Szablony klas, zastosowanie szablonów w programach

1. Szablony klas i funkcji
2. Szablon klasy obsługującej uniwersalną tablicę wskaźników
3. Zastosowanie metody zwracającej przez return referencję do składowych klasy
4. Tworzenie zbioru rachunków zawierających zakupy różnych produktów
5. Agregacja silna – rola konstruktora kopiującego, przeciążonego operatora= oraz destruktora

Szablony klas, zastosowanie szablonów w programach

1. Szablony klas i funkcji

1.1. Szablony - wiadomości wstępne

Szablon lub inaczej wzorzec (**template**) definiuje rodzinę typów lub funkcji.

deklaracja-wzorca:

template <*lista-argumentów-wzorca*> *deklaracja*

lista-argumentów-wzorca:

argument-wzorca

lista-argumentow-wzorca, argument-wzorca

argument-wzorca:

argument-typu

deklaracja-argumentu

argument-typu:

class *nazwa-typu*

Uwagi:

- ***deklaracja*** w ***deklaracja-wzorca*** deklaruje lub definiuje funkcję lub klasę
- ***argument-typu*** definiuje nazwę typu obowiązującą w zasięgu ***deklaracja-wzorca***.
- ***deklaracja-wzorca*** jest globalna i podlega zwykłym regułom zasięgu i kontroli dostępu, ***nazwa-typu*** obowiązuje w zasięgu ***deklaracja-wzorca*****.**
- W programach wieloplikowych deklaracje szablonów powinny być zawarte w plikach nagłówkowych ze względu na ich wielożywalność!

1.2. Szablony klas

1.2.1. Definiowanie wzorca klasy

Definicja wzorca klasy *wektor* o parametrze *T .*

```
template <class T> class wektor
        {  T * p;
           int ile, rozmiar;
         public:
         wektor (int rozmiar_)
            {  ile = 0;
               p = new T[rozmiar=rozmiar_]; }
         int zakres(int i)
             {  return i>=0 && i<ile; }
          T& element(int i)
             {  return p[i];  }
          T& operator[] (int i)
             {  return element(i); }
        };
```

1.2.2. Deklarowanie wzorca klasy

nazwa-klasy-wzorca:

nazwa-wzorca <lista-arg-wzorca>

lista-arg-wzorca:

arg-wzorca

lista-arg-wzorca, arg-wzorca

arg-wzorca:

wyrażenie

nazwa-typu

nazwa-klasy-wzorca (np. wektor) musi być unikatowa w programie.

Typy ***arg-wzorca*** wyspecyfikowane w ***lista-arg-wzorca*** ***w deklaracji nazwa-klasy-wzorca*** muszą pasować do typów podanych w ***lista-argumentów-wzorca.***

Argumentami ***arg-wzorca*** mogą być: typy, wyrażenia-stałe, adresy obiektów lub funkcji łączonych zewnętrznie lub statyczne składowe klasy.

1.2.3. Przykłady deklaracji klasy generowanej ze wzorca

1) generowanie klas wzorca na podstawie danego szablonu np. wektor

```
wektor<int> w1(20)
wektor<complex> w2(30)
typedef wektor<complex> cwekt;    //cwekt synonimem wektor<complex>
cwekt w3(40);
w2[1] = w3.element(4) = complex(7, 8);
```

2) używanie *nazwy-klasy-wzorca* wszędzie tam, gdzie można używać nazwy klasy jako argumentu wzorca:

```
class wektor<Punkt*>;
wektor<Ramka>* biezaca_ramka;
class wektor_fig : public wektor<Punkt*> {.......};
```

1.2.4. Definiowanie metod klasy wzorca na zewnątrz bloku szablonu

Metoda klasy wzorca jest niejawnie funkcją wzorca, której argumentami są argumenty wzorca jej klasy:

```
template<class T>  T& wektor<T>::operator[](int i) {.....}
```

1.3. Szablony funkcji

Wzorzec funkcji specyfikuje, jak można konstruować poszczególne funkcje. Specyfikuje on nieograniczony zbiór (przeciążonych) funkcji.

Każdy ***argument-wzorca*** podany w ***liście-argumentów*** wzorca musi być użyty jako argument we wzorcu funkcji.

1.3.1. Generowanie funkcji wzorca

np. rodzinę funkcji sortujących można zadeklarować:

```
template<class T> void sortuj(wektor<T>& w)
{  tutaj można używać metod obiektu w klasy wektor<T> oraz typu T  }

wektor<complex> wc(100);
wektor<int> wi(200);
void f (wektor <complex>& wc, wektor<int>& wi)
    {
        sortuj(wc);
        sortuj(wi);
    }
```

Błędne definicje wzorca funkcji

W obu przykładach w liście parametrów wzorca funkcji brakuje argumentów wzorca

(1) **template**<**class** T> T* utworz();

(2) **template**<**class** T> **void** f()

{ T a..... }

1.3.2. Algorytm rozróżniania funkcji wzorca i innych funkcji o tej samej nazwie

1) Poszukiwanie dokładnie pasującej funkcji

- Poszukiwanie wzorca funkcji, z którego można wygenerować funkcję, którą można wywołać z dokładnym dopasowaniem
- zwykły algorytm rozróżnienia przeciążonych funkcji

Rozróżnianie przeciążonych funkcji wzorca i innych funkcji (2)

```
(2) template<class T> T max (T a, T b)
      {   return a>b ? a : b;  }

void f (int a, int b, char c, char d)
 {
   int m1 = max (a, b);  // (2)max(int, int),
   int m2 = max (c, d);  // (2)max(char, char),
   int m3 = max (a, c);  // (2)błąd: nie można wykonać max(int, int(char))
 }
```

```
(2)(3) template<class T> T max (T a, T b)
        {  return a>b ? a : b;  }

    int max (int, int);          // (2)funkcja domyślnie wygenerowana z wzorca

    void f (int a, int b, char c, char d)
       {
          int m1 = max (a, b);       // (2)max(int, int),
          int m2 = max (c, d);       // (2)max(char, char),
          int m3 = max (a, c);       // (3)można wywołać max(int, int),
                    // ponieważ dokonuje się konwersja do listy parametrów
        }           // (int, int(char)) funkcji wygenerowanej ze wzorca
```

```
(2) template<class T1, class T2> T1 max (T1 a, T2 b)
        {  return a>b ? a : b;  }

    void f (int a, int b, char c, char d)
       {
          int m1 = max (a, b);       // (2)max(int, int),
          int m2 = max (c, d);       // (2)max(char, char),
          int m3 = max (a, c);       // (2)max(int, char),
       }
```

Przykład - wzorzec funkcji sortującej bąbelkowo

```
const n= 5;
template <class T> void sortuj (T *w, const int ile);
void main()
{   //wygenerowane dwa egzemplarze kodu funkcji sortuj
    int t1[n] ={2,1,4,3,5};
    float t2[n] = {3.2, 1.2, 5.3, 6.4, 4.5};
    sortuj(t1, n); //wywołanie kodu funkcji sortuj dla tablicy elementów typu int
    sortuj(t2, n); //wywołanie kodu funkcji sortuj dla tablicy elementów typu float
}
// wzorzec funkcji, który stanie się kodem konkretnej funkcji, jeżeli kompilator
// zauważy wywołanie wzorca z konkretnymi danymi zamiast parametru T
template <class T> void sortuj (T *w, const int ile)
 {
    for (int i= 0; i < ile-1; i++)
     for (int j=0; j <ile-1-i; j++)
          if (w[j] > w[j+1])
           {        T pom = w[j];
                    w[j] = w[j+1];
                    w[j+1]= pom;  }
 }
```

Szablony klas, zastosowanie szablonów w programach

1. **Szablony klas i funkcji**
1. **Szablon klasy obsługującej uniwersalną tablicę wskaźników**

C:\Settings\dydaktyka\Programowanie_obiektowe\w8_1\KOL2.h

main1.cpp | KOL2.h

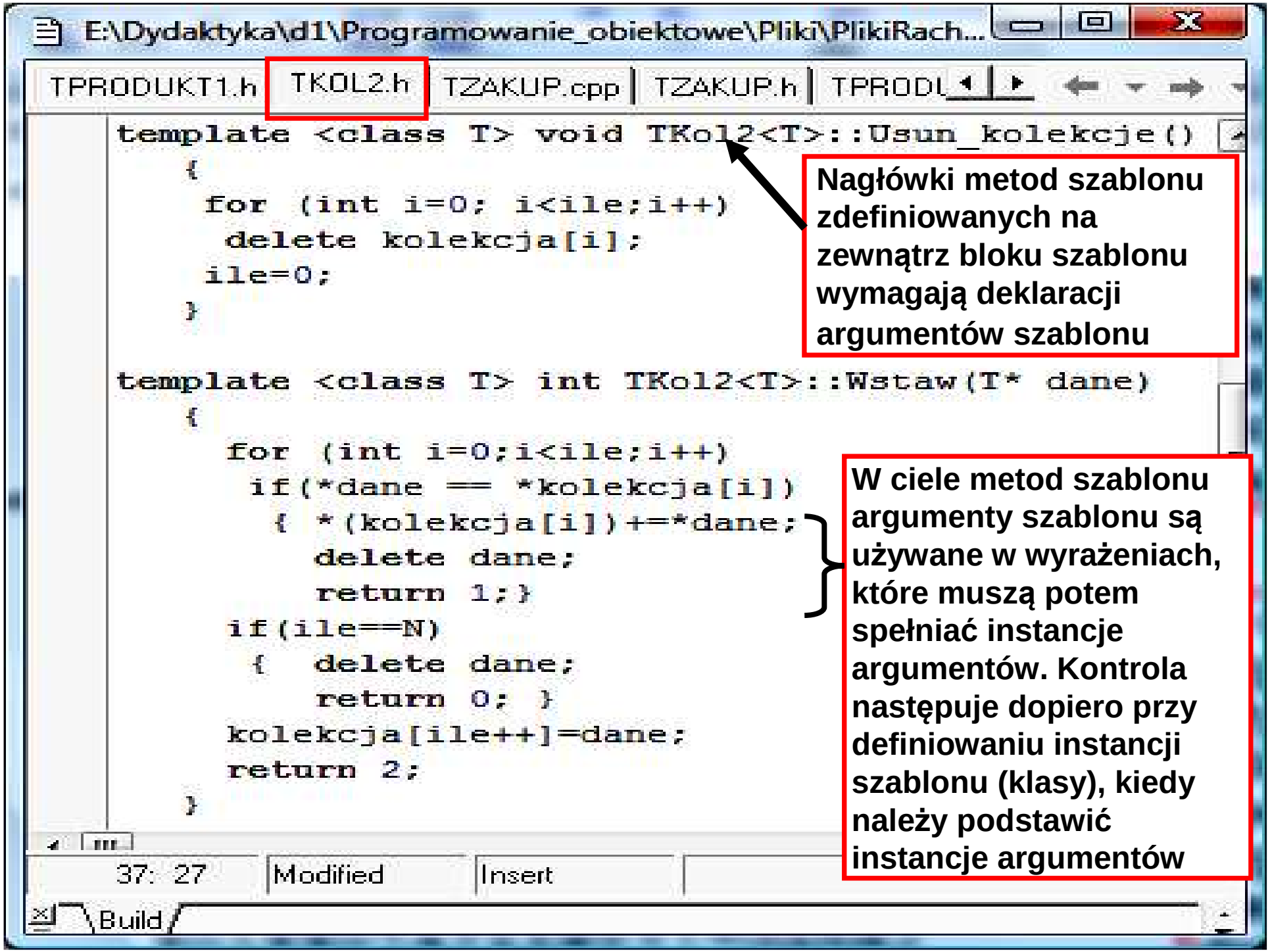
```
#ifndef _TKOL2
#define _TKOL2
#include <string.h>
#include "Abstrakcyjny.h"
const int N=5;
template <class T>
class TKol2
 {protected:
  T* kolekcja[N];
  int ile;
  int biezacy;
  public:
  TKol2(int a=0)
    {ile = a;
     biezacy = 0; }
  ~TKol2()                {  }
  int Pusta()             { return ile==0;}
  void Zeruj()            { biezacy=0;}
  int Koniec()            { return biezacy==ile; }
  T*& Podaj_nast()        { return kolekcja[biezacy++]; }
 void Usun_kolekcje();
 int Wstaw(T* dane);
 T* Podaj(T* dane);
 string toString();
 friend ostream& operator<<(ostream& wy, TKol2<T>& kol)
   { return wy<<kol.toString(); }
};
#endif
```

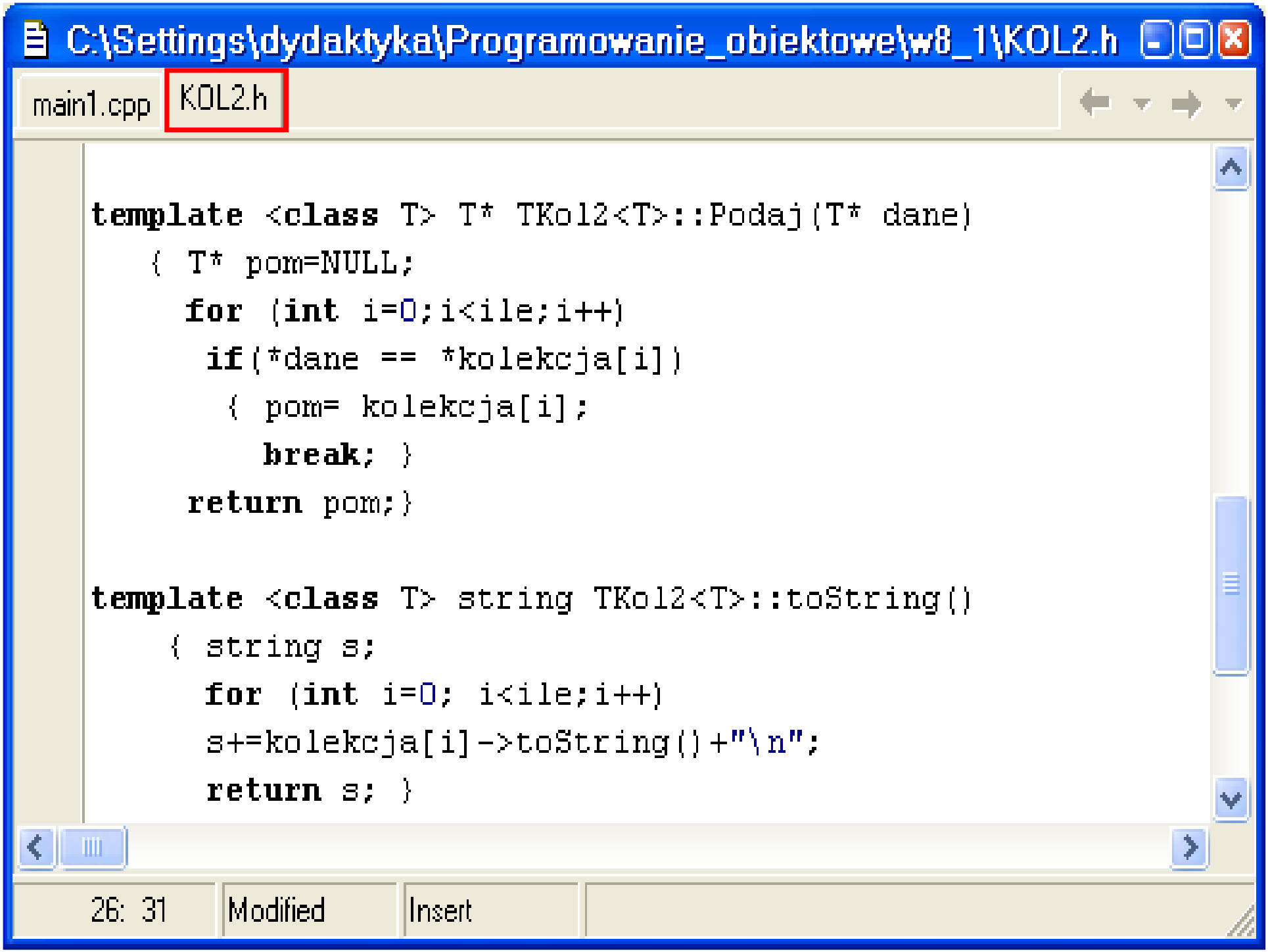
Zwracanie referencji do elementu tablicy kolekcja pozwala na wywołania tej metody:

- **z prawej strony operatora przypisania (pobieranie wartości elementu tablicy kolekcja)**
- **z lewej strony operatora przypisania (pobieranie referencji do elementu tablicy kolekcja i możliwość przypisania do tego elementu wartości z prawej strony operatora przypisania)**

Funkcja zaprzyjaźniona szablonu musi użyć w nagłówku parametry typów, które są związane z argumentami szablonu

26: 31


E:\Dydaktyka\d1\Programowanie_obiektowe\Pliki\PlikiRach...
TPRODUKT1.h
TKOL2.h
TZAKUP.cpp
TZAKUP.h
TPRODU
template <class T> void TKol2<T>::Usun_kolekcje()
{
for (int i=0; i<ile;i++)
delete kolekcja[i];
ile=0;
}
template <class T> int TKol2<T>::Wstaw(T* dane)
{
for (int i=0;i<ile;i++)
if(*dane == *kolekcja[i])
{ *(kolekcja[i])+=*dane;
delete dane;
return 1;}
if(ile==N)
{ delete dane;
return 0; }
kolekcja[ile++]=dane;
return 2;
}
Nagłówki metod szablonu zdefiniowanych na zewnątrz bloku szablonu wymagają deklaracji argumentów szablonu
W ciele metod szablonu argumenty szablonu są używane w wyrażeniach, które muszą potem spełniać instancje argumentów. Kontrola następuje dopiero przy definiowaniu instancji szablonu (klasy), kiedy należy podstawić instancje argumentów
37: 27
Modified
Insert
Build


C:\Settings\dydaktyka\Programowanie_obiektowe\w8_1\KOL2.h
main1.cpp
KOL2.h
template <class T> T* TKol2<T>::Podaj(T* dane)
{ T* pom=NULL;
for (int i=0;i<ile;i++)
if(*dane == *kolekcja[i])
{ pom= kolekcja[i];
break; }
return pom;}
template <class T> string TKol2<T>::toString()
{ string s;
for (int i=0; i<ile;i++)
s+=kolekcja[i]->toString()+"\n";
return s; }
26: 31
Modified
Insert

Przykład programu 1 – generowanie kodu dwóch klas z jednego szablonu

main1.cpp | PRODUKT1.CPP | PRODUKT2.CPP | ZA

1) Dwie instancje kodu kolekcji dla dwóch typów argumentów: TKol2<TProdukt1>, TKol2<TZakup>
2) Dwa obiekty różnych typów: produkty, zakupy

```
#include "Zakup.h"
#include "kol2.h"
 TKol2<TProdukt1> produkty;
 TKol2<TZakup> zakupy;
 void Wstaw_zakup(TProdukt1* p, int ilosc);

void main()
{
 TProdukt1* p1;
 TProdukt2* p2;
//1 p1 = new TProdukt1("zeszyt", 1.0);    produkty.Wstaw(p1);
//2 p1 = new TProdukt1("zeszyt", 1.0);    produkty.Wstaw(p1);
//3 p1 = new TProdukt1("zeszyt", 2.0);    produkty.Wstaw(p1);
//4 p2 = new TProdukt2("olowek",0.80,7);  produkty.Wstaw(p2);
//5 p2 = new TProdukt2("pioro",4.80,20);  produkty.Wstaw(p2);
//6 p2 = new TProdukt2("pioro",4.80,20);  produkty.Wstaw(p2);
```

33: 15 | Modified | Insert

Przykład programu 1 – generowanie kodu dwóch klas z jednego szablonu

main1.cpp

```
 Wstaw_zakup(new TProdukt1("zeszyt", 1.0),1);
 Wstaw_zakup(new TProdukt1("zeszyt", 1.0),2);
 Wstaw_zakup(new TProdukt1("zeszyt", 2.0),3);
 Wstaw_zakup(new TProdukt2("olowek",0.80,7),4);
 Wstaw_zakup(new TProdukt2("pioro",4.80,20),5);
 Wstaw_zakup(new TProdukt2("pioro",4.80,20),6);

 cout<<produkty<<endl;   //cout<<produkty.toString()<<endl;
 cout<<zakupy<<endl;     //cout<<zakupy.toString()<<endl;
 produkty.Usun_kolekcje();
 zakupy.Usun_kolekcje();
 cin.get();
}
void Wstaw_zakup(TProdukt1* poszukiwany, int ilosc)
 {  TProdukt1* znaleziony;
    TZakup* nowy;
    znaleziony=produkty.Podaj(poszukiwany);
    delete poszukiwany;
    if (znaleziony!=NULL)
    {nowy = new TZakup(znaleziony,ilosc);
     zakupy.Wstaw(nowy); }
 }
```

//7
//8
//9
//10
//11
//12

Wynik testów metody *Wstaw* w wygenerowanej instancji klasy TKol2<TProdut1> - (nie powinna wstawiać ponownie tych samych produktów typu TProdukt1 oraz TProdukt2) oraz metody *Wstaw* w wygenerowanej instancji klasy TKol2<TZakup> - (nie powinna wstawiać zakupów z tymi samymi produktami, tylko zwiększać ilość tych zakupów)

Instancja kolekcji elementów typu TProdukt1 – z kolekcji pobierane są elementy typu TProdukt1* i niejawnie typu TProdukt2*

24: 2 | Modified | Insert

Przykład programu 2 – generowanie kodu jednej klasy z jednego szablonów

main1.cpp | KOL2.h | PRODUKT1.CPP | PRODUKT2.CPP

1) Jedna instancja kodu kolekcji TKol2<TAbstrakcyjny> dla jednego typu argumentu
2) dwa obiekty typu TKol2<TAbstrakcyjny> elementów dziedziczących po typie Abstrakcyjny: produkty, zakupy

```
#include "Zakup.h"
#include "kol2.h"
 TKol2<TAbstrakcyjny> produkty;
 TKol2<TAbstrakcyjny> zakupy;
 void Wstaw_zakup(TProdukt1* p, int ilosc);
void main()
{
 TProdukt1* p1;
 TProdukt2* p2;
 p1 = new TProdukt1("zeszyt", 1.0);    produkty.Wstaw(p1);
 p1 = new TProdukt1("zeszyt", 1.0);    produkty.Wstaw(p1);
 p1 = new TProdukt1("zeszyt", 2.0);    produkty.Wstaw(p1);
 p2 = new TProdukt2("olowek",0.80,7);  produkty.Wstaw(p2);
 p2 = new TProdukt2("pioro",4.80,20);  produkty.Wstaw(p2);
 p2 = new TProdukt2("pioro",4.80,20);  produkty.Wstaw(p2);
```

8: 40 | Modified | Insert

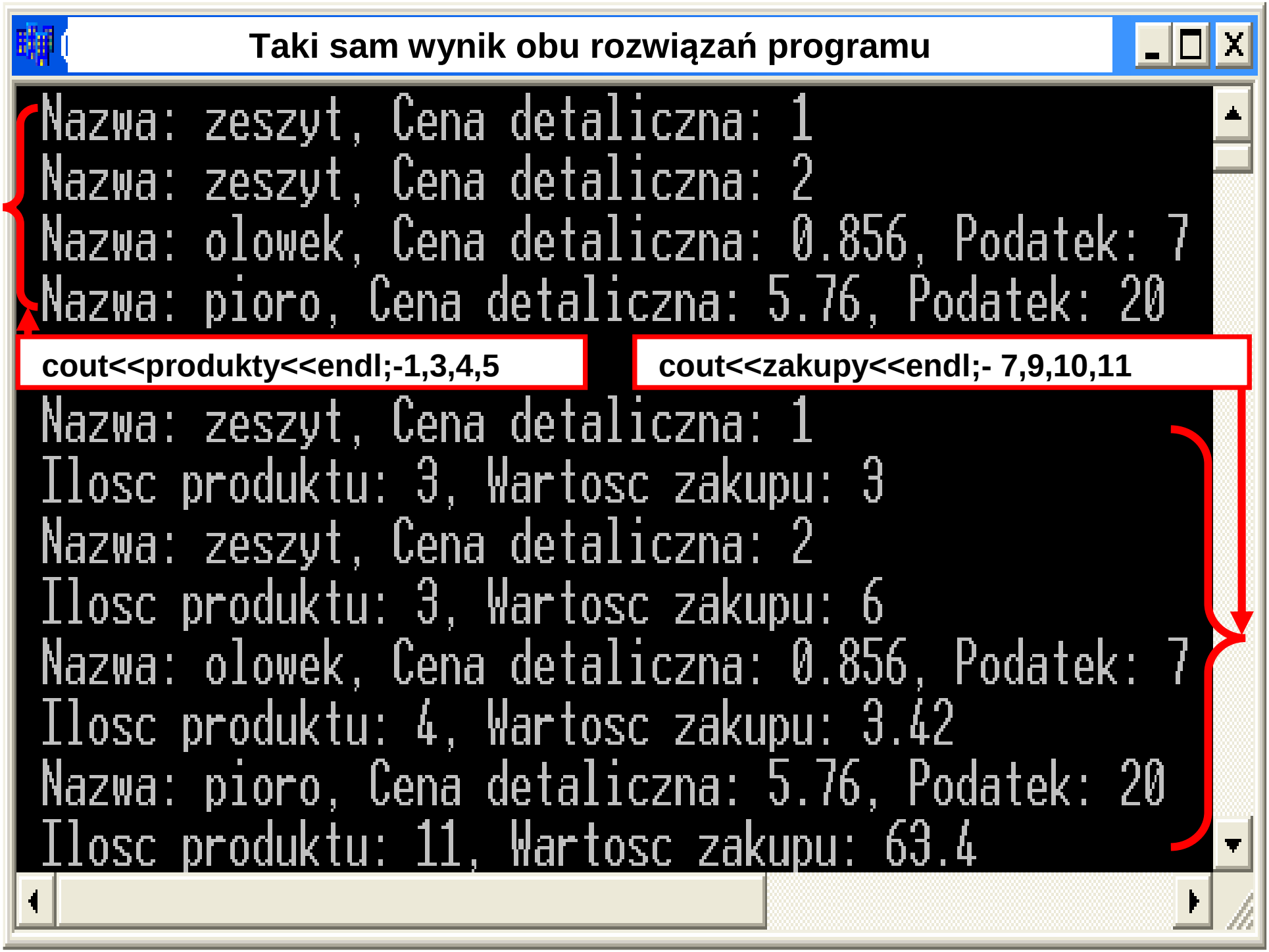
Przykład programu 2 – generowanie kodu jednej klasy z jednego szablonów

main1.cpp | KOL2.h

```
 Wstaw_zakup(new TProdukt1("zeszyt", 1.0),1);
 Wstaw_zakup(new TProdukt1("zeszyt", 1.0),2);
 Wstaw_zakup(new TProdukt1("zeszyt", 2.0),3);
 Wstaw_zakup(new TProdukt2("olowek",0.80,7),4);
 Wstaw_zakup(new TProdukt2("pioro",4.80,20),5);
 Wstaw_zakup(new TProdukt2("pioro",4.80,20),6);

 cout<<produkty<<endl;   //cout<<produkty.toString()<<endl;
 cout<<zakupy<<endl;     //cout<<zakupy.toString()<<endl;
 produkty.Usun_kolekcje();
 zakupy.Usun_kolekcje();
 cin.get();
}
void Wstaw_zakup(TProdukt1* poszukiwany, int ilosc)
 {  TProdukt1* znaleziony;
    TZakup* nowy;
    znaleziony=(TProdukt1*)produkty.Podaj(poszukiwany);
    delete poszukiwany;
    if (znaleziony!=NULL)
    {nowy = new TZakup(znaleziony,ilosc);
     zakupy.Wstaw(nowy); }
 }
```

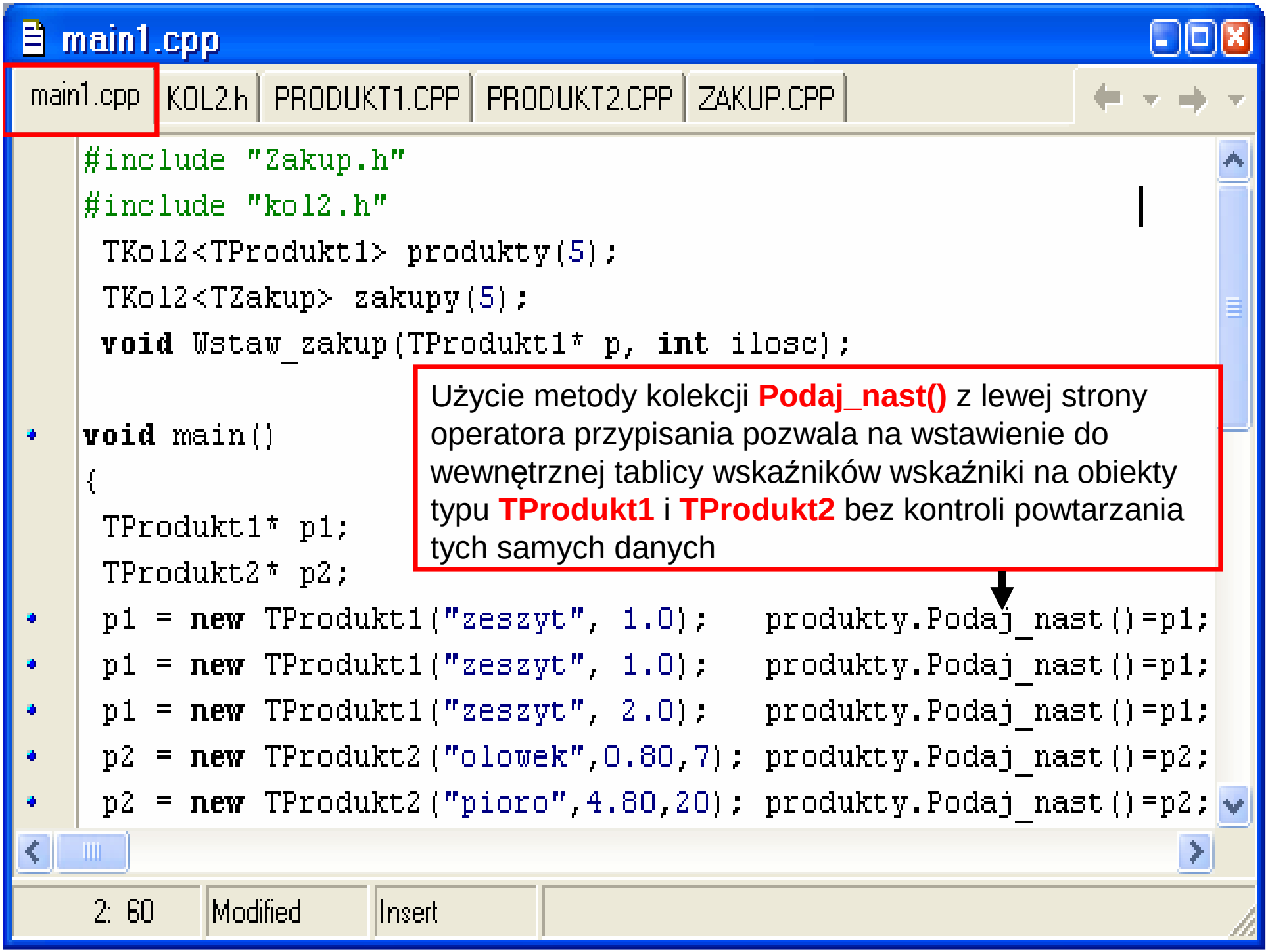
28: 29

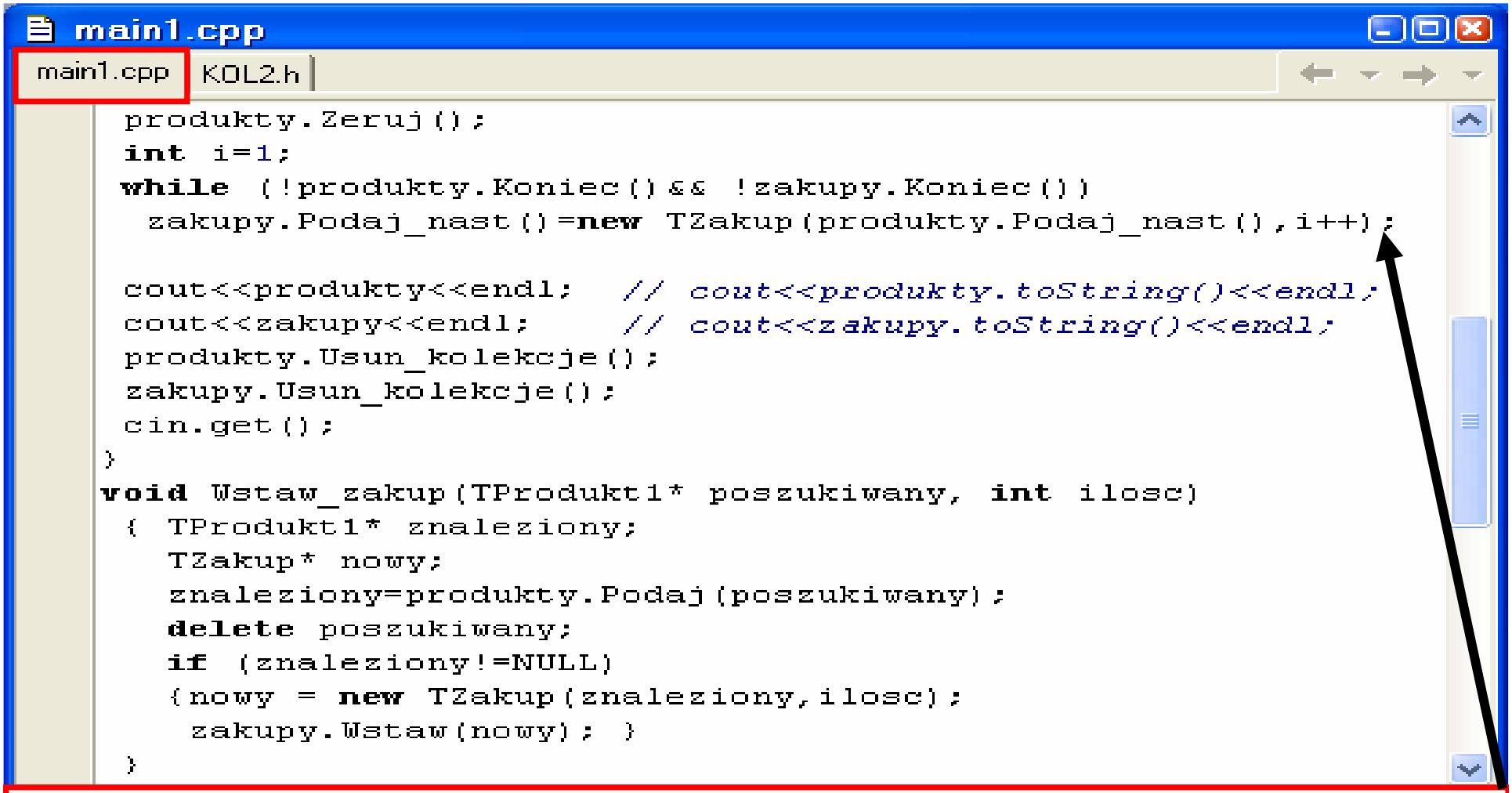
Instancja kolekcji elementów typu TAbstrakcyjny – przy pobraniu elementów z kolekcji trzeba rzutować do typu wskaźnika klasy implementującej typ TAbstrakcyjny, czyli TProdukt1


Taki sam wynik obu rozwiązań programu
Nazwa: zeszyt, Cena detaliczna: 1
Nazwa: zeszyt, Cena detaliczna: 2
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
cout<<produkty<<endl;-1,3,4,5
cout<<zakupy<<endl;- 7,9,10,11
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 3, Wartosc zakupu: 3
Nazwa: zeszyt, Cena detaliczna: 2
Ilosc produktu: 3, Wartosc zakupu: 6
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 4, Wartosc zakupu: 3.42
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
Ilosc produktu: 11, Wartosc zakupu: 63.4

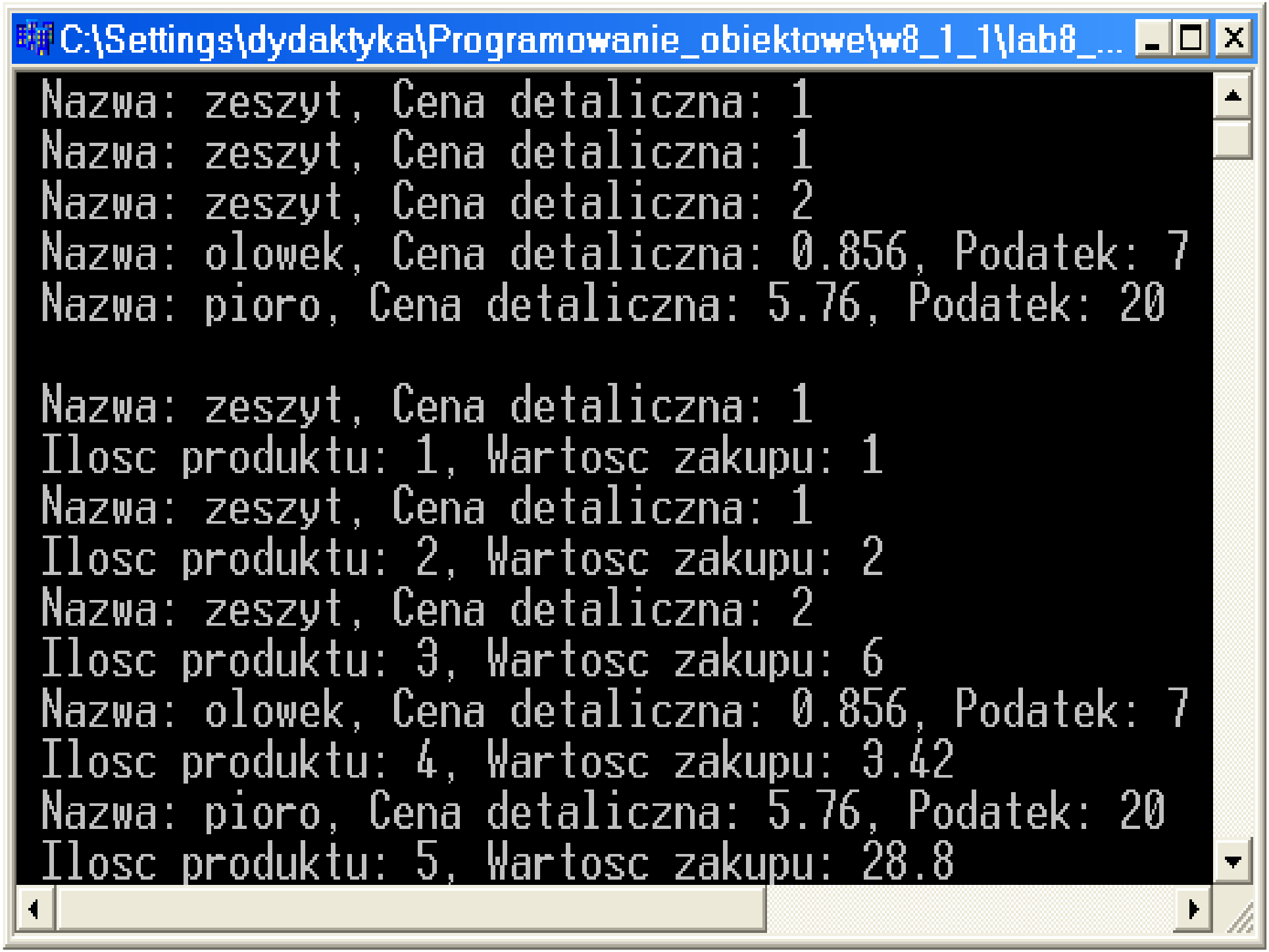
Szablony klas, zastosowanie szablonów w programach

1. Szablony klas i funkcji
- Szablon klasy obsługującej uniwersalną tablicę wskaźników
3. Zastosowanie metody zwracającej przez return referencję do składowych klasy


main1.cpp
main1.cpp
KOL2.h
PRODUKT1.CPP
PRODUKT2.CPP
ZAKUP.CPP
#include "Zakup.h"
#include "kol2.h"
TKol2<TProdukt1> produkty(5);
TKol2<TZakup> zakupy(5);
void Wstaw_zakup(TProdukt1* p, int ilosc);
void main()
{
TProdukt1* p1;
TProdukt2* p2;
p1 = new TProdukt1("zeszyt", 1.0); produkty.Podaj_nast()=p1;
p1 = new TProdukt1("zeszyt", 1.0); produkty.Podaj_nast()=p1;
p1 = new TProdukt1("zeszyt", 2.0); produkty.Podaj_nast()=p1;
p2 = new TProdukt2("olowek",0.80,7); produkty.Podaj_nast()=p2;
p2 = new TProdukt2("pioro",4.80,20); produkty.Podaj_nast()=p2;
Użycie metody kolekcji Podaj_nast() z lewej strony operatora przypisania pozwala na wstawienie do wewnętrznej tablicy wskaźników wskaźniki na obiekty typu TProdukt1 i TProdukt2 bez kontroli powtarzania tych samych danych
2: 60
Modified
Insert



Użycie metody **Podaj_nast()** kolekcji **zakupy** z lewej strony operatora przypisania (**kontekst „do zapisu” metody Podaj_nast()**) pozwala na wstawienie do wewnętrznej tablicy wskaźniki na obiekty dynamiczne typu **TZakup** bez kontroli poprawności danych. Obiekty dynamiczne są tworzone z prawej strony operatora pobierając metodą **Podaj_nast()** wskaźniki na obiekty typu **TProdukt1** i **TProdukt2** z kolekcji **produkty**. Te wskaźniki są kopiowane do składowej **produkt** w tworzonym obiekcie typu **TZakup** (**kontekst „do odczytu” metody Podaj_nast()**)


C:\Settings\dydaktyka\Programowanie_obiektowe\w8_1_1\lab8_...
Nazwa: zeszyt, Cena detaliczna: 1
Nazwa: zeszyt, Cena detaliczna: 1
Nazwa: zeszyt, Cena detaliczna: 2
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 1, Wartosc zakupu: 1
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 2, Wartosc zakupu: 2
Nazwa: zeszyt, Cena detaliczna: 2
Ilosc produktu: 3, Wartosc zakupu: 6
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 4, Wartosc zakupu: 3.42
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
Ilosc produktu: 5, Wartosc zakupu: 28.8

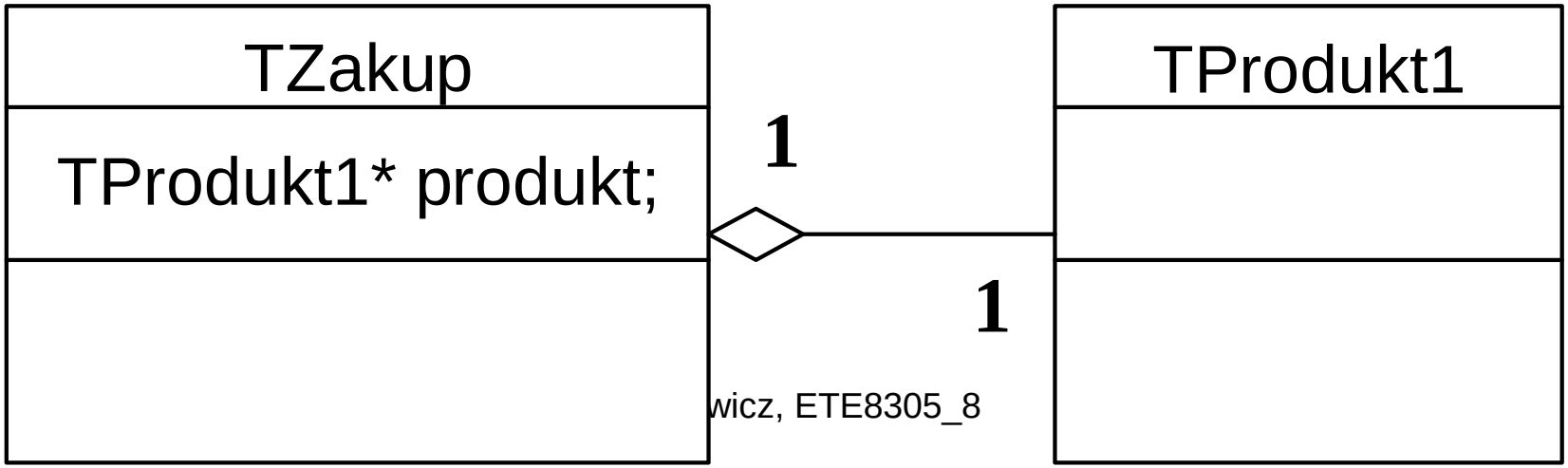
Szablony klas, zastosowanie szablonów w programach

1. **Szablony klas i funkcji**
- **Szablon klasy obsługującej uniwersalną tablicę wskaźników**
- **Zastosowanie metody zwracającej przez return referencję do składowych klasy**

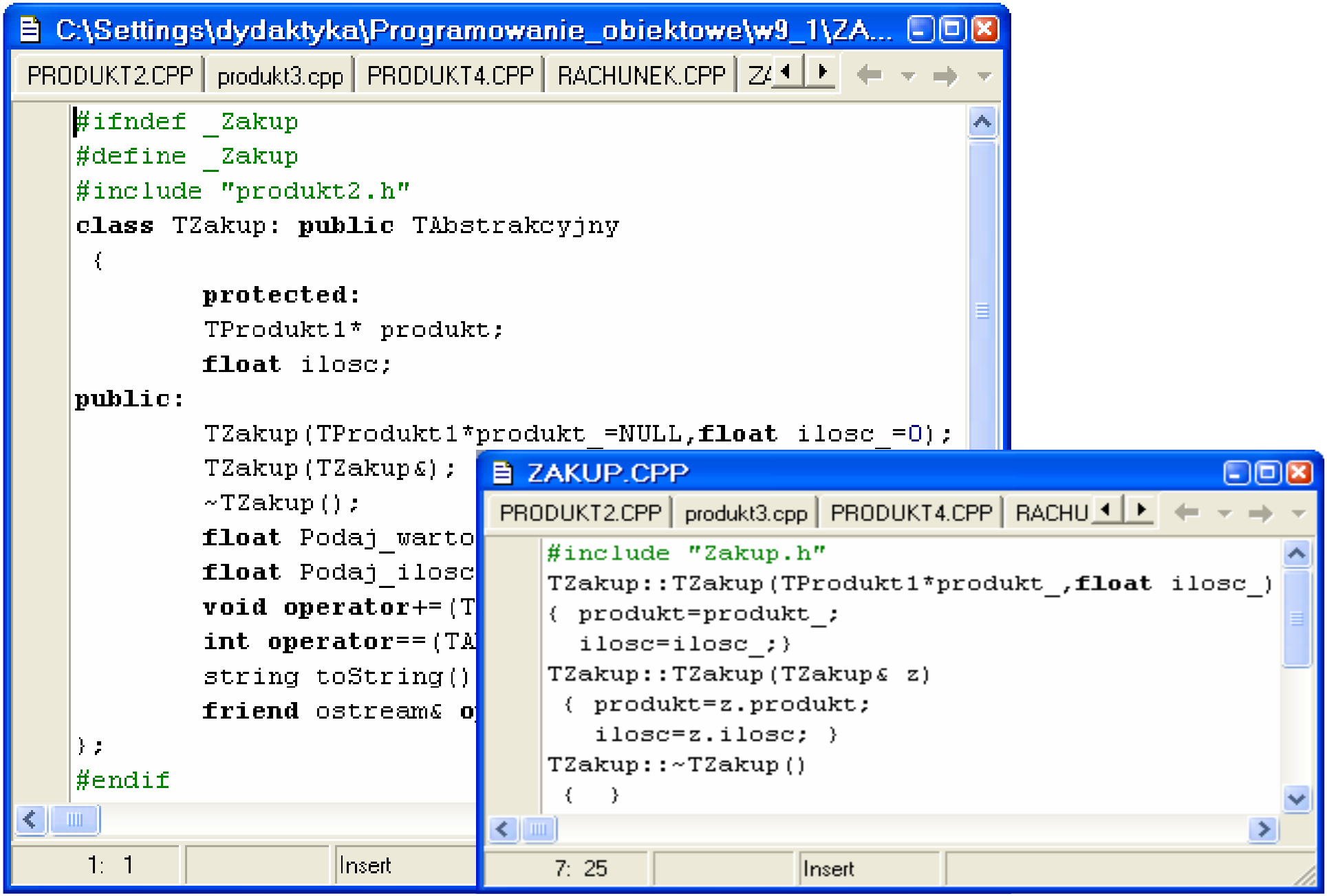
4. **Tworzenie zbioru rachunków zawierających zakupy różnych produktów**

Klasa złożona TZakup agreguje słabo obiekt klasy TProdukt1

```
#include "Zakup.h"
TZakup::TZakup(TProdukt1*produkt_,float ilosc_)
  {  produkt=produkt_;       // nie przydziela pamięci w kostruktorze
     ilosc=ilosc_; }

TZakup::TZakup(TZakup& z)
  {  produkt=z.produkt;      // nie tworzy kopii dynamicznego obiektu podczas
     ilosc=z.ilosc; }        // tworzenie kopii całego obiektu TZakup

TZakup::~TZakup()            // destruktor w klasie TZakup nie zwalnia pamięci
  {      }                   // przydzielonej dla obiektu dynamicznego TProdukt1
```



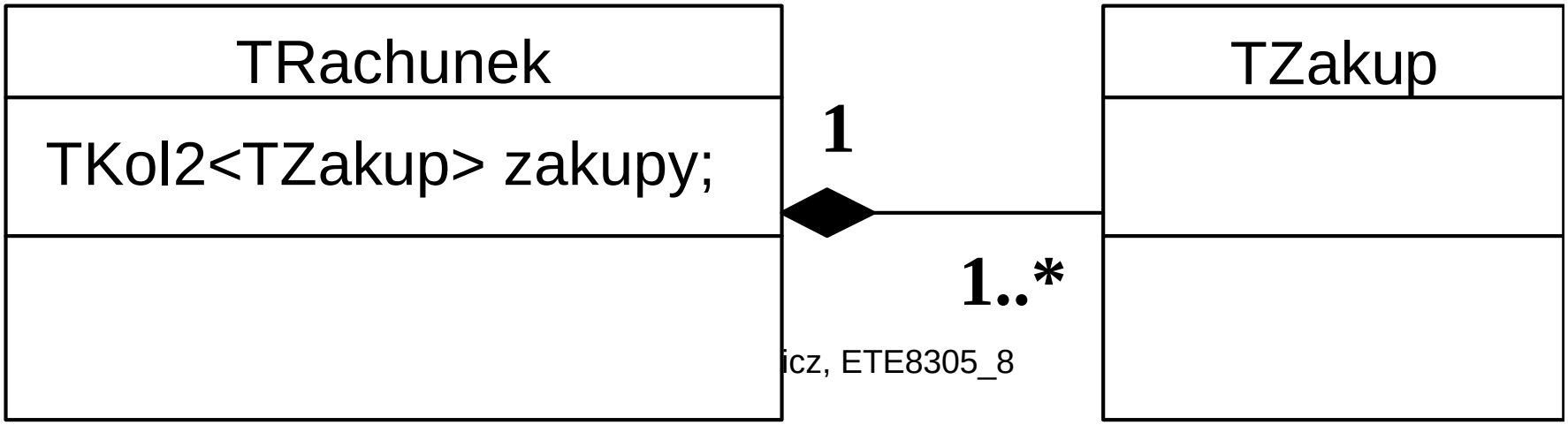
Implementacja agregacji słabej

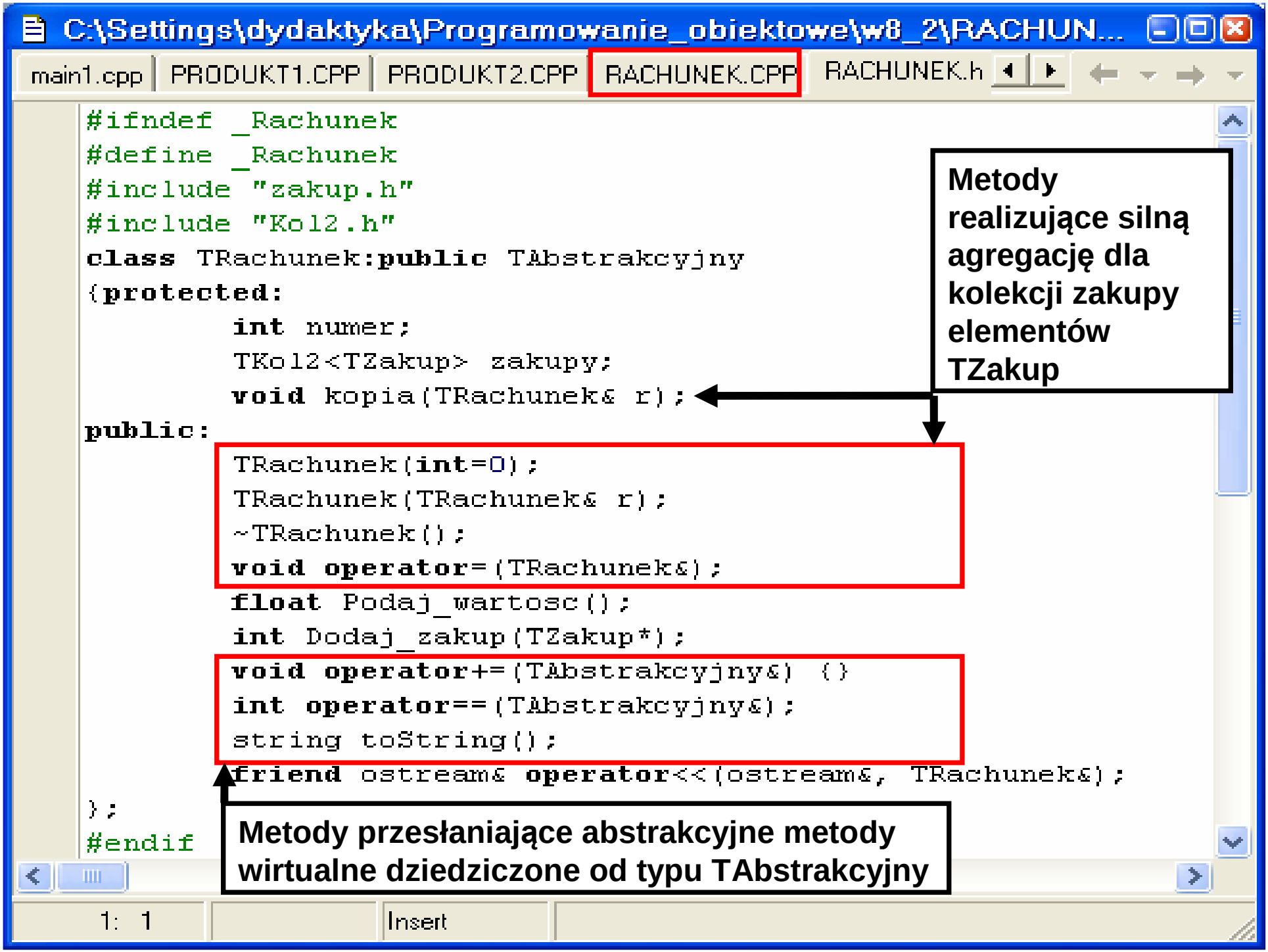


Klasa złożona TRachunek agreguje mocno obiekt klasy TZakup

Kolekcja **TKol2** zawiera wskaźniki do własnych obiektów **TZakup**. Jest to umożliwione również w sytuacjach, gdy powstaje obiekt **TRachunek** przy użyciu konstruktora kopiującego oraz przy użyciu przeciążonego operatora przypisania dla obiektów klasy **TRachunek**.

Kolekcja **TKol2** jest obiektem, który automatycznie jest zwalniany z pamięci, jeśli zwalnia się obiekt klasy **TRachunek** (pierwszy destruktor klasy **TRachunek**, który zwalnia pamięć na elementy kolekcji metodą kolekcji **Usun_kolekcje(),** a potem uruchamiany jest pusty destruktor klasy **TKol2**, Umożliwia to realizację agregacji słabej lub silnej obiektom zawierającym obiekt klasy **TKol2**.




C:\Settings\dydaktyka\Programowanie_obiektowe\w8_2\RACHUN...
main1.cpp
PRODUKT1.CPP
PRODUKT2.CPP
RACHUNEK.CPP
RACHUNEK.h
#ifndef _Rachunek
#define _Rachunek
#include "zakup.h"
#include "Kol2.h"
class TRachunek:public TAbstrakcyjny
{protected:
int numer;
TKol2<TZakup> zakupy;
void kopia(TRachunek& r);
public:
TRachunek(int=0);
TRachunek(TRachunek& r);
~TRachunek();
void operator=(TRachunek&);
float Podaj_wartosc();
int Dodaj_zakup(TZakup*);
void operator+=(TAbstrakcyjny&) {}
int operator==(TAbstrakcyjny&);
string toString();
friend ostream& operator<<(ostream&, TRachunek&);
};
#endif
Metody realizujące silną agregację dla kolekcji zakupy elementów TZakup
Metody przesłaniające abstrakcyjne metody wirtualne dziedziczone od typu TAbstrakcyjny
1: 1
Insert

RACHUNEK.CPP

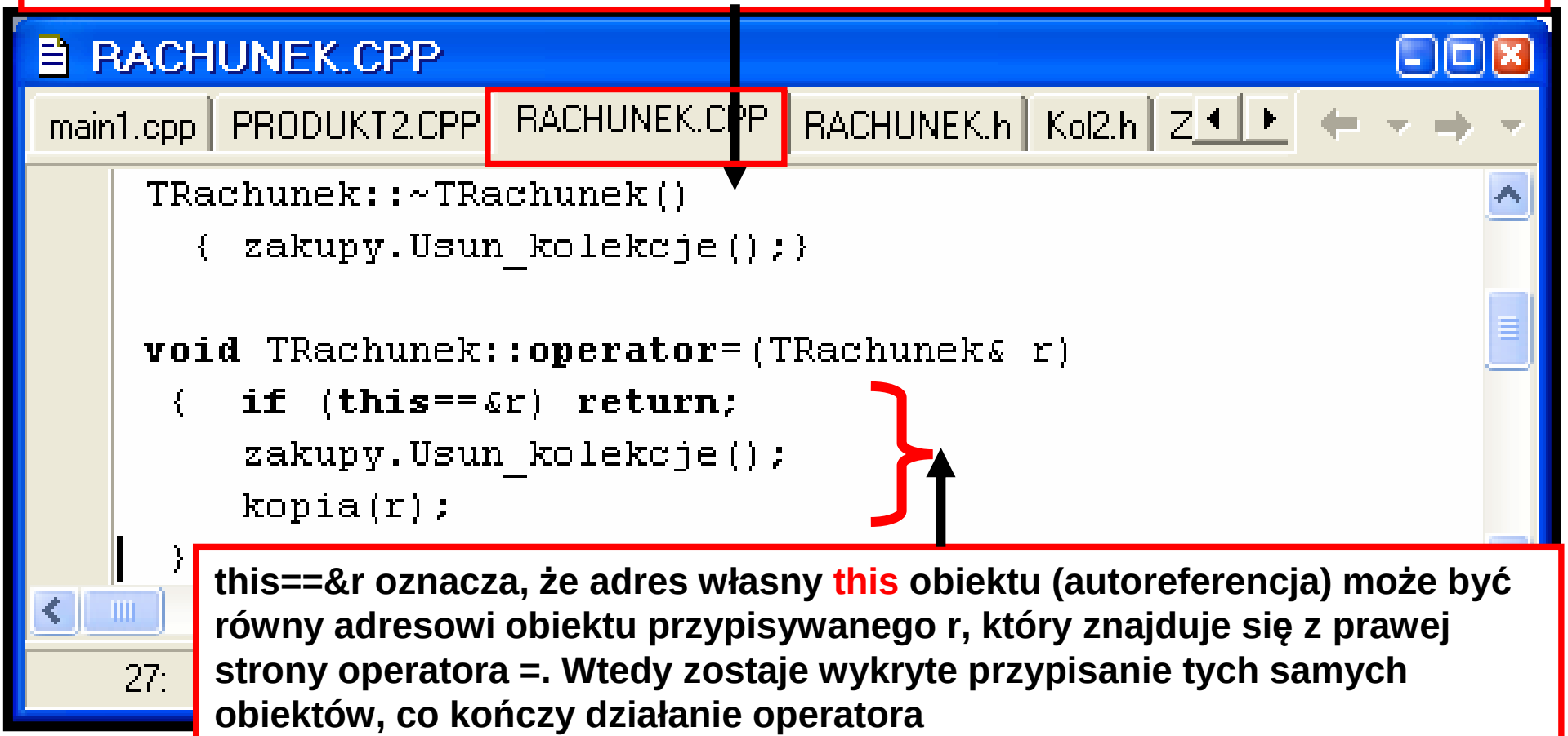
PRODUKT1.CPP | PRODUKT2.CPP | RACHUNEK.CPP | R

```
#include "rachunek.h"
TRachunek::TRachunek(int nr)
  {  numer=nr;}

void TRachunek::kopia(TRachunek& r)
 {   TZakup* z,*pom;
     numer= r.numer;
     zakupy=r.zakupy;
     r.zakupy.Zeruj();
     zakupy.Zeruj();
     while (!r.zakupy.Koniec())
       {  z=r.zakupy.Podaj_nast();
          pom =new TZakup(*z);
          zakupy.Podaj_nast()=pom;  }
 }

TRachunek::TRachunek(TRachunek& r)
  {  kopia(r);  }
```

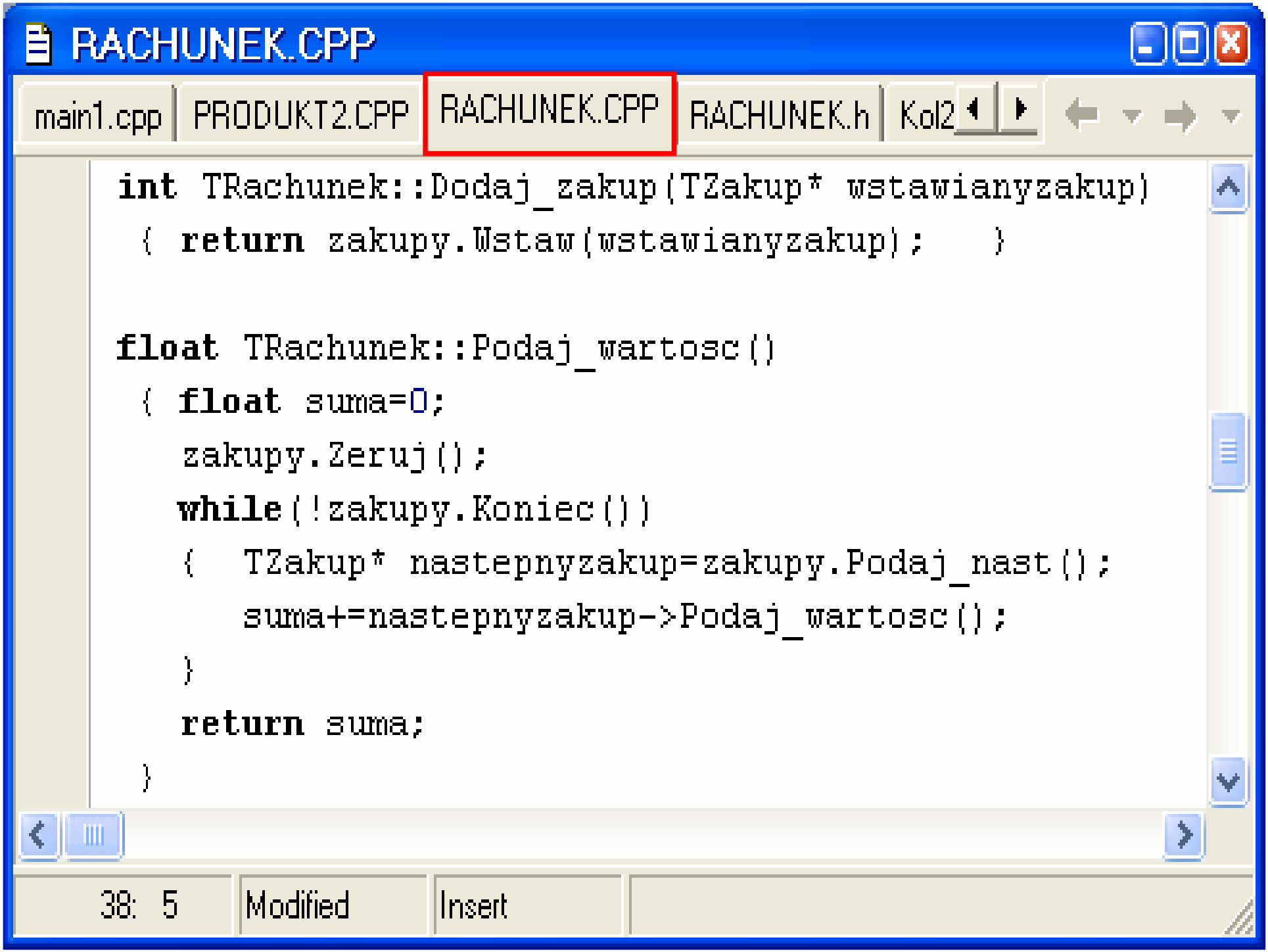
1: 1 | Insert

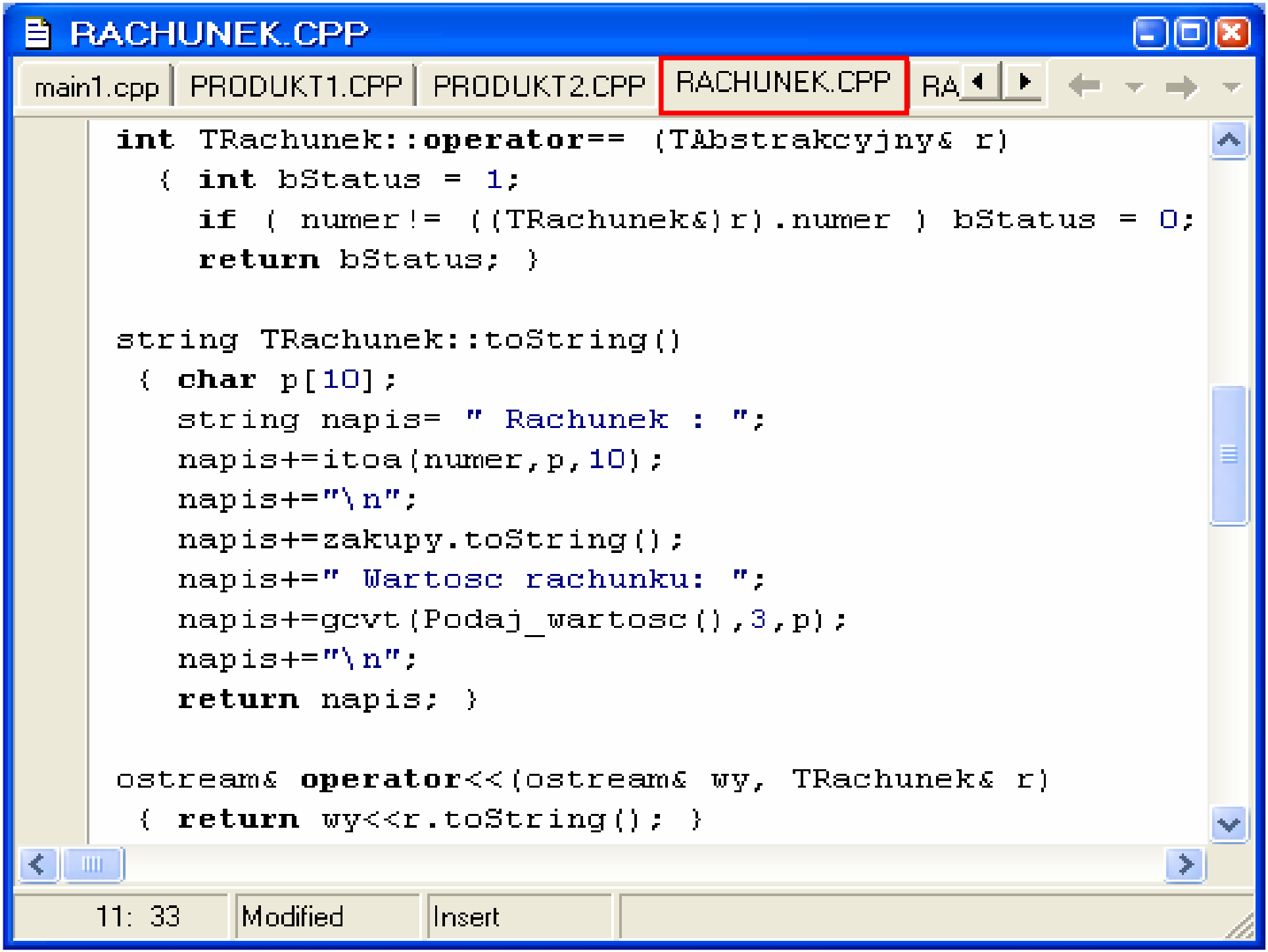
Konstruktor kopiujący pozwala na realizację silnej agregacji – w nowym rachunku najpierw tworzona kopia zakupy obiektu TKol2 z kopiami wskaźników do tych samych obiektów typu TZakup. Następnie w obiekcie zakupy nowego obiektu tworzone są wskaźniki do nowych obiektów TZakup jako kopii obiektów TZakup z kolekcji zakupy z kopiowanego rachunku r

Destruktor realizujący silną agregację między zakupami i rachunkiem – każdy rachunek usuwa własne obiekty typu TZakup



this==&r oznacza, że adres własny this obiektu (autoreferencja) może być równy adresowi obiektu przypisywanego r, który znajduje się z prawej strony operatora =. Wtedy zostaje wykryte przypisanie tych samych obiektów, co kończy działanie operatora
W przeciwnym wypadku usuwane są obiekty typu TZakup z kolekcji zakupy rachunku z lewej strony operatora =.
Następnie za pomocą metody pomocniczej kopia tworzona jest kopia kolekcji obiektów dynamicznych TZakup w obiekcie z lewej strony operatora =, czyli kolekcji zakupy należącej do obiektu r typu TRachunek z prawej strony operatora.


RACHUNEK.CPP
main1.cpp
PRODUKT2.CPP
RACHUNEK.CPP
RACHUNEK.h
Kol2
int TRachunek::Dodaj_zakup(TZakup* wstawianyzakup)
{ return zakupy.Wstaw(wstawianyzakup); }
float TRachunek::Podaj_wartosc()
{ float suma=0;
zakupy.Zeruj();
while(!zakupy.Koniec())
{ TZakup* nastepnyzakup=zakupy.Podaj_nast();
suma+=nastepnyzakup->Podaj_wartosc();
}
return suma;
}
38: 5
Modified
Insert


RACHUNEK.CPP
main1.cpp
PRODUKT1.CPP
PRODUKT2.CPP
RACHUNEK.CPP
RA
int TRachunek::operator== (TAbstrakcyjny& r)
{ int bStatus = 1;
if (numer!= ((TRachunek&)r).numer) bStatus = 0;
return bStatus; }
string TRachunek::toString()
{ char p[10];
string napis= " Rachunek : ";
napis+=itoa(numer,p,10);
napis+="\n";
napis+=zakupy.toString();
napis+=" Wartosc rachunku: ";
napis+=gcvt(Podaj_wartosc(),3,p);
napis+="\n";
return napis; }
ostream& operator<<(ostream& wy, TRachunek& r)
{ return wy<<r.toString(); }
11: 33
Modified
Insert

Przykład programu – generowanie kodu dwóch klas z jednego szablonu

main1.cpp | Kol2.h

```
#include "Rachunek.h"
 TKol2<TProdukt1> produkty;
 TKol2<TRachunek> rachunki;
 void Wstaw_zakup(TProdukt1* poszukiwanyprodukt,
                  int ilosc, int numer);
void main()
{
 TProdukt1* p1;
 TProdukt2* p2;
 p1 = new TProdukt1("zeszyt", 1.0);    produkty.Wstaw(p1); //1
 p1 = new TProdukt1("zeszyt", 1.0);    produkty.Wstaw(p1); //2
 p1 = new TProdukt1("zeszyt", 2.0);    produkty.Wstaw(p1); //3
 p2 = new TProdukt2("olowek",0.80,7);  produkty.Wstaw(p2); //4
 p2 = new TProdukt2("pioro",4.80,20);  produkty.Wstaw(p2); //5
 p2 = new TProdukt2("pioro",4.80,20);  produkty.Wstaw(p2); //6
 cout<<produkty<<endl;   //cout<<produkty.toString()<<endl;
```

Dwie instancje kodu kolekcji dla różnych typów argumentów: TKol2<TProdukt1>, TKol2<TZakup> oraz dwa obiekty: produkty, rachunki

Test metody *Wstaw* w wygenerowanej instancji klasy TKol2<TProdukt1> - nie powinna wstawiać tych samych produktów 2,6

21: 22 | Modified | Insert

Przykład programu – generowanie kodu dwóch klas z jednego szablonu

main1.cpp | PRODUKT1.CPP | PRODUKT2.CPP | RACHUNEK.CPP | ZAKUP.CPP | Kol2.h

```
rachunki.Wstaw(new TRachunek(1));          //7
rachunki.Wstaw(new TRachunek(1));          //8
rachunki.Wstaw(new TRachunek(2));          //9
cout<<rachunki<<endl;   //cout<<rachunki.toString();

//11
Wstaw_zakup(new TProdukt1("zeszyt", 1.0),1,1);   //10
Wstaw_zakup(new TProdukt1("zeszyt", 1.0),2,1);
Wstaw_zakup(new TProdukt1("zeszyt", 2.0),3,1);   //12
Wstaw_zakup(new TProdukt2("olowek",0.80,7),4,1);  //13
Wstaw_zakup(new TProdukt2("pioro",4.80,20),5,1);  //14
Wstaw_zakup(new TProdukt2("pioro",4.80,20),6,1);
                                                  //15
```

21: 22 | Modified | Insert

I - Test metody *Wstaw* w wygenerowanej instancji klasy TKol2<TRachunek> - nie powinna wstawiać tych samych rachunków- 8

Test metody *Wstaw* w instancji klasy TKol2<TZakup> w klasie TRachunek- nie powinna wstawiać ponownie zakupów z identycznymi produktami- 11,15

Przykład programu – generowanie kodu dwóch klas z jednego szablonu

main1.cpp | PRODUKT1.CPP | PRODUKT2.CPP | RACHUNEK.CPP | ZAKUP.CP

```
Wstaw_zakup(new TProdukt1("zeszyt", 1.0),1,2);
Wstaw_zakup(new TProdukt1("zeszyt", 3.0),3,2);
Wstaw_zakup(new TProdukt1("zeszyt", 3.0),5,2);
Wstaw_zakup(new TProdukt2("olowek",0.90,7),7,2);
Wstaw_zakup(new TProdukt2("pioro",4.80,20),9,2);
Wstaw_zakup(new TProdukt2("pioro1",4.80,20),11,2);
Wstaw_zakup(new TProdukt2("pioro",4.80,20),9,3);

cout<<rachunki<<endl; //cout<<rachunki.toString();
produkty.Usun_kolekcje();
rachunki.Usun_kolekcje();
cin.get();
}
```

//16
//17
//18
//19
//20
//21
//22

38: 22 | Modified | Insert

II - Wynik testu funkcji *Wstaw_zakup*, Nie powinna ona wstawiać:

- **ponownie zakupów z powtarzającymi się produktami (11,15) – test metody Wstaw kolekcji rachunki**
- **zakupów z nie istniejącymi produktami w kolekcji produkty (brak zakupów z produktami: 17,18,19,21) - test metody Podaj w kolekcji produkty**
- **do nie istniejącego rachunku w kolekcji rachunki (22) - test metody Podaj w kolekcji rachunki**

Przykład programu – generowanie kodu dwóch klas z jednego szablonu

main1.cpp | PRODUKT1.CPP | PRODUKT2.CPP | RACHUNEK.CPP | RACHUNEK.h | ZAKUP.CPP

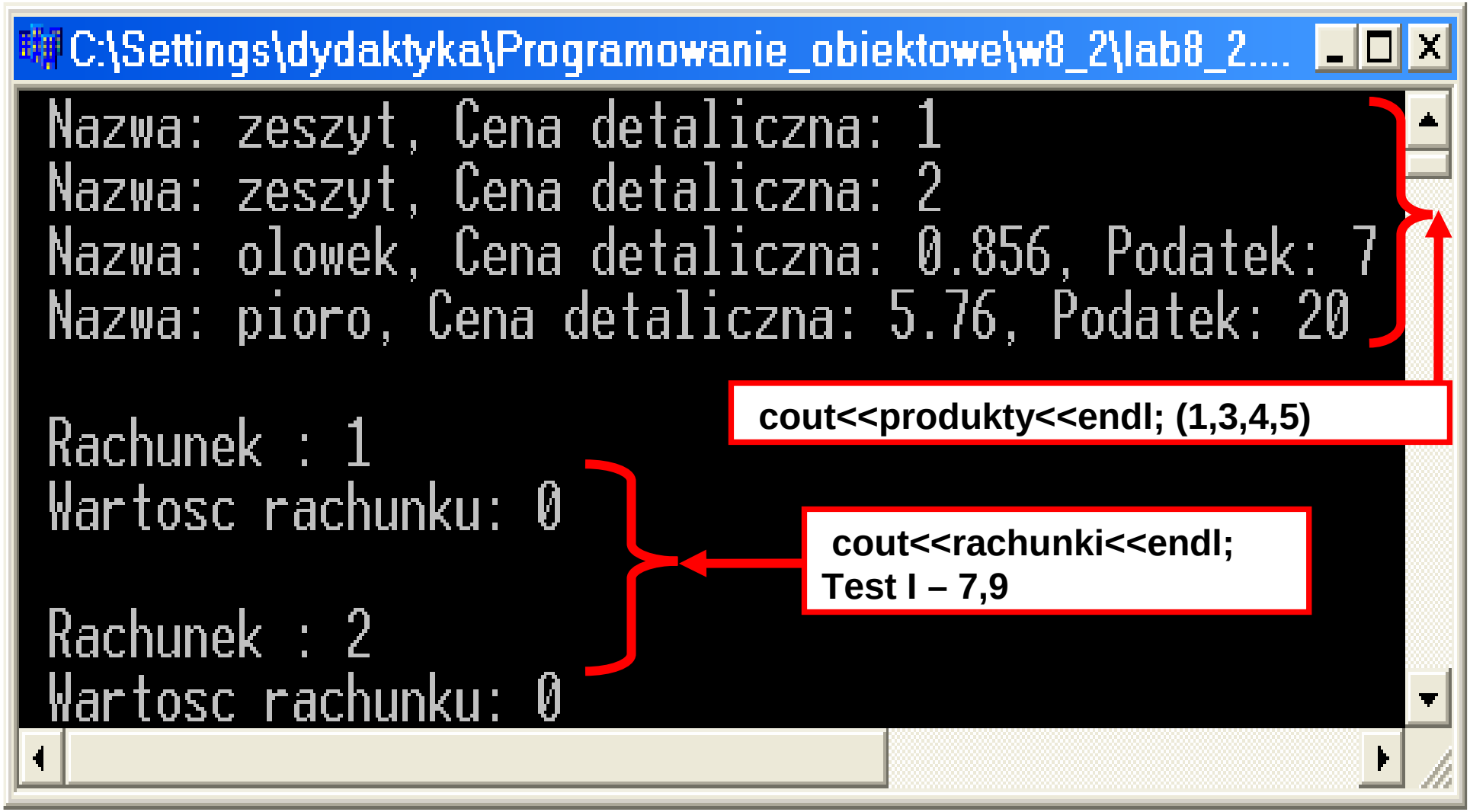
```
void Wstaw_zakup(TProdukt1* poszukiwanyprodukt, int ilosc, int numer)
 { TRachunek* poszukiwanyrachunek=new TRachunek(numer);
   TRachunek* znalezionyrachunek;
   znalezionyrachunek=rachunki.Podaj(poszukiwanyrachunek);
   if (znalezionyrachunek!=NULL)
    { TProdukt1* znalezionyprodukt=produkty.Podaj(poszukiwanyprodukt);
      if(znalezionyprodukt!=NULL)
        znalezionyrachunek->Dodaj_zakup(new TZakup(znalezionyprodukt,ilosc));
    }
   delete poszukiwanyrachunek;
   delete poszukiwanyprodukt;
 }
```

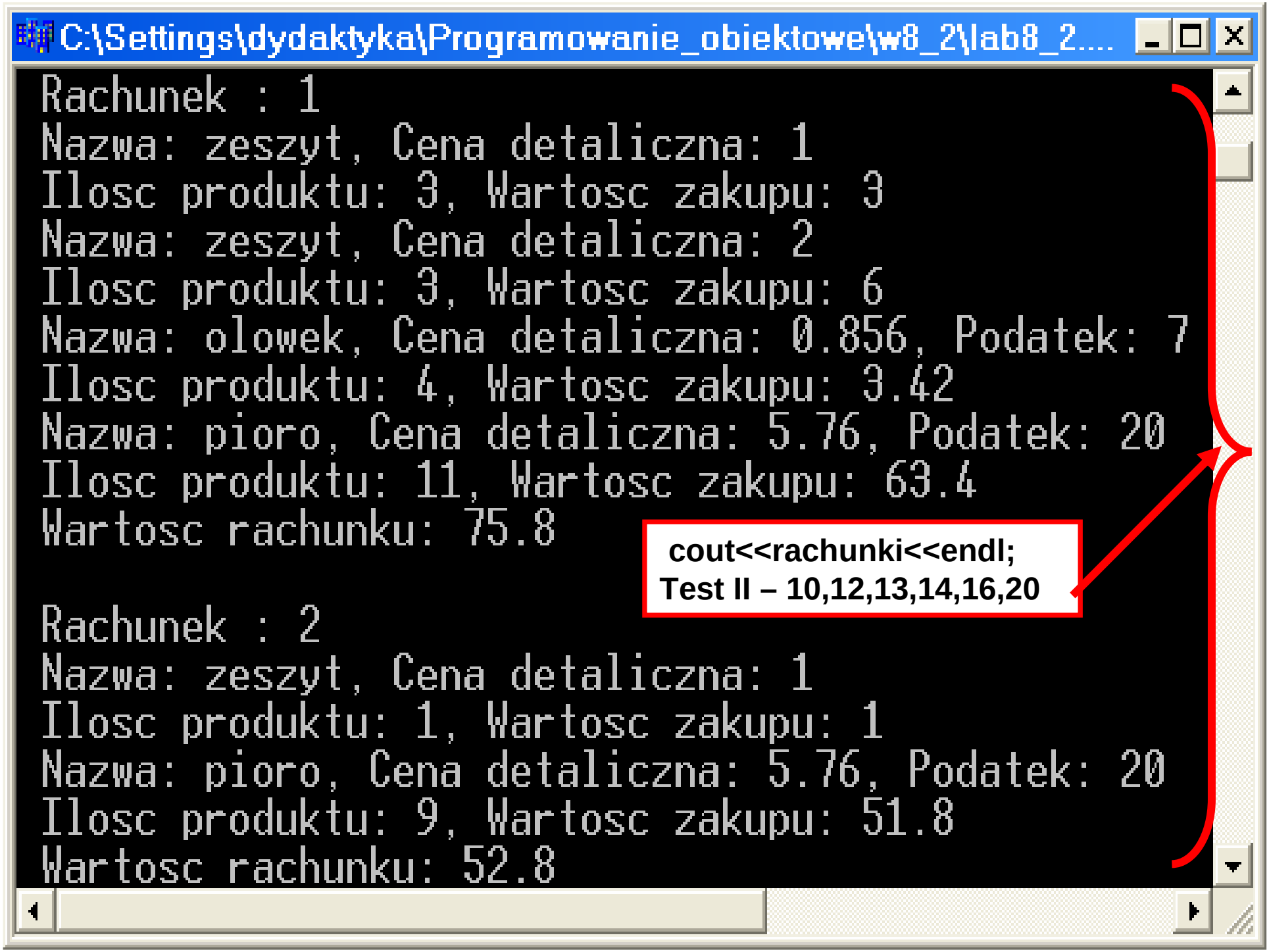
//22

//17,18,19,21

//11,15

6: 1 | Insert

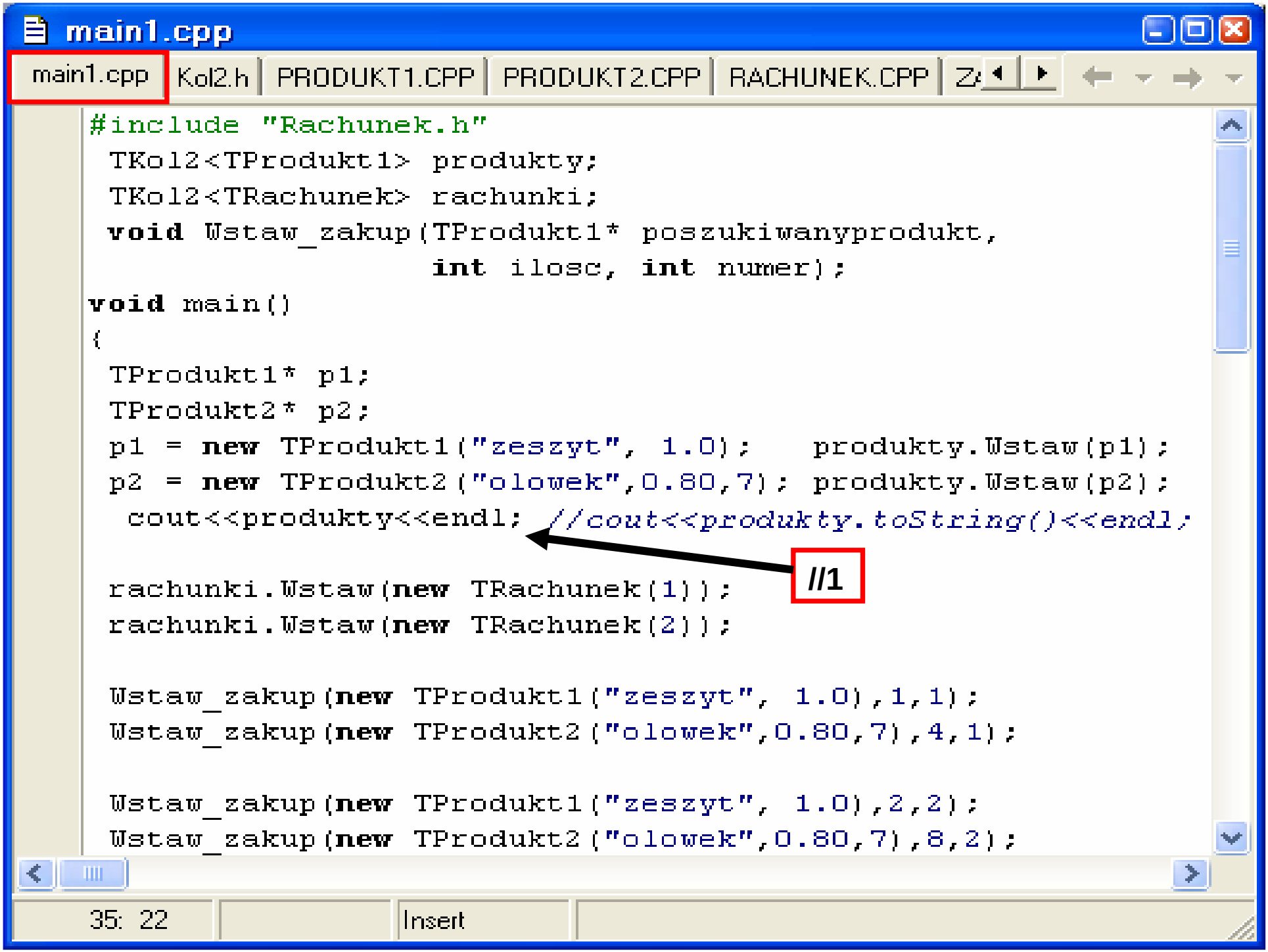

C:\Settings\dydaktyka\Programowanie_obiektowe\w8_2\lab8_2....
Nazwa: zeszyt, Cena detaliczna: 1
Nazwa: zeszyt, Cena detaliczna: 2
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
Rachunek : 1
Wartosc rachunku: 0
Rachunek : 2
Wartosc rachunku: 0
cout<<produkty<<endl; (1,3,4,5)
cout<<rachunki<<endl;
Test I – 7,9

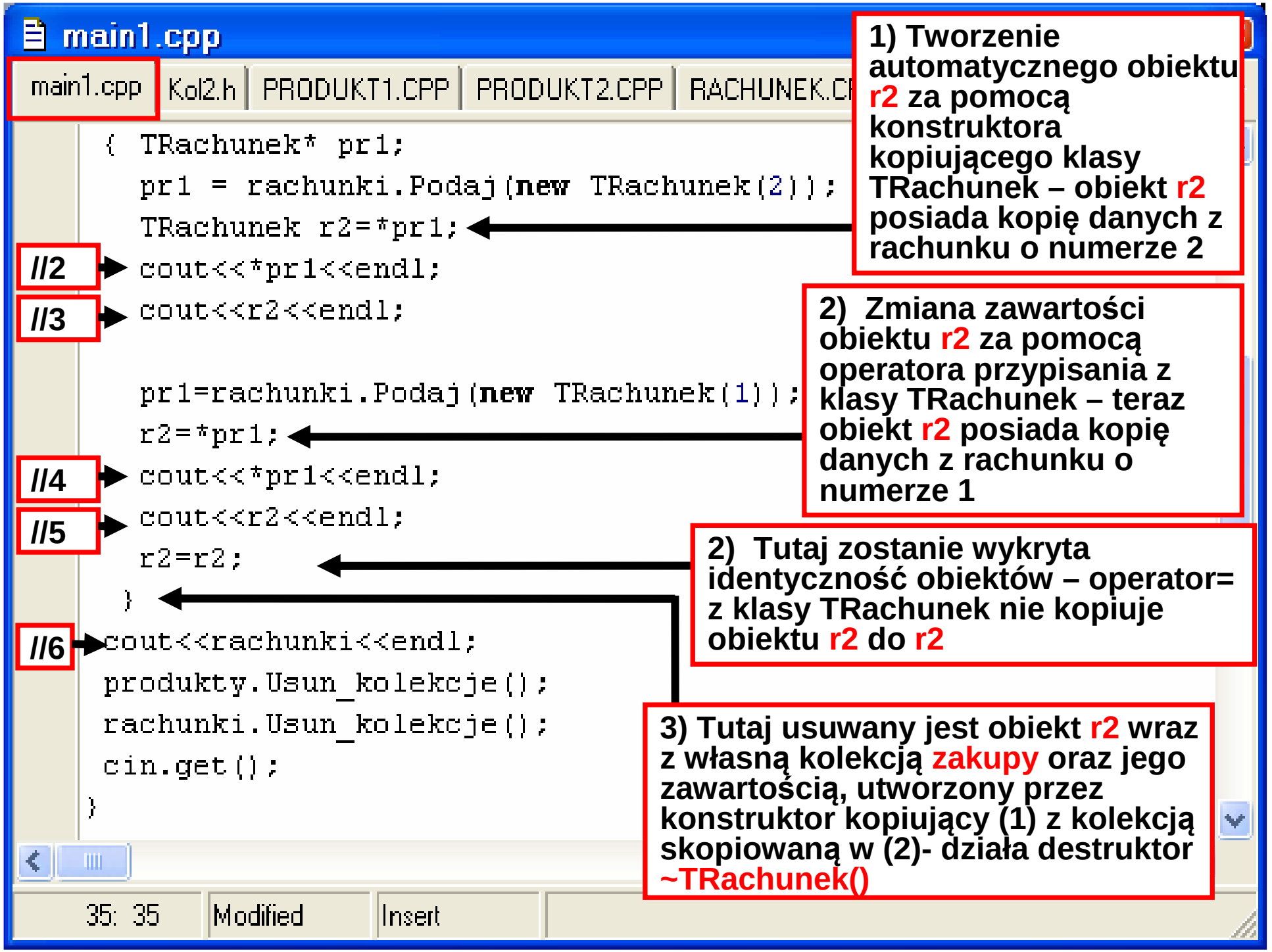

C:\Settings\dydaktyka\Programowanie_obiektowe\w8_2\lab8_2....
Rachunek : 1
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 3, Wartosc zakupu: 3
Nazwa: zeszyt, Cena detaliczna: 2
Ilosc produktu: 3, Wartosc zakupu: 6
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 4, Wartosc zakupu: 3.42
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
Ilosc produktu: 11, Wartosc zakupu: 63.4
Wartosc rachunku: 75.8
Rachunek : 2
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 1, Wartosc zakupu: 1
Nazwa: pioro, Cena detaliczna: 5.76, Podatek: 20
Ilosc produktu: 9, Wartosc zakupu: 51.8
Wartosc rachunku: 52.8
cout<<rachunki<<endl;
Test II – 10,12,13,14,16,20

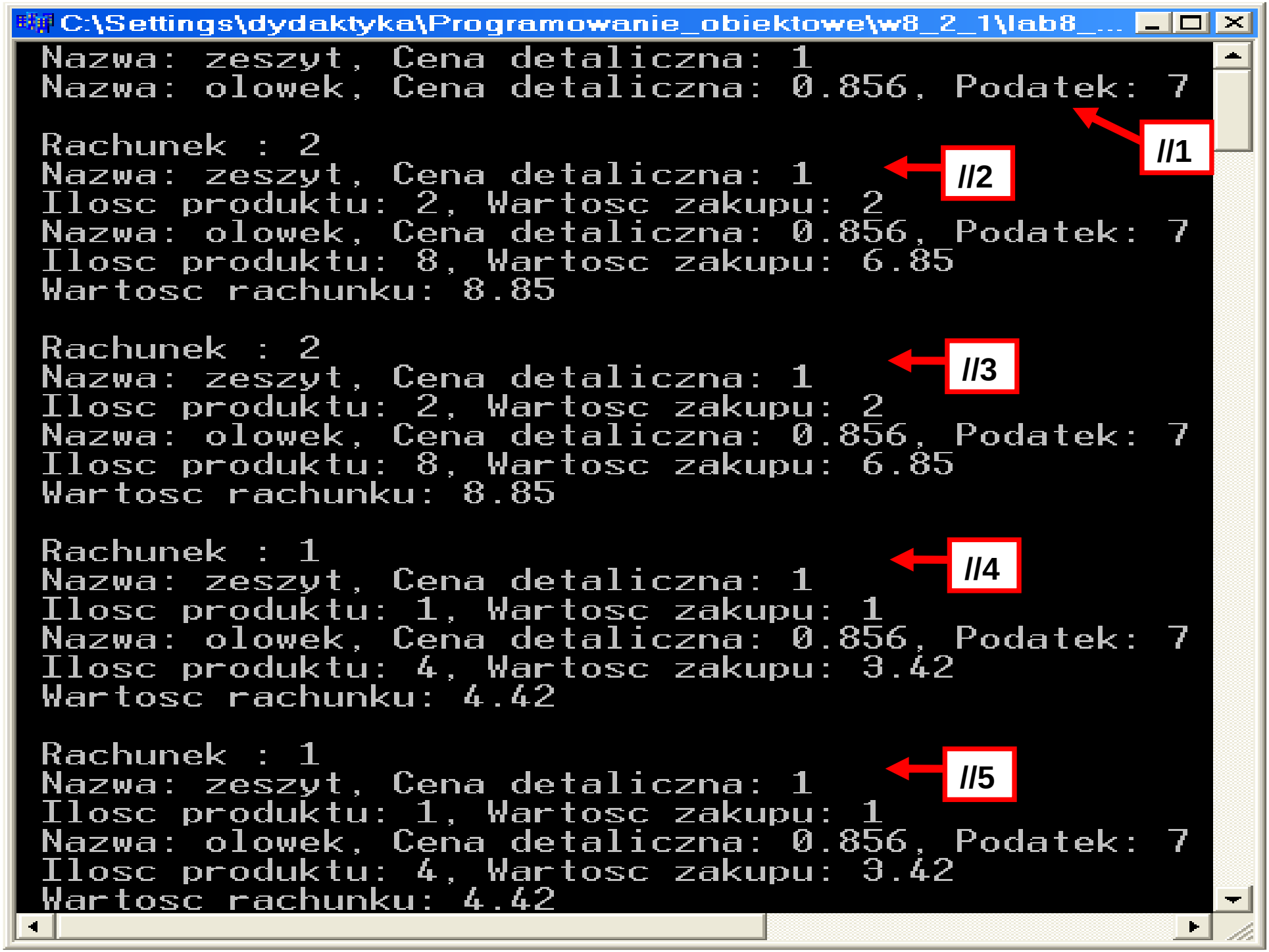
Szablony klas, zastosowanie szablonów w programach

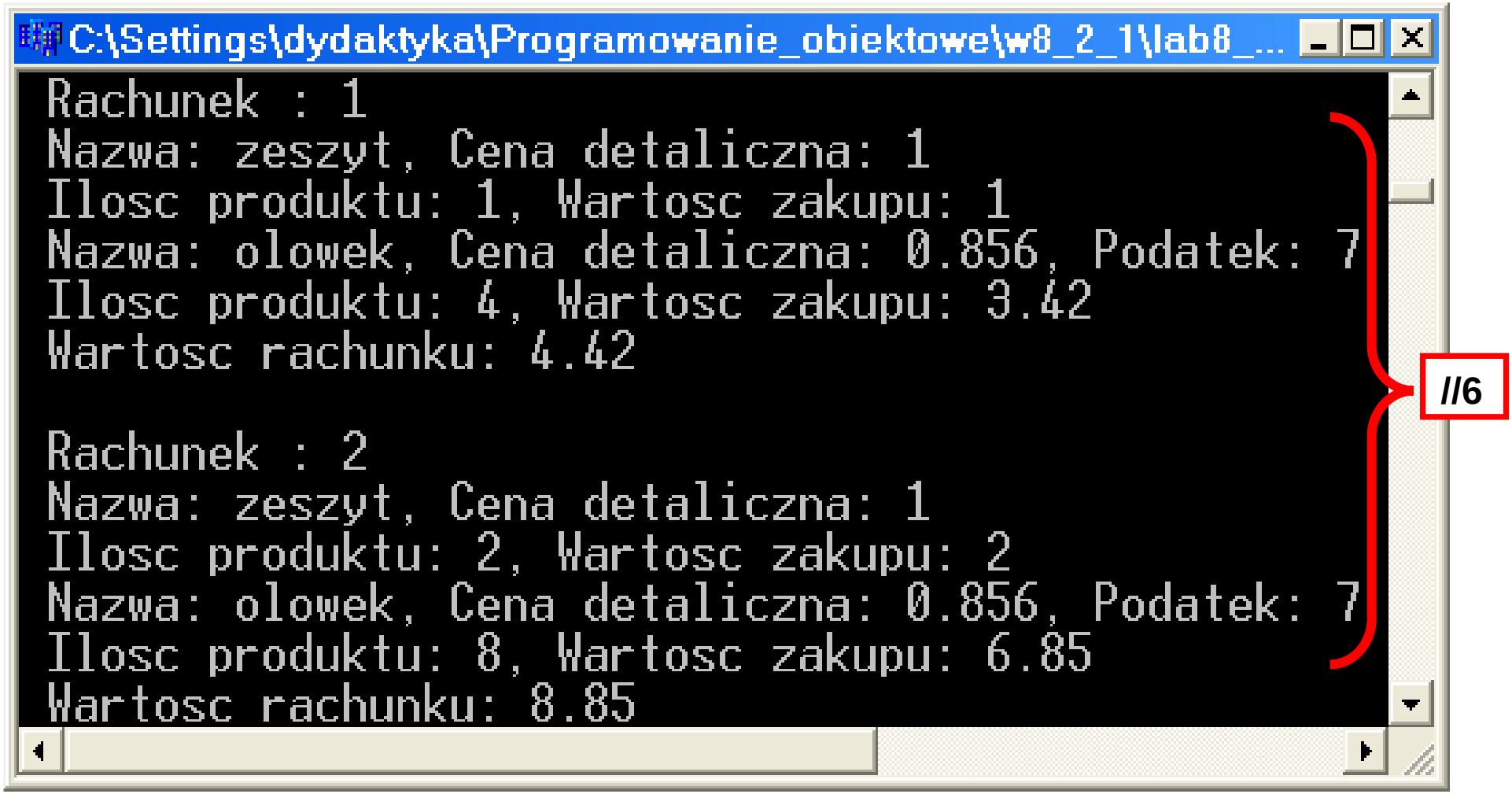
1. Szablony klas i funkcji
- Szablon klasy obsługującej uniwersalną tablicę wskaźników
- Zastosowanie metody zwracającej przez return referencję do składowych klasy
- Tworzenie zbioru rachunków zawierających zakupy różnych produktów

5. Agregacja silna – rola konstruktora kopiującego, przeciążonego operatora= oraz destruktora


main1.cpp
main1.cpp
Kol2.h
PRODUKT1.CPP
PRODUKT2.CPP
RACHUNEK.CPP
#include "Rachunek.h"
TKol2<TProdukt1> produkty;
TKol2<TRachunek> rachunki;
void Wstaw_zakup(TProdukt1* poszukiwanyprodukt,
int ilosc, int numer);
void main()
{
TProdukt1* p1;
TProdukt2* p2;
p1 = new TProdukt1("zeszyt", 1.0); produkty.Wstaw(p1);
p2 = new TProdukt2("olowek",0.80,7); produkty.Wstaw(p2);
cout<<produkty<<endl; //cout<<produkty.toString()<<endl;
//1
rachunki.Wstaw(new TRachunek(1));
rachunki.Wstaw(new TRachunek(2));
Wstaw_zakup(new TProdukt1("zeszyt", 1.0),1,1);
Wstaw_zakup(new TProdukt2("olowek",0.80,7),4,1);
Wstaw_zakup(new TProdukt1("zeszyt", 1.0),2,2);
Wstaw_zakup(new TProdukt2("olowek",0.80,7),8,2);
35: 22
Insert


main1.cpp
main1.cpp
Kol2.h
PRODUKT1.CPP
PRODUKT2.CPP
RACHUNEK.CP
{ TRachunek* pr1;
pr1 = rachunki.Podaj(new TRachunek(2));
TRachunek r2=*pr1;
//2
cout<<*pr1<<endl;
//3
cout<<r2<<endl;
pr1=rachunki.Podaj(new TRachunek(1));
r2=*pr1;
//4
cout<<*pr1<<endl;
//5
cout<<r2<<endl;
r2=r2;
}
//6
cout<<rachunki<<endl;
produkty.Usun_kolekcje();
rachunki.Usun_kolekcje();
cin.get();
}
1) Tworzenie automatycznego obiektu r2 za pomocą konstruktora kopiującego klasy TRachunek – obiekt r2 posiada kopię danych z rachunku o numerze 2
2) Zmiana zawartości obiektu r2 za pomocą operatora przypisania z klasy TRachunek – teraz obiekt r2 posiada kopię danych z rachunku o numerze 1
2) Tutaj zostanie wykryta identyczność obiektów – operator= z klasy TRachunek nie kopiuje obiektu r2 do r2
3) Tutaj usuwany jest obiekt r2 wraz z własną kolekcją zakupy oraz jego zawartością, utworzony przez konstruktor kopiujący (1) z kolekcją skopiowaną w (2)- działa destruktor ~TRachunek()
35: 35
Modified
Insert


C:\Settings\dydaktyka\Programowanie_obiektowe\w8_2_1\lab8_...
Nazwa: zeszyt, Cena detaliczna: 1
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Rachunek : 2
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 2, Wartosc zakupu: 2
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 8, Wartosc zakupu: 6.85
Wartosc rachunku: 8.85
Rachunek : 2
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 2, Wartosc zakupu: 2
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 8, Wartosc zakupu: 6.85
Wartosc rachunku: 8.85
Rachunek : 1
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 1, Wartosc zakupu: 1
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 4, Wartosc zakupu: 3.42
Wartosc rachunku: 4.42
Rachunek : 1
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 1, Wartosc zakupu: 1
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 4, Wartosc zakupu: 3.42
Wartosc rachunku: 4.42
//1
//2
//3
//4
//5


C:\Settings\dydaktyka\Programowanie_obiektowe\w8_2_1\lab8_...
Rachunek : 1
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 1, Wartosc zakupu: 1
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 4, Wartosc zakupu: 3.42
Wartosc rachunku: 4.42
Rachunek : 2
Nazwa: zeszyt, Cena detaliczna: 1
Ilosc produktu: 2, Wartosc zakupu: 2
Nazwa: olowek, Cena detaliczna: 0.856, Podatek: 7
Ilosc produktu: 8, Wartosc zakupu: 6.85
Wartosc rachunku: 8.85
//6