeżeniem, że skok przez instrukcję deklaracyjną nie jest dozwolony. Zasięg deklaracji rozciąga się od miejsca jej wystąpienia do końca bloku.

Zmienne można deklarować dokładnie tam, gdzie pojawia się potrzeba ich użycia, na przykład

```
for(int i=0; i<n; i++) . . .
```

Nie można jednak wykonywać skoków przez instrukcje deklaracyjne. W szczególności zabronione jest użycie deklaracji wewnątrz instrukcji pętli, instrukcji *if* oraz *switch*. Na przykład nie są dozwolone instrukcje deklarujące zmienną *j* w poniższej instrukcji *if*

```
if(n<=0x7FFF) int j; else long j;
```

Nie powinno się też definiować zmiennych wewnątrz pętli, tak jak poniżej zdefiniowano zmienną *j*.

```
for(int i=0; i<n; i++)
   for(int j=0; j<n; j++)  . . .
```

1.1.3. Wyrażenia strukturowe

Dla każdej struktury (i klasy) jest niejawnie definiowany operator przypisania (=). W rezultacie jest możliwe użycie struktury jako: lewego i prawego argumentu przypisania, aktualnego i formalnego argumentu funkcji oraz wyniku funkcji, np.

```
struct Para {int x, y;} X={1, 2}, Z;
Z=X;                    // przepisanie zawartości struktury X do struktury Z
```

1.1.4. Obiektowe operacje wejścia i wyjścia

Użycie obiektowych operacji wejścia i wyjścia wymaga włączenia do programu pliku nagłówkowego *iostream.h* albo pliku *fstream.h* (w miejsce *stdio.h*).

W każdym programie są predefiniowane następujące obiektowe strumienie:

cin – standardowy strumień wejściowy (jak *stdin*),
cout – standardowy strumień wyjściowy (jak *stdout*),
cerr – wyjściowy strumień diagnostyczny (jak *stderr*),
clog – wyjściowy strumień rejestrujący.

Obiektowe operatory wejścia << i wyjścia >> są łączne lewostronnie. Ich lewym argumentem jest odpowiednio obiektowy strumień wejściowy lub wyjściowy, a wynikiem jest zawsze lewy argument. Prawym argumentem dla operatora wejścia << jest L-wartość (zmienna, obiekt), a dla operatora wyjścia >> wyrażenie. Na przykład:

```
cin >> x >> y;                      // wprowadzenie x i y
cout<<"\nX="<<x<<"  Y="<<y;         // wyprowadzenie wartości x i y
```

Przed wykonaniem operacji na *cin*, *cerr* i *clog* następuje wyprowadzenie zawartości bufora *cout*.

Możliwości formatowania obiektowego wejścia i wyjścia są szersze niż nieobiektowego. Wejście i wyjście obiektowe można kojarzyć z plikami lub buforami w pamięci operacyjnej.

1.1.5. Pojęcie klasy

Klasa jest odmianą struktury. Semantyka odwoływania się do pól klas jest taka sama jak do pól struktur. W języku obiektowym sama nazwa klasy oraz sama nazwa struktury (a nie wyrażenie *struct Nazwa*) jest nazwą typu.

Tam, gdzie w języku C używano konstrukcji *struct Nazwa*, w języku C++ wystarczy tylko *Nazwa*. Na przykład aby zdefiniować struktury *X* oraz *Z* typu *Para*, wystarczy napisać

```
Para X={1, 2}, Z;
```

W języku C++ komponentami struktur i unii mogą być zarówno dane, jak i funkcje (metody). Funkcję, która jest składową danej klasy (również struktury i unii), można aktywować (wywołać) na rzecz obiektu lub wskaźnika do obiektu tej klasy (podobnie jak przy odwoływaniu się do pól struktury i klasy) za pomocą operatorów **.** (kropka) lub **–>** (minus, większe). Obiekt ten jest niejawnym argumentem tej funkcji i jest on wewnątrz niej wskazywany przez niejawnie predefiniowaną zmienną o nazwie *this*.

Komponenty klasy (również struktury) mogą być publiczne lub niepubliczne. Komponenty zdefiniowane (lub zadeklarowane) w sekcji publicznej są dostępne wszędzie (tak jak komponenty struktury w języku C). Komponenty, które nie są publiczne, mogą być używane tylko wewnątrz klasy, to znaczy tylko przez funkcje składowe klasy (lub funkcje zaprzyjaźnione).

Jeśli więc na przykład w klasie *Klasa* zdefiniowano w sekcji publicznej składową funkcję *void put*(*void*)

```
class Klasa {
 . . .
  void put();
 . . .
```

a także zdefiniowano obiekt *D* tej klasy oraz wskaźnik *p* do tego obiektu

```
Klasa D, *p=&D;
```

to funkcję *put* można wywołać na rzecz obiektu *D* (obiekt *D* jest niejawnym argumentem tej funkcji), pisząc wyrażenia *D.put*( ); lub *p–>put*( ).

W ciele funkcji *put* (pełną nazwą tej funkcji jest *Klasa*::*put*) można używać komponentów prywatnych klasy *Klasa*, podczas gdy np. w funkcji *main* używać ich nie wolno.

Zasadniczą różnicą między klasą a strukturą jest to, że domyślnie klasa zaczyna się sekcją prywatną, podczas gdy struktura – sekcją publiczną. Przykładowa definicja uproszczonej klasy liczb zespolonych może być następująca

```
class ZESP {
  private:                              // początek sekcji prywatnej
     double Re, Im;                     // pola prywatne Re, Im
  public:                               // początek sekcji publicznej
    ZESP(double re=0, double im=0)      // definicja konstruktora
```

```
      {Re=re;
        Im=im;}
    ZESP &operator+(ZESP &Z);          // deklaracja operatora +
    void put()                         // definicja funkcji
      {cout<<'('<<Re<<','<<Im<<')';}
};                                     // koniec definicji klasy
```

Komponentami prywatnymi klasy *ZESP* są pola (zmienne) *Re* oraz *Im* typu *double*. Komponentami publicznymi są tu funkcje: *ZESP* (konstruktor), *operator+* oraz *put*. Funkcja operatora *plus* jest tylko zadeklarowana. Jej definicja zostanie podana poza klasą albo dołączona z biblioteki lub z innego pliku.

1.1.6. Zmienne ustalone

Zmienne ustalone są definiowane z atrybutem *const* i mogą występować wszędzie tam, gdzie wymagane są wartości stałe, np. w definicjach tablic

```
const int Max=50;
double X[Max];
```

Zasadniczą zaletą zmiennych ustalonych w porównaniu z dyrektywą *#define* preprocesora jest to, że zmienne ustalone (jak i inne zmienne) mają swój określony typ i każde ich użycie jest dokładniej analizowane.

Wartości zmiennych ustalonych nie mogą być zmieniane. Ustalone mogą być wskaźniki oraz funkcje składowe klas. Stałe funkcje składowe nie mogą modyfikować obiektów, na rzecz których są wywoływane.

Przykłady definicji stałych:

```
const int *p;        // wskaźnik na stałą – p może być modyfikowane, *p zaś nie,
int *const q;        // stały wskaźnik – *p może być modyfikowane p zaś nie,
const int *const r;                    // stały wskaźnik na stałą typu int,
int f() const;       // stała funkcja (predefiniowane const Klasa *this;).
```

Zmienne ustalone (stałe) mogą być parametrami i wynikami funkcji.

1.1.7. Operator zakresu

W wielu kompilatorach nieobiektowych zaimplementowano operator globalności **::**. Nazwa zmiennej poprzedzona tym operatorem oznacza nazwę zmiennej globalnej (najczęściej przesłoniętej). Poprzedzenie nazwy zmiennej lub funkcji nazwą klasy i znakiem **::** oznacza odwołanie się do komponentu tej klasy.

Na przykład wyrażenie *ZESP*::*get*( ); oznacza wywołanie funkcji *get* zdefiniowanej w klasie *ZESP*, nie zaś funkcji globalnej.

Definicja konstruktora w klasie *ZESP* może mieć postać

```
ZESP(double Re=0,double Im=0)
 { ZESP::Re=Re;
   ZESP::Im=Im;}
```

Pola *Re*, *Im* klasy *ZESP* są tu przesłonięte parametrami formalnymi o takich samych nazwach. Operator zakresu umożliwia dostęp do przesłoniętych komponentów *Re*, *Im* klasy *ZESP*. Tak więc w ciele konstruktora wyrażenie *ZESP*::*Re* daje pole *Re* klasy *ZESP*, podczas gdy wyrażenie *Re* daje argument konstruktora.

1.1.8. Zmienne referencyjne

Zmienna referencyjna to zmienna tożsama z inną zmienną. Na przykład definicja

```
int k, &R=k;
```

definiuje zmienną *k* i zmienną referencyjną *R* utożsamianą ze zmienną *k*. W rezultacie zmienna *k* jest osiągalna pod dwiema nazwami: *k* oraz *R*. Referencyjny może być parametr formalny funkcji oraz wynik funkcji.

Przekazywanie parametru funkcji przez referencję polega na tym, że do funkcji przekazywany jest parametr aktualny, a nie tylko jego wartość.

Zwykle argumenty funkcji są przekazywane przez wartość. To znaczy, że funkcja dla argumentu formalnego definiuje własną lokalną zmienną, której nadaje wartość argumentu aktualnego. Wszystkie operacje funkcja wykonuje na własnej zmiennej. Jeśli argument formalny jest typu referencyjnego, np.

```
void dodaj2(int &k) {k+=2;}
```

to parametrem aktualnym musi być *L*-wartość (*L*-value), funkcja nie tworzy własnej zmiennej, lecz używa argumentu aktualnego. Tak więc powyżej zdefiniowana funkcja wywołana jako *dodaj*2(*K*); zwiększy wartość zmiennej *K* o 2. Parametrów referencyjnych używa się głównie wtedy, gdy trzeba wyprowadzić wynik przez parametr oraz aby uniknąć kopiowania dużych obiektów (jak np. struktura) do zmiennych wewnętrznych funkcji (które w przypadku parametrów referencyjnych nie są tworzone).

Jeśli funkcja ma wynik typu referencyjnego, to może być też użyta tam, gdzie wymaga się *L*-wartości, bowiem jej wynikiem jest zmienna.

Na przykład

```
int A;
int &Fun1() {return A;}
int &Fun2(int &x) {return x;}

main()
  {int K;
   Fun1()=30;          // podstawi A=30
   Fun2(K)=15;         // podstawi K=15
```

Typowe zastosowania funkcji o wynikach referencyjnych to:
- przekazanie w wyniku jednego z parametrów referencyjnych,
- przekazanie nowego zaalokowanego obiektu.

Na przykład mogą być to definicje operatorów dla poprzednio zdefiniowanej klasy ZESP:

```
ZESP& ZESP::operator+(ZESP &Z)
  {ZESP *t = new ZESP;     // alokacja pamięci dla obiektu ZESP
   t->Re = Re + Z.Re;      // obliczenie wartości pola t->Re
   t->Re = Im + Z.Im;      // obliczenie wartości pola t->Im
   return *t;              // zwrot obiektu
  }
ostream &operator<<(ostream &wy, ZESP &Z)
  { return wy<<'('<<Z.Re<<','<<Z.Im<<')'; }
```

Funkcja *operator*+ daje w wyniku zaalokowany w niej obiekt klasy *ZESP*. Wynikiem funkcji *operator*<< jest jej pierwszy argument *wy*. Argument ten jest referencyjnym wynikiem globalnego operatora wyjścia <<. Zauważmy, że aby można było przekazać w wyniku referencję do argumentu formalnego, argument ten musi też być typu referencyjnego. Inaczej bowiem argumentowi formalnemu odpowiadałaby wewnątrz funkcji automatyczna zmienna lokalna, która przestaje istnieć po wykonaniu się tej funkcji.

Jako wyniku funkcji nie wolno przekazywać referencji do lokalnych zmiennych automatycznych tej funkcji. Te zmienne przestają istnieć, gdy funkcja zostanie

wykonana. W takim przypadku wynikiem byłaby referencja do już nieistniejącej zmiennej.

1.1.9. Funkcje przeciążone

Różne funkcje o tej samej nazwie nazywamy funkcjami przeciążonymi. Funkcje przeciążone mają wspólną nazwę, ale muszą różnić się liczbą parametrów lub typami parametrów. Sama różnica w typie wyniku tu nie wystarcza. Przykładem są funkcje statyczne *int rozmiar*( ) oraz *int rozmiar*(*int*) zdefiniowane w klasie

```
class TABLICA {
    static int Rozmiar;
     . . .
  public:
    static int rozmiar(){return Rozmiar;}
    static int rozmiar(int n)
       {Rozmiar=n; return Rozmiar;}
     . . .
    };
```

Pierwsza funkcja daje w wyniku rozmiar tablic, druga natomiast służy do ustawiania nowego rozmiaru

Dobrym zwyczajem jest wywoływanie funkcji przeciążonych z dokładnymi typami parametrów aktualnych. Jeśli na przykład są zdefiniowane funkcje *fun*(*int*) i *fun*(*float*), to która z nich będzie wywołana w instrukcji *fun*(*L*); jeśli *L* jest typu *long*? Jeśli zdefiniowano funkcje *Fun*(*int*, *float*) oraz *Fun*(*float*, *int*), to próby wywołania *Fun*(*i*, *j*) oraz *Fun*(*x*, *y*) zakończą się błędem kompilacji; gdy oba argumenty *i*, *j* są typu *int*, a *x*, *y* są typu *float*. W tych bowiem przypadkach kompilator nie potrafi zdecydować, które funkcje należy wywołać.

1.1.10. Argumenty domniemane

W definicji lub deklaracji funkcji można podać domyślne wartości dla wszystkich lub kilku ostatnich argumentów. Wywołując tę funkcję można opuścić maksymalnie tyle argumentów aktualnych, ile argumentów formalnych ma wartości domyślne. Te właśnie wartości nadaje się automatycznie brakującym ostatnim argumentom.

Jeśli na przykład zdefiniowano:

```
int suma(int x, int y=3, int z=10) {return x+y+z; }
```

to instrukcje

```
A = suma(50, 7);          // to samo co A=suma(50,7,10);
B = suma(100);            // to samo co A=suma(100,3,10);
```

podstawią $A = 67$ oraz $B = 113$.

1.1.11. Funkcje otwarte

Funkcje deklarowane ze słowem kluczowym *inline* umieszczonym przed nagłówkiem funkcji są funkcjami otwartymi. Kod wynikowy funkcji otwartej może być przez kompilator wpisany w każdorazowym miejscu jej wywołania.

Na przykład

```
inline int suma(int x, int y=3, int z=10)
     {return x+y+z;}
```

W odróżnieniu od makrodefinicji dla funkcji otwartych dokonuje się kontroli typów, konwersji argumentów itp.

Funkcjami otwartymi na ogół są bardzo krótkie funkcje, których czas wykonania jest krótszy od czasu ich wywołania i powrotu z wywołania. Zyskuje się tu na czasie obliczeń, a czasem nawet skraca się kod wynikowy programu.

Jeśli funkcja zawiera pętle iteracyjne (instrukcje *while*, *do*, *for*), to kompilator zignoruje specyfikację *inline*.

Funkcje zdefiniowane wewnątrz opisu klasy (np. funkcja *put* w klasie *ZESP* lub funkcje *rozmiar* w klasie *TABLICA*) **są domyślnie funkcjami otwartymi**.

1.1.12. Operatory *new* i *delete*

Operator *new* służy do alokacji pamięci (podobnie jak funkcje *malloc* i *calloc*) obiektom i tablicom obiektów. Przy alokacji obiektów (ale nie tablic) jest możliwa ich inicjalizacja.

Operator *new* ma postać:

```
new typ                 // alokacja obiektu
new typ (wartość)       // alokacja obiektu z inicjacją
new typ[rozmiar]        // alokacja tablicy obiektów
```

Typ wskaźnika jest dopasowany do typu alokowanego obiektu, tak że nie są potrzebne żadne konwersje wskaźnikowe. Jeśli alokacja nie jest udana, *new* zwraca wskaźnik pusty *NULL*.

Operator *delete* służy do zwalniania przydzielonej pamięci. Ma on jedną z postaci:

```
delete ptr;          // zwolnienie obiektu lub tablicy prostych obiektów
                        (nie tworzonych przez konstruktory)
delete [n] ptr;      // zwolnienie tablicy obiektów (stare kompilatory)
delete [] ptr;       // zwolnienie tablicy obiektów (nowe kompilatory)
```

Postać operatora *delete* z nawiasami [ ] jest stosowana tylko wtedy, gdy tablica zaalokowana instrukcją *new typ*[*rozmiar*]; jest tablicą obiektów, czyli identyfikator *typ* jest nazwą klasy. W tej klasie musi być zdefiniowany (jawnie lub domyślnie) konstruktor bezargumentowy.

Przykłady

```
ZESP *px=new ZESP,                  // obiekt klasy ZESP
     *py=new ZESP(2,7),             // obiekt klasy ZESP z inicjacją
     *tab1=new ZESP[n],             // tablica n obiektów klasy ZESP
     (*tab2)[8]=new ZESP[n][8];     // tablica n wierszy po 8 obiektów

ZESP **tab=new ZESP*[n];            // tablica n wskaźników
    . . .
```

W dalszej części programu można używać wyrażeń *tab*1[*i*], *tab*2[*i*][*j*] oraz *tab*[*i*] dla $i = 0, 1, ..., n-1$ oraz $j = 0, 1, ..., 7$. Zaalokowane powyżej obszary pamięci po wykorzystaniu należy zwolnić następującymi instrukcjami:

```
delete px;
delete py;
delete [ ]tab1;
delete [ ]tab2;
delete tab;
```

Przykłady funkcji zwalniającej i alokującej prostokątną macierz obiektów klasy *ZESP* o *n* wierszach oraz *m* kolumnach:

```
void Deletetab(ZESP **A)
  {if(!A) return;
     for(int i=0; A[i]; i++) delete [ ] A[i];
     delete A;
  }

ZESP **Newtab(int n, int m)
  {ZESP **A=new ZESP*[n+1];
     if(!A) return A;
     for(int i=0; i<n; i++)
        {A[i]=new ZESP[m];
```

```
            if(!A[i]) {Deletetab(A); return NULL;}
          }
        A[n]=NULL;
        return A;
    }
```

Pytania i zadania

1.1. Napisz prototypy pięciu wybranych funkcji: a) obsługi ekranu, b) graficznych, c) operacji na tekstach.

1.2. Referencja jakich zmiennych może być wynikiem funkcji, a jakich nie i dlaczego?

1.3. Punkt może być położony na prostej, na płaszczyźnie lub w przestrzeni. Napisz jedną funkcję z argumentami domniemanymi, która obliczy odległość tego punktu od początku układu współrzędnych.

1.4. Kąt między wektorem (x, y) a osią X może być określony jednym argumentem w przedziale $(-\pi/2, \pi/2)$ jako arctg(y/x) lub w przedziale $(-\pi, \pi\rangle$ za pomocą dwu liczb: x, y oraz w szerszym przedziale jako $\varphi = \varphi_0 + 2\pi n$ za pomocą funkcji trzech zmiennych. Zdefiniuj trzy funkcje przeciążone oraz równoważną im jedną funkcję z argumentami domniemanymi.

1.5. Zakładając, że zmiennym N, M zostały nadane wartości, napisz instrukcje definiujące i inicjujące zmienną A, które używając operatora *new* przydzielą pamięć na:
a) tablicę N liczb typu *double*,
b) tablicę liczb typu *double*, o N wierszach i czterech kolumnach,
c) tablicę liczb typu double, o N wierszach i M kolumnach.
Jak zwolnić przydzieloną pamięć?

1.2. Przykładowy program ZESP

Przykładem będzie program dodawania dwu liczb zespolonych, napisany w nieobiektowym języku C oraz w języku obiektowym C++, napisany w stylu proceduralnym i obiektowym.

1.2.1. Program w języku C

```
#include <stdio.h>
#include <conio.h>
```

Definicja struktury dla liczby zespolonej

```
struct ZESP {
   double Re, Im;
   };
```

Definicja funkcji obliczania sumy **pc* liczb wskazywanych przez *pa* i *pb*

```
void
 dodaj(struct ZESP *pa,struct ZESP *pb,struct ZESP *pc)
   {pc->Re=pa->Re+pb->Re;
    pc->Im=pa->Im+pb->Im;
   }
```

Funkcja drukowania liczby zespolonej wskazywanej przez *p*

```
void putzesp(struct ZESP *p)
   {printf("(%.3lf, %.3lf)", p->Re, p->Im);
   }

main(void)
   {struct ZESP A={1.23, 3.14}, B={10, 20}, C;
    clrscr();
    dodaj(&A, &B, &C);
    putzesp(&C);
    return(0);
   }
```

1.2.2. Program proceduralny w języku C++

Zastosowano tu udogodnienia języka C++ (wyrażenia strukturowe, zmienne referencyjne, funkcje przeciążone, obiektowe operacje wyjścia) bez definiowania klasy. Wynikiem funkcji *dodaj* jest struktura zawierającą sumę dwu liczb zespolonych, które są przekazywane tej funkcji przez referencję.

```
#include <iostream.h>        // opisy funkcji wejściowych i wyjściowych
#include <iomanip.h>         // opisy manipulatorów
#include <conio.h>

struct ZESP {
   double Re, Im;
   };
```

Definicja funkcji obliczania sumy liczb *a* i *b* (przekazanych przez referencję)

```
ZESP dodaj(ZESP &a, ZESP &b)
   {ZESP c;
    c.Re=a.Re+b.Re;
    c.Im=a.Im+b.Im;
    return c;                    // wynikiem jest struktura
   }

void put(ZESP &z)                // przeciążona funkcja put
   {cout<<setprecision(3)<<'('<<z.Re<<", "<<z.Im<<')';
   }

main()
   {ZESP A={1.23, 3.14}, B={10, 20}, C;
    clrscr();
    C=dodaj(A, B);
    put(C);
    return(0);
   }
```

1.2.3. Program obiektowy

W miejsce struktury wprowadzono klasę i zdefiniowano w tej klasie dotychczasowe funkcje oraz wprowadzono funkcję konstruktora z argumentami domniemanymi. Funkcję *dodaj* przemianowano na funkcję *operator+*.

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

class ZESP {
   private:                                  // początek sekcji prywatnej
     double Re, Im;                          // prywatne pola Re, Im
   public:                                   // początek sekcji publicznej
     ZESP(double re=0, double im=0)          // definicja konstruktora
       { Re=re; Im=im;}
     ZESP operator+(ZESP &b)                 // definicja funkcji dodawania
       { return ZESP(Re+b.Re, Im+b.Im); }
     void put()                              // definicja funkcji drukowania
```

```
      {cout<<setprecision(3)<<'('<<Re<<", "<<Im<<')'; }
};

main()
   {ZESP A(1.23, 3.14), B(10, 20), C;
    clrscr();
    C=A+B;
    C.put();
    return(0);
   }
```

Funkcję *put* z klasy *ZESP* można zastąpić funkcją operatorową *operator*<< (por. p. 1.2.4).

1.2.4. Program z oprogramowaną klasą

Oddzielono definicję klasy od definicji funkcji składowych klasy. Całość podzielono na trzy pliki: *zesp.h*, *zesp.cpp* i *prog.cpp*.

Plik **zesp.h**. Plik *zesp.h* zawiera definicję klasy i jest przeznaczony do włączania do innych kompilowanych plików. Definicje funkcji zastąpiono ich deklaracjami, choć definicje krótkich funkcji korzystniej byłoby zostawić lub zdefiniować funkcje te z atrybutem *inline*.

```
#include <iostream.h>
#include <iomanip.h>

class ZESP {
   private:
     double Re, Im;
   public:
     ZESP(double=0, double=0);
     ZESP operator+(ZESP&);
     friend ostream &operator<<(ostream&, ZESP&);
};
```

Lewym argumentem funkcji operatorowej *operator*<< jest obiekt klasy *ostream*, a nie obiekt klasy *ZESP*. Tak więc gdyby funkcja *operator*<< była funkcją klasy, to będąc aktywowaną na rzecz lewego argumentu, mogłaby być tylko funkcją klasy *ostream*. Definicji klasy *ostream* nie można jednak zmieniać. Funkcja *operator*<<

może być zatem tylko funkcją globalną. Musi mieć ona jednak dostęp do prywatnych komponentów *Re* oraz *Im* klasy *ZESP*. Musi być więc zaprzyjaźniona z klasą *ZESP*.

Plik **zesp.cpp**. Plik *zesp.cpp* zawiera definicje funkcji składowych klasy: konstruktora, operatora + oraz operatora <<. Identyfikatory funkcji klasy *ZESP* muszą być poprzedzone kwalifikatorem *ZESP*::. Plik ten po skompilowaniu do postaci *zesp.obj* będzie dołączany przez linker do programów na etapie konsolidacji.

```
#include "zesp.h"

ZESP::ZESP(double Re, double Im): Re(Re), Im(Im)
   { }

ZESP ZESP::operator+(ZESP &b)
   { return ZESP(Re+b.Re, Im+b.Im); }

ostream &operator<<(ostream &wy, ZESP &z)
   {return
      wy<<setprecision(3)<<'('<<z.Re<<", "<<z.Im<<')';
   }
```

Plik **prog.cpp**. Plik *prog.cpp* jest plikiem kompilowanego programu.

```
#include <conio.h>
#include "zesp.h"

main()
   {ZESP A(1.23, 3.14), B(10, 20), C;
    clrscr();
    C=A+B;
    cout<<C;
    return(0);
   }
```

Podczas kompilacji tego pliku nie kompiluje się pliku *zesp.cpp*, lecz dołącza jego półskompilowaną postać *zesp.obj*. Aby linker dołączył plik *zesp.obj*, kompilacja powinna odbywać się zgodnie z projektem zawierającym kolejno pliki: *prog.cpp* i *zesp.obj*.

Aby utworzyć projekt należy:

1. Wybrać opcję „Project” (główne menu), a następnie „Open project...”, aby otworzyć okno dialogowe „Open Project File”. W tym oknie należy wpisać taką nazwę tworzonego projektu, jaką powinien otrzymać skompilowany program

wykonywalny. W rezultacie zostanie otwarte okno z zawartością projektu – puste podczas tworzenia nowego projektu.

2. Do projektu należy wstawić wszystkie pliki, które mają być uwzględnione przez kompilator i linker. Aby wstawić plik do projektu należy wybrać (w podmenu „Project") opcję „Add item..." lub nacisnąć klawisz Insert, aby otworzyć okno "Add to Project List". Za pomocą tego okna należy wstawić wymagane pliki do projektu. W przykładowym programie będą to pliki: prog.cpp oraz zesp.obj. Każdy zbędny element projektu można usunąć podświetlając go i naciskając klawisz Del.

Po otwarciu projektu kompilacja programu ("Make", "Link" oraz "Build all") jest realizowana zgodnie z projektem, a utworzony program wykonywalny otrzymuje nazwę projektu (z rozszerzeniem .exe).

Aby projekt został zapamiętany na dysku, musi być ustawiona opcja "Project" w oknie "Auto-Save" w opcjach "Options | Environment | Preferences".

1.3. Tworzenie bibliotek. Program TLIB

Pliki półskompilowane (*.obj) wymienione w projekcie są dołączane do programu wykonywalnego niezależnie od tego, czy są one rzeczywiście potrzebne czy nie. Selektywny wybór potrzebnych definicji jest dokonywany podczas przeglądania bibliotek (*.lib). Do tworzenia bibliotek służy program TLIB.EXE umieszczony w tym samym podkatalogu co program BC.EXE.

Aby dołączyć do biblioteki wybrane funkcje lub utworzyć nową bibliotekę z tymi funkcjami, należy:

1. Umieścić każdą funkcję w oddzielnym pliku źródłowym. Dobrze jeśli nazwa pliku pokrywa się całkowicie lub częściowo z nazwą funkcji (plus rozszerzenie .cpp).
2. Skompilować oddzielnie każdą funkcję (opcją "Compile | Compile") do postaci półskompilowanej z rozszerzeniem .obj. Należy zwrócić uwagę, aby wszystkie funkcje były skompilowane w tym samym modelu pamięci, w jakim będzie kompilowany program. Dla różnych modeli pamięci (*Tiny*, *Small*, *Medium*, *Compact*, *Large*, *Huge*) należy tworzyć różne biblioteki.
3. Użyć programu TLIB.EXE, aby umieścić półskompilowane funkcje w pliku bibliotecznym.

Uruchomienie programu TLIB.EXE ma postać

```
TLIB libname [/C] [/E] [/P] [/0] commands, listfile
```

gdzie: *libname* jest nazwą pliku bibliotecznego,
commands jest sekwencją nazw modułów poprzedzonych symbolami operacji,

listfile	jest opcjonalną nazwą pliku na listing,
/C	biblioteka z rozróżnianiem wielkości liter,
/E	kreowanie rozszerzonego słownika,
/P*size*	ustawienie wielkości strony na *size*,
/0	usunięcie komentarzy.

Symbole operacji umieszczane przed nazwami modułów:

+	dodaj moduł do biblioteki,
–	usuń moduł z biblioteki,
*	wyjmij moduł (do pliku *.obj) bez usuwania go z biblioteki,
–+ lub +–	zastąp moduł w bibliotece,
–* lub *–	wyjmij moduł i usuń go z biblioteki,
@	wykonaj moduł przetwarzania wsadowego.

Przykłady użycia programu TLIB

Utworzenie pliku bibliotecznego *moja.lib* z plików *x.obj*, *y.obj* oraz *z.obj*.

```
tlib moja +x +y +z
```

Utworzenie pliku *moja.lst* z wykazem modułów zawartych w pliku bibliotecznym *moja.lib*.

```
tlib moja, moja.lst    lub    tlib moja, moja
```

Dopisanie do pliku bibliotecznego *moja.lib* pliku *b.obj*.

```
tlib moja +b
```

Aktualizacja pliku *moja.lib*: zastąpienie *x* wersją *x.obj*, dopisanie *a.obj*, usunięcie *z.obj*.

```
tlib moja -+x +a -z
```

Utworzenie pliku *x.obj* z modułu *x* w *moja.lib* oraz umieszczenie listingu w *wykaz.lst*.

```
tlib moja *y, wykaz.lst    lub    tlib moja *y, wykaz
```

Utworzenie pliku *abc.lib* według pliku *abc.rsp* oraz umieszczenie listingu w pliku *abc.lst*.

```
tlib abc @abc.rsp, abc.lst  lub  tlib abc @abc.rsp, abc
```

Plik *abc.rsp* jest plikiem tekstowym zawierającym kolejne komendy do wykonania. Jeśli na przykład plik *abc.lib* powinien zawierać moduły *a.obj*, *b.obj*, *c.obj*, *d.obj*, *e.obj*, *f.obj* oraz *g.obj*, to plik *abc.rsp* powinien zawierać tekst

```
+a.obj +b.obj +c.obj +d.obj +e.obj +f.obj +g.obj
```

Jeżeli linia komend jest długa, to można ją kontynuować po znaku & w następnej linii tekstu. Tak więc plik *abc.rsp* może też zawierać tekst

```
+a.obj +b.obj +c.obj &
   +d.obj +e.obj +f.obj +g.obj
```

Rozszerzenia *.obj* mogą zostać pominięte.

Przykład utworzenia pliku bibliotecznego *zesp.lib* do programu z rozdziału 1.2.4

Niech pliki *zesp.cpp*, *plus.cpp*, *wy.cpp* zawierają kolejno definicje: konstruktora *ZESP*, funkcji *operator+* oraz funkcji *operator<<* poprzedzone dyrektywą kompilatora *#include* "*zesp.h*". Każdy z tych plików należy skompilować (wybierając z menu „Compile|Compile") w celu utworzenia półskompilowanych plików: *zesp.obj*, *plus.obj*, *wy.obj*. Plik biblioteczny *zesp.lib* należy utworzyć poleceniem

```
tlib zesp +zesp +plus +wy
```

Do skompilowania programu z pliku *prog.cpp* należy utworzyć projekt zawierający pliki: *prog.cpp* oraz *zesp.lib*.

1.4. Budowanie klas

Klasa z formalnego punktu widzenia jest odmianą struktury, która domyślnie zaczyna się sekcją prywatną. W języku C++ komponentami klasy i struktury mogą być nie tylko pola danych, ale również funkcje składowe zwane metodami. Zasady posługiwania się komponentami klasy są takie same jak komponentami struktury.

Definiując klasę tworzy się nowy typ. Zasadniczą ideą tworzenia klasy jest to, aby posługując się obiektami tej klasy (zmiennymi typu tej klasy) można było odwoływać się do tych obiektów jako do całości, bez możliwości ingerencji do ich wnętrza.

Na przykład użytkownik prawidłowo zbudowanej klasy, której obiektem będzie tekst, nie powinien się martwić tym, jak obiekty pamiętają i przetwarzają teksty. Nie powinien więc zabiegać o odpowiednie bufory na wprowadzane teksty ani martwić się o to, by połączone teksty zmieściły się w buforze. O te i o inne sprawy powinny dbać same obiekty. Podobnie użytkownik klasy wektorów nie powinien mieć bezpośredniego dostępu do elementów wektora, a jedynie do wektora jako całości. Wszystkie operacje (wczytywanie, drukowanie, dodawanie, itp.) powinny być wykonywane na całych wektorach.

1.4.1. Hermetyzacja danych i metod

Wewnętrzna struktura obiektu powinna zawsze być ukryta przed swobodnym dostępem. Poszczególne komponenty klasy (dane i metody) ukrywa się definiując (lub deklarując) je, zależnie od stopnia ukrycia, jako prywatne lub zabezpieczone.

Komponenty klasy (i struktury) mogą być zadeklarowane w sekcjach: **prywatnej** (*private*), **zabezpieczonej** (*protected*) i **publicznej** (*public*).

Na przykład

```
class Klasa {
  private:          // początek sekcji prywatnej
    . . .           // definicje komponentów prywatnych
  protected:        // początek sekcji zabezpieczonej
    . . .           // definicje komponentów zabezpieczonych
  public:           // początek sekcji publicznej
    . . .           // definicje komponentów publicznych
};                  // koniec definicji klasy
```

Komponenty prywatne i zabezpieczone są dostępne jedynie w funkcjach swojej klasy oraz w funkcjach zaprzyjaźnionych z tą klasą (funkcjach nie należących do klasy, ale zadeklarowanych w tej klasie z atrybutem *friend*).

Komponenty zabezpieczone klas bazowych mogą być dostępne (jako zabezpieczone) w klasach pochodnych, podczas gdy komponenty prywatne nie mogą być dostępne.

Komponenty publiczne są dostępne wszędzie.

Na przykład jeśli *z* jest obiektem klasy *ZESP*, to wyrażenie *Z.Re* może być użyte tylko w funkcjach klasy *ZESP* (np. w funkcji *put*) lub funkcjach zaprzyjaźnionych z klasą *ZESP*. Wyrażenia tego nie można użyć nigdzie poza klasą *ZESP*, np. w funkcji *main*, ponieważ komponent *Re* zdefiniowano w sekcji prywatnej. Wyrażenie *Z.put*( ) może być użyte wszędzie, ponieważ funkcję *put* zadeklarowano w sekcji publicznej. Zauważmy, że w funkcji *put* użyto nazw *Re* oraz *Im* bez wiązania ich z jakimkolwiek obiektem. W tym przypadku odnoszą się one do tego obiektu, na rzecz którego funkcja *put* została wywołana – w wyrażeniu *Z.put*( ); odnoszą się do pól *Z.Re* i *Z.Im* obiektu *Z*. Obiekt ten jest wskazywany przez predefiniowaną zmienną *this* (wyrażenie **this* daje tu obiekt *Z*).

1.4.2. Pola i funkcje statyczne

Pola statyczne są deklarowane z atrybutem *static*. Są to pola wspólne wszystkim obiektom danej klasy i istnieją niezależnie od obiektów tej klasy. Pole statyczne zajmuje tylko jedno miejsce w pamięci niezależnie od liczby istniejących obiektów. Pola statyczne klas globalnych można inicjować w normalny sposób.

Funkcje statyczne są deklarowane z atrybutem *static*. Funkcje statyczne nie są wywoływane na rzecz obiektów klasy, tak więc funkcje te nie mają zmiennej *this* i nie

mogą się odwoływać do niestatycznych komponentów klasy. Funkcje statyczne wywołuje się na rzecz klasy reprezentowanej przez swoją nazwę, obiekt lub wskaźnik (np. *Klasa*::*fun*( ); lub *x*.*fun*( ); lub *p*–>*fun*( );), na przykład:

```
class TABLICA {
  static int Rozmiar;                    // deklaracja pola statycznego
   . . .
public:
  static int rozmiar(){return Rozmiar;}
  static int rozmiar(int n)
      {Rozmiar=n; return Rozmiar;}
   . . .
  };                                              // koniec deklaracji klasy
 . . .
int TABLICA::Rozmiar=44;                          // definicja inicjująca
main()
{ int N=TABLICA::rozmiar();     // wywołanie funkcji rozmiar( ), N=44
  TABLICA x;
  x.rozmiar(50);            // wywołanie funkcji rozmiar(int), Rozmiar=50
```

Pola statyczne najczęściej służą do przechowywania danych wspólnych wszystkim obiektom oraz do przekazywania danych między obiektami. W powyższym przykładzie wspólną cechą wszystkich obiektów klasy *TABLICA* jest ich rozmiar zapamiętany w polu statycznym o nazwie *Rozmiar*.

Wspólną cechą wszystkich obiektów może być sposób ich wyprowadzania. Na przykład liczby zespolone można wyprowadzać w postaci (*Re*, *Im*) lub *Re*+*Im*·*j* z precyzją *pn* cyfr po kropce dziesiętnej. Ponieważ sposób prezentacji liczb zespolonych jest w programie (lub jego części) jednakowy, nie ma sensu pamiętać go w każdym obiekcie osobno. W tym celu zostaną zdefiniowane prywatne pola statyczne *pn* oraz *postac*. Definicja klasy *ZESP* będzie teraz następująca:

```
class ZESP {
  private:
     static int pn, postac;
     double Re, Im;
  public:
    . . .
     void put()
      {cout<<setprecision(pn);
       if(postac)
        { cout.setf(showpos); cout<<Re<<Im<<"*j";}
```

```
            else
            {cout.unsetf(showpos);
cout<<'('<<Re<<','<<Im<<')';}
          }
   };
```

Pola statyczne zadeklarowane w klasie globalnej muszą zostać zdefiniowane i inicjowane na poziomie globalnym w zwykły sposób przed definicją jakiegokolwiek obiektu tej klasy, na przykład

```
int TABLICA::Rozmiar=44;
int ZESP::pn=3, ZESP::postac=0;
```

Tak więc statyczne komponenty *pn* i *postac* klasy *ZESP* mają charakter zmiennych globalnych. Są to jednak prywatne komponenty klasy *ZESP* i tylko funkcje tej klasy (lub funkcje z nią zaprzyjaźnione) mogą się odwoływać do tych komponentów.

Późniejsza zmiana wartości pól statycznych powinna odbywać się za pomocą funkcji statycznych zadeklarowanych (albo zdefiniowanych) w sekcji publicznej. Na przykład za pomocą funkcji *int rozmiar*(*int n*) w klasie *TABLICA* lub funkcji *precyzja* oraz *ustaw_postac* w klasie *ZESP*.

```
class ZESP {
    . . .
  public:
    . . .
      static void precyzja(int n)
                    {pn=(n<0)?0:(n>6)?6:n;}
      static void ustaw_postac(int n) {postac=n;}
};
```

Funkcje statyczne mogą być wywoływane z nazwą klasy nawet przed utworzeniem obiektów klasy, np.

```
ZESP::precyzja(1);
ZESP::ustaw_postac(1);
```

Gdy istnieją obiekty klasy lub wskaźniki (np. *ZESP Z*, **pz*;), to funkcje statyczne można też aktywować tak jak inne funkcje klasy, np.

```
Z.precyzja(2);
pz->precyzja(3);
```

Funkcje statyczne nie są aktywowane na rzecz żadnego obiektu klasy (podanie obiektu lub wskaźnika służy tylko do określenia z jakiej klasy ma być aktywowana funkcja). Wewnątrz funkcji statycznej nie można zatem odwoływać się do niestatycznych komponentów klasy bez jawnego podania obiektów (i operatorów **.** lub –>), natomiast do statycznych komponentów można odwoływać się bezpośrednio. Na przy-

kład wewnątrz funkcji *precyzja* odwołujemy się bezpośrednio do pola *pn*, natomiast nie można w taki sposób odwołać się do pola *Re* ani *Im*. Pola te bowiem są indywidualne dla każdego obiektu i w odwołaniu należy określić właściwy obiekt, np. *Z.Re*.

1.4.3. Wzorce klas i funkcji

Czasami zachodzi konieczność definiowania wielu podobnych klas dla różnych typów danych. Na przykład klasę *TAB* można by zdefiniować jako klasę tablic zmiennych typu *int*, *double*, *char*, *ZESP* itp. Poszczególne definicje różniłyby się tylko typem elementów tablicy. W tej sytuacji warto zdefiniować wspólny wzorzec klasy, w którym nazwa typu elementów tablicy wystąpiłaby w postaci parametru.

W języku C++ można definiować wzorce klas i funkcji. Każdy wzorzec rozpoczyna się wyrażeniem złożonym ze słowa kluczowego *template* oraz listy parametrów w nawiasach < >. Na przykład napis

```
template <class Typ>
```

rozpoczyna definicję (klasy lub funkcji) w której identyfikator *Typ* oznacza nazwę typu (lub klasy). Jeśli definiowany jest szablon klasy o nazwie *TAB*, to generowana klasa nazywa się *TAB<Typ>*, np. *TAB<int>*, *TAB<char>* lub *TAB<ZESP>*. Przykładowy program definiuje szablon klasy *TAB* i używa go do zdefiniowania powyższych trzech klas.

```
#include<iostream.h>
#include<iomanip.h>
#include<string.h>
#include<zesp.h>

template <class Typ> class TAB{   // początek definicji klasy
   static int w;                  // szerokość pola wydruku
   int tmp, N;                    // rozmiar tablicy
   Typ *A;                        // wskaźnik tablicy
 public:
   TAB():tmp(0),N(0),A(0) {}      // konstruktor bezargumentowy
   TAB(Typ s):tmp(0),N(1),A(new Typ)
     {*A=s;}                      // konwersja z Typ do TAB<Typ>
   TAB(Typ[], int);               // konstrukcja z tablicy
   TAB(TAB&);                     // konstruktor kopiujący
   ~TAB();
   TAB &operator=(TAB&);
   friend ostream &operator<<(ostream&,TAB&);
```

```
};
```

Zauważmy, że definiując funkcje na zewnątrz opisu klasy, należy ich nazwy poprzedzić kwalifikacją klasy *TAB<Typ>*::.

```
template <class Typ>
TAB<Typ>::TAB(Typ a[], int N):tmp(0),N(N),A(new Typ[N])
{if(A) for(int i=0; i<N; i++) A[i]=a[i];
}

template <class Typ>
TAB<Typ>::TAB(TAB<Typ> &S):
          tmp(0),N(S.N),A(S.A?new Typ[N]:0)
{if(A) for(int i=0; i<N; i++) A[i]=S.A[i];
 if(S.tmp) delete &S;
}

template <class Typ>
TAB<Typ>::~TAB()
{if(A) delete A;
 A=0;
}

template <class Typ>
TAB<Typ> &TAB<Typ>::operator=(TAB<Typ> &S)
{if(this==&S) return *this;
 if(A) delete A;
 N=S.N;
 A=S.A?new Typ[N]:0;
 if(A) for(int i=0; i<N; i++) A[i]=S.A[i];
 if(S.tmp) delete &S;
 return *this;
}

template <class Typ>
ostream &operator<<(ostream &wy,TAB<Typ> &S)
{for(int i=0; i<S.N; i++) wy<<setw(S.w)<<S.A[i];
 if(S.tmp) delete &S;
 return wy;
}

int TAB<int>::w=4;      // definicja zmiennej statycznej klasy TAB<int>
int a[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
```

```
int TAB<char>::w=1;     // definicja zmiennej statycznej klasy TAB<char>
char *t="Tablica znakow";

int TAB<ZESP>::w=3;     // definicja zmiennej statycznej klasy TAB<ZESP>
ZESP b[]={ZESP(1.5,2.7), ZESP(3.8,4.4), ZESP(5.1,6.2),
          ZESP(7.3,8.4), ZESP(9.5,0.6)};

void main()
{TAB<int> X(a,sizeof(a)/sizeof(a[0]));
 TAB<char> Y(t,strlen(t));
 TAB<ZESP> Z(b,sizeof(b)/sizeof(b[0]));
 cout<<"\nX= "<<X<<"\nY= "<<Y<<"\nZ= "<<Z;
}
```

Funkcja *main* definiuje zmienną *X* typu *TAB<int>*, zmienną *Y* typu *TAB<char>* oraz zmienną *Z* typu *TAB<ZESP>*.

Pytania i zadania

1.6. Dla zdefiniowanej wcześniej klasy *ZESP* zaproponuj funkcję *get* do wprowadzania obiektów swojej klasy.

1.7. W klasie *ZESP* zaproponuj komponent typu całkowitego, który będzie zawierać liczbę zdefiniowanych obiektów tej klasy. Napisz funkcję, której wynikiem będzie ta liczba. W jakich sekcjach klasy należy umieścić deklarację zmiennej całkowitej oraz deklarację funkcji i dlaczego?

1.5. Obiektowe wejście i wyjście

W języku C++ dostępne są operatory wejścia >> i wyjścia <<, których lewym argumentem jest obiekt strumieniowy. Ten sam obiekt jest wynikiem operacji. Aby używać tych operatorów, należy włączyć do programu plik *iostream.h*.

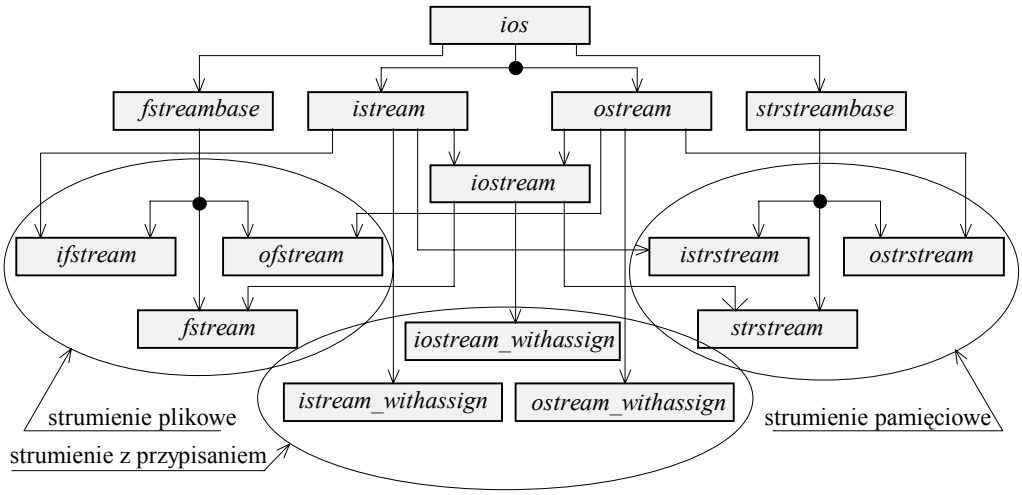
1.5.1. Obiekty strumieniowe

W każdym programie jest predefiniowany strumień wejściowy *cin* oraz strumienie wyjściowe: *cout*, *cerr*, *clog*. Istnieją też możliwości definiowania własnych strumieni

plikowych i pamięciowych. W tym przypadku należy włączyć pliki nagłówkowe *fstream.h* lub *strstrea.h*.

Istnieje wiele klas strumieni. Wszystkie one bazują na klasie *ios*. Bezpośrednio na klasie *ios* bazują klasy: wejściowa *istream*, wyjściowa *ostream*, buforów dyskowych *fstreambase* oraz buforów pamięciowych *strstreambase*. Inne klasy wejściowe bazują na klasie *istream*, natomiast inne klasy wyjściowe bazują na klasie *ostream*. Klasy wejściowo-wyjściowe bazują na klasie *iostream*, a ta bazuje na obu klasach *istream* oraz *ostream*. Klasy plikowe (*ifstream*, *ofstream*, *fstream*) bazują dodatkowo na klasie buforów dyskowych *fstreambase*, natomiast klasy pamięciowe – na klasie buforów pamięciowych *strstreambase*. Strumień *cin* jest klasy *istream_withassign*, strumienie *cout*, *cerr*, *clog* są natomiast klasy *ostream_withassign*. Te klasy mają dodatkowo zdefiniowany operator przypisania. Hierarchię dziedziczenia klas strumieniowych przedstawia schemat na rys. 1.1.

Obiekt strumieniowy można porównać z zerem. Wartość równa zero świadczy o błędnym stanie strumienia (uniemożliwiającym dalszy odczyt lub zapis) wywołanym nieudanym otwarciem pliku lub błędem operacji na strumieniu. Na przykład poprawność odczytu z *cin* można sprawdzić za pomocą instrukcji



Rys. 1.1. Hierarchia klas strumieniowych

```
if(cin==0) cerr<<"Źle wprowadzone dane!\n";
if(!cin) cerr<<"Źle wprowadzone dane!\n";
```

Do bardziej szczegółowego testowania stanu strumieni służą funkcje:

`int eof()` – wystąpił koniec strumienia wejściowego,

`int bad()` – wystąpił błąd zapisu lub odczytu (*failbit*|*badbit*|*hardbit*≠0),
`int fail()` – wystąpił błąd zapisu lub odczytu (*badbit*|*hardbit*≠0),
`int good()` – nie wystąpił żaden błąd,
`int rdstate()` – wynikiem jest wartość zmiennej stanu strumienia,
`void clear(int n=0)` – ustawienie wartości zmiennej stanu strumienia.

Powyższe funkcje testują flagi stanu strumienia. Flagi te zostały zdefiniowane w klasie *ios*

`ios::goodbit = 0x00` – operacje poprawne,
`ios::eofbit = 0x01` – wystąpił koniec strumienia,
`ios::failbit = 0x02` – operacja zakończona niepowodzeniem,
`ios::badbit = 0x04` – nieprawidłowa operacja na strumieniu,
`ios::hardfail = 0x80` – błąd fatalny (np. błąd urządzeniowy).

Funkcje *eof*, *bad*, *fail* dają wynik różny od zera, jeśli wystąpił właściwy dla nich błąd. Funkcja *good* daje wynik różny od zera, gdy nie wystąpił żaden błąd, np.:

```
if(cin.eof()) cerr<<"Koniec pliku!\n";
if(!cin.good()) cerr<<"Błędy odczytu!\n";
```

Aby po wystąpieniu błędu można było kontynuować operacje na strumieniu, należy wyzerować zmienną stanu strumienia.

Do tego celu służy funkcja *clear*, np.

```
cin.clear();
```

1.5.2. Wprowadzanie i wyprowadzanie

Wprowadzanie znaków jest oprogramowane przez funkcje:

`istream& get(unsigned char&)`
`istream& get(signed char&)`
`int get()` – zwraca EOF w przypadku błędu,
`int peek()` – podgląda następny znak do wprowadzenia,
`istream& putback(char)` – zwraca znak do bufora wejściowego.

Funkcja *peek* zwraca EOF na końcu strumienia. Funkcja *putback* daje efekt, gdy w buforze wejściowym jest miejsce na zwracany znak. Na przykład dwa znaki można wczytać do znakowych zmiennych *C*1 i *C*2 instrukcjami:

```
cin.get(C1).get(C2);
```

lub

```
C1=cin.get(); C2=cin.get();
```

Wprowadzanie tekstów bez redagowania jest oprogramowane przez funkcje:

```
istream& get(signed char *p, int n, char d='\n')
istream& get(unsigned char *p, int n, char d='\n')
istream& getline(signed char *p, int n, char d='\n')
istream& getline(unsigned char *p, int n, char d='\n')
istream& read(signed char *p, int n)
istream& read(unsigned char *p, int n)
istream& ignore(int n, int c=EOF)
int gcount()
```

Funkcje *get* i *getline* wprowadzają (włącznie ze znakiem '\0') do bufora wskazanego przez *p* tekst aż do znaku *d* (na ogół bez tego znaku), ale nie więcej niż *n* znaków. Funkcje *get* pozostawiają znak *d* w buforze wejściowym, natomiast funkcje *getline* usuwają ten znak z bufora. Jeśli znak *d* pozostanie w buforze, to kolejne użycie funkcji *get* z tym samym znakiem *d* wprowadzi tekst pusty.

Funkcje *read* wprowadzają *n* bajtów do bufora wskazanego przez *p*. Funkcje te są najczęściej używane do czytania plików binarnych.

Funkcja *ignore* usuwa ze strumienia wejściowego znaki aż do napotkania znaku *c* lub końca pliku, ale nie więcej niż *n* znaków.

Funkcja *gcount* zwraca liczbę znaków wczytanych ze strumienia podczas ostatniej operacji *get*, *getline* lub *read*.

Aby na przykład wczytać jedną linię tekstu do bufora *buf*1, pominąć tekst do końca zdania, wczytać całe zdanie (bez znaku kropki) do *buf*2 i wyznaczyć liczbę znaków (razem z kropką) w tym zdaniu, można użyć instrukcji:

```
cin.get(buf1,128).ignore(256,'.').getline(buf2,256,'.');
n=cin.gcount();
```

Do zmiennej znakowej oraz do bufora znakowego można wprowadzić kolejno znak i wyraz (z redagowaniem podobnie jak formatem %c%s) za pomocą operatora >>, np.:

```
char znak,B[80];
cin >> znak >> B;
```

Wyprowadzanie znaków i tekstów bez redagowania realizują funkcje:

```
ostream& put(char c)
ostream& write(signed char *p, int n)
ostream& write(unsigned char *p, int n)
```

Funkcja *put* wyprowadza znak *c*. Funkcje *write* wyprowadzają *n* znaków (bajtów) z bufora wskazywanego przez *p* i są najczęściej używane do wyprowadzania do plików binarnych.

Prawidłowo zakończone teksty można wyprowadzać (z redagowaniem) za pomocą operatora <<. Na przykład jeśli bufor tekstowy *B* zawiera tekst, to można go wyprowadzić instrukcją

```
cout << "Zawartosc bufora B:\n" << B;
```

Wprowadzanie i wyprowadzanie zredagowane realizuje się za pomocą obiektowych operatorów wejścia >> i wyjścia <<. W zależności od typu prawego argumentu aktywowany jest właściwy operator dokonujący konwersji wejściowego tekstu na wewnętrzną postać binarną lub odwrotnie – z postaci binarnej na tekst wyjściowy.

Przykład wprowadzania i wyprowadzania tekstów

```
char Imie[30],Nazwisko[50];
cerr<<"Podaj imie i nazwisko: ";
cin>>Imie>>Nazwisko;
cout<<"Nazywasz sie "<<Imie<<' '<<Nazwisko<<endl;
```

W powyższych instrukcjach kompilator automatycznie wywoła operatory << oraz >> wyprowadzania i wprowadzania tekstów.

Przykład wprowadzania elementów tablicy liczb rzeczywistych

```
const int Max=100;
int i, N;
double A[Max];
cerr << "Podaj rozmiar tablicy: ";
cin >> N;
if(N>Max) N=Max;
for(i=0; i<N; i++)
  {cerr << "A[" << (i+1) << "]= ";
   cin >> A[i];
  }
```

Podane wyżej instrukcje aktywują dwa różne operatory << oraz dwa różne operatory >>. Kompilator automatycznie wywoła w trzech miejscach operator wyprowadzania tekstu i w jednym miejscu operator wyprowadzania liczby typu *int*. W liniach wprowadzania danych ze strumienia *cin* są wywoływane różne operatory: operator wprowadzania liczby typu *int* oraz operator wprowadzania liczby typu *double*.

W powyższym przykładzie można zabezpieczyć się przed błędem wprowadzenia liczby oraz przed wprowadzeniem niewłaściwego rozmiaru tablicy tak, jak pokazano poniżej

```
cerr << "Podaj rozmiar (N<" << Max << ") tablicy: ";
int x=wherex(), y=wherey();          // zapamiętanie pozycji na ekranie
```

```
while(!(cin>>N) || N<1 || N>=Max)      // wczytanie N i sprawdzenie
  {gotoxy(x, y);
     clreol();
     cin.clear();          // odblokowanie wejścia po ewntualnym błędzie
     cin.ignore(0x7FFF, '\n');
                                  // usunięcie tekstu pozostałego w strumieniu
  }
```

1.5.3. Formatowanie wejścia i wyjścia

Wprowadzane i wyprowadzane liczby oraz teksty można formatować określając dla nich minimalną wielkość pola, precyzję, podstawę systemu i znak wypełnienia. Do tego służą manipulatory sparametryzowane:

setw(int w) – ustawia szerokość pola (wejście i wyjście) na *w* znaków,
setprecision(int p) – ustawia precyzję na *p* znaków,
setbase(int b) – ustawia podstawę systemu na *b* (domyślnie 10),
setfill(int c) – ustawia dopełnianie pola znakiem *c*,
setiosflags(long f) – ustawienie flag formatujących na *f*,
resetiosflags(long f) – zerowanie flag według *f*.

oraz manipulatory bezparametrowe:

dec – ustawienie systemu dziesiętnego,
hex – ustawienie systemu heksagonalnego (o podstawie 16),
oct – ustawienie systemu ósemkowego,
ws – pominięcie spacji wiodących,
endl – wyprowadzenie zawartości bufora i przejścia do nowej linii,
ends – wyprowadzenie znaku zerowego '\0' do bufora,
flush – fizyczne wyprowadzenie zawartości bufora.

Użycie manipulatorów sparametryzowanych wymaga zwykle włączenia pliku nagłówkowego *iomanip.h*. Podane niżej instrukcje

```
cout<<setprecision(2)<<setfill('*')<<"X="<<setw(7)
   <<x<<endl;
cout<<"N="<<n<<"    ("<<oct<<n<<" - osemkowo)\n";
```

gdy x=3.1415 i n=12, wyprowadzą napisy: "X=***3.14" i "N=12 (14 – ósemkowo)". Oba napisy będą zakończone znakiem przejścia do nowej linii. Użycie manipulatora *setw*(*w*) zmienia szerokość pola z 0 na *w* tylko dla jednej operacji wejścia–wyjścia.

Do operowania na flagach formatujących służą następujące funkcje składowe klasy *ios*:

long flags() – zwraca flagę *x_flag*,
long flags(long f) – ustawia flagę *x_flag* = *f* oraz zwraca starą flagę,
long setf(long s) – ustawia w *x_flag* bity, które są ustawione w *s*,
long setf(long s,long f)
– ustawia w *x_flag* bity z *s*&*f* i zeruje bity z ~*s*&*f*,
long unsetf(long s) – zeruje w *x_flag* bity, które są ustawione w *s*.

Dwuargumentowa funkcja *setf*(*s*,*f*) ustawia te bity flagowe, które są ustawione równocześnie w obu jej argumentach *s* oraz *f*. Funkcja ta zeruje też te bity flagowe, które są ustawione w *f* i są jednocześnie wyzerowane w *s*. Inaczej mówiąc, funkcja *setf*(*s*,*f*) przepisuje do *x_flag* te bity argumentu *s*, które są ustawione w *f*.

W klasie *ios* zdefiniowano następujące bity flagi formatującej:

skipws – pominięcie wiodących odstępów przy wprowadzaniu,
left – wyrównanie w polu wyjściowym w lewo,
right – wyrównanie w polu wyjściowym w prawo,
dec – dane (na wejściu i wyjściu) w postaci dziesiętnej,
oct – dane (na wejściu i wyjściu) w postaci ósemkowej,
hex – dane (na wejściu i wyjściu) w postaci szesnastkowej,
showbase – pokazanie bazy systemu (0 lub 0x lub 0X),
showpoint – wymusza wyprowadzenie kropki dziesiętnej i nieznaczących zer liczb rzeczywistych,
uppercase – użycie dużych liter (A–F, E, X) przy wyprowadzaniu liczb,
showpos – wyprowadzanie znaku + przed dodatnimi liczbami,
scientific – naukowa notacja dla liczb rzeczywistych (np. 1.23e+00),
fixed – notacja dla liczb rzeczywistych z kropką dziesiętną.

Na przykład, aby wyprowadzić do *cout* wartość zmiennej rzeczywistej *x* z dokładnością trzech cyfr po kropce z wyprowadzeniem nieznaczących zer, można użyć instrukcji

```
cout.setf(ios::showpoint);
cout<<"X="<<setprecision(3)<<x;
```

Aby wyprowadzić liczbę całkowitą *K* w postaci szesnastkowej z prefiksem 0X oraz z użyciem dużych liter, należy wykonać

```
cout.setf(ios::showbase|ios::uppercase);
cout<<"K="<<hex<<x;
```

Wśród bitów flagi formatującej są trzy zestawy konkurujących bitów: (*left*, *right*), (*dec*, *oct*, *hex*) oraz (*scientific*, *fxed*). W każdym zestawie musi być ustawiony tylko jeden bit. Aby uniknąć sprzecznego ustawienia justowania (np. równocześnie *right* i *left*), bazy liczb całkowitych (np. równoczesnego ustawienia bitów *dec* i *hex*), i prezentacji liczb rzeczywistych, należy zerować konkurujące bity, używając dwuargumentowej funkcji *setf* oraz identyfikatora:

ios::adjustfield	– do ustawienia justowania (wyrównania prawo- lub lewostronnego),
ios::basefield	– do ustawienia bazy liczb całkowitych,
ios::floatfield	– do ustawienia prezentacji liczb rzeczywistych.

Na przykład ustawienie notacji naukowej powinno wyglądać następująco

```
cout.setf(ios::scientific, ios::floatfield);
```

Podany niżej przykładowy program wydrukuje następujące trzy wiersze tekstu:

```
X=***3.14  X=3.14
X=3.14e+00
K=160    K=0XA0    K=0240
```

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

main()
{ double x=3.1415;
  int K=160;
  clrscr();
  cout<<setprecision(2)<<setfill('*')<<"X="<<setw(7);
  cout<<x<<" X="<<x<<endl;
  cout.setf(ios::scientific, ios::floatfield);
  cout<<x<<endl<<"K="<<K;
  cout.setf(ios::left,ios::adjustfield);
  cout.setf(ios::uppercase|ios::showbase);
  cout<<"    K="<<hex<<K<<"    K="<<oct<<K<<endl;
  return(0);
}
```

Aby wyprowadzić liczby typu *double* z tablicy *A* w kolumnie z ustawieniem kropek dziesiętnych w jednej linii pionowej, należy ustawić flagę *showpoint*. W przeciwnym razie nieznaczące pozycje dziesiętne nie będą drukowane. Przykładowe

instrukcje wyprowadzą liczby w układzie tabelarycznym na ośmiu znakach z precyzją trzech cyfr po kropce

```
cout<<setprecision(3);
cout.setf(ios::showpoint);
for(int i=0;i<8;i++) cout<<'['<<setw(8)<<A[i]<<" ]\n";
```

1.5.4. Strumienie plikowe

Na klasie *ios* bazują klasy: *istream*, *ostream* oraz *iostream*, na nich zaś bazują klasy: *ifstream*, *ofstream* oraz *fstream*, przeznaczone kolejno do definiowania strumieni: wejściowych, wyjściowych oraz dwukierunkowych. Konstruktory bezparametrowe tych klas tylko budują swoje obiekty. Konstruktory parametrowe kojarzą budowane obiekty z plikami.

```
ifstream(char *nazwa, int mode=ios::in,
                int prot=filebuf::openprot);
ofstream(char *nazwa, int mode=ios::out,
                int prot=filebuf::openprot);
fstream(char *nazwa, int mode, int prot=filebuf::openprot);
```

W klasie *ios* zdefiniowano następujące trzy podstawowe tryby otwarcia:

`ios::in` – otwarcie do odczytu (bez kreowania),
`ios::out` – otwarcie do zapisu (z kasowaniem zawartości po otwarciu),
`ios::app` – otwarcie do dopisywania (dane dopisuje na końcu),

oraz modyfikacje trybów otwarcia:

`ios::ate` – z ustawieniem bieżącej pozycji strumienia na końcu pliku,
`ios::trunc` – z kasowaniem zawartości po otwarciu,
`ios::nocreate` – bez kreowania (otwierany plik musi istnieć),
`ios::noreplace` – bez modyfikacji (otwierany plik nie może istnieć),
`ios::binary` – w trybie binarnym.

Dla klasy *ifstream* obowiązuje tryb *ios*::*in*, natomiast dla klasy *ofstream* – tryb *ios*::*out*.

Przykłady trybów otwarcia

`ios::in|ios::out` – do zapisu i odczytu z ewentualnym kreowaniem,
`ios::in|ios::app` – jak wyżej, ale ustawieniem pliku na koniec,
`ios::in|ios::out|ios::ate` – jak wyżej,
`ios::out|ios::noreplace` – kreowanie pliku o nowej nazwie.

Destruktory obiektów strumieniowych automatycznie zamykają skojarzone z nimi pliki.

Na rzecz obiektów strumieniowych można wywoływać funkcje *open*(*char* **nazwa*, *int mode*) i *close*( ), aby w sposób jawny otwierać i zamykać pliki.

Przykładowe definicje i otwarcia

```
ifstream
   we1="DANE.TXT", we2("DANE.BIN", ios::in|ios::binary);

ofstream wy1="WYNIKI1", wy2;

fstream wewy("BAZA.DAT", ios::in|ios::out|ios::binary);
wy2.open("WYNIKI.ALL", ios::out | ios::ate);
wy2.open("WYNIKI.ALL", ios::app);        // to samo co wyżej
    . . .
wy2.close();
```

Obiekt *wewy* jest obiektem dwukierunkowym i może służyć zarówno do wprowadzania, jak i do wyprowadzania danych. Obiekt *wy*2 jest budowany przez konstruktor bezparametrowy i musi być jawnie kojarzony z plikiem wyjściowym. Czy otwarcie się udało, można sprawdzić, niezależnie od sposobu definiowania i kojarzenia obiektu z plikiem, dowolną z poniższych metod, np.:

```
if(!we1) cerr << "Nieudane otwarcie !";
if(wy2==0) cerr << "Nieudane otwarcie !";
if(we2.fail()) cerr << "Nieudane otwarcie !";
if(we2.bad()) cerr << "Nieudane otwarcie !";
if(!wy1.good()) cerr << "Nieudane otwarcie !";
```

Wprowadzanie i wyprowadzanie danych odbywa się podobnie jak do/ze strumieni predefiniowanych, na przykład:

```
we1>>x>>y;
wy2<<"\nN="<<n<<"\nX="<<x<<"   Y="<<y<<endl;
wy1<<"N="<<hex<<n<<"(szesnastkowo)\n";
```

Do sterowania strumieniami z klas *istream* i *ostream* służą funkcje:

```
long tellg();
long tellp();
istream& seekg(long n, seek_dir p=ios::beg);
ostream& seekp(long n, seek_dir p=ios::beg);
ostream& flush();
```

Funkcje *tellg* i *tellp* zwracają aktualną pozycję pliku (kolejno *istream* i *ostream*) w bajtach względem początku.

Funkcje *seekg* i *seekp* ustawiają plik w zadanej pozycji *n* bajtów od początku, końca lub aktualnej pozycji zależnie od parametru *p*, który może przyjmować wartości: *ios*::*beg*, *ios*::*end*, *ios*::*cur*.

Funkcja *flush* wyprowadza fizycznie zawartość bufora do pliku wyjściowego.

Na przykład, następujące instrukcje przewijają pliki (patrz definicje obiektów *we*1, *wy*2 i *we*2) "DANE.TXT", "WYNIKI.ALL" oraz "DANE.BIN":

```
we1.seekg(0L);                  – na początek,
wy2.seekp(0L, ios::end);        – na koniec,
wy2.seekp(-16L, ios::end);      – 16 bajtów przed końcem,
we2.seekg(24L, ios::cur);       – 24 bajty za pozycją aktualną.
```

W strumieniach wejściowo-wyjściowych można używać zarówno funkcji *seekg* i *telg*, jak również *seekp* i *telp*, pamiętając, że odnoszą się one do tej samej pozycji strumienia.

Do zapisu i odczytu binarnego służą funkcje:

```
istream& read(char *bufor, int n);
ostream& write(char *bufor, int n);
```

Na przykład odczytu 12 liczb typu *double* z pliku "DANE.BIN" do tablicy *A* można dokonać instrukcją

```
we2.read((char*)A, 12*sizeof(*A));
```

1.5.5. Strumienie pamięciowe

Na strumieniach pamięciowych można wykonywać te same operacje co na strumieniach plikowych, tyle że informacja jest przesyłana do lub z pamięci. Do posługiwania się strumieniami pamięciowymi utworzono klasy *istrstream*, *ostrstream* oraz *strstream*.

Co do wejściowych strumieni klasy *istrstream* zakłada się, że zawierają one dane zapisane w buforze. W klasie *istrstream* są zdefiniowane dwa konstruktory:

```
istrstream(char *buf);
istrstream(char *buf, int len);
```

Konstruktor jednoargumentowy sam określa wielkość bufora na podstawie długości łańcucha znakowego. Wskazany bufor musi zawierać zatem tekst zakończony ogranicznikiem '\0'.

Konstruktor dwuargumentowy ma podaną w drugim argumencie długość bufora, który może zawierać zarówno dane tekstowe, jak i binarne.

Wyjściowe strumienie pamięciowe klasy *ostrstream* mogą być tworzone przez konstruktory:

```
ostrstream();
ostrstream(char *buf, int len, int mode=ios::out);
```

Konstruktor bezparametrowy tworzy strumień z buforem dynamicznym. Jeśli wynikowy łańcuch nie mieści się w buforze, to sam obiekt dokonuje realokacji bufora do wymaganej wielkości.

Konstruktor trójparametrowy tworzy strumień z podanym statycznym buforem o ustalonej i podanej w drugim parametrze długości.

Tekst wyprowadzany do bufora nie jest kończony znakiem '\0'. Znacznik końca tekstu należy zawsze dopisać w sposób jawny, używając manipulatora *ends*, np.

```
ostrstream S;
S << "X =" << x << endl << "Koniec" << ends;
```

W klasie *ostrstream* zdefiniowane są funkcje:

`char *str();` – wskaźnik do pierwszego znaku w buforze wyjściowym,
`int pcount();` – liczba bajtów między bieżącą pozycją a końcem pliku.

Na przykład instrukcja

```
cout << S.str();
```

wyprowadzi do *cout* tekst umieszczony w strumieniu *S*.

Użycie funkcji *str* zamraża bufor (przestaje on być relokowalny). Aby odmrozić bufor, trzeba użyć funkcji *freeze* z klasy *strstreambuf*, np.:

```
S.rdbuf()->freeze(0);
```

gdzie wyrażenie *S.rdbuf*( ) daje wskaźnik do bufora strumienia *S*.

Funkcja *pcount* nie potrzebuje do działania znacznika końca tekstu, tak więc przykładowa instrukcja

```
liczba_znakow=S.pcount();
```

działa zawsze poprawnie, nawet gdy bufor strumienia *S* zawiera dane binarne.

Strumienie wejściowo-wyjściowe klasy *strstream* mogą być tworzone konstruktorami:

```
strstream();
strstream(char *buf, int len, int mode);
```

Podobnie jak w klasie *ostrstream* konstruktor bezparametrowy tworzy strumień z buforem dynamicznym, który jest zamrażany użyciem funkcji *str* i może być odmrożony funkcją *freeze* wywołaną z zerowym argumentem.

W klasie *strstream* nie ma funkcji *pcount*. Aby zatem uzyskać liczbę znaków wpisanych do bufora strumienia *Swewy*, należy użyć funkcji *out_waiting* zdefiniowanej w klasie *strstreambuf*, np.

```
liczba_znakow=Swewy.rdbuf()->out_waiting();
```

1.5.6. Strumienie ekranowe

W nieobiektowym języku C bezpośrednie wyjście na ekran realizują funkcje opisane w pliku *conio.h*. W wersji 3.1 kompilatora Borland C++ zaimplementowano obiektowe strumienie bezpośrednio związane z ekranem. Strumienie te są obiektami klasy *constream*, zdefiniowanej w pliku *constrea.h*. Metody klasy *constream* używają funkcji opisanych w *conio.h*, tak więc plik *conio.h* jest automatycznie dołączany do pliku *constrea.h*.

Można zdefiniować równocześnie kilka strumieni ekranowych i powiązać każdy strumień z innym oknem na ekranie. Klasa *constream* bazuje na klasie *ostream*. Na strumieniach ekranowych można zatem wykonywać wszystkie operacje dozwolone dla strumieni klasy *ostream* oraz operacje ekranowe (funkcje i manipulatory) zdefiniowane w klasie *constream*.

Definicja klasy *constream* zawarta w pliku *constrea.h*:

```
class constream : public ostream {
public:
  constream();
  conbuf* rdbuf();                  // pobranie wskaźnika do bufora
  void clrscr();                    // czyszczenie ekranu
  void window(int,int,int,int);     // definicja okna
  void textmode(int);               // zmiana trybu tekstowego
  static int isCon(ostream&);       // test konsoli
private:
  static long isCon_;
  conbuf buf;
};
```

Funkcje klasy *constream* działają tak jak funkcje *conio.h* o analogicznych nazwach.

W klasie *constream* zdefiniowane są następujące manipulatory bezargumentowe:

```
ostream& clreol(ostream&);
ostream& highvideo(ostream&);
ostream& lowvideo(ostream&);
ostream& normvideo(ostream&);
```

```
ostream& delline(ostream&);
ostream& insline(ostream&);
```

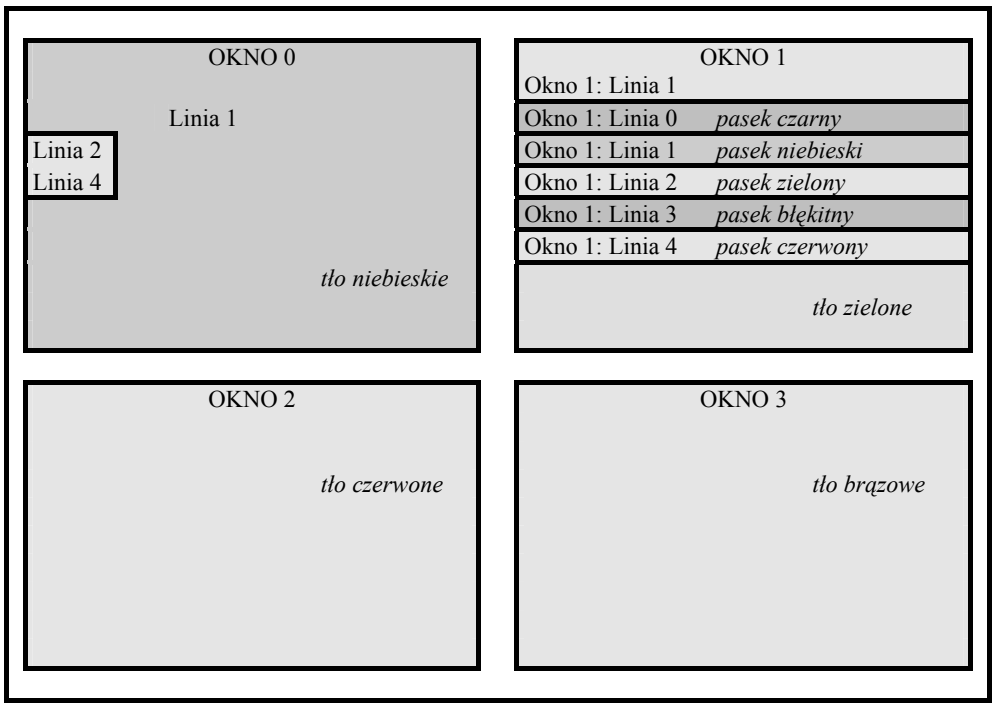
oraz manipulatory jedno- i dwuargumentowe:

```
omanip_int setcrsrtype(int);      // typ 0 ukrywa kursor
omanip_int setattr(int);          // ustawienie bajtu atrybutu
omanip_int setbk(int);            // ustawienie koloru tła
omanip_int setclr(int);           // ustawienie koloru tuszu
omanip_int_int setxy(int,int);    // ustawienie pozycji kursora
```

Przykładowe użycie strumieni obiektowych ilustruje poniższy program.

```
#include <constrea.h>

main()
{ constream cc,c[4];              // definicja pięciu strumieni ekranowych
  cc<<setbk(BLACK);               // textbackground(BLACK);
  cc.clrscr();                    // czyszczenie całego ekranu
  c[0].window(2,2,38,12);         // definicje okien tekstowych
  c[1].window(42,2,80,12);
  c[2].window(2,14,38,24);
  c[3].window(42,14,80,24);
  c[0]<<setbk(BLUE);              // definicje tła w oknach
  c[1]<<setbk(GREEN);
  c[2]<<setbk(RED);
  c[3]<<setbk(BROWN);
  for(int i=0;i<4;i++)            // pokazanie czterech prostokątów z nazwami
     {c[i].clrscr();
      c[i]<<setxy(16,1)<<"OKNO "<<i<<endl;
     }
  c[0]<<setxy(10,3)<<setclr(YELLOW)<<"Linia 1"<<endl<<
              setattr(0x70) <<"Linia 2";
  c[1]<<setclr(WHITE)<<"Okno 1: Linia 1"<<setbk(RED);
  c[0]<<endl<<"Linia 4"<<endl;
  for(i=0;i<5;i++)                // pięć kolorowych pasków w oknie nr 1
   c[1]<<setbk(i)<<endl<<"Okno 1: Linia "<<i<<clreol;
  getch();
  cc.clrscr();                    // czyszczenie całego ekranu
  return 0;
}
```

W rezultacie na czarnym tle ekranu powstaną cztery izolowane kolorowe prostokąty, nazwane oknami (od OKNO 0 do OKNO 3). W górnych oknach będą widoczne napisy na kolorowych tłach, jak pokazano schematycznie na rys. 1.2.



Rys. 1.2. Wynik działania przykładowego programu

Pytania i zadania

1.8. Data jest pamiętana w trzech zmiennych całkowitych *D*, *M*, *R*, oznaczających dzień, miesiąc i rok. Wyprowadź tę datę w postaci *dd.mm.rrrr*.

1.9. Tablice *X* i *Y* zawierają liczby rzeczywiste. Wyprowadź te liczby parami w dwu kolumnach z precyzją trzech cyfr. Liczby z tabeli *X* powinny być wyprowadzone w postaci zwykłej, natomiast liczby z tabeli *Y* w postaci naukowej (scientific). Wartości X_i są z przedziału od – 100 do 100.

1.10. Napisz instrukcje deklaracyjne, które otworzą:
   a) plik tekstowy w celu dopisania do niego kolejnych wyników,
   b) istniejący plik binarny do odczytu,
   c) plik binarny do edycji z ewentualnym kreowaniem, jeśli nie istnieje.

1.11. Napisz instrukcje, które otworzą plik tekstowy i wyprowadzą zawartość jego 20 linii na ekran: a) w całości, b) opuszczając w liniach znaki wystające poza prawy brzeg ekranu.

1.12. Plik binarny zawiera liczby typu *double*. Napisz instrukcje, które otworzą ten plik i zamienią w nim liczby ujemne na dodatnie, pozostawiając pozostałe liczby bez zmian.

1.13. Graficzna funkcja *outtext* wykreśla teksty. Jeśli tekst ma zawierać wartości zmiennych, to należy go wcześniej sformatować. Napisz instrukcje, które użyją strumienia pamięciowego do przygotowania tekstu z wartościami zmiennych x, y według wzorca "X = 123.12 Y = 123.12" oraz wykreślą ten tekst. Przyjmij, że system jest już w trybie graficznym, a kursor jest we właściwej pozycji.

1.6. Podejście obiektowe

Istota programowania obiektowego polega na tym, że w miejsce rozłącznie traktowanych funkcji i danych wprowadza się pojęcie obiektów, w których dane i funkcje ich przetwarzania są wzajemnie zintegrowane. Podejście obiektowe umożliwia programowanie w kategoriach problemu, które za pomocą abstrahowania tworzą klasy i ich obiekty, a następnie przedstawiają algorytmy, posługując się tymi obiektami i arytmetyką stworzoną dla nich.

Jeżeli na przykład rozwiązywane problemy opisuje się metodami algebry liniowej, to wygodniej jest posługiwać się takimi obiektami, jak macierze, wektory wierszowe i wektory kolumnowe niż elementami tych obiektów. Jeśli y jest iloczynem macierzy A i wektora x, co notuje się jako $y = A\,x$, to w programie obiektowym można to zapisać w bardzo podobnej postaci $y=A*x$;, podczas gdy w programie proceduralnym należałoby wywołać funkcję z pięcioma argumentami (x, y oraz A z liczbą wierszy i kolumn).

Najważniejsze mechanizmy programowania obiektowego to:
- **hermetyzacja** danych i metod,
- **dziedziczenie**,
- **przeciążanie** funkcji i operatorów,
- **polimorfizm**.

1.6.1. Hermetyzacja danych i metod

Hermetyzacja danych i metod polega na ukryciu wewnętrznej struktury obiektów i wewnętrznych metod przetwarzania w sekcji prywatnej oraz w sekcji zabezpieczonej.

Na przykład w klasie liczb zespolonych (*ZESP*) ukryta jest reprezentacja (*Re*, *Im*) liczby zespolonej i zakazany jest dostęp do pól *Re* oraz *Im*. Programista korzystający z liczb zespolonych nie musi wiedzieć, czy liczby te są pamiętane jako część rzeczywista i urojona, czy jako moduł i argument. Programista ten nie wie, jak jest zrealizowane dodawanie liczb zespolonych, ale potrafi (mając funkcję *operator+*) je dodawać. Nie może więc błędnie wykonać dodawania czy wyprowadzenia liczb zespolonych. Nie mając dostępu do pól *Re* i *Im*, programista nie może uszkodzić pamiętanej liczby. Wektory i macierze powinny same wykrywać niezgodność swoich wymiarów (np. w dodawaniu czy mnożeniu) i powinny same umieć uporać się z tymi i podobnymi problemami. Przesuwane obiekty graficzne powinny same dbać o odświeżanie odsłanianych fragmentów ekranu.

O ile to możliwe, użytkownik klasy (programista) powinien operować na całych obiektach (np. wektorach, macierzach) bez dostępu do ich wnętrza (np. do elementów wektora czy macierzy). Zauważmy, że przykładowy użytkownik, nie mając dostępu do elementów wektora, nie jest w stanie odwołać się do nieistniejącego elementu (przekroczyć zakres indeksacji). W programowaniu proceduralnym natomiast taka możliwość jest częstą przyczyną błędnych wyników obliczeń lub błędów wykonania.

Dostęp do komponentów obiektu powinien zawsze odbywać się za pośrednictwem funkcji. Funkcje te powinny gwarantować, że żadne nielegalne operacje na obiekcie nie zostaną wykonane.

Hermetyzacja zwiększa uniwersalność i niezawodność oprogramowania.

1.6.2. Dziedziczenie

Dziedziczenie umożliwia tworzenie nowych klas na bazie klas już opracowanych bez wnikania do wnętrza klas bazowych, przejmując pożądane cechy tych klas.

Przykłady

Można utworzyć klasę wektorów trójwymiarowych bazując na klasie wektorów dwuwymiarowych. Operacje na trójwymiarowych wektorach (obiektach klasy) można

oprogramować, wykorzystując operacje wykonywane na płaszczyźnie i obliczając trzecią składową.

Klasę macierzy można utworzyć, bazując na klasie wektorów, traktując macierz jako wektor złożony z jej kolejnych wierszy. W algorytmach wprowadzania i wyprowadzania macierzy można przejąć metody wprowadzania i wyprowadzania wektorów itp.

Można też na przykład utworzyć klasę okien wypełnionych komunikatami, bazując na klasie okna pustego, przejmując z tej klasy między innymi mechanizmy zapamiętywania i odtwarzania ekranu oraz mechanizmy przesuwania i zmiany wymiarów okna.

Jeżeli obiekt graficzny będzie bazować na klasie opisującej punkt, to na przykład do przesuwania tego obiektu na ekranie można użyć dziedziczone metody przesuwania punktu, jak pokazano poniżej.

```
class PUNKT {
protected:
   double x, y;
public:
   PUNKT(double x=0, double y=0): x(x), y(y)
      { }
   void przesun(double dx, double dy)
      {x+=dx; y+=dy;}
   . . .
};

class KOLO: public PUNKT {
   double r;
public:
   KOLO(double x=0, double y=0, double r=1):
        PUNKT(x,y),r(r) {}
   . . .
};

main()
{ KOLO K(2, 3);              // definiuje x=2, y=3, r=1
  K.przesun(5, 10);          // zmienia na x=7, y=13, r=1
  . . .
}
```

Zauważmy, że funkcję *przesun* z klasy *PUNKT* aktywowano na rzecz obiektu klasy *KOLO*. Uzyskano przesunięcie punktu – środka koła, a tym samym przesunięcie koła. Klasa *KOLO*, dziedzicząc klasę *PUNKT*, przejęła nie tylko pola *x*, *y*, ale też

i metodę *przesun*. Gdyby dziedziczenie zastąpić polem obiektowym, na przykład zmieniając definicję klasy na

```
class KOLO {
   PUNKT s;
   double r;
public:
   . . .
};
```

to w klasie *KOLO* należałoby zdefiniować jej własną funkcję *przesun*. Należy więc podkreślić, że dziedziczenie to nie tylko przejęcie pól klasy bazowej, ale przede wszystkim przejęcie jej właściwości, czyli metod przetwarzania.

1.6.3. Przeciążanie funkcji i operatorów

Przeciążanie umożliwia stosowanie tradycyjnych jednakowych nazw wielu funkcji i jednakowych nazw wielu operatorów opracowanych do przetwarzania różnych obiektów.

Przykłady

Ten sam znak operatora dodawania (+) można stosować do dodawania liczb rzeczywistych, liczb zespolonych, wektorów dwu- i trójwymiarowych, okienek na ekranie oraz do innych obiektów.

Operator mnożenia może oznaczać nie tylko mnożenie liczb, ale i mnożenie wektorów, macierzy oraz macierzy przez wektor lub przez liczbę. Ten sam operator może też oznaczać powiększenie obiektu graficznego itp.

Podobnie można jednakowo nazwać funkcje obliczania modułu liczb rzeczywistych, zespolonych, długości wektorów itp.

W zależności od tego, jaki obiekt ma być przetwarzany, czyli w zależności od typów argumentów operatora lub funkcji ze zbioru przeciążonych funkcji wywoływana jest ta, która jest najbardziej odpowiednia. Mechanizmy przeciążania pozwalają więc na automatyczne dopasowywanie metod przetwarzania do przetwarzanych danych. Jeśli np. w wyrażeniu *A+B* dodawane identyfikatory są z klasy liczb zespolonych (*ZESP*), to będzie wykonana operacja dodawania zdefiniowana w tej klasie, a nie np. dodawanie wektorów lub liczb rzeczywistych.

1.6.4. Polimorfizm

Polimorfizm pozwala na automatyczny wybór funkcji zależnie od aktualnych, nie zaś od formalnych właściwości obiektów, na rzecz których funkcje te są aktywowane.

Nie potrzeba więc rozstrzygać na etapie pisania programu ani na etapie jego kompilacji, jaka funkcja ma być w danym miejscu wykonana. Wybór funkcji dokonywany jest dopiero podczas wykonywania się programu.

Na przykład instrukcja *p–>dlugosc*( ); aktywująca polimorficzną funkcję *dlugosc* na rzecz wskazywanego przez *p* obiektu aktywuje funkcję z tej klasy (np. wektorów dwu- lub trójwymiarowych), do której należy wskazywany obiekt niezależnie od typu zmiennej wskazującej *p*. Zadeklarowanie zmiennej *p* jako wskaźnika na obiekty określonej klasy nie przesądza zatem o wyborze funkcji *dlugosc* z tej właśnie klasy. O wyborze funkcji decyduje w chwili wykonywania się programu to, jakiej klasy obiekt jest rzeczywiście wskazywany przez zmienną *p*.

Niech na przykład w klasach *VECT* i *VECT*3*D* będą zdefiniowane funkcje *dlugosc* i niech klasa *VECT* będzie klasą bazową klasy *VECT*3*D*.

```
class VECT {
protected:
  double x, y;
public:
  . . .
  virtual double dlugosc() {return sqrt(x*x+y*y);}
};

class VECT3D: public VECT {
  double z;
public:
  . . .
  double dlugosc() {return sqrt(x*x+y*y+z*z);}
};

main()
{ VECT3D K(3, 4, 5);
  VECT *p;
  double dl;
  . . .
  p=&K;              // wskaźnik typu VECT* wskazuje obiekt klasy VECT3D
  dl=p->dlugosc();                          // dl= 7.071068, a nie 5.0
```

```
  . . .
}
```

Gdyby w definicji funkcji *dlugosc* w klasie *VECT* opuścić słowo *virtual*, funkcja ta nie byłaby funkcją polimorficzną. W tej sytuacji instrukcja *dl=p–>dlugosc*( ); aktywowałaby funkcję *dlugosc* z klasy *VECT*, ponieważ *p* jest typu *VECT** i zmienna *dl* otrzymałaby nieprawidłową wartość *dl*=5. Ponieważ funkcja *dlugosc* została zadeklarowana jako polimorficzna (wirtualna), instrukcja *dl=p–>dlugosc*( ); aktywuje (też polimorficzną) funkcję *dlugosc* z klasy *VECT*3*D*, ponieważ *p* faktycznie wskazuje na obiekt klasy *VECT*3*D*. Teraz wynik będzie prawidłowy *dl*= 7,071068.

Funkcja polimorficzna niejawnie deklaruje się w każdej klasie pochodnej. Jeśli jakaś funkcja klasy bazowej używa funkcji wirtualnych, to nie wiadomo, z jakiej klasy funkcje te będą aktywowane. Zostaje to rozstrzygnięte dopiero podczas wykonywania się programu. Być może zostaną aktywowane funkcje klas, które będą zdefiniowane dopiero w przyszłości. Tak więc polimorfizm umożliwia budowanie funkcji, których szczegóły algorytmu nie są znane na etapie oprogramowywania klas bazowych.

Tworząc na przykład klasę wektorów dwuwymiarowych na bazie pewnej klasy, można już w tej klasie bazowej zdefiniować funkcję wprowadzania wektorów bez precyzowania, o jaką liczbę wymiarów chodzi. Ta funkcja wykonywałaby tylko to, co nie zależy od liczby wymiarów. Pozostałe czynności można dla niej definiować w każdej klasie oddzielnie. Na przykład funkcja operatorowa

```
istream &operator>>(istream &we, klasa_bazowa &w)
  { if(w.A) delete w.A;
    w.A=NULL;
    if(we==cin) cerr<<"Rozmiar wektora: ");
    we >> w.N;
    if(we==cin) cerr<<"Wspolrzedne wektora:\n");
    w.getwsp(we); // wirtualna funkcja z klasy obiektu podstawionego pod w
    return we;
  }
```

usuwa niepotrzebny bufor *A*, warunkowo wyprowadza odpowiednie napisy, gdy wejście jest skojarzone z klawiaturą, wprowadza rozmiar wektora i zwraca lewy argument. Sposoby alokacji bufora i wprowadzania współrzędnych wektora nie są znane. Zależą one od typu (klasy) samych współrzędnych i zostaną zdefiniowane w przyszłości. Zauważmy, że funkcja *operator>>* została napisana do wprowadzania obiektów klasy bazowej i klas pochodnych, które będą zdefiniowane w przyszłości. W każdej z tych klas należy zdefiniować odpowiednią funkcję wirtualną *getwsp*(*istream&*), która tylko zaalokuje bufor i wprowadzi do niego współrzędne wektora.

Pytania i zadania

1.14. Funkcja wymierna jest ułamkiem, którego licznik i mianownik są wielomianami. Na przykładzie klasy funkcji wymiernych wyjaśnij pojęcia hermetyzacja, dziedziczenie oraz funkcja polimorficzna.

1.15. Algebra zbiorów nie zależy od tego, jaki sens mają elementy zbioru. Można tworzyć klasy zbiorów liczb, zbiorów nazw itp. Można też utworzyć klasę, która nie ma zdefiniowanych elementów, ale ma zdefiniowany sposób określania, czy dany element należy do zbioru, czy nie należy oraz ma zdefiniowane działania na zbiorach. Na przykładzie takich klas wyjaśnij pojęcia hermetyzacji, dziedziczenia oraz polimorfizmu.

Zadania laboratoryjne

1.16. Napisz programy, które przetestują rozwiązania zadań:
a) 1.8 i 1.9, b) 1.10 i 1.11, c) 1.12, d) 1.13.

1.17. Przeprowadź kompilację programu z oprogramowaną klasą z punktu 1.2.4 zapisanego w trzech plikach.

1.18. Używając programu *tlib* utwórz bibliotekę z funkcjami do programu z punktu 1.2.4. Skompiluj do postaci wykonywalnej plik *prog.cpp*, dołączając użyte funkcje z utworzonej biblioteki.

2. Konstruktory i destruktory

Konstruktorem jest funkcja o nazwie klasy (*Klasa*), bez typu (nawet *void*), nie zwracająca wartości. Jeśli nie został zdefiniowany żaden konstruktor, to niejawnie jest zdefiniowany konstruktor bezparametrowy *Klasa*( ) i konstruktor kopiujący *Klasa*(*Klasa*&).

Konstruktor jest niejawnie wywoływany zawsze wtedy, gdy tworzony jest obiekt jego klasy:

- w definicjach zmiennych obiektowych,
- podczas alokacji obiektów za pomocą operatora *new*,
- podczas przekazania obiektowego argumentu i wyniku funkcji przez wartość.

Zadaniem konstruktora jest zorganizowanie tworzonego obiektu. Domyślny konstruktor jedynie przydziela pamięć na obiekt, co w wielu zastosowaniach nie wystarcza. Często konstruktory nadają polom obiektu wartości początkowe i alokują bufory pamięciowe potrzebne obiektowi.

Przykłady

Tworząc obiekty klasy *ZESP* można nadawać wartości początkowe ich polom *Re* i *Im*.

```
ZESP(double re, double im} {Re=re; Im=im;}
```

Przykładowa klasa *TEXT* ma dwa pola całkowite: długość tekstu i znacznik tymczasowości oraz jedno pole typu wskaźnik do tekstu. Początek definicji tej klasy może być następujący

```
class TEXT {
  private:
    int len, tmp;       // długość tekstu i znacznik tymczasowości
    char *txt;          // wskaźnik na tekst
    . . .
```

Konstruktor tworzący obiekt z podanym tekstem powinien zaalokować bufor na ten tekst, przepisać go do tego bufora. Długość tekstu powinna zostać zapisana do pola całkowitego *len*. Pole *tmp* powinno być różne od zera dla obiektu tymczasowego. Konstruktor klasy *TEXT* będzie więc bardziej rozbudowany, na przykład

```
TEXT(char *t)
  { len=strlen(t);          // zapamiętanie długości tekstu
    tmp=0;                  // wyzerowanie znacznika tymczasowości
    txt=new char[len+1];    // alokacja bufora na tekst
    strcpy(txt, t);         // przepisanie tekstu do bufora obiektu
  }
```

2.1. Konstruktor bezparametrowy

Konstruktor bezparametrowy ma nagłówek

```
Klasa();
```

Konstruktor ten ma za zadanie stworzyć obiekt, który nie jest inicjowany żadnymi wartościami.

Konstruktor bezparametrowy jest zawsze aktywowany, gdy jest definiowany obiekt bez inicjacji, co zwykle ma miejsce, gdy definiowane są zmienne klasy *Klasa* oraz gdy alokowana jest pamięć na obiekty tej klasy za pomocą operatora *new*.

Na przykład poniższe instrukcje definiują obiekty aktywując konstruktory bezparametrowe:

```
ZESP A, B[10], *p=new ZESP, *q=new ZESP[n];
TEXT TA, TB[20], *pt=new[n];
```

Jeżeli został zdefiniowany dowolny inny konstruktor, to kompilator nie dołączy własnej definicji konstruktora bezparametrowego.

Domyślny konstruktor bezparametrowy ma definicyjną postać

```
Klasa() { }
```

W przykładowych klasach *ZESP* i *TEXT* konstruktory zerują pola *Re* i *Im* oraz *tmp*, *len* i *txt*. Konstruktor

```
ZESP() { Re=Im=0.0; }
```

tworzy obiekt o zerowych polach, natomiast konstruktor

```
TEXT() { tmp=len=0; txt=NULL; }
```

tworzy obiekt bez tekstu i bez bufora.

Pytania i zadania

2.1. Napisz taki konstruktor bezparametrowy dla klasy *TEXT*, który utworzy obiekt z pustym tekstem w jednobajtowym buforze.

2.2. Obiekt klasy *RECT* zawiera ekranowe współrzędne lewego górnego i prawego dolnego rogu prostokąta na ekranie (tak jak w funkcji *window*) oraz wskaźnik do bufora na zawartość tej części ekranu, którą prostokąt ten może przykryć. Zaproponuj konstruktor bezparametrowy, który utworzy obiekt:
a) pusty, b) wielkości całego ekranu, c) wielkości jednego znaku na pozycji kursora.

2.2. Konstruktor kopiujący

Konstruktor kopiujący ma nagłówek

```
Klasa(Klasa&);
Klasa(const Klasa&);
```

Konstruktor kopiujący ma za zadanie stworzyć obiekt i zainicjować go innym obiektem tej samej klasy.

Konstruktor kopiujący jest aktywowany podczas przekazywania obiektów przez wartość. Jeśli argument funkcji jest przekazywany przez wartość (a nie przez referencję), to wewnątrz funkcji tworzony jest lokalny obiekt za pomocą konstruktora kopiującego. Konstruktor ten jest też aktywowany, gdy definiowana zmienna obiektowa jest inicjowana innym obiektem tej samej klasy.

Na przykład poniższe instrukcje definiują obiekty aktywując konstruktory kopiujące, jeśli *X*, *Y* są obiektami klasy *ZESP*, a *T* jest obiektem klasy *TEXT*:

```
ZESP A(X), B=X, *p=new ZESP(X), C[]={X, Y};
TEXT TA(T), TB=T;
```

Jeśli konstruktor kopiujący nie został zdefiniowany, to kompilator niejawnie dołącza własną definicję tego konstruktora.

Domyślny konstruktor kopiujący kopiuje do tworzonego obiektu pole po polu. Jeśli kopiowane pole jest typu obiektowego, to jest ono tworzone konstruktorem kopiującym swojej klasy.

Przykłady konstruktorów kopiujących w klasach *ZESP* i *TEXT*

```
ZESP(ZESP &z)
  { Re=z.Re; Im=z.Im; }

TEXT(TEXT &s)
  { tmp=0;
    if(s.txt)
    { len=s.len;
      txt=new char[len+1];   // alokacja bufora na tekst
      strcpy(txt,s.txt);     // przepisanie tekstu do bufora obiektu
    } else {txt=NULL; len=0;}
    if(s.tmp) delete &s;     // usunięcie obiektu tymczasowego
  }
```

Pytania i zadania

2.3. Uprość konstruktor kopiujący z wyżej podanego przykładu zakładając, że w klasie *TEXT* zdefiniowano konstruktor bezparametrowy jak w zad. 2.1 i każdy obiekt posiada swój bufor.

2.4. Napisz konstruktor kopiujący dla klasy opisanej w zad. 2.2.

2.5. Wyjaśnij, dlaczego konstruktory kopiujące nie kopiują zawartości pól statycznych.

2.3. Konwersja konstruktorowa

Konstruktor definiujący konwersję ma nagłówek

```
Klasa(typ);
```

Konstruktor ten definiuje konwersję z typu *typ* do typu *Klasa*. Na przykład w klasie *ZESP* można zdefiniować konwersję z typu *double* (do typu *ZESP*)

```
ZESP(double re) {Re=re; Im=0;}
```

W klasie *TEXT* można na przykład zdefiniować konwersję z typu *char**

```
TEXT(char *t)
```

```
{ tmp=0;
  if(t)
    {len=strlen(t);
       txt=new char[len+1];  // alokacja bufora na tekst
       strcpy(txt, t);       // przepisanie tekstu do bufora obiektu
    } else {txt=NULL; len=0;}
}
```

Jeśli w pewnym miejscu programu występuje typ *typ*, a wymagany jest typ *Klasa*, to obiekt tej klasy jest automatycznie tworzony za pomocą konwersji konstruktorowej. Jeśli na przykład została zdefiniowana funkcja operatora dodawania *Klasa* + *Klasa*, a w programie występuje wyrażenie *Klasa* + *typ*, to drugi argument typu *typ* zostanie poddany konwersji konstruktorowej (jeśli taka została zdefiniowana) do typu *Klasa*.

W poniższych przykładach zostaną aktywowane konwersje konstruktorowe. Tworzą one obiekty klasy *ZESP* inicjowane jedną liczbą rzeczywistą oraz obiekty klasy *TEXT* inicjowane tekstem.

```
ZESP A=5, B(4), *p=new ZESP(7);
B=A+12.5;
B=ZESP(2.71);                 // to samo co B=2.71;
TEXT TA="Borland", TB("programowanie");
TA+=" C++";
```

Na przykładzie dodawania *A* + 12,5 zauważmy, że jeśli jest zdefiniowane dodawanie dwu obiektów klasy *ZESP*, to aby dodać do obiektu *A* liczbę 12,5 należy najpierw dokonać konwersji tej liczby na obiekt klasy *ZESP*. Konwersja ta dokonywana jest automatycznie. Tak więc faktycznie wykonywane jest

```
B=A+ZESP(12.5);       // tożsame z B=A.operator+(ZESP(12.5));
```

Liczba 12,5 jest konwertowana do typu *ZESP*, ponieważ funkcja *operator*+ aktywowana na rzecz obiektu *A* wymaga argumentu typu *ZESP*.

Zauważmy, że w wyrażeniu 12,5 + *A* funkcja *operator* + jest aktywowana na rzecz liczby typu *double*, podczas gdy taka funkcja wymaga argumentu (prawego) typu *double*. Tak więc w tym wyrażeniu obiekt *A* musi być poddany konwersji do typu *double*, a jest to możliwe, jeśli konwersja ta zostanie zdefiniowana. Wynikiem wyrażenia 12,5 + *A* będzie więc liczba typu *double*.

Pytania i zadania

2.6. Zaproponuj konwersję konstruktorową:
a) z typu *double* do klasy *ZESP*, tworzącą obiekt o jednakowych *Re* i *Im*,

b) z typu znakowego *char* do klasy *TEXT*,
c) z typu całkowitego *int* do klasy *TEXT*.

2.7. Zaproponuj dla klasy *RECT* opisanej w zad. 2.2 konwersję konstruktorową z typu: a) *int*, b) *ZESP*, c) *TEXT*.

2.4. Konstruktory wieloargumentowe

Poza konstruktorami wymienionymi w każdej klasie można definiować konstruktory dwu- i więcej argumentowe. Zadaniem tych konstruktorów jest tworzenie obiektów opierając się na większej liczbie danych.

Konstruktor wieloargumentowy jest aktywowany, gdy tworzony obiekt jest inicjowany wieloma parametrami. Podobnie jak przy jednym argumencie, definiując zmienną typu obiektowego, można wymienić za nią w nawiasach okrągłych argumenty konstruktora.

Przykłady aktywowania konstruktorów

```
ZESP A(-10, 3.14), *p=new ZESP(7.5, 2);
A=ZESP(35, 1.5);
```

Przykłady różnych konstruktorów
W klasach *ZESP* i *VECT* konstruktory

```
ZESP(double=0, double=0);
VECT(double=0, double=0);
```

są dzięki zastosowaniu argumentów domniemanych równocześnie konstruktorami bezparametrowymi i definiującymi konwersje z typu *double* do typu *ZESP* i typu *VECT*.

Poza tworzeniem obiektów konstruktory mogą wykonywać inne zadania (np. nadawać wartości polom klasy, wyprowadzać komunikaty itp.). Ciało funkcji konstruktora jest wykonywane po utworzeniu obiektu.

Na przykład w przypadku klasy *ZESP* konstruktor powinien zainicjować wartościami pola *Re* i *Im*

```
ZESP::ZESP(double re, double im)
   { Re=re; Im=im; }
```

Jeżeli w powyższej definicji konstruktora nazwy parametrów będą takie same jak nazwy pól, to pola klasy zostaną przesłonięte przez parametry. Aby odwołać się do pól klasy, należy ich nazwy poprzedzać kwalifikatorem

```
ZESP::ZESP(double Re, double Im)
```

```
{ ZESP::Re=Re; ZESP::Im=Im; }
```

Polom klasy można nadawać wartości podczas tworzenia obiektu (przed wykonaniem ciała funkcji konstruktora) za pomocą listy inicjacyjnej. Lista inicjacyjna występuje zaraz po nagłówku definicji konstruktora i jest poprzedzona dwukropkiem. Elementami listy są nazwy pól (niestatycznych) z wyrażeniami ujętymi w okrągłe nawiasy. Elementy te oddziela się przecinkami.

Klasa::Nagłówek_konstruktora : lista_inicjacyjna { ciało konstruktora }

Lista inicjacyjna ma postać

Pole_1(Wyrażenie_1), Pole_2(Wyrażenie_2), ..., Pole_n(Wyrażenie_n),

Na przykład listy inicjacyjne w konstruktorach

```
ZESP::ZESP(double re, double im): Re(re), Im(im) { }
VECT::VECT(double xx, double yy): x(xx),y(yy) { }
```

nadają polom *Re*, *Im* klasy *ZESP* wartości *Re*=*re*; *Im*=*im* oraz polom *x*, *y* klasy *VECT* wartości *x*=*xx*; *y*=*yy*;.

W powyższych przykładach *Re*, *Im*, *x*, *y* są nazwami pól, natomiast *re*, *im*, *xx*, *yy* są wyrażeniami – w tym przypadku parametrami formalnymi konstruktora. Ponieważ na liście inicjacyjnej wiadomo, co jest nazwą pola, a co parametrem, nazwy parametrów nie przesłaniają tu nazw pól. Poprawne są więc definicje

```
ZESP::ZESP(double Re, double Im): Re(Re), Im(Im) { }
VECT::VECT(double x, double y): x(x), y(y) { }
TEXT(char *t, int tmp): tmp(tmp), len(t?strlen(t):0)
  {if(t)
   {txt=new char[len+1];
    strcpy(txt, t);
   } else txt=NULL;
  }
```

Polom będącym referencjami lub obiektami można nadawać wartości jedynie poprzez listę inicjacyjną.
Jeśli pole obiektowe nie jest inicjowane w liście inicjacyjnej, to wywoływany jest dla niego konstruktor bezparametrowy.

Pytania i zadania

2.8. Dla klasy *TEXT* napisz konstruktor o nagłówku *TEXT*(*int n*, *char c*); tworzący obiekt z tekstem zawierającym *n* podanych znaków *c* (oraz znak '\0'). Co będzie, gdy drugi argument będzie miał wartość domyślną?

2.9. Dla klasy *TEXT* napisz konstruktory o nagłówkach *TEXT*(*char**, *int*=0); oraz *TEXT*(*TEXT*&, *int*=0);, w których ostatni argument nadaje wartość polu *tmp*.

2.10. Napisz konstruktor dla klasy *RECT* opisanej w zad. 2.2, tworzący obiekt dla podanego okna na ekranie.

2.5. Destruktor

Destruktorem jest bezargumentowa funkcja o nazwie klasy poprzedzonej znakiem tyldy ~*Klasa*( ), bez typu (nawet *void*), nie zwracająca wartości. Jeśli nie został zdefiniowany żaden destruktor, to jest on zdefiniowany niejawnie.

Destruktor jest niejawnie wywoływany zawsze wtedy, gdy usuwany jest obiekt jego klasy (gdy obiekty automatyczne przestają istnieć po zakończeniu programu w operatorach *delete*). Samo aktywowanie destruktora na rzecz obiektu nie powoduje usunięcia tego obiektu, lecz tylko wykonanie się funkcji destruktora. Domyślne definicje destruktorów w klasach *ZESP* i *TEXT* byłyby następujące:

```
~ZESP() { }
~TEXT() { }
```

Jeśli konstruktor zaalokował dla obiektu bufory, to domyślny destruktor ich nie usunie i bufory te będą ciągle zajmować pamięć. Bufory takie powinny być zwalniane przez odpowiednio zdefiniowany destruktor.

Destruktor, podobnie jak konstruktor, może wykonywać też i inne zadania. Ciało funkcji destruktora jest automatycznie wykonywane przed usunięciem obiektu.

Przykład

Aby w klasie *TEXT* destruktor zwalniał pamięć bufora z tekstem, a w klasie *VECT* destruktor (w celach dydaktycznych) wyprowadzał zawartość usuwanego obiektu, można konstruktory te zdefiniować następująco:

```
~VECT()
```

```
    { cerr<<" Destruktor2"<<*this;}
  ~TEXT()
    { if(txt) delete txt;
      txt=NULL;
    }
```

Warunkowe zwolnienie pamięci (*if*(*txt*) *delete txt*;) oraz przypisanie *txt*=*NULL*; jest konieczne, aby nie zwalniać raz już zwolnionego bufora. Można bowiem aktywować jawnie funkcję destruktora wiele razy na rzecz tego samego obiektu (bez jego usuwania). Tak więc zwalnianie buforów należy realizować zawsze według powyższego wzoru.

Jeśli obiekt zawiera pola wskaźnikowe do zaalokowanych (np. przez konstruktor) dla niego obszarów pamięci, **to należy zdefiniować własny destruktor**, który będzie zwalniał te obszary pamięci. W tym przypadku należy też zdefiniować własny **konstruktor kopiujący** oraz **operator przypisania**, bowiem:

- domyślny destruktor nie zwolni zaalokowanego dla obiektu obszaru pamięci,
- domyślny konstruktor kopiujący i operator przypisania będą kopiować tylko wartości wskaźników, zamiast alokować odpowiednie obszary i przepisywać do nich zawartość z kopiowanych obiektów.

Pytania i zadania

2.11. Uzupełnij konstruktory oraz destruktor klasy *TEXT*, tak aby wypisywały one na ekranie tworzone lub usuwane teksty (do celów dydaktycznych).

2.12. Zaproponuj destruktor dla klasy *RECT* opisanej w zad. 2.2.

2.6. Przykład klasy TEXT

Przykładowo przyjmijmy następującą definicję klasy *TEXT*:

```
class TEXT {
  private:
    int len, tmp;       // długość tekstu i znacznik tymczasowości
    char *txt;          // wskaźnik na tekst
  public:
```

```
    TEXT();              // konstruktor bezparametrowy
    TEXT(TEXT &);        // konstruktor kopiujący
    TEXT(char *);        // konstruktor konwersji z typu char*
    ~TEXT();             // destruktor
    TEXT &operator=(const TEXT&);       // operator przypisania
    const TEXT &operator+(const TEXT&);
                                        // operator dodawania (+)
    TEXT& operator+=(TEXT &);           // operator dopisania (+=)
    friend const TEXT& operator+(char *, TEXT &);
    friend ostream &operator<<(ostream&,const TEXT&);
};
```

W tej klasie tej obowiązkowo zdefiniowano:

konstruktor bezparametrowy

```
TEXT::TEXT(): len(0), tmp(0), txt(NULL)
   { }
```

konstruktor kopiujący

```
TEXT::TEXT(TEXT &s): len(s.len), tmp(0)
   { if(s.txt)
      { txt = new char[len+1];     // alokacja bufora
        if(txt) strcpy(txt, s.txt); else len = 0;
                                   // kopiowanie tekstu do bufora
      } else txt=NULL;
    if(s.tmp) delete &s;           // usunięcie obiektu tymczasowego
   }
```

destruktor

```
TEXT::~TEXT()
   {if(txt) delete txt;            // zwolnienie bufora z tekstem
    txt=NULL;                      // oznaczenie, że brak bufora
   }
```

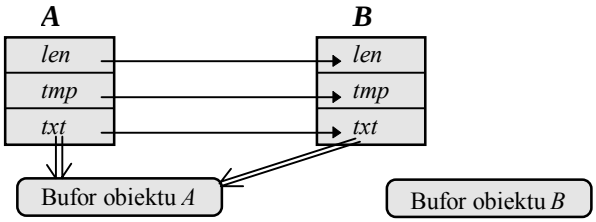
operator przypisania

```
TEXT& TEXT::operator=(const TEXT &s)
   {if(&s == this) return *this; // przypisanie tożsame (np. X=X;)
    delete txt;                  // zwolnienie bufora z tekstem
    len = s.len;
    if (s.txt)
    { txt = new char[len+1];     // alokacja bufora na nowy tekst
```

```
    if(txt) strcpy(txt, s.txt); else len = 0;
                                        // kopiowanie tekstu do bufora
  } else  txt = NULL;
  if(s.tmp) delete &s;                  // usunięcie obiektu tymczasowego
  return *this;                         // zwrot lewego argumentu
}
```

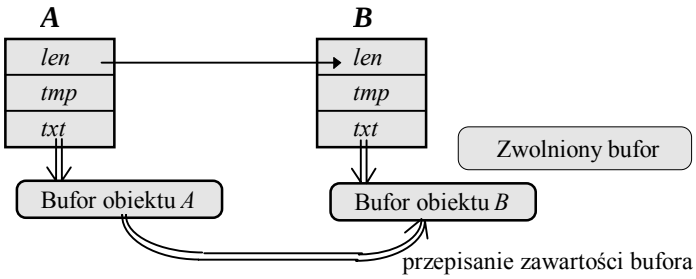
Zauważmy, że w instrukcji $X=A+B$; (A, B, X są obiektami klasy *TEXT*) prawym argumentem operatora przypisania jest wynik dodawania tekstów ($A+B$). Wynik ten jest pamiętany obiekcie tymczasowym, który po wykorzystaniu (wykonaniu przypisania) powinien zostać usunięty. Podobnie jeśli zostanie wywołana funkcja z argumentem $A+B$ przekazywanym przez wartość, to tymczasowy obiekt zawierający tę sumę zostanie usunięty przez konstruktor kopiujący.

Na rysunku 2.1 pokazano działanie domyślnego konstruktora kopiującego i domyślnego operatora przypisania. Kopiują one kolejne pola. Tak więc z obiektu A do obiektu B zostanie przepisana wartość wskaźnika *txt*. W rezultacie oba obiekty będą miały wspólny bufor z tekstem. Zmiana tekstu w obiekcie B spowoduje też zmianę tekstu w A, a tego najczęściej chcemy uniknąć. W przypadku przypisania $B=A$; stary bufor z tekstem obiektu B nie zostaje usunięty i zajmuje pamięć operacyjną. Bufor ten nie jest dostępny, ponieważ poprzednia wartość wskaźnika *txt* nie istnieje i nic na ten bufor nie wskazuje.



Rys. 2.1. Domyślne przypisanie $B=A$ lub stworzenie obiektu B w klasie *TEXT*

Prawidłowy operator przypisania usunie stary bufor, zaalokuje bufor na nowy tekst. Do tego bufora przekopiuje tekst z bufora A. Podobnie postąpi prawidłowy konstruktor kopiujący, z tym że tworzy on nowy obiekt i nie potrzebuje usuwać żadnego bufora. Prawidłowe działanie tego konstruktora i operatora przypisania pokazano na rys. 2.2.



Rys. 2.2. Prawidłowe przypisanie *B*=*A* lub stworzenie obiektu *B* w klasie *TEXT*

Destruktor po zwolnieniu bufora wskazywanego przez *txt* wpisuje *txt*=*NULL*, aby zapobiec przypadkowemu ponownemu zwolnieniu bufora.

Innym, już nieobowiązkowym, ale przydatnym konstruktorem jest konstruktor definiujący konwersję z typu *char** do typu *TEXT*. Tak więc w klasie *TEXT* zdefiniowano

konstruktor konwersji z typu *char** (konwersję konstruktorową)

```
TEXT::TEXT(char *t): len(t?strlen(t):0), tmp(0)
   { if(t)
      { txt = new char[len+1];      // alokacja bufora
        if(txt)strcpy(txt, t); else len = 0;
                                    // kopiowanie tekstu do bufora
      } else txt=NULL;
   }
```

W podobny sposób można definiować konwersje z innych typów (np. *char*) do typu *TEXT*.

Zadania laboratoryjne

2.13. Przeanalizować i uruchomić poniższy program. Wyniki oglądać po całkowitym zakończeniu programu. Określić, kiedy i jakie konstruktory i destruktory są aktywowane. W funkcji *operator+* klasy *VECT* zmienić przekazywanie parametru *A* z przekazywania przez wartość na przekazywanie przez referencję. Jak zmieniła się liczba konstruowanych obiektów?
W pierwotnym tekście programu uaktywnić instrukcje ujęte w komentarze usuwając /* */. Dlaczego kompilator wykazuje błędy? Zmienić sekcję pól *x*, *y* w klasie *VECT* z prywatnej na zabezpieczoną i ponowić kompilację. Dlaczego teraz nie ma błędów? W funkcji *main* ująć w komentarz pięć pierwszych linii (poprzednio aktywnych – po nawiasie klamrowym {). Określić, kiedy i jakie

konstruktory i destruktory są aktywowane. W funkcjach *operator+* obu klas zmienić przekazywanie parametru *A* z przekazywania przez wartość na przekazywanie przez referencję. Jak zmieniła się liczba konstruowanych obiektów?

```
#include <iostream.h>
#include <conio.h>

int N=(clrscr(),1);
                    // czyszczenie ekranu przed rozpoczęciem się funkcji main

class VECT {
 private:           // po dopisaniu VECT3D zmienić na protected:
  double x,y;
 public:
  VECT(double=0,double=0);          // konstruktor
  VECT(VECT&);                      // konstruktor kopiujący
  ~VECT();                          // destruktor
  VECT operator+(VECT);
  friend ostream &operator<<(ostream&,VECT&);
 };
/*
class VECT3D:public VECT      // klasa pochodna
 {double z;
 public:
  VECT3D(double=0,double=0,double=0);
  VECT3D(VECT&,double=0);     // konstruktor kopiujący
  ~VECT3D();                  // destruktor
  VECT3D operator+(VECT3D&);
  friend ostream &operator<<(ostream&,VECT3D&);
 };
*/
VECT::VECT(double xx,double yy):x(xx),y(yy)
                                          // xx, yy zmienić na x, y
 {cerr<<"\n Konstruktor2"<<*this;}

VECT::VECT(VECT &A):x(A.x),y(A.y)
 {cerr<<"\n Konst. kopiujacy"<<*this;}

VECT::~VECT()
 {cerr<<" Destruktor2"<<*this;}
```

```
VECT VECT::operator+(VECT A) {return VECT(x+A.x,y+A.y);}
ostream &operator<<(ostream &c,VECT &w)
 {return c<<" ("<<w.x<<','<<w.y<<") ";}

VECT P(333,444);                    // definicja zmiennej globalnej

/*
VECT3D::VECT3D(double xx,double yy,double zz):
                                          VECT(xx,yy),z(zz)
 {cerr<<" Konstruktor3"<<*this;}

VECT3D::~VECT3D()
 {cerr<<"\n Destruktor3"<<*this;}

VECT3D::VECT3D(VECT &A,double zz):VECT(A),z(zz)
 {cerr<<" Konstruktor3a"<<*this;}
VECT3D VECT3D::operator+(VECT3D &A)
 {return VECT3D((VECT)(*this)+(VECT)A,z+A.z);}

ostream &operator<<(ostream &c,VECT3D &w)
 {return c<<" ("<<w.x<<','<<w.y<<','<<w.z<<") ";}

VECT3D R(123,456,789);              // definicja zmiennej globalnej

*/
void main()
{cerr<<"\nDeklaracja VECT D(1,2),E(5,6),C, A(D), B=E:";
 VECT D(1,2), E(5,6), C, A(D), B=E;
 cerr<<"\nObliczanie C=A+B:";
 C=A+B;
 cout<<" C= "<<C<<endl;
/*

 cerr<<"\nDeklaracja VECT3D X(10,20,30),Y(40,50,60),Z:";
 VECT3D X(10,20,30),Y(40,50,60),Z;
 cerr<<"\nObliczanie Z=X+Y:";
 Z=X+Y;
 cout<<" Z= "<<Z<<endl;
*/
```

```
 cerr<<"\nKoniec programu:\n";      // nie zatrzymywać programu!!
}
```

3. Funkcje składowe i zaprzyjaźnione

3.1. Właściwości funkcji składowych

Przypomnijmy najważniejsze właściwości funkcji składowych.

Funkcje składowe:
1. Mogą być prywatne, publiczne i zabezpieczone.
2. Są w zakresie klasy i mają dostęp do wszystkich (również prywatnych) komponentów klasy.
3. Mogą być statyczne i niestatyczne.
4. Niestatyczne funkcje są aktywowane na rzecz obiektów swojej klasy i otrzymują obiekt ten jako niejawny argument wskazywany przez niejawnie zdefiniowaną zmienną *this*. Odwołania do pól obiektu bez użycia operatorów wyboru (. –>) oznaczają odwołania do pól obiektu, na rzecz którego aktywowana jest funkcja. Odwołania do składowych (funkcji) bez użycia operatorów wyboru oznaczają aktywowanie tych funkcji na rzecz niejawnego argumentu **this*.
5. Statyczne funkcje są aktywowane na rzecz klasy, nie zaś obiektu. Nie mają one zatem zmiennej *this*. Bez użycia operatorów wyboru można w funkcjach statycznych odwoływać się tylko do komponentów (pól i funkcji) statycznych.

Funkcje składowe są w zakresie klasy, to znaczy, że:
- mają dostęp do komponentów prywatnych (i innych) swojej klasy,
- wszelkie konflikty identyfikatorów rozstrzyga się na rzecz komponentów klasy.

Nawet gdy funkcja jest zdefiniowana poza klasą, to identyfikatory globalne są przesłaniane przez identyfikatory komponentów.

Na przykład identyfikatory *len* i *txt* w funkcjach klasy *TEXT* odnoszą się zawsze do pól klasy, nawet gdy zdefiniuje się zmienne globalne *len* lub *txt*. W poniższym

przykładzie funkcja *put* wyprowadzi tekst *Obiekt A* związany z obiektem *A*, nie zaś liczbę 13.

```
class TEXT {
   char *txt;          // txt – pole klasy
   . . .
   void put();         // funkcja klasy
  };

int txt=13;            // txt – zmienna globalna
TEXT::put()            // definicja funkcji klasy TEXT
   {cout << txt;}      // txt – pole klasy TEXT (a nie zmienna globalna)

main()
{ TEXT A("Obiekt A");
  A.put();             // wyprowadzi napis Obiekt A
  . . .
```

Nazwa *txt* w ciele funkcji *put* jest nazwą pola obiektu, na rzecz którego ta funkcja jest aktywowana. Ponieważ obiekt ten jest wskazywany przez niejawnie zdefiniowaną zmienną *this*, to wyrażenie *txt* należy rozumieć jako

```
this -> txt          oraz          (*this).txt
```

W powyższym przykładzie funkcja *put* jest aktywowana na rzecz obiektu *A* w instrukcji *A.put*( );. Tak więc zmienna *this* wskazuje obiekt *A* (*this*=&*A*), natomiast nazwa *txt* w ciele funkcji (w instrukcji *cout*<<*txt*;) jest nazwą pola obiektu *A*. Wyrażenie *txt* oznacza tu

```
(&A) -> txt          oraz          A.txt
```

Funkcje statyczne deklaruje się poprzedzając słowem *static*, np

```
static void precyzja(int);
```

Jeśli definicja funkcji znajduje się poza klasą, to słowo *static* należy w tej definicji opuścić, np.

```
void ZESP::precyzja(int p) {pn=(n<0)?0:(n>6)6:n;}
```

Jeśli do jakiejś funkcji ma być przekazany (jako argument) wskaźnik do funkcji statycznej (wskaźnikiem do funkcji jest jej nazwa), to nazwa funkcji statycznej zawsze musi być poprzedzona nazwą klasy, np.

```
func(... , ZESP::precyzja, ...);
```

Funkcje stałe deklaruje i definiuje się kończąc nagłówek funkcji słowem *const*, na przykład

```
void fun() const;
void Klasa::fun() const {  /* ciało funkcji */  }
```

Funkcje stałe nie mogą modyfikować obiektu, na rzecz którego są aktywowane.

Przyjrzyjmy się na przykład poniższej funkcji składowej klasy *ZESP*

```
double ZESP::Realis() const
  { Im=0;                  // niedozwolona modyfikacja
    return Re;
  }
```

Przypisanie $Im=0$ modyfikuje pole *Im* obiektu, na rzecz którego wywoływana jest funkcja *Realis*, a tego stała funkcja robić nie może.

Na rzecz obiektów stałych można aktywować tylko funkcje stałe.

Nie należy mylić funkcji stałych z funkcjami zwracającymi obiekty stałe. Te funkcje deklaruje się ze słowem *const* na początku, np.

```
const Klasa Fun(...);
```

Funkcja stała może zwracać w wyniku obiekt stały. Deklaruje (i definiuje) się ją wtedy ze słowami *const* na początku i na końcu, np.

```
const Klasa Fun(...) const;
```

W ciele niestatycznej funkcji składowej jest dostępna (niejawnie zdefiniowana) lokalna zmienna *this*, która wskazuje obiekt, na rzecz którego funkcja jest aktywowana. Niejawna definicja jest postaci

```
Klasa *const this;          // w funkcjach niestałych
const Klasa *const this;    // w funkcjach stałych
```

Zauważmy, że wyrażenie **this* daje obiekt, na rzecz którego funkcja jest aktywowana, a instrukcja

```
return *this;
```

zwraca obiekt ten jako wynik funkcji. Zauważmy na przykład, jak destruktor klasy *VECT*

```
~VECT() { cerr<<"Destrukcja "<<*this; }
```

wyprowadza na ekran usuwany obiekt. Podobnie przykładowe konstruktory tej klasy wyprowadzają (w celach dydaktycznych) tworzone obiekty.

Pytania i zadania

3.1. Napisz deklarację i definicję stałej funkcji obliczającej moduł liczby zespolonej typu *ZESP*, czyli pierwiastek z sumy kwadratów wartości pól *Re* oraz *Im*.

3.2. Bezparametrowa stała funkcja *Fun* klasy *Klasa* zwraca referencję do stałego obiektu swojej klasy. Napisz deklarację tej funkcji.

3.3. Funkcja bezparametrowa *Zamiana* zwraca referencję do obiektu, w którym pola *Re* i *Im* mają wartości kolejno pól *Im* i *Re* obiektu, na rzecz którego ta funkcja jest aktywowana. Napisz deklarację i definicję poza klasą tej funkcji zakładając, że:
a) funkcja nie może modyfikować niejawnego argumentu,
b) funkcja nie może być użyta po lewej stronie operatora przypisania,
c) jest to stała funkcja zwracająca stały obiekt.

3.2. Funkcje zaprzyjaźnione

Funkcje zaprzyjaźnione mogą odwoływać się do prywatnych (też do zabezpieczonych i publicznych) komponentów klasy. Poza tym nie różnią się od zwykłych funkcji.

W szczególności funkcje zaprzyjaźnione nie są w zakresie klasy, nie mogą być prywatne ani zabezpieczone niezależnie od tego, w której sekcji są zadeklarowane – nie są to bowiem funkcje klasy.

Funkcję zaprzyjaźnia się z klasą umieszczając w jej definicji deklarację tej funkcji z atrybutem *friend*. Definicja funkcji nie ma tego atrybutu.

Przykład

```
class TEXT {
   . . .
   friend int strlen(TEXT&);
   . . .
 };

int strlen(TEXT &S) {return s.len;}
```

Jedna funkcja może być zaprzyjaźniona z wieloma klasami i wtedy jej definicja musi wystąpić po definicji wszystkich tych klas.

Z daną klasą można zaprzyjaźnić funkcję składową innej klasy. Jeśli na przykład funkcja *put* w klasie *ZESP* miałaby wypisywać tekst z obiektu *S* klasy *TEXT* przed wyprowadzanym obiektem klasy *ZESP*, to aby mieć dostęp do prywatnego pola *txt*, funkcja *put* musi być zaprzyjaźniona z klasą *TEXT*.

```
class ZESP;

class TEXT {
   . . .
   friend void ZESP::put(TEXT&);
   . . .
  };

class ZESP {
   . . .
   void put(TEXT &S)
     { cout<<S.txt<<" ("<<Re<<','<<Im<<") "; }
   . . .
  };
```

Zauważmy, że w definicji klasy *TEXT* musi być znany identyfikator klasy *ZESP*. Stąd przed definicją klasy *TEXT* umieszczono deklarację klasy *ZESP*. Aby wewnątrz funkcji *put* skompilować wyrażenie *S.txt*, musi być znana pełna definicja klasy *TEXT*. Definicja klasy musi wystąpić zatem przed definicją funkcji.

Z daną klasą można zaprzyjaźnić wszystkie funkcje innej klasy. Definicje funkcji zaprzyjaźnionych, w których są odwołania do komponentów obu klas, muszą wystąpić po definicjach obu tych klas.

Jeśli na przykład zaprzyjaźnimy wszystkie funkcje klasy *ZESP* z klasą *TEXT*

```
class ZESP;
class TEXT {
   . . .
   friend class ZESP;
   . . .
  };
```

to wszystkie funkcje klasy *ZESP* będą mogły odwoływać się do wszystkich komponentów (również prywatnych) klasy *TEXT*. Definicje funkcji klasy *ZESP* powinny więc wystąpić po definicjach obu klas oraz przed definicją klasy *TEXT* musi wystąpić deklaracja klasy *ZESP*.

Pytania i zadania

3.4. Napisz niezależną funkcję, która wyprowadzi tekst z obiektu klasy *TEXT* do podanego strumienia wyjściowego. Zaprzyjaźnij tę funkcję z klasą *TEXT*, aby funkcja ta mogła odwoływać się do pola *txt*.

3.5. Napisz niezależną funkcję, która wyprowadzi tekst z obiektu klasy *TEXT* oraz wartości pól obiektu klasy *ZESP*. Podaj w odpowiedniej kolejności definicje klas i definicję tej funkcji.

3.6. Funkcję z poprzedniego zadania napisz jako funkcję klasy: a) *TEXT*, b) *ZESP*. W każdym przypadku podaj też definicje klas.

3.3. Funkcje operatorowe

Na obiektach tworzonych klas można definiować działania. Do definiowania działań operatorów służą **funkcje operatorowe**. Nazwa takiej funkcji składa się ze słowa kluczowego **operator**, po którym następuje symbol operatora. Na przykład: *operator*+ , *operator*( ), *operator*++ .

Działanie operatorów można definiować dowolnie. Nie można zmienić liczby argumentów operatora ani jego priorytetu, ani wiązania. Aby zachować dotychczasowe znaczenie operatorów dla standardowych typów, co najmniej jeden argument przeciążonego operatora musi być typu obiektowego (za wyjątkiem *new* i *delete*).
Nie można przeciążać operatorów: **. .* :: ?: sizeof** ani symboli preprocesora: # ##.

Specyfikacji formalnych argumentów funkcji operatorowej dokonuje się tak jak dla każdej innej funkcji. Jeśli funkcja operatorowa jest funkcją danej klasy, to obiekt na rzecz którego jest ona wywoływana stanowi lewy argument definiowanego operatora dwuargumentowego (lub jedyny argument operatora jednoargumentowego). Jeśli funkcja operatorowa jest tylko zaprzyjaźniona z daną klasą, to jej pierwszy argument jest lewym argumentem operatora. Co najmniej jeden z argumentów musi być typu obiektowego *KlasaX*.

Tak więc jedno- i dwuargumentowe funkcje operatorowe *operator@* definiuje się jako funkcje klasy

```
Typ_wyniku Klasa::operator@()
Typ_wyniku Klasa::operator@(Typ arg_prawy)
```

lub jako niezależne funkcje zaprzyjaźnione z klasą

```
Typ_wyniku operator@(KlasaX arg_lewy)
Typ_wyniku operator@(Typ arg_lewy, Typ arg_prawy)
```

Na przykład:

```
VECT VECT::operator+(VECT &A)
   {return VECT(x+A.x,y+A.y);}

ostream &operator<<(ostream &c,VECT &w)
   {return c<<" ("<<w.x<<','<<w.y<<") ";}
```

Funkcja *operator+* jest tu funkcją klasy *VECT* i definiuje (przeciąża) dwuargumentowy operator dodawania. Funkcja *operator<<* nie może być funkcją klasy *VECT*, ponieważ jej lewy argument nie jest obiektem tej klasy; nie może być ona wywoływana na rzecz obiektu klasy *VECT*.

Funkcja operatorowa może być wywołana jak każda inna funkcja lub przez proste użycie operatora. Jeśli na przykład *A*, *B*, *C* są obiektami klasy *VECT*, a *xwy* jest strumieniem wyjściowym klasy *ostream*, to instrukcje:

```
C = A + B;
xwy << C;
xwy << "Wektor C = " << C;
```

należy kolejno rozumieć w kontekście (i tak też można je wywołać):

```
C = A.operator+(B);
operator<<(xwy, C);
operator<<(operator<<(xwy, "Wektor C = "), C);
```

Działanie przeciążonych operatorów może być definiowane dowolnie i nie należy oczekiwać na przykład związku między operatorami: +, ++ oraz +=.

Aby zminimalizować liczbę przypisań, argumenty typu obiektowego przekazuje się do funkcji przez referencję. Przekazywanie obiektu przez wartość polega na utworzeniu wewnątrz funkcji lokalnego obiektu, do którego przekopiowuje się wartość argumentu aktualnego.

Przekazanie w wyniku wartości typu obiektowego zawsze tworzy automatyczny tymczasowy obiekt, który po wykorzystaniu jest usuwany w nieokreślonym momencie, co może prowadzić do rozrostu liczby niepotrzebnych już obiektów. Aby tego uniknąć, lepiej jest, aby w wyniku funkcja przekazywała referencję do obiektu otrzymanego jako referencyjny parametr wejściowy (np. funkcja *operator<<*) lub referencję do zaalokowanego obiektu tymczasowego, który zostanie w kontrolowany sposób zwolniony po wykorzystaniu. Zmodyfikowany operator dodawania może mieć tu nagłówek postaci

```
VECT& VECT::operator+(VECT &A)
```

Zauważmy, że tak zdefiniowana funkcja umożliwia przypisanie do utworzonego tymczasowego obiektu zawierającego sumę, np. $A+B=C$ lub wczytanie danych do takiego obiektu, np $cin>>(A+B)$;. Lewym argumentem operatora przypisania nie może być obiekt stały. Jeśli więc funkcja *operator+* będzie dawać w wyniku obiekt stały, nie będzie on mógł wystąpić po lewej stronie operatora przypisania. Należy ją zatem zdefiniować jako

```
const VECT& VECT::operator+(VECT &A)
```

Po tej poprawce kompilator ostrzega, że w wyrażeniach np. $A+B+C$ prawy operator + jest aktywowany na rzecz obiektu stałego. Rzeczywiście lewym argumentem tego operatora jest stały obiekt $(A+B)$, a niestała funkcja nie może być aktywowana na rzecz obiektu stałego. Wyrażenie $A+B+C$ jest bowiem realizowane jako $(A+B).operator+(C)$. Ponieważ funkcja *operator+* nie modyfikuje swojego lewego argumentu, możemy ją zdefiniować jako funkcję stałą. Funkcja nie modyfikuje też prawego argumentu i aby umożliwić realizację takich wyrażeń jak np. $A+(B+C)$ argument ten zadeklarujemy jako stały

```
const VECT& VECT::operator+(const VECT &A) const
  {VECT *t=new VECT;        // alokacja obiektu
   t->x=x+A.x;
   t->y=y+A.y;              // zapisanie wyniku operacji
   t->tmp=1;                // ustawienie znacznika tymczasowości
   if(this->tmp) delete this;  // zwolnienie pierwszego argumentu
   if(A.tmp) delete &A;     // zwolnienie drugiego argumentu
   return *t;
  }
```

Jeśli argumentem jest obiekt tymczasowy (dodatkowe pole *tmp* różne od zera), to funkcja go usunie. Funkcja ta zwraca obiekt stały (*const VECT&*), aby uczynić niepoprawnymi instrukcje typu

```
A+B=C;          oraz          cin>>(A+B);
```

Przykłady operatorów + oraz += zdefiniowanych w klasie *TEXT*

```
const TEXT& TEXT::operator+(const TEXT &s) const
  {TEXT *t = new TEXT;       // tymczasowy obiekt na wynik
   t->len = len+s.len;       // wynikowa długość tekstu
   t->tmp = 1;               // znacznik tymczasowości
   if(txt || s.txt)
     {t->txt = new char[t->len+1];
      t->txt[0]='\0';
      if(txt) strcat(t->txt, txt);
```

```
       if(s.txt) strcat(t->txt, s.txt);
      }
        else  t->txt=NULL;   // gdy oba składniki są puste
    if(tmp) delete this;     // usunięcie obiektu tymczasowego
    if(s.tmp) delete &s;     // usunięcie obiektu tymczasowego
    return *t;               // zwrot wyniku
   }

TEXT& TEXT::operator+=(const TEXT &s)
   {if(s.len==0) return *this;     // gdy obiekt s jest pusty
    char *t=new char[(len+=s.len)+1];
    t[0]='\0';
    if(txt) strcat(t, txt);
    strcat(t, s.txt);
    if(txt) delete txt;      // usunięcie starego bufora
    txt=t;                   // przypisanie nowego bufora
    if(s.tmp) delete &s;     // usunięcie obiektu tymczasowego
    return *this;
   }
```

Zauważmy, że dzięki konwersji konstruktorowej *TEXT*(*char**); prawym argumentem obu tych operatorów może być zwykły tekst (lub wskaźnik typu *char**). Lewym argumentem musi być obiekt klasy *TEXT*, ponieważ na rzecz tego obiektu te funkcje operatorowe są aktywowane. Jeśli na przykład *P*, *R* i *Q* są obiektami typu (klasy) *TEXT*, to dozwolone są instrukcje:

```
P+=Q;  P+="Borland";          P=Q+R;
P=Q+"Borland";
```

ale tak długo nie jest dozwolona instrukcja

```
P="Borland"+Q;
```

jak długo nie zostanie zdefiniowana poza klasą *TEXT* zaprzyjaźniona funkcja operatorowa

```
const TEXT& operator+(char *p, const TEXT &s)
   {TEXT *t=new TEXT;         // tymczasowy obiekt na wynik
    t->len=s.len+strlen(p); // wynikowa długość tekstu
    t->tmp=1;                // znacznik tymczasowości
    if(txt || s.txt)
     {t->txt=new char[t->len+1];
      strcpy(t->txt, p);
     }
```

```
      else  t->txt=NULL;
    if(s.txt) strcat(t->txt, s.txt);
    if(s.tmp) delete &s;      // usunięcie obiektu tymczasowego
    return *t;                // zwrot wyniku
   }
```

Podobnie operator wyjścia << należy zdefiniować poza klasą jako funkcję zaprzyjaźnioną, ponieważ jego lewy argument musi być typu *ostream*&, nie zaś typu *TEXT*&.

```
ostream &operator<<(ostream &wy, const TEXT &s)
   {wy<<s.txt;                // wyprowadzenie tekstu
    if(s.tmp) delete &s;      // usunięcie obiektu tymczasowego
    return wy;                // zwrot strumienia wyjściowego
   }
```

Przykładowy program, podany niżej, wyprowadzi trzy wiersze tekstu:

```
Borland C++
Jezyk  C++
Borland Pascal
```

```
#include <iostream.h>
#include <string.h>
#include <conio.h>

class TEXT {
   int len, tmp; // długość tekstu i znacznik obiektu tymczasowego
   char *txt;            // wskaźnik na bufor z tekstem
 public:
   TEXT():tmp(0),len(0),txt(0){ }
   TEXT(const TEXT &s);
   TEXT(const char *t);
   ~TEXT();
   const TEXT& operator+(const TEXT &s) const;
   TEXT& operator+=(const TEXT &s);
   const TEXT& operator+(char *p, const TEXT &s);
   friend ostream &operator<<(ostream &wy,const TEXT &s);
   friend istream &operator>>(istream &wy,const TEXT &s);
   TEXT& operator=(const TEXT &s);
};

      // Tutaj wstawić podane wcześniej definicje funkcji
```

```
main()
  {TEXT A("Borland"), B=" C++", C;
   clrscr();
   C=A+B;
   cout<<C<<endl<<("Jezyk "+B)<<endl<<(A+" Pascal");
   return(0);
  }
```

Funkcje operatorowe można podzielić na cztery kategorie:

1. Operatory ogólne, które albo są niestatycznymi składowymi klasy, albo mają co najmniej jeden argument typu obiektowego. Należy do niej większość operatorów.
2. Operatory *new* i *delete*, które nie muszą być składowymi klasy, ani nie muszą posiadać argumentów typu obiektowego, a jeśli są funkcjami klasy, to są to funkcje statyczne.
3. Operatory specjalne, które mogą być tylko niestatycznymi składowymi klasy. Są to operatory = **( )** **[ ]** **–>** .
4. Konwertery – jednoparametrowe niestatyczne funkcje składowe klasy bez typu (nawet *void*), definiujące konwersje z typu klasy do innego typu.

Operatory ogólne opisano w tym rozdziale, a poszczególne ich przypadki zostaną skomentowane poniżej.

Operatory ++ oraz –– mogą być przedrostkowe (++*p*) lub przyrostkowe (*p*++). W starych wersjach kompilatorów C++ przy przeciążaniu nie rozróżnia się tych form. Dopiero od wersji 3.0 kompilatora Borland C++ wprowadzono możliwość definiowania przyrostkowej formy (*p*++, *p*––) tych operatorów z drugim argumentem typu *int*. Ten drugi argument nie może być używany wewnątrz funkcji.

Operator pobrania adresu & jest często używanym operatorem o charakterze uniwersalnym. Dlatego choć można mu nadać dowolny sens, należy zachować jego tradycyjne znaczenie dostarczania adresu obiektu.

Przeanalizujmy na przykładzie w jakim celu przeciąża się operator pobrania adresu. Niech *A*, *B*, *C* będą obiektami klasy *ZESP*. Rozpatrzmy następujący fragment programu

```
ZESP *p=&(A+B);      // p otrzymuje wskazanie na obiekt tymczasowy,
cout << *p;          // operator<< likwiduje obiekt tymczasowy,
C=*p+A;              // lewy argument dodawania nie istnieje.
```

Program stanie się poprawny, gdy funkcja *operator*& zmieni status obiektu na stały, na przykład gdy zostanie zdefiniowana funkcja

```
ZESP *operator&()
  {tmp=0; return this;}
```

Pojawią się jednak problemy, kto i kiedy usunie ten obiekt, jeśli nie zrobi tego programista – przecież nie on dokonywał alokacji. Aby pobrać adres bez zmiany statusu obiektu tymczasowego, należy zdefiniować funkcję, np.

```
ZESP *Adres() { return this;}
```

Operator pobrania adresu obiektu klasy *Klasa* może mieć ogólną postać

```
Klasa *operator&()
  {
      /* Ciało funkcji */
    return this;
  }
```

Operator wyłuskania * ma jeden argument, który jest wskaźnikiem. Operator przeciążony jest funkcją klasy aktywowaną na rzecz obiektu tej klasy. Tak więc argumentem przeciążonego operatora jest obiekt klasy, nie zaś wskaźnik. W zależności od typu argumentu aktywowany jest więc albo operator globalny, albo przeciążony. Nie zachodzi tu zatem potrzeba definiowania specjalnej funkcji udostępniającej operator globalny.

Na przykład w wyrażeniu **A*, gdy *A* jest wskaźnikiem, operator * jest funkcją globalną, natomiast gdy *A* jest obiektem, operator * jest funkcją klasy tego obiektu.

Operatory new i delete w najprostszej postaci powinny być deklarowane jako

```
void *operator new(size_t);
void operator delete(void*);
```

Operatorów tych używa się tak jak operatorów ogólnych

```
Typ *ptr = new Typ;
```

Wywołanie to przekazuje argumentowi typu *size_t* (często zdefiniowanemu jako *unsigned*) rozmiar obiektu typu *Typ* w bajtach *sizeof*(*Typ*). Operator sam dokonuje konwersji typu zwracanego wskaźnika z *void** do *Typ**.

Kwalifikator globalności zawsze umożliwia wywołanie operatora globalnego, np.

```
Typ *ptr = ::new Typ;
```

Aby na przykład operator *new* tworzył obiekty klasy *TEXT* z wartością *tmp*=2, można zdefiniować w tej klasie

```
void *TEXT::operator new(size_t k)
  { TEXT *p=(TEXT*)new char[k]; // k==sizeof(TEXT)
    p->tmp=2;
    return p;
  }
```

Jeśli na przykład zażądamy, aby operator *delete* zwalniał tylko obiekty mające ustawioną wartość *tmp*=2, to należy zdefiniować

```
void *TEXT::operator delete(void *p)
  { if(((TEXT*)p)->tmp==2) delete p;
  }
```

Zauważmy, że wewnątrz powyższej funkcji zmienna *p* jest wskaźnikiem typu *void**, zatem w instrukcji *if* będzie aktywowany globalny operator *new*, nie zaś operator z klasy *TEXT*. Operatorów zdefiniowanych powyżej używa się tak jak operatorów globalnych, np.

```
TEXT *r=new TEXT;    // alokacja obiektu klasy TEXT
delete r;            // zwolnienie obiektu klasy TEXT
```

Istnieje możliwość przekazania funkcji *new* dodatkowych argumentów. Należy wówczas zadeklarować tę funkcję w postaci

```
void *operator new(size_t, Typ1, Typ2, ... , TypN);
```

Wywołanie ma postać

```
Typ *ptr = new (Arg1, Arg2, ... , ArgN)Typ;
```

gdzie: *Arg*1, *Arg*2, ..., *ArgN* są przekazywanymi argumentami typu kolejno *Typ*1, *Typ*2, ..., *TypN*, natomiast *Typ* jest identyfikatorem typu tworzonego obiektu.

Przykład operatora *new* alokującego obiekt klasy *TEXT* o statusie *st* (tymczasowy lub nie) z buforem na tekst o podanej długości *n*

```
void *operator new(size_t k, int st, int n)
  { TEXT *p=(TEXT*)new char[k]; // alokacja obiektu
    p->tmp=st;                  // nadanie statusu
    p->len=0;                   // aktualna długość tekstu
    p->txt=new char[n+1];       // alokacja bufora na tekst
    p->txt[0]='\0';             // wpisanie tekstu pustego
    return p;                   // zwrot wskazania na obiekt
  }
```

Aby zaalokować obiekt klasy *TEXT* o statusie *tmp*=1 z buforem na 80 znaków, wystarczy instrukcja

```
TEXT *q=new(1,80)TEXT;
```

Pytania i zadania

3.7. Napisz dla klasy *ZESP* funkcję *operator–*, będącą funkcją klasy oraz funkcję niezależną, która umożliwi odejmowanie obiektu klasy *ZESP* od liczby rzeczywistej. Czy potrzebne są obie funkcje równocześnie?

3.8. Napisz dla klasy *ZESP* operatory: a) mnożenia dwu obiektów tej klasy, b) mnożenia obiektu przez liczbę, c) mnożenia liczby przez obiekt.

3.9. Napisz dla klasy *TEXT* operator mnożenia: a) obiektu przez liczbę całkowitą, b) liczby całkowitej przez obiekt. Wynikiem w obu przypadkach powinien być obiekt z tekstem powielonym podaną liczbę razy.

3.10. Napisz funkcję *operator>>* realizującą wprowadzanie obiektów klasy: a) *ZESP*, b) *TEXT*.

3.11. Napisz w klasie *TEXT* funkcję *operator new*, która przydzieli pamięć na podaną liczbę obiektów i zainicjuje każdy obiekt: a) podanym (jednakowym) tekstem, b) kolejnymi tekstami zawartymi w tablicy typu *char* * t[ ].

3.4. Operatory specjalne

Operatory specjalne to: **operator indeksowania [ ], operator wywołania funkcji ( ), operator wyboru komponentu –>** oraz **operator przypisania =**. Operatory specjalne mogą być definiowane tylko jako niestatyczne funkcje klasy. Są one więc zawsze aktywowane na rzecz obiektu swojej klasy, a zatem ich lewym argumentem może być tylko obiekt tej klasy.

Operator indeksowania [ ] przeciążony ma jako lewy argument obiekt swojej klasy, natomiast operator globalny jako lewy argument ma wskaźnik. Dzięki tej różnicy operator przeciążony nie przesłania operatora globalnego. W każdym wyrażeniu *A*[*k*] lewym argumentem jest *A*, natomiast prawym jest *k*.

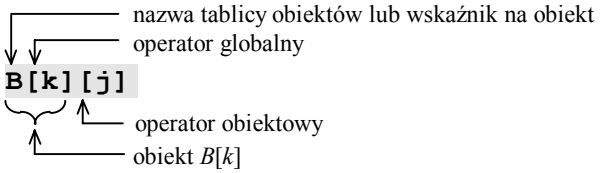
Przykład operatora w klasie *TEXT*, który udostępnia wybrane znaki tekstu

```
char &operator[](int n)
  {if(n<0) n=0; else if(n>=len) n=len-1;
   return txt[n];
  }
```

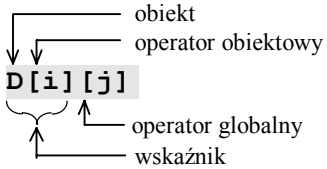
Jeżeli *A* jest obiektem klasy *TEXT*, to wyrażenie *A*[*k*] da w wyniku *k*-ty znak tekstu przechowywanego w obiekcie *A*. Zauważmy, że obiekt sam dba, aby nie

przekroczyć sensownych wartości indeksu i jeśli jest spoza dopuszczalnego przedziału, to wynikiem jest początkowy albo końcowy znak tekstu.

Jeśli *B* jest nazwą tablicy obiektów lub wskaźnikiem na obiekt (nie zaś obiektem), to aktywowany jest globalny operator i wyrażenie *B*[*k*] daje w wyniku *k*-ty obiekt za wskazywanym przez *B*. Wyrażenie *B*[*k*][*j*] jest opracowywane jako (*B*[*k*])[*j*] i aktywuje najpierw operator globalny, a potem przeciążony. Jeśli *B* wskazuje na obiekt klasy *TEXT* (lub *B* jest nazwą tablicy), to wyrażenie *B*[*k*][*j*] da w wyniku *j*-ty znak w *k*-tym obiekcie.



Jeżeli *D* jest obiektem, a wynikiem wyrażenia *D*[*i*] jest wskaźnik, to w wyrażeniu *D*[*i*][*j*] najpierw jest aktywowany operator obiektowy, a następnie operator globalny.



Przykład

Niech klasa *TAB* będzie klasą prostokątnych tablic liczb rzeczywistych. Użytkownikowi prezentuje się liczby ułożone w *N* wierszach i *M* kolumnach. Obiekty klasy *TAB* przechowują liczby te w tablicy jednoindeksowej. Początek definicji klasy *TAB* jest zatem następujący

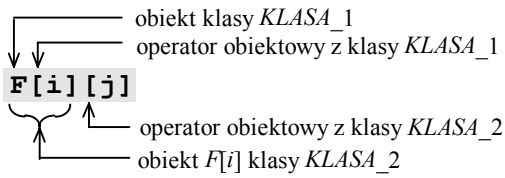
```
class TAB {
   int N, M, tmp;
   double *a;
public:
  double *operator[](int i) {return a+i*M;}
  . . . .
};
```

Pola *N*, *M* oznaczają liczbę wierszy i liczbę kolumn, natomiast wskaźnik *a* wskazuje na spójny obszar pamięci zawierający *N*×*M* liczb typu *double*. Liczby

umieszczone są wierszami, tak więc jeśli D jest obiektem klasy *TAB*, to wyrażenie $D[i]$ daje wskazanie i-tego wiersza tablicy. Zatem wyrażenie $D[i][j]$ daje element a_{ij} w i-tym wierszu i j-tej kolumnie.

Zauważmy, że jeśli wynikiem operatora indeksacji jest obiekt innej klasy, to w wyrażeniu $F[i][j]$ aktywowany jest najpierw operator obiektowy z klasy obiektu F, a następnie operator obiektowy z klasy obiektu $F[i]$. Nagłówek w definicji operatora jest wtedy następujący

```
KLASA_2 &KLASA_1::operator[](int i)
```



Operator wywołania funkcji ( ) ma dwa argumenty. Lewym argumentem operatora globalnego jest wskaźnik do funkcji (nazwa funkcji), a prawym argumentem jest lista wyrażeń. Na przykład w wyrażeniu $sin(x)$ lewym argumentem jest stała wskaźnikowa *sin*, a prawym jest wyrażenie x. Lewym argumentem operatora przeciążonego jest obiekt klasy. Tak więc i tu przeciążenie nie przesłania operatora globalnego.

Przykładowa funkcja w klasie *TEXT* da w wyniku obiekt z tekstem zawartym w lewym argumencie i ograniczonym do znaków zawartych między indeksem p a indeksem k (włącznie).

```
TEXT& TEXT::operator() (int p, int k)
  {
   TEXT *t=new TEXT;
   t->tmp=1;
   if(p>k) return *t;
   if(p<0) p=0;
   if(k>=len) k=len-1;
   t->len=k-p+1;
   t->txt=new char[t->len+1];
   strncpy(t->txt, txt+p, t->len);
   t->txt[t->len]='\0';
   if(tmp) delete this;
   return *t;
  }
```

Aby zabezpieczyć tymczasowy obiekt wynikowy przed użyciem go jako *l*-wartość oraz aby można było aktywować operator ten na rzecz obiektów stałych, należy go zadeklarować jako

```
const TEXT &operator() (int, int) const;
```

Przykładowy niżej podany fragment programu wyprowadzi napis BOR.

```
TEXT A="Jezyk BORLAND C++";
cout<<A(6, 8);
```

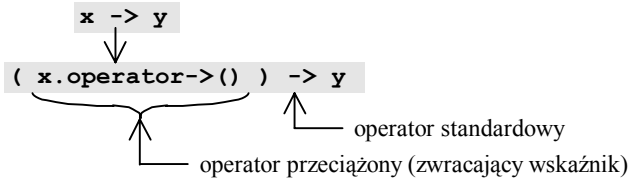
Operator pośredniego wyboru komponentu –> przeciążony jest jednoargumentowy, podczas gdy standardowy operator jest dwuargumentowy. Lewym (i jedynym) argumentem operatora przeciążonego jest obiekt klasy, natomiast lewym argumentem operatora standardowego jest wskaźnik na obiekt. W zależności od typu lewego argumentu jest więc aktywowany właściwy operator.

Operator pośredniego wyboru komponentu jest wywoływany w wyrażeniu *x*–>*y*, gdzie *x* jest lewym argumentem, a *y* jest identyfikatorem komponentu klasy. Jeśli *x* jest obiektem (nie zaś wskaźnikiem), to aktywowany jest operator przeciążony i jego wynik staje się lewym argumentem operatora pośredniego wyboru. Jeśli tym wynikiem jest obiekt, to ponownie jest aktywowany operator przeciążony. Jeśli wynikiem jest wskazanie obiektu, to aktywowany jest operator globalny z prawym argumentem *y*, co kończy opracowywanie wyrażenia *x*–>*y*.

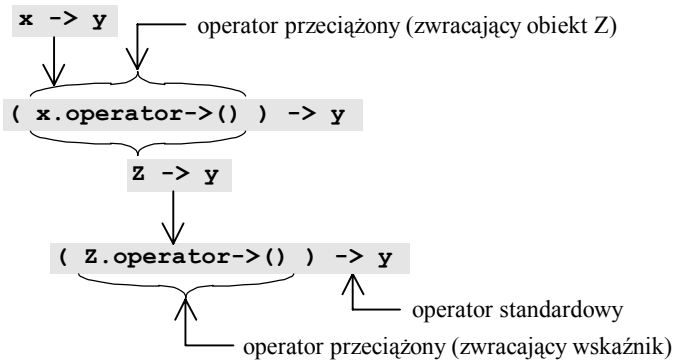
Jak widać, wynikiem przeciążonego operatora –> może być wskazanie obiektu dowolnej klasy albo obiekt innej klasy. Jeśli operator ten zadeklarowano w klasie *KlasaA*, to jego deklaracje mogą być następujące

```
KlasaA *operator->();
KlasaB *operator->();
KlasaB  operator->();
KlasaB &operator->();
```

Jeśli wynikiem jest wskazanie obiektu, to wyrażenie *x*–>*y* jest interpretowane następująco



Jeśli wynikiem jest obiekt innej klasy, to w tej klasie musi też istnieć przeciążony operator –> i jest on aktywowany na rzecz tego obiektu wynikowego, jak pokazano na poniższym rysunku



Jeśli na przykład zdefiniujemy w klasie *TEXT*

```
TEXT *operator->()  {return this;}
```

to gdy *X* jest obiektem klasy *TEXT*, wyrażenia *X.txt* oraz *X–>txt* dadzą taki sam wynik.

Operator przypisania = jest operatorem dwuargumentowym. W wyrażeniu $x=y$ obiekt *x* jest jego lewym argumentem. Jeśli w danej klasie (*KlasaX*) nie zdefiniujemy operatora = dla obiektów tej klasy, to do definicji klasy zostanie niejawnie dodana funkcja operatorowa postaci

```
KlasaX& operator=(const KlasaX&);
```

realizująca przypisanie obiektów metodą „pole po polu".

Metoda „pole po polu" polega na tym, że niezależnie do każdego pola używa się operatora przypisania właściwego temu polu. Na przykład jeśli przepisywane pole jest typu obiektowego, to zostanie użyty operator przypisania zdefiniowany w klasie tego pola.

Aby możliwe były instrukcje wielokrotnego przypisania (np. $A=B=C$), funkcja *operator*= musi zwracać swój lewy argument (wynik typu referencja do).

Tam, gdzie konstruktory alokują pamięć dla obiektów, powinno się definiować własny operator przypisania, który lewemu argumentowi (obiektowi) zwolni zaalokowaną pamięć, zaalokuje pamięć o rozmiarze takim jak w prawym argumencie i przepisze zawartość pamięci z prawego do lewego obiektu. Niejawnie dodany operator skopiowałby tylko wskazanie na zaalokowaną dla prawego argumentu pamięć, jak pokazano na rysunkach 2.1 oraz 2.2.

Przykładem operatora przypisania w klasie *TEXT* jest zdefiniowana wcześniej funkcja o nagłówku

```
TEXT& TEXT::operator=(const TEXT &s)
{
  if(&s==this) return *this;    // przypisanie tożsame (np. X=X;)
  if(txt) delete txt;           // zwolnienie bufora z tekstem
  len=s.len;
  if (s.txt)
   {txt=new char[len+1];        // alokacja bufora na nowy tekst
    strcpy(txt, s.txt);         // kopiowanie tekstu do bufora
   }
     else txt=NULL;
 if(s.tmp) delete &s;           // usunięcie obiektu tymczasowego
 return *this;                  // zwrot lewego argumentu
}
```

Prawy argument może być dowolnego typu. Powyższy operator może być przeciążony, np. funkcjami przypisania tekstu lub znaku do obiektu klasy. W tych przypadkach operatory te będą deklarowane jako

```
TEXT &operator=(const char*);
TEXT &operator=(const char);
```

Uwaga: Operator przypisania nie podlega dziedziczeniu.

Pytania i zadania

3.12. Napisz w klasie *ZESP* operator indeksowania taki, że wyrażenie *Z*[*k*] da w wyniku *Z.Re* gdy $k=1$, *Z.Im* gdy $k=2$ oraz $Z.Re^2+Z.Im^2$ dla pozostałych *k*. Zinterpretuj wyrażenie *A*[*i*][*j*], gdzie *A* jest zdefiniowane jako *ZESP A*[100];.

3.13. Napisz w klasie *TEXT* operator wywołania funkcji, który da w wyniku obiekt z tekstem złożonym z *k* pierwszych znaków tekstu argumentu niejawnego (na rzecz którego funkcja jest aktywowana) poprzedzających cały tekst z argumentu jawnego.

3.14. Klasa *TAB* zaczyna się definicją pól *tmp*, *n*, *A*: *class TAB* {*int tmp*, *n*; *double* **A*; ...};. Pole $tmp \neq 0$ oznacza obiekt tymczasowy, *n* – liczbę elementów w tablicy *A*, *A* wskazuje na początkowy element. Napisz operator przypisania i operator indeksowania dla obiektów tej klasy.

3.5. Konwertery

Konwertery są jednoargumentowymi funkcjami operatorowymi (bez argumentów jawnych) bez podanego typu wyniku (nawet *void*) formułującymi **konwersje definiowane** z typu klasy do innego typu.
Konwerter powinien być zadeklarowany w postaci

```
operator typ();
```

gdzie słowo *typ* jest nazwą typu, do którego dokonywana jest konwersja i typ ten jest równocześnie typem wyniku konwertera.

Na przykład w klasach *VECT* i *TEXT* można zdefiniować konwersje

```
operator double() {return sqrt(x*x+y*y);}
operator char*()  {return txt;}
```

Konwersja z typu *VECT* do typu *double* daje w wyniku długość wektora. Konwersja z typu *TEXT* do typu *char** zamienia obiekt klasy *TEXT* na wskaźnik do zawartego w nim tekstu.

Konwersje w przeciwną stronę do typu *KlasaX* (np. do typu *VECT*) są konwersjami konstruktorowymi. Na przykład konwersja *VECT*(*double d*):*x*(*d*),*y*(0) { } konstruuje wektor o składowej $x=d$, oraz o składowej $y=0$, dokonując w ten sposób konwersji z typu *double* do typu *VECT*.

Identyfikatory *cin* oraz *cout* oznaczają obiekty strumieniowe. Takie wyrażenia jak *if*(*cin*) ... lub *if*(*cout*) ... są poprawne, ponieważ w klasach strumieniowych jest zdefiniowany konwerter z typu klasy do typu wskaźnikowego *void** postaci

```
operator void*() { return fail() ? NULL : this; }
```

Jeśli stan strumienia umożliwia operacje na nim, to wynikiem konwersji jest wskazanie obiektu strumieniowego. W przeciwnym razie wynikiem konwersji jest wskazanie puste *NULL*.

Zauważmy, że gdyby w klasach strumieniowych zdefiniować jakikolwiek konwerter do typu skalarnego (*int*, *char*, *float*, ...), wówczas np. w wyrażeniu *if*(*cin*) ... kompilator nie wiedziałby, której konwersji użyć (wartość jakiego typu ma być porównana z zerem). Zauważmy też, że jeśli np. *Z* jest obiektem klasy *ZESP* i jest zdefiniowana konwersja z typu *ZESP* do typu *double*, to nie wiadomo czy wyrażenie $Z+5$ interpretować jako

```
Z+ZESP(5)
```

czy też jako

`(double)Z+5`.

Tak więc na ogół nie należy definiować zbyt wielu konwerterów, a dodanie konwertera do istniejącego programu może program ten uczynić niepoprawnym.

Konwersje definiowane mogą być aktywowane (tak jak konwersje standardowe) jawnie na żądanie programisty lub niejawnie przez kompilator, tam, gdzie jednoznacznie wymagana jest konwersja w celu interpretacji wyrażenia.

Pytania i zadania

3.15. Zdefiniuj konwersję z klasy *TEXT* do typu *int*, taką która daje w wyniku długość tekstu lub –1, gdy bufor na tekst nie jest zaalokowany.

3.16. Zdefiniuj konwersję z klasy *TAB* (opisanej w zadaniu 3.14) do typu *double**, która da w wyniku wskazanie na tablicę przechowywanych liczb.

3.17. W klasie *ZESP* zdefiniowano konwersję *operator double*( ), a w klasie *TEXT* zdefiniowano konwersje *operator char**( ) oraz *operator int*( ). Zmienne *A* i *B* są typu *ZESP*, a zmienne *X*, *Y* są typu *TEXT*. Jak zinterpretować wyrażenia: a) `X[(int)A]`, b) `*X`, c) `X+A`.

Zadania laboratoryjne

3.18. W klasach *VECT* i *VECT3D* w programie z ćwiczenia 2.13 zdefiniuj funkcje operatorowe (*operator–*, *operator**, *operator>>*) odejmowania wektorów, iloczynu skalarnego oraz wprowadzania. Zdefiniuj też w obu klasach konwersje funkcyjne do typu *double*, które każdemu wektorowi przyporządkują jego długość.

Do funkcji *main*( ) dopisz instrukcję *C*=*A*+80; i przetestuj program. Dopisz instrukcję *C*=60+*A*;. Dlaczego teraz program jest błędny?

Istniejącą funkcję *operator+* zdefiniuj jako funkcję globalną (nie należącą do klasy *VECT*) i zaprzyjaźnij ją z klasą *VECT*. Dlaczego program stał się poprawny?

4. Dziedziczenie

Dziedziczenie polega na przejmowaniu właściwości klasy bazowej przez klasę pochodną. Dziedziczenie uzupełnia zestaw pól klasy bazowej zestawem pól klasy pochodnej z zachowaniem ich kolejności. Tak więc w skład obiektów klasy pochodnej wchodzą pola klasy bazowej, a następnie pola klasy pochodnej. Poza tym wszystkie obiekty klasy pochodnej mają wspólne pola statyczne (*static*) zdefiniowane w obu klasach.

Do obiektów klasy pochodnej można (z pewnymi ograniczeniami) stosować operacje zdefiniowane przez funkcje składowe klasy bazowej.

Istnieją standardowe konwersje z typów klasy pochodnej do typów publicznej klasy bazowej.

> Dziedziczeniu nie podlegają ani konstruktory, destruktory, operator=, ani zaprzyjaźnienia.

4.1. Klasy bazowe i pochodne

Definiowana klasa może przejąć właściwości innych klas, zwanych klasami bazowymi. W nagłówku klasy pochodnej po jej nazwie i po dwukropku występuje lista nazw klas bazowych. Każda nazwa jest poprzedzona słowem *private* albo *public*, które określa stopień ukrycia komponentów klasy bazowej w klasie pochodnej. Klasy pochodne można definiować następująco:

```
class BAZOWA {
   . . .
  };
class POCHODNA: private/public BAZOWA {
   . . .
```

```
};
```

Przykład 4.1

Definicja klasy *VECT3D* na bazie klasy *VECT* może być następująca

```
class VECT {
 protected:
  double x,y;
  . . .
 };
class VECT3D: public VECT {
  double z;
  . . .
 };
```

Obiekty klasy *VECT3D* zawierają kolejno pola: *x*, *y*, *z*.

Jeśli klasa bazowa jest uprywatniona, to jej publiczne i zabezpieczone komponenty stają się prywatne w klasie pochodnej. Jeśli klasa bazowa jest upubliczniona, to jej publiczne i zabezpieczone komponenty stają się odpowiednio publiczne i zabezpieczone w klasie pochodnej. W każdym przypadku komponenty prywatne klasy bazowej nadal pozostają jej prywatnymi komponentami.

Komponenty prywatne klasy bazowej nie mogą być używane przez funkcje klasy pochodnej. Gdyby pola *x*, *y* były polami prywatnymi klasy *VECT*, to żadna funkcja klasy *VECT3D* nie mogłaby odwoływać się do tych pól.

Komponenty publiczne uprywatnionej klasy bazowej mogą być używane wobec obiektów klasy pochodnej tylko wewnątrz funkcji tej klasy (i funkcji z nią zaprzyjaźnionych). Wobec obiektów klasy pochodnej komponenty te są prywatne. Wobec obiektów klasy bazowej – pozostają publiczne. Dla przykładu jeśli zdefiniowano:

```
class BAZOWA {
   . . .
public: void put();                    // deklaracja funkcji publicznej,
   . . .
  };
class POCHODNA: private BAZOWA {       // prywatna klasa bazowa,
   . . .
  };
  . . .
BAZOWA B;       // definicja obiektu B klasy BAZOWA,
POCHODNA P;     // definicja obiektu P klasy POCHODNA,
```

```
B.put();        // instrukcja dozwolona wszędzie,
P.put();        // instrukcja niedozwolona poza klasą POCHODNA.
```

Instrukcja *P.put*( ) jest dozwolona tylko w zakresie klasy *POCHODNA*, nie jest natomiast dozwolona poza klasą, ponieważ składowa *put* jest prywatna w klasie *POCHODNA* (choć jest publiczna w klasie *BAZOWA*).

Zdefiniowanie w klasie pochodnej komponentu o takiej samej nazwie jak komponent klasy bazowej oznacza **przesłonięcie** komponentu klasy bazowej, a nie jego przeciążenie.

Poszukiwanie komponentu zaczyna się od klasy obiektu, na rzecz którego nastąpiło odwołanie się do tego komponentu i poszukiwanie kontynuuje się w klasach bazowych.

Przy odwołaniu kwalifikowanym (nazwa komponentu poprzedzona nazwą klasy, na przykład *P.BAZOWA*::*put*( );), poszukiwanie rozpoczyna się od klasy, której nazwą komponent jest kwalifikowany (tu od klasy *BAZOWA*).

Jeśli na przykład *X* jest obiektem klasy *VECT3D* (patrz zad.2.13 i przykład 4.1), a w publicznej klasie bazowej *VECT* jest zdefiniowana w sekcji publicznej funkcja

```
void skaluj(double dx, double dy) {x*=dx; y*=dy;}
```

to instrukcja `X.skaluj(2, 2);` podwoi wartości pól *x* oraz *y* w obiekcie *X*, pozostawiając bez zmian pole *z*. Jeśli teraz zdefiniujemy w klasie *VECT3D* funkcję

```
void skaluj(double dz) {z*=dz;}
```

to instrukcja `X.skaluj(2, 2);` jest błędna, ponieważ dwuargumentowa funkcja *skaluj* jest w klasie obiektu *X* przesłonięta funkcją jednoargumentową. Aby aktywować funkcję *skaluj* z klasy *VECT*, należy użyć instrukcji

```
X.VECT::skaluj(2, 2);
```

Istnieją standardowe konwersje typów z klasy pochodnej do klasy bazowej dla obiektów, wskaźników i referencji. Tak więc dla klas (typów) *Bazowa* i *Pochodna* istnieją konwersje:

```
Bazowa   ⇐   Pochodna
Bazowa*  ⇐   Pochodna*
Bazowa&  ⇐   Pochodna&
```

Jeśli klasa bazowa jest publiczna, to powyższe konwersje są dostępne też poza zakresem klasy pochodnej (czyli wszędzie). Na przykład jeśli zdefiniowano

```
VECT  A(1, 2), C, *p;
VECT3D X(10, 20, 30), Z, *q;
```

to dzięki standardowym konwersjom dozwolone są instrukcje:

```
C=X;    p=q;          p=&X;
```

natomiast bez odrębnego zdefiniowania konwersji błędne są instrukcje:

```
Z=A;    q=p;          q=&A;
```

Można jednak rzutować wskaźniki. Tak więc poprawne są instrukcje

```
q=(VECT3D*)p;              q=(VECT3D*)&A;
```

W instrukcji

Obiekt_klasy_bazowej = Obiekt_klasy_pochodnej;

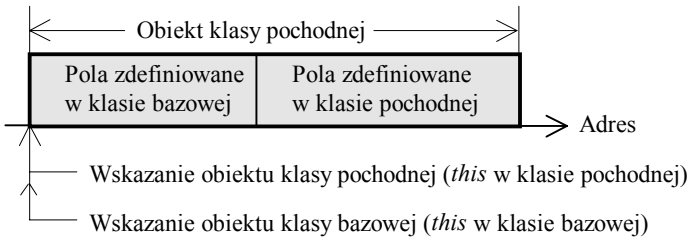
przypisaniu podlega tylko część wspólna obiektów. Przykładowa instrukcja

```
C=X;
```

podstawi zatem $C=(10, 20)$.

W pamięci komputera obiekt klasy pochodnej zaczyna się polami obiektu klasy bazowej, po których występują pola danych zdefiniowane w klasie pochodnej jak pokazano na rys. 4.1. Tak więc obiekt klasy pochodnej oraz wchodzący w jego skład obiekt klasy bazowej zaczynają się od tego samego adresu. Wskaźnik na obiekt klasy pochodnej wskazuje też na zawarty w nim obiekt klasy bazowej.

Pola statyczne klasy bazowej są wspólne dla wszystkich obiektów klasy pochodnej i klasy bazowej.



Rys. 4.1. Organizacja obiektu klasy pochodnej

Przed wykonaniem konstruktora klasy pochodnej zawsze jest wywoływany konstruktor klasy bazowej. Do konstruktora klasy bazowej można odwołać się tylko na liście inicjacyjnej (w konstruktorze klasy pochodnej). Jeśli takiego odwołania nie ma, to wywołany będzie bezparametrowy konstruktor klasy bazowej.

Jeśli na przykład w klasie *VECT* zdefiniowano konstruktor

```
VECT::VECT(double x=0, double y=0): x(x), y(y) {}
```

to konstruktor klasy *VECT3D*:

```
VECT3D::VECT3D(double Z): z(Z) {}
```

konstruuje obiekt o wartościach pól $x=0$, $y=0$ oraz $z=Z$, natomiast konstruktor

```
VECT3D::VECT3D(double X, double Y, double Z):
                    VECT(X, Y), z(Z)
  {}
```

konstruuje obiekt o wartościach pól $x=X$, $y=Y$ oraz $z=Z$.

Aby mieć konwersje z klasy bazowej do klasy pochodnej, należy zdefiniować konwersje konstruktorowe typu

```
POCHODNA(BAZOWA &);
```

Na przykład konstruktor

```
VECT3D::VECT3D(VECT &V, double Z=0): VECT(V), z(Z) {}
```

dzięki domyślnemu drugiemu argumentowi może być wywołany z jednym argumentem i definiuje wtedy obiekt klasy *VECT3D* z polem $z=0$ i z pozostałymi polami przepisanymi z obiektu klasy *VECT*. Konstruktor ten definiuje konwersję z typu *VECT* do typu *VECT3D*.

Pytania i zadania

4.1. Zdefiniuj klasę *PUNKT*, której komponentami prywatnymi będą współrzędne *x*, *y*, a publicznymi: konstruktor, funkcja przesuwania bezwzględnego *przesundo* oraz przesuwania względnego *przesun*. Bazując na klasie *PUNKT* zdefiniuj:
a) klasę *KOLO* z komponentem *r* określającym promień koła,
b) klasę *PROST* z komponentami *a*, *b* oznaczającymi długości boków prostokąta.
Napisz konstruktor w klasie pochodnej. Czy funkcje *przesundo* i *przesun* klasy *PUNKT* mogą być aktywowane na rzecz obiektów klasy *KOLO* lub *PROST* i dlaczego?

4.2. W klasie *PUNKT* z zadania 4.1 zdefiniuj wspólne dla wszystkich obiektów pola *xmax* i *ymax*, określające maksymalne wartości współrzędnych. W klasach pochodnych zdefiniuj funkcje *przesundo* i *przesun* takie, które zapewnią, że obiekt tej klasy (koło lub prostokąt) nie wysunie się (nawet częściowo) poza limity współrzędnych. Załóż $xmin=ymin=1$. Co trzeba zmienić w klasie *PUNKT*, aby funkcje klas pochodnych miały dostęp do komponentów *x*, *y* klasy *PUNKT*? Jak zrealizować przesunięcie bez ograniczeń, nie definiując nowych funkcji?

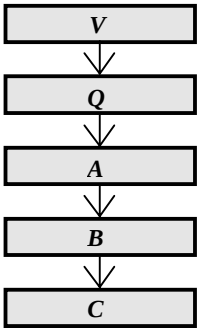
4.3. Dla klas *KOLO* i *PROST* pochodnych względem klasy *PUNKT* z zadania 4.1 napisz konstruktory definiujące konwersje konstruktorowe z klasy *PUNKT*. Czy potrzeba definiować konwersje w przeciwną stronę czyli do klasy *PUNKT*?

4.2. Dziedziczenie sekwencyjne

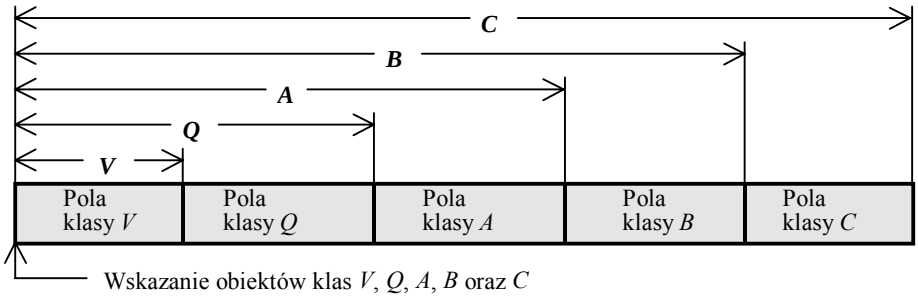
Przy dziedziczeniu sekwencyjnym klasy dziedziczą kolejno jedna po drugiej, np.

```
class V { . . . };
class Q: public V { . . . };
class A: public Q { . . . };
class B: public A { . . . };
class C: public B { . . . };
```

Hierarchię powyższych klas pokazano na rys. 4.2, a organizację obiektu klasy *C* pokazano na rys 4.3.



Rys. 4.2. Hierarchia klas dziedziczonych sekwencyjnie



Rys. 4.3. Organizacja obiektu klasy *C* dziedziczącej sekwencyjnie pozostałe klasy

Konstruktor klasy *C*, inicjując swoje pola przekazuje na liście inicjacyjnej parametry konstruktorowi swojej klasy bazowej. Konstruktor klasy *C* ma więc postać

```
C::C(...): B(...) { ... }
```

Jeśli na liście inicjacyjnej konstruktora klasy pochodnej nie ma konstruktora klasy bazowej, to aktywowany jest jej konstruktor bezparametrowy.

Dzięki standardowym konwersjom do klasy bazowej istnieją standardowe konwersje z klasy *C* do wszystkich pozostałych klas. Jeśli są potrzebne konwersje z klas bazowych do klas pochodnych, to należy je zdefiniować.

Na przykład z klasy *A* istnieją standardowe konwersje tylko do klas *Q* oraz *V*, natomiast konwersje do klas *B* i *C* można zdefiniować jako konwersje konstruktorowe *B*(*A*&); oraz *C*(*A*&); w postaci np.:

```
B::B(A &a): A(a) { . . . }
C::C(A &a): B(a) { . . . }
                        // wykorzystano poprzedni konstruktor B(A&);
```

Przykład 4.2

Rozważmy poniższą sekwencję klas z konstruktorami i funkcją *przesun*

```
class PUNKT {
  protected:
    double x,y;
  public:
    PUNKT(double x=0, double y=0): x(x), y(y) { }
    void przesun(double dx, double dy)
         {x+=dx, y+=dy;}
    . . .
};

class KOLO: public PUNKT {
  protected:
    double R;
  public:
    KOLO(double x=0, double y=0, double R=1):
                                  PUNKT(x, y), R(R) { }
    . . .
};

class WALEC: public KOLO {
  protected:
    double h;
  public:
```

```
    WALEC(double x=0,double y=0,double R=1,double h=0):
                                KOLO(x, y, R), h(h) { }
    . . .
  };
class RURA: public WALEC {
protected:
   double r;
public:
   RURA(): r(0.8)
      { }
   . . .
};
```

Na liście inicjacyjnej konstruktora *RURA*( ); nie umieszczono konstruktora klasy bazowej. W tej sytuacji zostanie użyty konstruktor bezparametrowy, co daje równorzędną definicję

```
    RURA(): WALEC(), r(0.8)
       { }
```

która konstruuje obiekt o wartościach pól $x=0$, $y=0$, $R=1$, $h=0$, $r=0{,}8$. Instrukcje

```
RURA X;              // x=0, y=0, R=1, h=0, r=0,8
X.przesun(1,2);      // aktywowana jest funkcja z klasy PUNKT
```

tworzą obiekt o wartościach pól $x=1$, $y=2$, $R=1$, $h=0$, $r=0{,}8$.
Aktywowanie funkcji *przesun* na rzecz obiektu *X* jest możliwe dzięki standardowym konwersjom z klas pochodnych do klas bazowych, a więc z klasy *RURA* do klasy *WALEC*, z klasy *WALEC* do klasy *KOLO* oraz z klasy *KOLO* do klasy *PUNKT*.

Konwersje w przeciwną stronę muszą zostać zdefiniowane. Przykładowe konstruktory definiujące konwersje konstruktorowe z klas bazowych do klas pochodnych powinny nadawać wartości domyślne polom zdefiniowanym w klasach pochodnych. Pola klasy bazowej zwykle są inicjowane przez konstruktor kopiujący (zdefiniowany albo domyślny), np.:

```
KOLO::KOLO(PUNKT &P): PUNKT(P), R(1) { }
WALEC::WALEC(KOLO &K): KOLO(K), h(0) { }
RURA::RURA(WALEC &W): WALEC(W), r(0.8*R) { }
```

Zauważmy, że w ostatnim konstruktorze można użyć wyrażenia $0{,}8 \cdot R$, ponieważ konstruktor podobiektu *WALEC* określił wartość pola *R* dla tworzonego obiektu klasy *RURA*. Funkcje klasy *RURA* mogą używać komponentu *R*, ale tylko w odniesieniu do obiektów swojej klasy. W stosunku do obiektów innej klasy (np. klasy *WALEC*) nie mogą, bo pole *R* jest komponentem zabezpieczonym tej innej klasy. Tak więc na

przykład inicjacja pola *r* w konstruktorze *RURA*::*RURA*(*WALEC &W*); wyrażeniem *W.R* byłaby błędna, podobnie jak byłaby błędna inicjacja *r*(0,8**W.R*)

Konwersje definiowane nie nakładają się w sposób domyślny. Tak więc w poniższym przykładzie przypisanie *X*=*P*; wymaga użycia jawnego rzutowania.

```
PUNKT P(10,20);
KOLO K=P;
RURA X=(WALEC)(KOLO)P;
```

Pytania i zadania

4.4. W przykładowej sekwencji klas zdefiniuj w klasie *KOLO* funkcję *pole*, która obliczy pole koła oraz w klasie *WALEC* funkcję *poleb*, która obliczy pole powierzchni bocznej walca. Jak obliczyć pole powierzchni całkowitej obiektu klasy *WALEC*? Na rzecz obiektów których klas można aktywować te funkcje?

4.5. W klasie *RURA* zaproponuj konstruktor, który umożliwi nadanie wartości wszystkim polom tworzonego obiektu. Podaj, jakie konstruktory i w jakiej kolejności będą aktywowane podczas definiowania obiektu klasy *RURA*?

4.6. Na wzór sekwencji klas od *PUNKT* do *RURA* zaproponuj inną sekwencję trzech lub więcej klas wraz z odpowiednimi konstruktorami.

4.7. Na wzór konstruktorów definiujących konwersje konstruktorowe w klasach *KOLO*, *WALEC* i *RURA* zaproponuj odpowiednie operatory przypisania. Jaki jest pożytek z wprowadzenia tych operatorów?

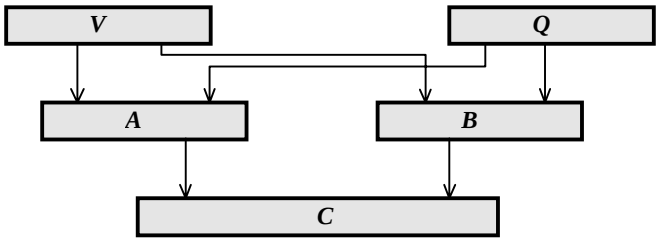
4.3. Dziedziczenie wielobazowe

W przypadku dziedziczenia wielobazowego **konwersje z klasy pochodnej do klasy bazowej istnieją tylko wtedy, gdy są jednoznaczne** – to znaczy, gdy pola klasy bazowej występują w klasie pochodnej jednokrotnie. Gwarancję jednoznaczności daje dziedziczenie wirtualne (*virtual*), np.

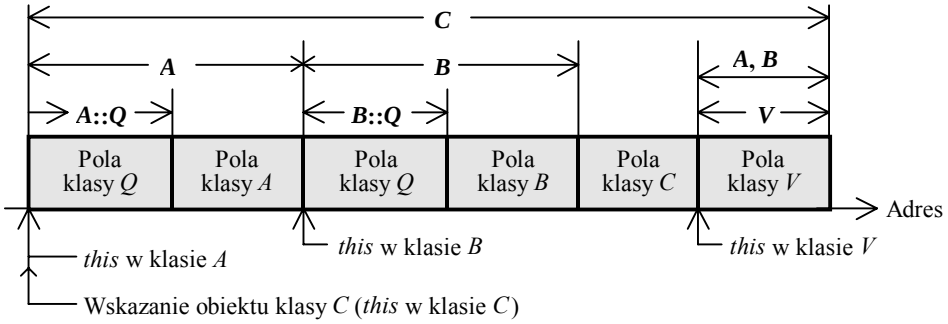
```
class V { . . . };
class Q { . . . };

class A: virtual public V, public Q { . . . };
class B: virtual public V, public Q { . . . };
```

```
class C: public A, public B { . . . };
```

Hierarchię powyżej zdefiniowanych klas pokazano na rys. 4.4. W klasie *C* zawarto jedną klasę *V* i dwie klasy *Q*, jak pokazano na rys. 4.5.



Rys. 4.4. Hierarchia klas *A*, *B*, *C*, *Q* oraz *V*



Rys. 4.5. Organizacja obiektu klasy *C* w wersji 3.1 kompilatora Borland C++

Uwaga: Klasa *V* musi być dziedziczona wirtualnie w obu klasach *A* oraz *B*, aby była wirtualna w klasie *C*.

Obiekt klasy pochodnej składa się z pól klas bazowych oraz z własnych pól. Pola klas bazowych występują w kolejności występowania tych klas w nagłówku deklaracji klasy pochodnej, przy czym pola klas dziedziczonych wirtualnie występują tylko raz i mogą być zgrupowane na końcu (Borland C++ 3.1).

Konstruktory aktywowane są w następującej kolejności:
1. Konstruktory klas dziedziczonych wirtualnie w kolejności ich występowania w nagłówku deklaracji klasy.
2. Konstruktory pozostałych klas w kolejności jak wyżej.
3. Konstruktory podobiektów klasy pochodnej.

4. Konstruktor klasy pochodnej.
Destruktory są aktywowane w kolejności odwrotnej.

Istnieją standardowe konwersje z klasy *C* do klas *A* i *B* oraz do klasy *V*, natomiast nie istnieją konwersje z klasy *C* do klasy *Q*.

W nagłówku konstruktora klasy pochodnej należy przekazać parametry klasom bazowym i klasom wirtualnym.

Na przykład konstruktor klasy *C*

```
C::C(...): A(...),B(...),V(...)
      {...}
```

przekazuje w liście inicjacyjnej parametry klasom bazowym *A* i *B* oraz klasie wirtualnej *V*, natomiast nie przekazuje parametrów klasie *Q*. Klasa *Q* występuje w klasie *C* dwukrotnie i otrzymuje parametry od klasy *A* i klasy *B*.

Konstruktor klasy *VECT3D* przekazuje parametry klasie *VECT*, np.

```
VECT3D::VECT3D(double xx,double yy,double zz):
                                        VECT(xx,yy),z(zz)
      { }
```

Komponenty klasy pochodnej przesłaniają tak samo nazwane komponenty klasy bazowej. Na przykład dla zdefiniowanych obiektów *X*, *Y* w klasie *VECT3D* oraz obiektu *C* w klasie *VECT* wyrażenie *Y.x* odwołuje się do nieprzesłoniętego pola *x* w klasie *VECT*, natomiast wyrażenie *C=X+Y* aktywuje funkcję *operator+* z klasy *VECT3D*, nie zaś przesłoniętą funkcję z klasy *VECT*. Gdyby w klasie *VECT3D* nie zdefiniowano funkcji *operator+*, to aktywowana byłaby funkcja z klasy *VECT*. W przypadku dziedziczenia wielobazowego należy zwrócić uwagę na jednoznaczność odwoływania się do nieprzesłoniętych komponentów klas bazowych. Na przykład jeżeli w wyżej zdefiniowanych klasach *Q*, *V*, *A*, *B* i *C* komponenty (pola lub funkcje) o nazwie *N* są nieprzesłonięte w klasach *A* i *B*, a *P* jest obiektem klasy *C*, to wyrażenie *P.N* nie jest jednoznaczne, bo nie wiadomo, o który komponent chodzi (z klasy *A*, czy z klasy *B*). Poprawne i jednoznaczne są wyrażenia z kwalifikacją klasy przed nazwą komponentu

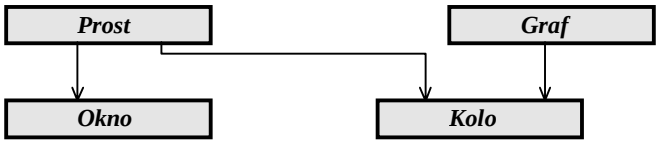
`P.A::N` oraz `P.B::N`

nawet wtedy, gdy komponenty *N* w klasach *A* i *B* są przesłonięte.

Zauważmy (por. rys. 4.5), że każdy komponent klasy *Q* wystąpi w obiekcie klasy *C* dwukrotnie, raz w podobiekcie klasy *A* i raz w podobiekcie klasy *B*. Zatem aby odwołać się w klasie *C* do dowolnego (nieprywatnego) pola klasy *Q*, należy użyć kwalifikatora, aby określić, czy chodzi o pole zawarte w klasie *A* czy w klasie *B*.

Przykład 4.3

Poniższy program ilustruje wyprowadzanie na ekran prostokątów w trybie tekstowym i kół w trybie graficznym. Figury tworzone są przez konstruktory obiektów klasy *Okno* oraz klasy *Kolo*. Hierarchię klas z poniższego programu pokazano na rys. 4.6.



Rys. 4.6. Hierarchia klas *Prost*, *Okno*, *Graf*, oraz *Kolo*

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <graphics.h>
const char path[]="C:\\bc3\\bgi";

class Prost {                                  // klasa definiująca prostokąt
   protected:
     int x1, y1, x2, y2;                  // współrzędne rogów prostokąta
   public:
     Prost(int x1=1, int y1=1, int x2=1, int y2=1):
                 x1(x1), y1(y1), x2(x2), y2(y2)  {}
   };

class Okno: public Prost {          // klasa prostokąta na ekranie tekstowym
   protected:
     char *buf;                      // bufor na przesłonięty fragment ekranu
     int tlo;                                          // kolor tła prostokąta
   public:
     Okno(int=1, int=1, int=80, int=25, int=0);
     ~Okno();
   };

Okno::Okno(int x1, int y1, int x2, int y2, int c):
                                    Prost(x1,y1,x2,y2),tlo(c)
   {buf=new char[2*(x2-x1+1)*(y2-y1+1)];          // alokacja bufora
    gettext(x1,y1,x2,y2,buf);   // zapamiętanie części ekranu pod oknem
    window(x1,y1,x2,y2);
    textbackground(c);
```

```
    clrscr();                                          // wpisanie tła w okno
   }

Okno::~Okno()
   {if(!buf) return;
    puttext(x1,y1,x2,y2,buf);                          // odtworzenie ekranu
    delete buf;                                        // usunięcie bufora
    buf=NULL;
   }

class Graf {                                           // obsługa trybu graficznego
   protected:
     static int N_ob;                                  // licznik obiektów graficznych
   public:
     Graf();
     ~Graf();
   };

int Graf::N_ob=0;                  // definicja licznika (pola statycznego klasy)

Graf::Graf()
   {if(!N_ob)                                          // pierwszy obiekt inicjuje grafikę
        {int Driver=DETECT, Mode, e;
         initgraph(&Driver, &Mode, path);
         if((e=graphresult())!=grOk)
                                          cerr<<grapherrormsg(e);
        }
    N_ob++;                                            // zliczenie obiektu
   }

Graf::~Graf()
   { N_ob--;
     if(!N_ob) closegraph();                           // ostatni obiekt zamyka grafikę
   }

class Kolo: public Prost, public Graf {
   protected:
     char *buf;                                        // bufor na przesłonięty fragment ekranu
     int x, y, r, tlo;                                 // środek, promień i kolor koła
   public:
     Kolo(int=200, int=180, int=100, int=LIGHTGRAY);
     ~Kolo();
   };
```

```
Kolo::Kolo(int x, int y, int r, int tlo):
                  Prost(x-r,y-r,x+r,y+r),r(r),tlo(tlo)
   { buf=new char[imagesize(x1,y1,x2,y2)];
     getimage(x1,y1,x2,y2,buf);
     setfillstyle(SOLID_FILL,tlo);
     pieslice(x,y,0,360,r);
   }
Kolo::~Kolo()
   { if(!buf) return;
     putimage(x1,y1,buf,COPY_PUT);
     delete buf;
     buf=NULL;
   }

main()
   { Okno  P,                  // zapamiętanie i wypełnienie ekranu czarnym tłem
           P1(10,10,50,18,RED),        // wpisanie czerwonego prostokąta
           P2(35,12,60,22,GREEN);      // wpisanie zielonego prostokąta
     getch();
       {
         Kolo K1, *K2=new Kolo(300,260,120,BLUE);
         getch();                  // pokazanie koła szarego i niebieskiego
         delete K2;                          // usunięcie koła niebieskiego
         getch();
       }                           // destruktor obiektu K1 usuwa koło szare
     return 0;
   }                               // destruktor obiektu P odtwarza ekran
```

Pytania i zadania

4.8. Jeżeli klasy *V*, *Q*, *A*, *B* oraz *C* zdefiniowano jak w przykładzie na rys. 4.4, to w jakiej kolejności i jakie konstruktory będą aktywowane w poniższych definicjach: a) *A a*; b) *B b*; c) *C c*;

4.9. Na wzór hierarchii klas *V*, *Q*, *A*, *B* oraz *C* zaproponuj definicje konkretnych klas z przykładowymi polami oraz konstruktorami. Narysuj hierarchię tych klas oraz organizację obiektu klasy pochodnej dziedziczącej wielobazowo.

4.10. Prostokąt na ekranie ma swoje wymiary oraz swoje położenie. Niech położenie prostokąta określa punkt przecięcia się jego przekątnych. Taki prostokąt można wypełnić tłem, obwieść ramką lub wpisać do niego tekst. Można to pokazać na

ekranie i usunąć z ekranu. Zaproponuj definicję hierarchii klas, w której klasa tekstu w ramce bazuje na klasie ramki i na klasie tekstu w prostokącie. Klasa ramki bazuje na klasie *prostokąt*, a ta na klasie *punkt*. Klasa tekstu w prostokącie bazuje na klasie *prostokąt* i na zdefiniowanej w poprzednich rozdziałach klasie *TEXT*. Zaproponuj dla definiowanych klas konstruktory, destruktory (jeśli będą potrzebne), pola danych oraz funkcje *pokaz* i *usun* pokazujące i usuwające obiekty z ekranu.

4.4. Funkcje polimorficzne

Funkcje polimorficzne (wirtualne) to funkcje zadeklarowane ze specyfikatorem *virtual* oraz wszystkie funkcje tego samego typu (te same nazwy, typy wyniku, liczby i typy argumentów) zawarte w dowolnej klasie pochodnej w ciągu klas pochodnych. Funkcją polimorficzną mogą być destruktory pomimo różnych nazw w różnych klasach.

Wywołanie funkcji polimorficznej na rzecz wskazywanego obiektu dokonuje się z klasy tego obiektu, nie zaś z klasy wskaźnika, który go wskazuje.
W konstruktorach i destruktorach funkcje polimorficzne zachowują się tak jak zwykłe funkcje.

Na przykład niech w klasach *BAZOWA* i *POCHODNA* funkcje *fun*2 będą polimorficzne, a *fun*1 nie, jak zadeklarowano poniżej.

```
class BAZOWA {
   public:
      fun1(...);          // kropki symbolizują parametry funkcji
      virtual fun2(...);
      . . .
   };

class POCHODNA: public BAZOWA {
   public:
      fun1(...);
      fun2(...);
      . . .
   };
```

Jeśli zdefiniujemy obiekt *S* i wskaźnik *p* jak poniżej

```
POCHODNA S;
BAZOWA *p=&S;
```

to instrukcja

```
p->fun1(...);
```

wywoła funkcję *fun*1 z klasy *BAZOWA* na rzecz podobiektu klasy *BAZOWA* zawartego w obiekcie *S*, bo *p* jest wskaźnikiem do typu *BAZOWA*, natomiast instrukcja

```
p->fun2(...);
```

wywoła wirtualną funkcję *fun*2 z klasy *POCHODNA* na rzecz obiektu *S*, bo wskaźnik *p* wskazuje faktycznie na obiekt *S* klasy *POCHODNA*.

Przykład 4.4

Jeśli w klasach *VECT* i *VECT*3*D* określimy funkcje obliczania sumy współrzędnych wektorów i polimorficzną funkcje obliczania długości wektorów

```
class VECT {
      . . .
   public:
      . . .
      double suma();
      virtual double dlugosc();
   };

class VECT3D: public VECT {
      . . .
   public:
      . . .
      double suma();
      double dlugosc();
   };

double VECT::suma() {return x+y;}
double VECT::dlugosc() {return sqrt(x*x+y*y);}

double VECT3D::suma() {return x+y+z;}
double VECT3D::dlugosc() {return sqrt(x*x+y*y+z*z);}
```

i zdefiniujemy

```
VECT3D X(-10, 20, -20);
VECT *p=&X;
```

to wyrażenie

```
p->suma()
```

aktywuje funkcję *suma* z klasy *VECT* (bo *p* jest typu *VECT**) i da w wyniku 10 (bo $-10+20=10$), natomiast wyrażenie

```
p->dlugosc()
```

aktywuje funkcję *dlugosc* z klasy *VECT3D* (*p* wskazuje na *X* czyli na obiekt klasy *VECT3D*) i da w wyniku 30 (bo $(-10)^2+20^2+(-20)^2=30^2$).

Innym typowym zastosowaniem funkcji polimorficznych jest aktywowanie ich na rzecz formalnych argumentów referencyjnych. Aktywowana jest funkcja polimorficzna nie z klasy argumentu formalnego, lecz z klasy argumentu aktualnego.

Przykład 4.5

Zrezygnujmy z definicji funkcji *operator*<< oraz *operator*>> z prawym argumentem z klasy *VECT3D*. Zdefiniujmy te funkcje operatorowe tylko z argumentem klasy *VECT*:

```
ostream &operator<<(ostream &wy, VECT &v)
   { wy<<" (";
     v.put(wy);                  // aktywowanie funkcji polimorficznej
     wy<<") ";
     return wy;
   }
istream &operator>>(istream &we, VECT &v)
   { int x, y;
     if(we==cin)
       { cerr<<"Wspolrzedne wektora: ";
         x=wherex();
         y=wherey();
       }
     v.get(we);                  // aktywowanie funkcji polimorficznej
     if(we!=cin) return we;
     while(!we)
       { gotoxy(x, y);
         clreol();
         we.clear();
         we.ignore(0x7FFF,'\n');
         v.get(we);              // aktywowanie funkcji polimorficznej
       }
     return we;
```

```
}
```

Deklaracje (lub definicje) funkcji w definicjach klas *VECT* i *VECT3D* będą następujące

```
class VECT {
  protected:
    float x,y;
    virtual void put(ostream &wy)    { wy<<x<<','<<y;}
    virtual void get(istream &we)    { we>>x>>y;}
  public:
    . . .
    friend ostream &operator<<(ostream&,VECT&);
    friend istream &operator>>(istream&,VECT&);
  };
class VECT3D:public VECT {
    float z;
    void put(ostream &wy)    { wy<<x<<','<<y<<','<<z;}
    void get(istream &we)    { we>>x>>y>>z;}
    . .
  };
```

Dzięki standardowym konwersjom z klasy pochodnej do bazowej obiekty klasy *VECT3D* mogą być wprowadzane i wyprowadzane operatorami przeciążonymi w klasie *VECT*. Indywidualne (dla tych klas) czynności są realizowane prze wirtualne funkcje *get* oraz *put* aktywowane na rzecz referencyjnego argumentu *v* (*v.get*(*we*); *v.put*(*wy*);). W miejsce tego argumentu jest podstawiany bowiem obiekt klasy *VECT* lub klasy *VECT3D*. Funkcja wirtualna rozpoznaje aktualny typ obiektu i aktywowana jest funkcja z jego klasy. Reasumując:

W wyrażeniach *v.get*(*we*) oraz *v.put*(*wy*) o wyborze funkcji wirtualnej *get* oraz *put* nie decyduje formalny typ zmiennej referencyjnej *v*, lecz typ zmiennej aktualnej.

Pytania i zadania

4.11. Dla sekwencji klas *PUNKT*, *KOLO*, *WALEC* i *RURA* z przykładu 4.2 napisz wspólne operatory wejścia i wyjścia z użyciem funkcji wirtualnych.

4.5. Czyste funkcje wirtualne

Czyste funkcje wirtualne to funkcje polimorficzne o niezdefiniowanym algorytmie. W deklaracji nagłówek takiej funkcji przyrównuje się do zera, np.

```
virtual int fun(int,int) = 0;
```

Klasa, w której zadeklarowano choć jedną czystą funkcję wirtualną, jest **klasą abstrakcyjną** (klasą fundamentalną). Nie można tworzyć bezpośrednio obiektów klasy abstrakcyjnej. Można jednak deklarować wskaźniki do obiektów tej klasy.

Czyste funkcje wirtualne w klasach pochodnych klasy abstrakcyjnej muszą być zdefiniowane lub ponownie zadeklarowane jako czyste.

Zastosowanie klas abstrakcyjnych pozwala na oprogramowanie ogólnych, wspólnych cech obiektów z różnych klas pochodnych. Szczegóły algorytmów będą oprogramowywane dopiero na etapie tworzenia poszczególnych klas. Zwiększa to niezawodność oprogramowania, gwarantuje bowiem zachowanie pewnych generalnych właściwości we wszystkich klasach pochodnych.

Na przykład wspólną cechą wyprowadzania wektorów *VECT* i *VECT*3*D* jest ujęcie wartości ich współrzędnych w nawiasy. Można to zagwarantować, opierając klasę *VECT* na klasie abstrakcyjnej o przykładowej nazwie *VECT*0. Funkcję *operator*<< zdefiniuje się tylko w klasie *VECT*0 (definicje z klas *VECT* i *VECT*3*D* zostaną usunięte). W klasach *VECT* i *VECT*3*D* zostaną zdefiniowane funkcje *out*( ), które wyprowadzą składowe wektora swojej klasy. Wymagana w klasie *VECT*0 funkcja *out*( ) będzie czysto wirtualna. W podobny sposób można by oprogramować wprowadzanie (*operator*>>).

```
class VECT0 {
   public:
     virtual void out(ostream&)=0;          // funkcja czysto wirtualna
     friend ostream &operator<<(ostream&, VECT0&);
                          // funkcja wspólna dla wszystkich klas pochodnych
   };

class VECT: public VECT0 {
      . . .
   public:
      . . .
      void out(ostream&);                   // funkcja wirtualna w klasie VECT
   }
```

```
class VECT3D: public VECT {
      . . .
   public:
      . . .
      void out(ostream&);          // funkcja wirtualna w klasie VECT3D
   }

ostream &operator<<(ostream &wy, VECT0 &A)
    {wy<<'(';
       A.out(wy);        // aktywacja funkcji out z klasy aktualnego obiektu A
       return wy<<')';
    }
void VECT::out(ostream &wy)
      {wy<<x<<','<<y;}

void VECT3D::out(ostream &wy)
      {wy<<x<<','<<y<<','<<z;}
 . . .
```

Przykład 4.6

Przyjmijmy, że chcemy oprogramować klasę okna. Na wstępie nie zakładamy fizycznej reprezentacji tego okna. Nie decydujemy, czy ma to być okno graficzne lub tekstowe na ekranie, czy też pewna organizacja zapisu w pamięci, na dysku lub na innym medium. Nie decydujemy też, jaki ma być wygląd tego okna. Zakładamy tylko jego prostokątny kształt.

Taki prostokąt będzie miał swoje położenie opisane współrzędnymi $(x1, y1)$ lewego górnego rogu i $(x2, y2)$ prawego dolnego rogu. Prostokąt musi spełniać ograniczenia $0 < x1 \le x2 \le Xmax$ oraz $0 < y1 \le y2 \le Ymax$ i może zmieniać swoje położenie oraz wymiary. Prostokąt może być widoczny lub nie. Po usunięciu prostokąta należy odtworzyć przykrytą przez niego powierzchnię.

Poniższy przykładowy program ilustruje zastosowanie klas abstrakcyjnych (fundamentalnych). Taką klasą jest w nim klasa *Prost*.

Definicja klasy abstrakcyjnej

```
#include <mem.h>              // potrzebny prototyp funkcji movmem

class Graf;                   // aby zaprzyjaźnić z klasą Prost
```

```
class Prost {
   protected:
      friend Graf;              // dostęp do Xmax, Ymax w klasie Graf
      static int Xmax, Ymax;
      int x1,y1,x2,y2;
      int Lenbuf;
      char *buf;
      virtual int buflen()=0;       // oblicza pojemność bufora
      virtual void wstaw()=0;       // uwidacznia obiekt
      virtual void usun()=0;        // odtwarza to, co obiekt zasłonił
      void korekta();
   public:
      Prost(int x1=1,int y1=1,int x2=Xmax,int y2=Ymax):
         x1(x1),y1(y1),x2(x2),y2(y2),Lenbuf(0),buf(NULL)
        {korekta();}
      Prost(Prost &);
      virtual ~Prost() {if(buf) delete buf; buf=NULL;}
      Prost &operator=(Prost&);
      Prost &operator-();
      Prost &operator+();
      Prost &operator>=(int dx)
        {przesun(dx, 0); return *this;}
      Prost &operator^=(int dy)
        {przesun(0, dy); return *this;}
      void przesun(int, int);
      void zmien(int,int,int,int);
   };
```

Definicje pól statycznych

```
int Prost::Xmax=80, Prost::Ymax=25;
```

Konstruktor kopiujący

```
Prost::Prost(Prost &A):
        x1(A.x1),y1(A.y1),x2(A.x2),y2(A.y2),
        Lenbuf(A.Lenbuf)
      { if(A.buf)
           movmem(A.buf, buf=new char[Lenbuf], Lenbuf);
        else buf=NULL;
      }
```

Operator przypisania

```
Prost &Prost::operator=(Prost &A)
      { x1=A.x1;
        y1=A.y1;
        x2=A.x2;
        y2=A.y2;
        Lenbuf=A.Lenbuf;
        if(A.buf)
          movmem(A.buf, buf=new char[Lenbuf], Lenbuf);
          else buf=NULL;
        return *this;
      }
```

Operator ukrycia obiektu (obiekt przestaje być widoczny)

Jeśli obiekt jest widoczny (*buf*≠*NULL*), to odtwarza się to, co on przysłonił i usuwa bufor pamiętający zasłonięty fragment.

```
Prost &Prost::operator-()
      { if(buf) {usun();
                 if(buf) {delete buf; buf=NULL;}
                }
        return *this;
      }
```

Operator pokazania obiektu

Jeśli obiekt nie jest widoczny (*buf*==*NULL*) oraz ma on zdolność zasłaniania (*Lenbuf*≠0), to stwórz bufor i pokaż obiekt zapamiętując to, co on zasłania w buforze.

```
Prost &Prost::operator+()
      { if(!Lenbuf) Lenbuf=buflen();
        if(!buf) buf=new char[Lenbuf];
        if(buf) wstaw();
        return *this;
      }
```

Przesunięcie obiektu

Jeśli przesunięcie nie jest zerowe, to ewentualnie skoryguj przesunięcie, tak aby obiekt nie wyszedł poza dozwolony obszar, usuń obiekt (bez usuwania bufora), gdy

jest on widoczny, zmień położenie obiektu i pokaż go jeśli był widoczny (posiada bufor).

```
void Prost::przesun(int dx, int dy)
      { if(!dx && !dy) return;
        if(x1+dx<=0) dx=1-x1;
          else
               if(x2+dx>Xmax) dx=Xmax-x2;
        if(y1+dy<=0) dy=1-y1;
          else if(y2+dy>Ymax) dy=Ymax-y2;
        if(buf) usun();
        x1+=dx; y1+=dy;
        x2+=dx; y2+=dy;
        if(buf) wstaw();
      }
```

Zmiana położenia i wymiarów

Usuń obiekt wraz z jego buforem (–**this*;). Zmień współrzędne i ewentualnie je skoryguj. Oblicz nową pojemność bufora. Jeśli obiekt był widoczny, to go pokaż (+**this*;).

```
void Prost::zmien(int X1, int Y1, int X2, int Y2)
      { int widoczny=buf!=NULL;
        -*this;
        x1=X1;
        y1=Y1;
        x2=X2;
        y2=Y2;
        korekta();
        Lenbuf=buflen();
        if(widoczny) +*this;
      }
```

Korekta współrzędnych prostokąta, tak aby spełniały ograniczenia

```
void Prost::korekta()
      { int xy;
        if(x2<x1)  {xy=x1; x1=x2; x2=xy;}
        if(x1<1)    x1=1;
        if(x2>Xmax) x2=Xmax;
        if(y2<y1)  {xy=y1; y1=y2; y2= xy;}
        if(y1<1)    y1=1;
```

```
        if(y2>Ymax) y2=Ymax;
      }
```

Definicja klasy okna na ekranie tekstowym

```
#include <conio.h>

class Okno: public Prost {
   protected:
      int tlo;
      int buflen()
          {return 2*(x2-x1+1)*(y2-y1+1);}
      void wstaw();
      void usun();
   public:
      Okno(int=1,int=1,int=80,int=25,int=0);
      ~Okno() {-*this;}
   };

Okno::Okno(int x1, int y1, int x2, int y2, int c):
                              Prost(x1,y1,x2,y2),tlo(c)
      { Lenbuf=buflen();
        +*this;
      }
```

Pokazanie prostokąta na ekranie po zapamiętaniu w buforze fragmentu ekranu

```
void Okno::wstaw()
      { gettext(x1,y1,x2,y2,buf);
        window(x1,y1,x2,y2);
        textbackground(tlo);
        clrscr();
      }
```

Odtworzenie zapamiętanego fragmentu ekranu

```
void Okno::usun()
      { puttext(x1,y1,x2,y2,buf);
      }
```

Definicja klasy obsługi ekranu graficznego

```
#include <iostream.h>
#include <graphics.h>
```

```
const char path[]="C:\\bc3\\bgi";

class Graf {
   protected:
      static int N_ob; // licznik obiektów graficznych
   public:
      Graf();
      ~Graf();
   };

int Graf::N_ob=0;

Graf::Graf()
      { if(!N_ob)              // przy pierwszym obiekcie inicjuj grafikę
            {int Driver=DETECT,Mode,e;
             initgraph(&Driver,&Mode,path);
             if((e=graphresult())!=grOk)
                       {cerr<<grapherrormsg(e); return;}
             Prost::Xmax=getmaxx();
             Prost::Ymax=getmaxy();
            }
        N_ob++;
      }

Graf::~Graf()
      { if(--N_ob) return;
        closegraph();          // przy ostatnim obiekcie zamknij grafikę
        Prost::Xmax=80;        // oraz wpisz rozmiary ekranu tekstowego
        Prost::Ymax=25;
      }
```

Definicja klasy rysunku koła

```
class Kolo: public Graf, public Prost {
   protected:
      int r,tlo;               // promień i kolor koła
      int buflen() {return imagesize(x1,y1,x2,y2);}
      void wstaw();
      void usun();
   public:
      Kolo(int=200,int=180,int=100,int=7);
      ~Kolo() {-*this;}
   };
```

```
Kolo::Kolo(int x, int y, int r, int tlo):
                 Prost(x-r,y-r,x+r,y+r),r(r),tlo(tlo)
      { Lenbuf=buflen();
        ++*this;
      }
```

Wykreślenie koła po zapamiętaniu powierzchni kwadratu opisanego na tym kole

```
void Kolo::wstaw()
      { getimage(x1,y1,x2,y2,buf);
        setfillstyle(SOLID_FILL,tlo);
        fillellipse(x1+r,y1+r,r,r);
      }
```

Ukrycie koła przez odtworzenie zapamiętanego prostokąta (kwadratu)

```
void Kolo::usun()
      { if(!buf) return;
        putimage(x1,y1,buf,COPY_PUT);
      }
```

Przykład użycia zdefiniowanych klas

```
#include <dos.h>

main()
   { Okno P, P1(10,10,50,18,RED), P2(35,12,60,22,GREEN);
     getch();                    // widoczne są dwa prostokąty na czarnym tle
     -P2; -P1;                          // usuń kolejno prostokąty z ekranu
     +P2; +P1;             // umieść prostokąt zielony i na wierzchu czerwony
     for(int i=0;i<40;i++,delay(20)) P1>=1;          // presuwaj P1
     -P1; -P2;                          // usuń kolejno prostokąty z ekranu
     +P1; +P2;             // umieść prostokąt czerwony i na wierzchu zielony
     P2.zmien(25,8,45,23);          // zmień kształt prostokąta zielonego P2
     getch();
     -P2;                                          // usuń zielony prostokąt P2
     getch();
      { Kolo K1, *K2=new Kolo(300,260,120,BLUE);
        getch();                  // szare i niebieskie koło na czarnym tle
        for(i=0;i<20;i++) *K2>=10; // koło niebieskie wędruje w prawo
        for(i=0;i<25;i++) *K2>=-10; // koło niebieskie wędruje w lewo
        getch();
        delete K2;                                 // koło niebieskie znika
        getch();
```

```
    }                          // destruktor koła szarego przywraca tryb tekstowy
  return 0;
}       // destruktor prostokąta P odtwarza ekran sprzed uruchomienia programu
```

Przykład 4.7

Chcemy oprogramować klasę *Menu* w postaci prostokąta z tytułem i kilkoma opcjami do wyboru. Ta klasa ma służyć do tworzenia programowego menu. Na wstępie nie zakładamy fizycznej reprezentacji tego menu ani też jego wyglądu. Przyjmujemy, że będzie ono miało właściwości prostokątnego okna z poprzedniego przykładu, tak więc klasa *Menu* będzie dziedziczyć klasę *Prost*. Przyjmujemy też, że sterowanie wyborem i przesuwaniem menu będzie się odbywać za pomocą klawiatury.

Poniższy przykładowy program ilustruje zastosowanie klas abstrakcyjnych (fundamentalnych). Taką klasą jest w nim klasa *Menu*. Dopiero obiekty klas pochodnych *Menu_T* oraz *Menu_G* mają swoją reprezentację fizyczną. Zauważmy, że abstrakcyjną klasę *Menu* wraz z jej funkcjami można oprogramować i przetestować zanim zostanie zdefiniowana np. klasa *Menu_G* wraz z jej wirtualnymi funkcjami.

Program ten zostanie przedstawiony z podziałem na trzy pliki: nagłówkowy (*prog*4*x.h*), z definicjami funkcji (*prog*4*x.cpp*) oraz plik z programem (*prog*47.*cpp*). Definicja klasy *Prost* razem z definicjami swoich funkcji została pominięta, ponieważ jest ona taka sama jak w przykładzie poprzednim.

Definicje klas – plik **prog4x.h**.

```
#include <mem.h>

class Graf;

class Prost {
        // Definicja klasy jak w przykładzie 4.6
  };

int Prost::Xmax=80, Prost::Ymax=25;

#include <iostream.h>
#include <graphics.h>

const char path[]="C:\\bc3\\bgi";

class Graf {
  protected:
    static int N_ob, ax, ay;        // dodatkowe zmienne skalujące
  public:
```

```
    Graf();
    ~Graf();
  };

int Graf::N_ob=0, Graf::ax, Graf::ay;
```

Klasa abstrakcyjna Menu

```
class Menu:public Prost {
  protected:
    char *tyt, *txt;
    int tlo, nt, poz;
    virtual int buflen()=0;      // pojemność bufora na zapamiętanie tła
    virtual void wstaw()=0;                    // uwidocznienie obiektu
    virtual void usun()=0;       // odtworzenie tła (usunięcie obiektu)
    virtual void ramka()=0;                    // wykreślenie ramki
    virtual void zmien(int)=0;   // zmiana prezentacji tekstu (kolory)
  public:
    Menu():Prost(1,1,80,25),txt(NULL),tyt(NULL),
                                        poz(0),tlo(0)
          {Lenbuf=4000;}
    Menu(char*, char*, int=1, int=1, int=1, int=1);
    operator int();
};
```

Klasa aplikacyjna na ekranie tekstowym

```
class Menu_T: public Menu {
  protected:
    void wstaw();
    void zmien(int);
    void usun();
    void ramka();
    int buflen()
        {return 2*(x2-x1+1)*(y2-y1+1); }
  public:
    Menu_T() { }
    Menu_T(char *t1,char *t2, int x=1, int y=1,
          int p=1, int c=1): Menu(t1, t2, x, y, p, c)
        { }
    ~Menu_T()
```

```
        { usun(); }
   };
```

Klasa aplikacyjna na ekranie graficznym

```
class Menu_G: public Graf, public Menu {
   private:
     void wstaw();
     void zmien(int);
     void usun();
     void ramka();
     int buflen()
           {return imagesize(x1, y1, x2+ax, y2+ay); }
   public:
   Menu_G(char*, char*, int=1, int=1, int=1, int=1);
   ~Menu_G() { usun(); }
   };
```

Definicje funkcji – plik **prog4x.cpp**.

Definicje funkcji klasy *Prost* jak w przykładzie 4.6

```
#include <graphics.h>

Graf::Graf()
   {int Driver=DETECT, Mode, e;
     if(!N_ob)
     {initgraph(&Driver, &Mode, path);
      if((e=graphresult())!=grOk)
                        {cerr<<grapherrormsg(e); return;}
      Prost::Xmax=getmaxx();
      Prost::Ymax=getmaxy();
      ax=getmaxx()/80+1;            // skaluj w poziomie
      ay=getmaxy()/25;              // skaluj w pionie
     }
     N_ob++;
   }

Graf::~Graf()
   { if(--N_ob) return;
     closegraph();
     Prost::Xmax=80; Prost::Ymax=25;
   }
```

Definicje funkcji klasy *Menu*

```
#include <conio.h>
#include <string.h>

Menu::Menu(char *q,char *p,int x,int y,int poz,
     int tlo):Prost(x, y),tyt(q),txt(p),poz(poz),tlo(tlo)
   { int m=strlen(tyt)+2;
     x1=x;
     y1=y;
     for(nt=0; (q=strchr(p,'\n'))!=NULL; p=q+1, nt++)
                            if(m < q-p)  m=(int)(q-p);
     m+=3;
     x2=x1+m;
     y2=y1+nt+1;
     korekta();
   }

Menu::operator int()
   { int k,c;
     wstaw();
     zmien(poz);
     k=poz;
     do
       { c=getch();
         switch(c)
         {case 27: return 0;
          case 13:
          case 10: poz=k; c=27; break;
          case 0:  c=getch();
                   zmien(k);
                   switch(c)
                        {case 72:if(k>1) k--; else k=nt;
                                 break;
                         case 73:k=1;
                                 break;
                         case 80:if(k<nt) k++; else k=1;
                                 break;
                         case 81:k=nt;
                                 break;
```

```
                        case 141:
                        case 152:przesun(0,-1);
                                 break;
                        case 145:
                        case 160:przesun(0, 1);
                                 break;
                        case 116:
                        case 157:przesun(1,0);
                                 break;
                        case 115:
                        case 155:przesun(-1,0);
                                 break;
                      }
                  zmien(k);
                  break;
        }
      } while(c!=27);
    return poz;
  }
```

Definicje funkcji klasy *Menu_T*

```
void Menu_T::wstaw()
  {
    if(!Lenbuf) Lenbuf=buflen();
    if(!buf) buf=new char[Lenbuf];
    gettext(x1, y1, x2, y2, buf);
    textbackground(tlo);
    window(x1, y1, x2, y2);
    clrscr();
    if(!txt) return;
    ramka();
    gotoxy(1, 2);
    for(char *p=txt; *p; p++)
          if(putch(*p)=='\n') putch('\r');
    if(!tyt) return;
    gotoxy(((x2-x1-3-strlen(tyt))>>1), 1);
    putch(' ');
    cputs(tyt);
    putch(' ');
  }
```

```
void Menu_T::zmien(int poz)
   {
     int i;
     int n=(x2-x1-1)<<1;
     char *t=new char[n];
     gettext(x1+1, y1+poz, x2-1, y1+poz, t);
     for(i=1; i<n; i+=2)
                       t[i]^=0x77;
     puttext(x1+1, y1+poz, x2-1, y1+poz, t);
     delete t;
   }

void Menu_T::usun()
   {
     if(!buf) return;
     puttext(x1, y1, x2, y2, buf);
     delete buf;
     buf=NULL;
   }

void Menu_T::ramka()
   { int i;
     window(x1, y1, x2+1, y2);
     gotoxy(1,1);
     putch('╔');          // kody znaków: 200, 201, 205, 188, 187, 186
     for(i=x1+1; i<x2; i++) putch('=');
     putch('╗');
     for(i=2; i<=y2-y1; i++)
       {gotoxy(1, i);
        putch('║');
        gotoxy(x2-x1+1, i);
        putch('║');
       }
     gotoxy(1, i);
     putch('╚');
     for(i=x1+1; i<x2; i++) putch('=');
     putch('╝');
     window(x1+2, y1, x2, y2);
   }
```

Definicje funkcji klasy *Menu_G*

```
Menu_G::Menu_G(char *t1,char *t2,int x,int y,
                int p,int c):Menu(t1, t2, x, y, p, c)
   { x1*=ax;
     y1*=ay;
     x2*=ax;
     y2*=ay;
   }

void Menu_G::wstaw()
   { char *p1, *p;
     int i;
     if(!Lenbuf) Lenbuf=buflen();
     if(!buf) buf=new char[Lenbuf];
     getimage(x1, y1, x2+ax, y2+ay, buf);
     setfillstyle(SOLID_FILL, tlo);
     bar(x1, y1, x2+ax, y2+ay);
     ramka();
     for(i=2, p1=p=txt; *p; p++)
        if(putch(*p)=='\n')
        {*p='\0';
          outtextxy(x1+ax, y1+i*ay, p1);
          *p='\n';
          p1=p+1;
          i++;
        }
     moveto(x1+((x2-x1-ax*(3+strlen(tyt)))>>1),
                 y1+(ay>>1));
     outtext(" ");
     outtext(tyt);
     outtext(" ");
   }

void Menu_G::zmien(int poz)
   { char *t=new char[imagesize(x1+ax, y1, x2, y1+ay)];
     int d=(ay>>1)+(ay>>2);
     getimage(x1+ax, y1+poz*ay+d, x2, y1+poz*ay+ay+d, t);
     putimage(x1+ax, y1+poz*ay+d, t, NOT_PUT);
     delete t;
   }
```

```
void Menu_G::usun()
   { if(!buf) return;
     putimage(x1, y1, buf, COPY_PUT);
     delete buf;
     buf=NULL;
   }

void Menu_G::ramka()
   { rectangle(x1+(ax>>1), y1+(ay>>2),
               x2+(ax>>1), y2+(ay>>1)+(ay>>2));
     line(x1+(ax>>1), y1+ay+(ay>>2),
               x2+(ax>>1), y1+ay+(ay>>2));
   }

void prostokat(int x1,int y1,int x2,int y2,int kolor)
   { window(x1,y1,x2,y2);
     textbackground(kolor);
     clrscr();
   }
```

Przykładowy program główny – plik **prog47.cpp**.

```
#include <graphics.h>
#include <conio.h>

main()
   { int petla;
     Menu_T AA, A("MENU",
                  "1. Ekran tekstowy\n"
                  "2. Ekran graficzny\n"
                  "Koniec (Esc)\n", 20, 7);
     Menu *B;
     +AA;
     do
       { switch(A)        // aktywacja głównego menu tekstowego
         {case 1:B=new Menu_T("EKRAN TEKSTOWY",
            "Kwadrat\nMinus\nPlus\nKoniec (Esc)\n", 25, 9);
              petla=1;
              do
               {+A;
                switch(*B)      // aktywacja menu tekstowego
```

```
     {case 1:prostokat(1, 1, 80, 25, BLACK);
             prostokat(20, 5, 60, 20, RED);
             break;
      case 3:prostokat(1, 1, 80, 25, BLACK);
             prostokat(38, 2, 42, 22, GREEN);
             prostokat(5, 12, 75, 13, GREEN);
             break;
      case 2:prostokat(1, 1, 80, 25, BLACK);
             prostokat(5, 12, 75, 13, MAGENTA);
             break;
      case 4:
     default:petla=0;  break;
     }
     if(petla) getch();
     prostokat(1, 1, 80, 25, BLACK);
    } while(petla);
 delete B;
 break;
case 2:B=new Menu_G("EKRAN GRAFICZNY",
     "Kolo\nElipsa\nKoniec (Esc)\n", 30, 11);
       petla=1;
       setbkcolor(BLACK);
       setcolor(WHITE);
       do {
        switch(*B)       // aktywacja menu graficznego
        {case 1:clearviewport();
            setfillstyle(SOLID_FILL, YELLOW);
            fillellipse(250, 150, 60, 60);
            break;
         case 2:clearviewport();
            setfillstyle(SOLID_FILL, MAGENTA);
            fillellipse(250, 150, 90, 40);
            break;
         case 3:default: petla=0;  break;
        }
       if(petla) getch();
       clearviewport();
      } while(petla);
     delete B;
     break;
case 3:
```

```
          default:B=NULL;
                  break;
         }
      } while(B);
     return 0;
   }
```

Przykład 4.8

Chcemy oprogramować abstrakcyjną klasę *Lista* operacjami wprowadzania, wyprowadzania, tworzenia sumy dwu list. Na bazie klasy *Lista* utworzyć klasę *Lista_O* zawierającą listę osób – obiektów klasy *Osoba*. W klasie abstrakcyjnej zostaną zadeklarowane następujące funkcje czysto wirtualne:

- *Lista *nowy*( ); alokująca obiekt przyszłej klasy pochodnej (tu klasy *Lista_O*),
- *int rozmiar*( ); dająca rozmiar elementu listy (np. obiektu klasy *Osoba*) w bajtach,
- *void kopy*(*void**, *int*); kopiująca określony numerem element listy pod wskazaną pamięć,
- *void zwolnij*(*int*); usuwająca określony numerem element listy,
- *void get*(*istream&*, *int*); wprowadzająca element listy,
- *void put*(*ostream&*, *int*); wyprowadzjąca element listy.

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
```

Abstrakcyjna klasa *Lista*

```
class Lista {                                          // lista abstrakcyjna
 protected:
   int tmp, N;                                         // N – rozmiar tablicy
   char *A;                                            // tablica elementów
   virtual Lista *nowy() const=0;                      // alokacja przyszłej listy
   virtual void put(ostream&,int)=0;                   // wydruk elementu listy
   virtual void get(istream&,int)=0;                   // wprowadzenie elementu listy
   virtual int rozmiar() const=0;                      // rozmiar elementu
   virtual void kopy(void*,int)const=0;                // kopiowanie elementu listy
   virtual void zwolnij(int)=0;                        // zwolnienie elementu listy
 public:
   Lista():tmp(0), N(0), A(NULL) { }
   virtual ~Lista(){if(A) delete A; A=NULL; N=0;}
   Lista(const Lista&);
```

```
   Lista &operator=(const Lista&);
   Lista &operator+(const Lista&) const;      // połączenie dwu list
   void *operator[](int i) {return A+i*rozmiar();}
   friend ostream &operator<<(ostream&,Lista&);
   friend istream &operator>>(istream&,Lista&);
};
Lista::Lista(const Lista &x):N(x.N),tmp(0)
   { char *p;
     int i, k=x.rozmiar();                    // rozmiar elementu listy x
     if(x.A)
     {p=A=new char[N*k];                              // alokacja tablicy
      for(i=0; i<N; i++, p+=k) x.kopy(p, i);
     } else A=NULL;
    if(x.tmp) delete &x;
   }

Lista &Lista::operator=(const Lista &x)
   { char *p;
     int i, k=x.rozmiar();
     if(A) {for(i=0; i<N; i++) zwolnij(i);
            delete A};
     N=x.N;
     if(x.A)
       { p=A=new char[N*k];
         for(i=0; i<N; i++, p+=k) x.kopy(p, i);
       } else A=NULL;
     if(x.tmp) delete &x;
     return *this;
   }

Lista &Lista::operator+(const Lista &x) const
   { Lista *t=nowy();          // utworzenie obiektu klasy lewego argumentu
     char *p;
     int i, k=x.rozmiar();
     t->tmp=1;
     t->N=N+x.N;
     p=t->A=(t->N)?new char[t->N*k]:NULL;
     for(i=0;   i<N; i++, p+=k) kopy(p,i);
     for(i=0; i<x.N; i++, p+=k) x.kopy(p,i);
     if(x.tmp) delete &x;
```

```
    if(tmp) delete this;
    return *t;
  }

ostream &operator<<(ostream &wy, Lista &x)
  { for(int i=0; i<x.N; i++)
        {x.put(wy, i); wy<<endl;}
    if(x.tmp) delete &x;
    return wy;
  }

istream &operator>>(istream &we, Lista &x)
{  if(x.A) { for(int i=0; i<N; i++) x.zwolnij(i);
             delete x.A;}                    // skasowanie starej tablicy
   do
   {if(we==cin) cerr<<"N= ";
    we>>x.N;                                 // liczba elementów listy
   }while((we?0:we.clear(),we.ignore(80,'\n'),1)||x.N<1);
   x.A=new char[x.N*x.rozmiar()];            // alokacja tablicy
   for(int i=0;i<x.N;i++) x.get(we,i);       // wprowadzanie elementów
   return we;
   }
```

Powyższa abstrakcyjna klasa *Lista* obsługuje podstawowe operacje wykonywane na liście elementów. Operacje te oprogramowano bez znajomości typu elementu listy. Bazując na tej abstrakcyjnej klasie można teraz tworzyć klasy list konkretnych obiektów. Dla przykładu przyjmijmy listę osób. W tym celu należy najpierw zdefiniować klasę *Osoba*, która będzie klasą elementu listy. Dla uproszczenia przyjmijmy, że klasa *Osoba* ma tylko trzy pola tekstowe o ustalonej długości. Wszelkie operacje na obiektach tej klasy wykonują jej metody (funkcje) oraz funkcje zaprzyjaźnione.

Definicja klasy elementu listy – klasy *Osoba*.

```
class Osoba {                                // element listy
    char imie[16], nazwisko[24], tel[20];
  public:
    friend istream &operator>>(istream&, Osoba&);
    friend ostream &operator<<(ostream&, Osoba&);
  };

istream &operator>>(istream &we, Osoba &o)
```

```
   { we>>setw(16)>>o.imie>>setw(24)>>o.nazwisko>>
        setw(22)>>o.tel;
     o.imie[15]=o.nazwisko[23]=o.tel[19]='\0';
     return we;
   }
ostream &operator<<(ostream &wy, Osoba &o)
   { wy.setf(ios::left);
     wy<<setw(22)<<o.tel<<setw(24)<<o.nazwisko<<
        setw(16)<<o.imie;
     return wy;
   }
```

Teraz przystąpimy do konstruowania klasy listy osób o nazwie *Lista_O*. W tej klasie elementem listy jest obiekt klasy *Osoba*. Ponieważ obiekty klasy *Osoba* są obsługiwane przez swoje funkcje, klasa *Lista_O* nie zależy od klasy *Osoba*. Na przykład funkcja *kopy* nie zależy od tego jakie pola należy kopiować w klasie *Osoba*, ponieważ aktywuje operator przypisania z tej klasy.

Definicja listy osób – klasy *Lista_O*.

```
class Lista_O:public Lista {        // lista obiektów klasy Osoba
   protected:
     Lista *nowy() const {return new Lista_O;}
     void put(ostream&, int);
     void get(istream&, int);
     int rozmiar() const {return sizeof(Osoba);}
     void kopy(void *p, int i) const
                 {*(Osoba*)p=((Osoba*)A)[i];}
     void zwolnij(int i) {(Osoba*)A)[i].~Osoba();}
   public:
     Lista_O(){}
     Lista_O(const Osoba &o):Lista()
                      { N=1; A=(char*)new Osoba(o);}
     Lista_O(const Lista &x):Lista(x) {}
     ~Lista_O();
     Osoba &operator[](int i){return ((Osoba*)A)[i];}
   };

Lista_O:~Lista_O()
   {for(int i=0; i<N; i++) zwolnij(i);}
```

```
void Lista_O::put(ostream &wy, int i)
   { wy<<((Osoba*)A)[i];      // aktywacja operatora z klasy Osoba
   }

void Lista_O::get(istream &we,int i)
   {if(we==cin)
    if(!i)
     cerr<<"Imie       Nazwisko          Telefon"<<endl;
    we>>((Osoba*)A)[i];      // aktywacja operatora z klasy Osoba
   }

main()
   { clrscr();
     Lista_O A,B,C;
     cerr<<"Lista A\n";
     cin>>A;
     cerr<<"Lista B\n";
     cin>>B;
     C=B+A;
     cout<<"\nSuma list B+A\n"<<(B+A);
     return 0;
   }
```

Zadania laboratoryjne

4.12. W przykładzie 4.5 dopisać klasę abstrakcyjną

```
class VBaza {
 protected:
   virtual void putv(ostream&)=0;
   virtual void getv(istream&)=0;
 public:
   friend ostream &operator<<(ostream&, VBaza&);
   friend istream &operator>>(istream&, VBaza&);
   virtual ~VBaza();
 };
```

Klasę *VBaza* uczynić klasą bazową klasy *VECT*. Dopisać konieczne definicje funkcji dla klasy *VBaza*, np:

```
ostream &operator<<(ostream &c,VBaza &w)
      {c<<'('; w.putv(c); return c<<')';
       }
```

```
VBaza::~VBaza() {cerr<<"Dest0()";}
```

Z klas *VECT* i *VECT3D* usunąć deklaracje i definicje funkcji *operator*<< oraz *operator*>>. Skompilować program; zastanowić się nad błędami kompilacji. Definicje funkcji *putv* i *getv* w klasach *VECT* i *VECT3D* ująć w komentarze. Skompilować program. Dlaczego definicje obiektów klas *VECT* i *VECT3D* są błędne? Usunąć komentarze w klasie *VECT*. Skompilować i uruchomić program. Jak wyprowadzane są obiekty klasy *VECT3D* i dlaczego? Usunąć komentarze w klasie *VECT3D* i sprawdzić poprawność działania programu.

4.13. W klasie *Prost* opisanej w przykładzie 4.6 zdefiniuj funkcję o nagłówku *Prost &operator+*(*int*), która zmieni rozmiary prostokąta, zmniejszając współrzędne jego lewego górnego rogu oraz zwiększając współrzędne prawego dolnego rogu o wartość prawego argumentu definiowanej funkcji operatorowej. W definicji wykorzystaj istniejące funkcje wirtualne. Przetestuj tę funkcję na ekranie tekstowym i graficznym.

4.14. W klasie *Lista* z przykładu 4.8 zdefiniuj sortowanie listy. Funkcja porównująca obiekty listy będzie funkcją wirtualną definiowaną dopiero w klasie pochodnej. Zdefiniuj taką funkcję w klasie *Lista_O* i przetestuj sortowanie osób według nazwisk i imion.

4.15. Bazując na klasie *Lista* z przykładu 4.8 napisz program obsługujący listę towarów. Niech operator dodawania towaru do listy dopisuje ten towar, gdy go nie ma na liście lub zwiększa jego ilość (pole w klasie *Towar*), gdy jest on na liście. Podobnie niech działa operator odejmowania skreślając towar z listy, gdy jego ilość osiągnie zero.

4.16. W przykładzie 4.8 w klasie *Osoba* zmień tablice znakowe *nazwisko*, *imie*, *tel* na wskaźniki do znaku (typ *char**). Zdefiniuj odpowiednie konstruktory (też konstruktor kopiujący) alokujące wymagane tablice znakowe, operator przypisania oraz destruktor. Do celów testowania destruktor może wyprowadzać komunikat. Czy w klasie *Osoba_O* należy też zdefiniować destruktor?

Zadania laboratoryjne (kompleksowe z całości materiału)

Podczas kolejnych ćwiczeń poza zadaniami doraźnymi należy tworzyć i doskonalić jeden z programów ilustrujących zastosowanie jednej z poniżej zaproponowanych klas. W każdym programie należy docelowo oprogramować klasę obiektów. Ta klasa powinna bazować na innej klasie i dalej na klasie abstrakcyjnej, z której

powinna dziedziczyć typowe funkcje wirtualne. W klasach (tam gdzie potrzeba) konstruktory powinny alokować wymagane obszary pamięci. W tych klasach należy zdefiniować odpowiednie destruktory, konstruktory kopiujące i operatory przypisania oraz zadbać o dobrą hermetyzację klas – aby użytkownik miał jak najmniejszy dostęp do wnętrza obiektów.

4.17. Oprogramować operacje na macierzach. Uwzględnić (poza dodawaniem, odejmowaniem i mnożeniem) operacje mnożenia macierzy przez liczbę i liczby przez macierz, wprowadzania, wyprowadzania. Zaproponować konwersję do typu *double*. Klasa macierz powinna bazować na klasie wektorów oraz na pewnej klasie abstrakcyjnej. Elementami macierzy niech będą liczby typu *double*. Niech klasa abstrakcyjna zawiera pole wskazujące na tablicę elementów oraz pola całkowite oznaczające: znacznik obiektu tymczasowego, liczbę elementów macierzy (wskazanej tablicy), precyzję i szerokość pola wydruku. Elementy macierzy powinny być pamiętane w jednoindeksowej tablicy tak jak elementy wektora, jednak dostęp do elementów tablicy spoza klasy powinien odbywać się przy użyciu dwu indeksów (wiersza i kolumny) za pośrednictwem przeciążonego operatora indeksacji. Jeżeli w działaniach macierze lub wektory nie mają zgodnych wymiarów, to należy założyć, że brakujące elementy, rzędy lub kolumny są zerowe (uzupełnić mniejszy wymiar do właściwego przez dopisanie elementów zerowych).

4.18. Oprogramować algebrę zbiorów. Klasa zbiorów powinna być niezależna od interpretacji elementów i powinna być bazą dla klas zbioru liczb oraz zbioru nazw (wyrazów). Oprogramować wprowadzanie, wyprowadzanie oraz wszystkie operacje mnogościowe (suma, różnica, różnica symetryczna, należenie, zawieranie, test czy zbiór jest pusty itp.). Zaproponować konwersję to typu *int*. W klasie abstrakcyjnej zbiory będą reprezentowane przez tablice liczb całkowitych. Kolejne elementy zbioru będą reprezentowane przez kolejne bity w tablicy liczb całkowitych. Bit ustawiony (równy jedności) oznacza, że odpowiedni element należy do zbioru. Bit wyzerowany oznacza, że odpowiedni element nie należy do zbioru. Operator wyjścia wyprowadzając zbiór wyprowadzi tylko te elementy, których bity są ustawione. Operator wejścia wczytując element do zbioru ustawi odpowiadający temu elementowi bit. Zbiór jest pusty, jeśli w reprezentującej go tablicy wszystkie liczby są zerowe. W klasach pochodnych należy zdefiniować przyporządkowanie bitów elementom oraz sposób wprowadzania i wyprowadzania elementów. Wszystkie operacje teoriomnogościowe powinny być zdefiniowane w abstrakcyjnej klasie bazowej.

4.19. Oprogramować algebrę wielomianów. Utworzyć abstrakcyjną klasę bazową zawierającą podstawowe właściwości algebry wielomianów oraz operacje wprowadzania i wyprowadzania, konwersję z typu *double*. Przeciążyć operator wywołania funkcji, aby w wyniku dawał wartość wielomianu dla zadanej liczby *x*. Bazując na tej klasie utworzyć klasę wielomianów liczb rzeczywistych oraz klasę wielomianów binarnych (współczynniki wielomianu są binarne, a dodawanie jest dodawaniem modulo 2 bez przeniesienia).

4.20. Obiektem jest lista towarów – tabela zawierająca nazwę towaru, jego cenę, ilość i ewentualnie inne dane. Bazując na klasie *Lista* z przykładu 4.8, oprogramować klasę *Lista_T* z operacjami wprowadzania, wyprowadzania, wyszukiwania, dopisywania, usuwania, zmiany stanu (odejmowanie lub dodawanie towarów i list). Zdefiniować klasę *Towar* opisującą dane towaru i operacje na nich.

5. Bibliografia

[1] Arnush C., *Turbo C++ 4.5 dla Windows w 21 dni*, Warszawa, Oficyna Wydawnicza Read Me, 1996.
[2] Barkakati N., *Borland C++ 4.*, Warszawa, Oficyna Wydawnicza Read Me, 1995.
[3] Barteczko K., *Praktyczne wprowadzenie do programowania obiektowego w języku C++*, Warszawa, Wydawnictwo Lupus, 1993.
[4] Bielecki J., *ANSI C++*, Warszawa, Intersoftland, 1997.
[5] Bielecki J., *Borland C++. Programowanie obiektowe*, Warszawa, Wydawnictwo PLJ, 1991.
[6] Chomicz P., Ulijasz R., *Programowanie w języku C i C++*, Warszawa, Wydawnictwo PLJ, 1992.
[7] Chomicz P., Ulijasz R., *Strumienie w C i C++*, Warszawa, Wydawnictwo PLJ, 1992.
[8] Coad P., Nicola J., *Programowanie obiektowe*, Warszawa, Oficyna Wydawnicza Read Me, 1993.
[9] Coad P., Yourdon F., *Analiza obiektowa*, Warszawa, Oficyna Wydawnicza Read Me, 1993.
[10] Coad P., Yourdon F., *Projektowanie obiektowe*, Warszawa, Oficyna Wydawnicza Read Me, 1994.
[11] Davis S.R., *C++ dla opornych*, Warszawa, Read Me, 1995.
[12] Faison T., *Borland C++ 4.5. Programowanie obiektowe*, Warszawa, Oficyna Wydawnicza Read Me, 1996.
[13] Gręboszl J., *Pasja C++. Szablony, pojemniki i obsługa sytuacji wyjątkowych w języku C++*, Kraków, Oficyna Kallimach, 1997.
[14] Heryk R., Gajewski P., *Borland C++*, Warszawa, Help, 1993.
[15] Hyman M., *Borland C++ dla opornych*, Warszawa, Oficyna Wydawnicza Read Me, 1995.
[16] Kain E., *Od C do C++*, Gliwice, Helion, 1993.
[17] Kerninghan B.W., Ritchie D.M., *Język ANSI-C*, Warszawa, WNT, 1994.
[18] Kisilewicz J., *Język C w środowisku Borland C++*, Wrocław, Oficyna Wydawnicza Politechniki Wrocławskiej, 2000.

[19] Kniat J., *Programowanie obiektowe w języku C++. Wprowadzenie*, Poznań, Politechnika Poznańska, 1995.
[20] Lippman S. B., *Podstawy języka C++*, Warszawa, WNT, 1997.
[21] Majczak A., *Praktyczne programowanie w C++*, Warszawa, Intersoftland, 1993.
[22] Martin J., Odell J.J., *Podstawy metod obiektowych*, Warszawa, WNT, 1997.
[23] Plauer P.J., *Biblioteka standardowa C++*, Warszawa, WNT, 1997.
[24] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-oriented modeling and design*, Prentice-Hall International, Inc. 1991.
[25] Shammas N. C., *Borland C++ 4. Kurs na zderzenie*, Warszawa, Intersoftland, 1993.
[26] Stroustrup B., *Język C++*, Warszawa, WNT, 1994.
[27] Stroustrup B., *Projektowanie i rozwój języka C++*, Warszawa, WNT, 1996.
[28] Stasiewicz A., *C++ Całkiem inny świat*, Gliwice, Helion, 1997.
[29] Zalewski A., *Programowanie w językach C i C++ z wykorzystaniem pakietu Borland C++*, Poznań, NAKOM,1994.