Dziedziczenie jednobazowe, poliformizm

1. Dziedziczenie jednobazowe
2. Polimorfizm – część pierwsza
3. Polimorfizm – część druga

Dziedziczenie jednobazowe, poliformizm

1. Dziedziczenie jednobazowe

1.1 Definicja nowych klas

Dziedziczenie to jeden z podstawowych paradygmatów programowania obiektowego (wykłady 1, 2).

Dziedziczenie pozwala na zdefiniowanie klasy pochodnej **B** na podstawie istniejącej klasy **A**, zwanej *klasą bazową*:

```
class B : public A // lub private A
{       //definicja nowych metod i pól typu dane
        // przedefiniowanie istniejących metod, czyli
        // definiowanie metod o identycznych nagłówkach
};
```

gdzie: **public** - dziedziczenie publiczne;
private - dziedziczenie prywatne

Uwaga:

Nazwy dowolnych składowych nie mogą się powtarzać w ramach klasy **A** lub **B**, natomiast w klasie **B** mogą wystąpić dowolne składowe o nazwach zastosowanych w klasie bazowej **A**. Wówczas należy je odróżniać za pomocą operatora **::**

1.2. Dostęp do składowych klasy bazowej A

1. w przypadku *dziedziczenia typu* **public:** dostęp klasy-następcy **B** do wszystkich składowych publicznych klasy bazowej **A**

2. w przypadku *dziedziczenia typu* **private:** składowe publiczne z klasy bazowej **A** stają się składowymi prywatnymi w klasie pochodnej (np. do całkowitej zmiany interfejsu).

3. w obu przypadkach dziedziczenia składowe prywatne nie są udostępniane klasie-następcy **B**

4. w celu ochrony składowych klasy bazowej **A** (hermetyzacja) można zastąpić specyfikator dostępu **private** specyfikatorem **protected**: takie składowe są publiczne dla klasy-następcy **B,** natomiast są prywatne dla jej dla użytkownika, który nie jest następcą.

5. metody dziedziczone bez przedefiniowania można wywołać bezpośrednio w ciele metod klasy- następcy **B**

   np. **metoda();**

6. metody dziedziczone na rzecz obiektu klasy **B** są wywoływane tak samo jak metody zdefiniowane w klasie **B**

   np. **B b;** **b.metoda ();**

   //metoda() jest dziedziczona od klasy **A**

---

7. jeśli metoda klasy **A** została przedefiniowana w klasie-następcy **B,** to w ciele metod klasy **B** można wywołać pierwotną metodę za pomocą operatora ::

   np. **A::metoda_p();**

8. metody dziedziczone z przedefiniowaniem na rzecz obiektu klasy **B** są wywoływane za pomocą operatora ::

   // metoda_p() jest dziedziczona z przedefiniowaniem od // klasy **A**

   np. **B b;** **b.A::metoda_p();**

1.3. Wywołanie konstruktorów i destruktorów

Tworzeniu obiektu typu **B** towarzyszy wywołanie (zazwyczaj niejawne) jego konstruktora i automatyczne wywołanie i wykonanie konstruktora z klasy **A**, a następnie wykonanie konstruktora klasy **B.**

- W przypadku braku listy argumentów w konstruktorze **B,** wywoływany i wykonywany jest konstruktor bezargumentowy klasy **A** (jawny lub domniemany).
- W przypadku zastosowania listy argumentów w klasie pochodnej **B,** można wybrać rodzaj konstruktora z klasy bazowej **A** do inicjowania pól dziedziczonych w **B**:

B(typ_1 a_1, ...typ_n a_n) : A(a_1, a_2);

Podczas usuwania obiektu typu B wywoływany i wykonany jest najpierw destruktor (jawny lub domniemany) klasy **B**, a potem destruktor klasy bazowej **A** (jawny lub domniemany).

C:\Settings\dydaktyka\Programowanie_obiektowe\w4_1\PRODUKT1.h

ZAKUP.h | MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PRODUKT2.CPP | PRI

```
#ifndef _Produkt1
#define _Produkt1
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
class TProdukt1
{protected:
        string nazwa;
        float cena;
public:
        TProdukt1(string nazwa_="bez nazwy",float cena_=0);// 1
        TProdukt1(TProdukt1&);
        ~TProdukt1();
        float Podaj_cene();
        int operator==(TProdukt1&);
        friend ostream& operator<<(ostream&, TProdukt1&); // 2
};
#endif
```

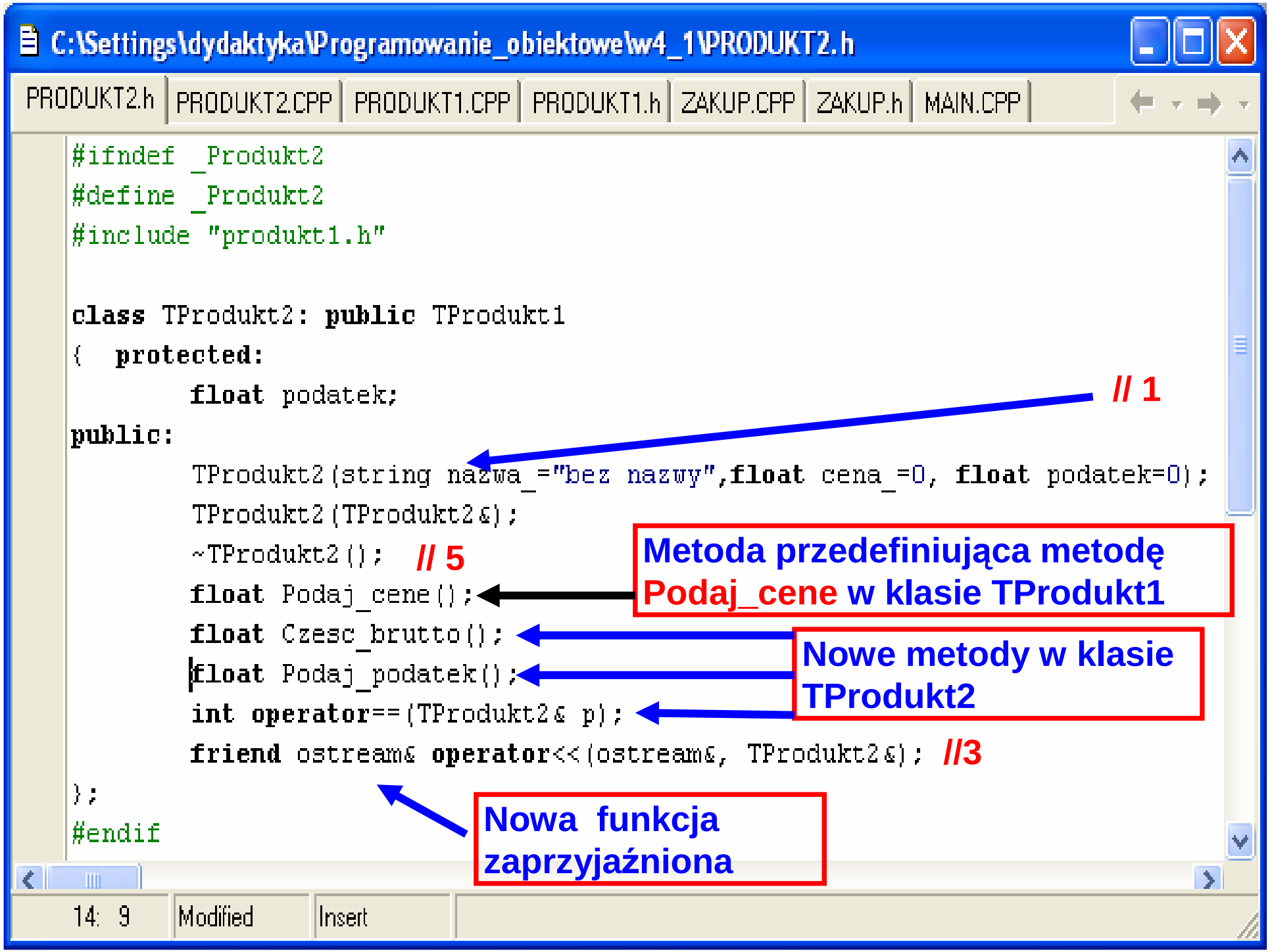
14: 1 | Insert

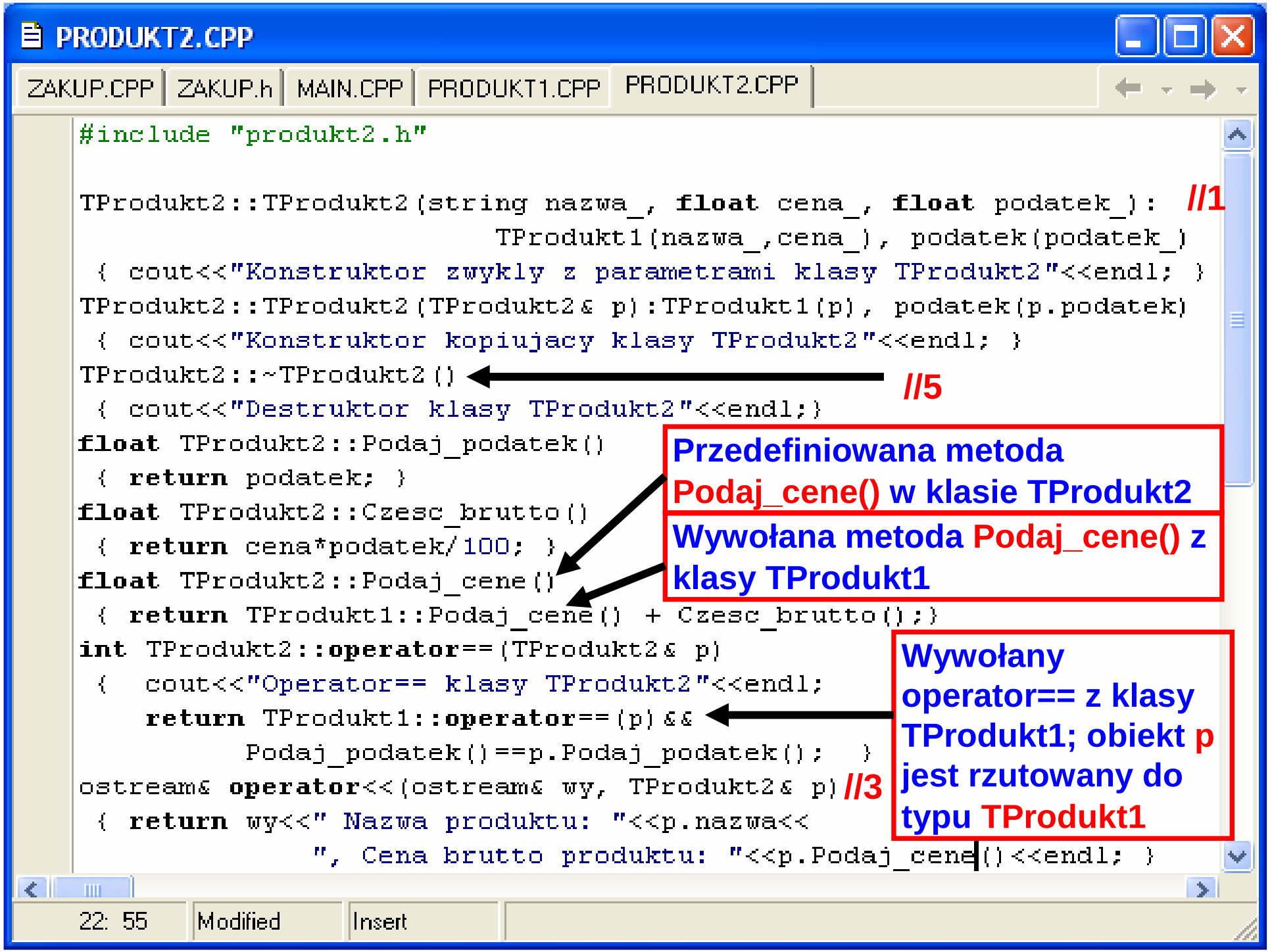
PRODUKT1.CPP

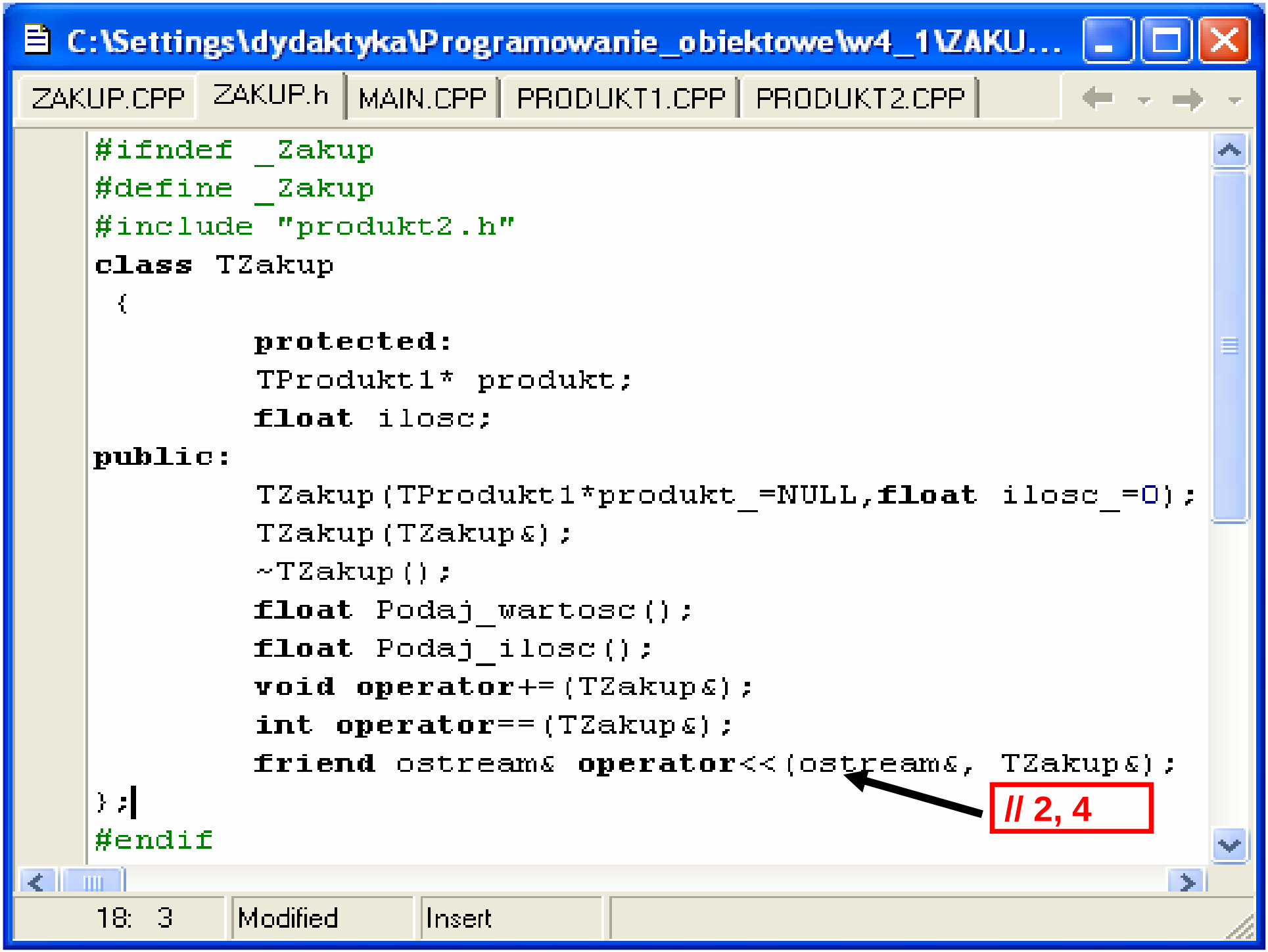
ZAKUP.CPP | ZAKUP.h | MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PRODUKT2.CPP | PF

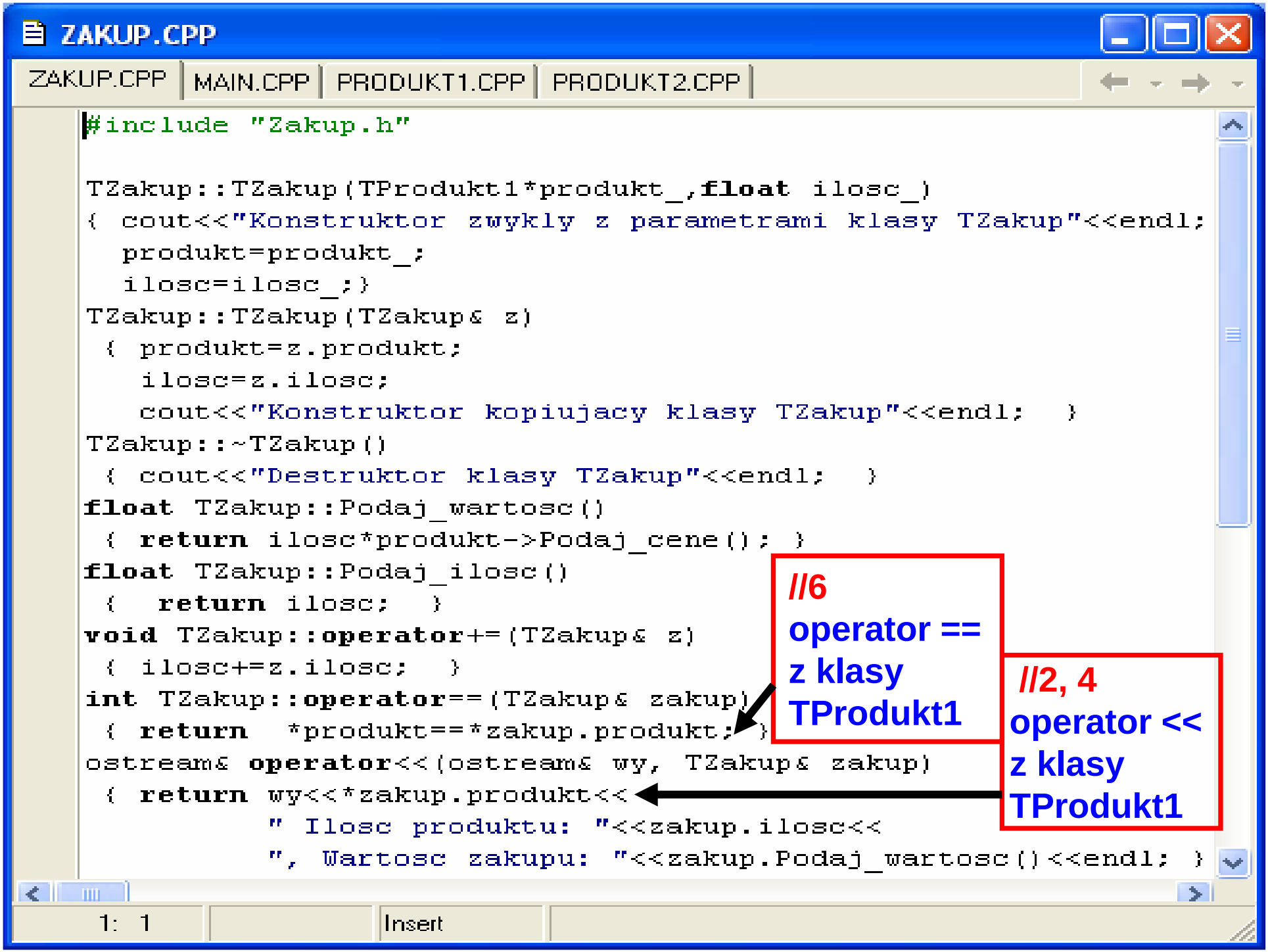
```
#include "produkt1.h"

TProdukt1::TProdukt1(string nazwa_,float cena_)      // 1
 { nazwa=nazwa_;
   cena=cena_;
   cout<<"Konstruktor zwykly z parametrami klasy TProdukt1"<<endl; }
TProdukt1::TProdukt1(TProdukt1& p)
 { nazwa=p.nazwa;
   cena=p.cena;
   cout<<"Konstruktor kopiujacy klasy TProdukt1"<<endl;}
TProdukt1::~TProdukt1()
 { cout<<"Destruktor klasy TProdukt1"<<endl; }
float TProdukt1::Podaj_cene()
 { return cena; }
int TProdukt1::operator==(TProdukt1& p)
 {  cout<<"Operator== klasy TProdukt1"<<endl;
    float a= Podaj_cene(), b= p.Podaj_cene();
    return nazwa==p.nazwa && a==b;   }
ostream& operator<<(ostream& wy, TProdukt1& p)      //2
 { return wy<<" Nazwa produktu: "<<p.nazwa<<
              ", Cena produktu: "<<p.Podaj_cene()<<endl; }
```

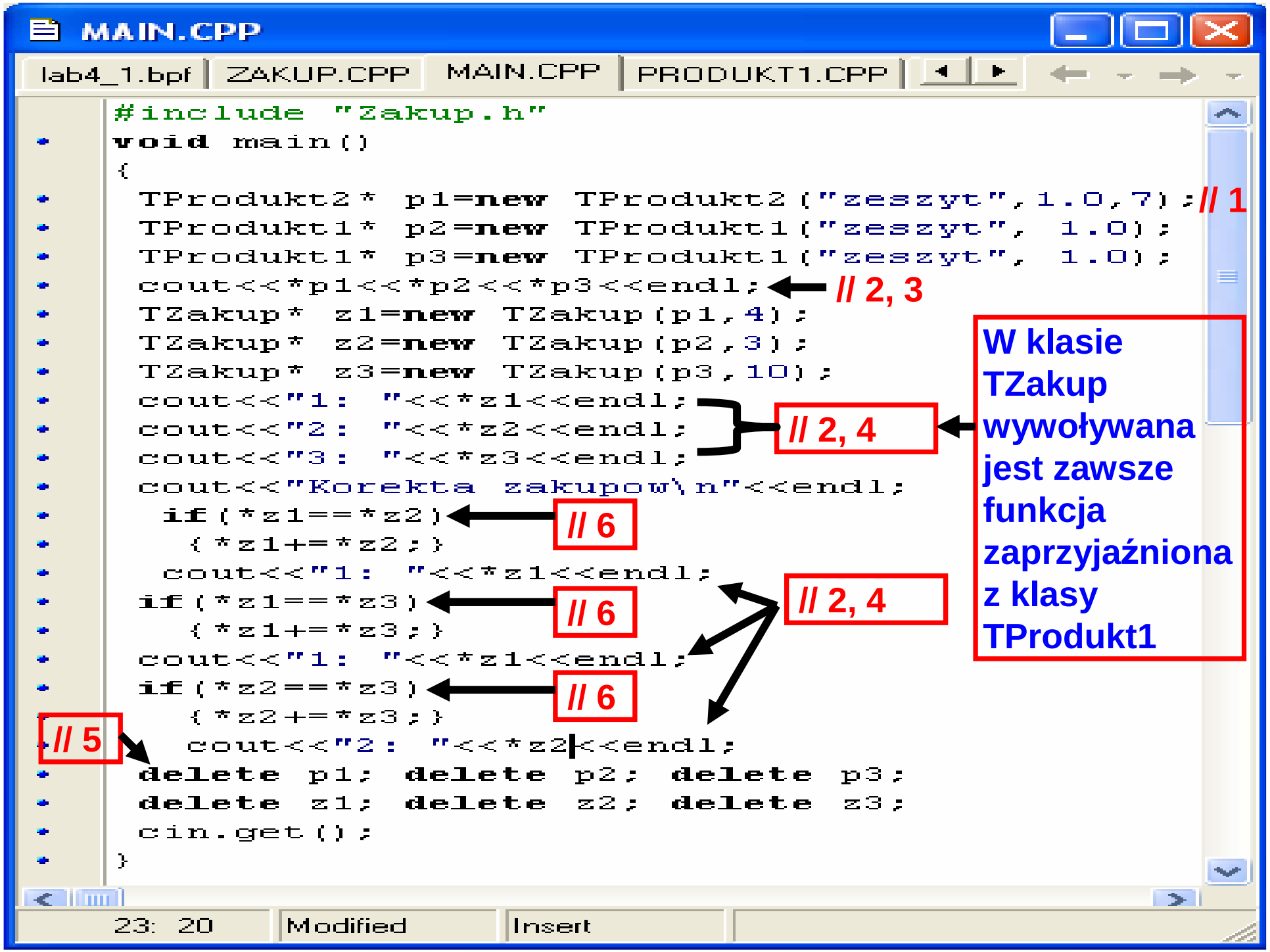
18: 34 | Modified | Insert

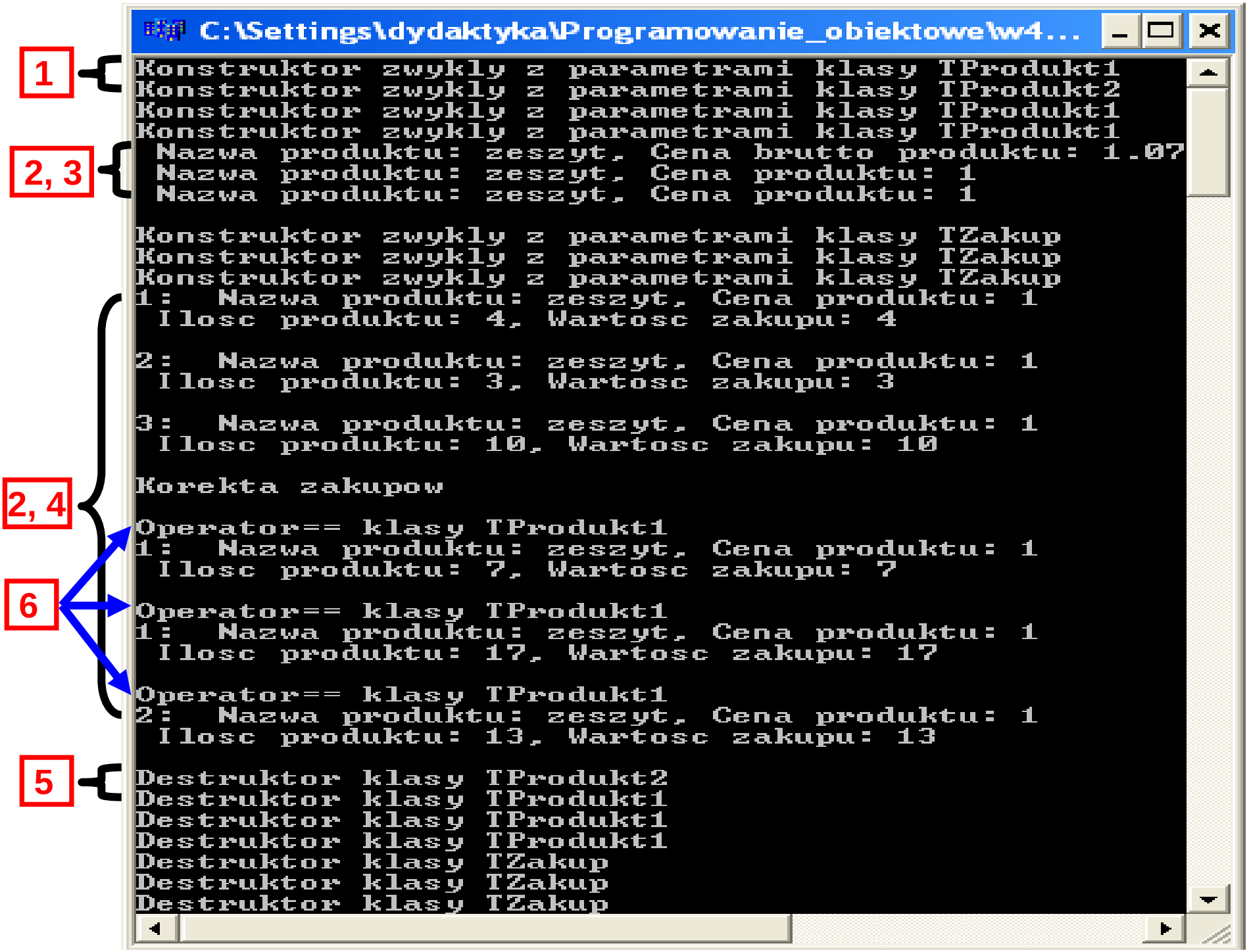

C:\Settings\dydaktyka\Programowanie_obiektowe\w4_1\PRODUKT2.h
PRODUKT2.h
PRODUKT2.CPP
PRODUKT1.CPP
PRODUKT1.h
ZAKUP.CPP
ZAKUP.h
MAIN.CPP
#ifndef _Produkt2
#define _Produkt2
#include "produkt1.h"
class TProdukt2: public TProdukt1
{ protected:
float podatek;
public:
TProdukt2(string nazwa_="bez nazwy",float cena_=0, float podatek=0);
TProdukt2(TProdukt2&);
~TProdukt2(); // 5
float Podaj_cene();
float Czesc_brutto();
float Podaj_podatek();
int operator==(TProdukt2& p);
friend ostream& operator<<(ostream&, TProdukt2&); //3
};
#endif
// 1
Metoda przedefiniująca metodę Podaj_cene w klasie TProdukt1
Nowe metody w klasie TProdukt2
Nowa funkcja zaprzyjaźniona
14: 9
Modified
Insert


PRODUKT2.CPP
ZAKUP.CPP
ZAKUP.h
MAIN.CPP
PRODUKT1.CPP
PRODUKT2.CPP
#include "produkt2.h"
TProdukt2::TProdukt2(string nazwa_, float cena_, float podatek_):
TProdukt1(nazwa_,cena_), podatek(podatek_)
{ cout<<"Konstruktor zwykly z parametrami klasy TProdukt2"<<endl; }
TProdukt2::TProdukt2(TProdukt2& p):TProdukt1(p), podatek(p.podatek)
{ cout<<"Konstruktor kopiujacy klasy TProdukt2"<<endl; }
TProdukt2::~TProdukt2()
{ cout<<"Destruktor klasy TProdukt2"<<endl;}
float TProdukt2::Podaj_podatek()
{ return podatek; }
float TProdukt2::Czesc_brutto()
{ return cena*podatek/100; }
float TProdukt2::Podaj_cene()
{ return TProdukt1::Podaj_cene() + Czesc_brutto();}
int TProdukt2::operator==(TProdukt2& p)
{ cout<<"Operator== klasy TProdukt2"<<endl;
return TProdukt1::operator==(p)&&
Podaj_podatek()==p.Podaj_podatek(); }
ostream& operator<<(ostream& wy, TProdukt2& p)
{ return wy<<" Nazwa produktu: "<<p.nazwa<<
", Cena brutto produktu: "<<p.Podaj_cene()<<endl; }
//1
//5
//3
Przedefiniowana metoda Podaj_cene() w klasie TProdukt2
Wywołana metoda Podaj_cene() z klasy TProdukt1
Wywołany operator== z klasy TProdukt1; obiekt p jest rzutowany do typu TProdukt1
22: 55
Modified
Insert


C:\Settings\dydaktyka\Programowanie_obiektowe\w4_1\ZAKU...
ZAKUP.CPP
ZAKUP.h
MAIN.CPP
PRODUKT1.CPP
PRODUKT2.CPP
#ifndef _Zakup
#define _Zakup
#include "produkt2.h"
class TZakup
{
protected:
TProdukt1* produkt;
float ilosc;
public:
TZakup(TProdukt1*produkt_=NULL,float ilosc_=0);
TZakup(TZakup&);
~TZakup();
float Podaj_wartosc();
float Podaj_ilosc();
void operator+=(TZakup&);
int operator==(TZakup&);
friend ostream& operator<<(ostream&, TZakup&);
};
#endif
// 2, 4
18: 3
Modified
Insert


ZAKUP.CPP
ZAKUP.CPP
MAIN.CPP
PRODUKT1.CPP
PRODUKT2.CPP
#include "Zakup.h"
TZakup::TZakup(TProdukt1*produkt_,float ilosc_)
{ cout<<"Konstruktor zwykly z parametrami klasy TZakup"<<endl;
produkt=produkt_;
ilosc=ilosc_;}
TZakup::TZakup(TZakup& z)
{ produkt=z.produkt;
ilosc=z.ilosc;
cout<<"Konstruktor kopiujacy klasy TZakup"<<endl; }
TZakup::~TZakup()
{ cout<<"Destruktor klasy TZakup"<<endl; }
float TZakup::Podaj_wartosc()
{ return ilosc*produkt->Podaj_cene(); }
float TZakup::Podaj_ilosc()
{ return ilosc; }
void TZakup::operator+=(TZakup& z)
{ ilosc+=z.ilosc; }
int TZakup::operator==(TZakup& zakup)
{ return *produkt==*zakup.produkt; }
ostream& operator<<(ostream& wy, TZakup& zakup)
{ return wy<<*zakup.produkt<<
" Ilosc produktu: "<<zakup.ilosc<<
", Wartosc zakupu: "<<zakup.Podaj_wartosc()<<endl; }
//6
operator ==
z klasy
TProdukt1
//2, 4
operator <<
z klasy
TProdukt1
1: 1
Insert

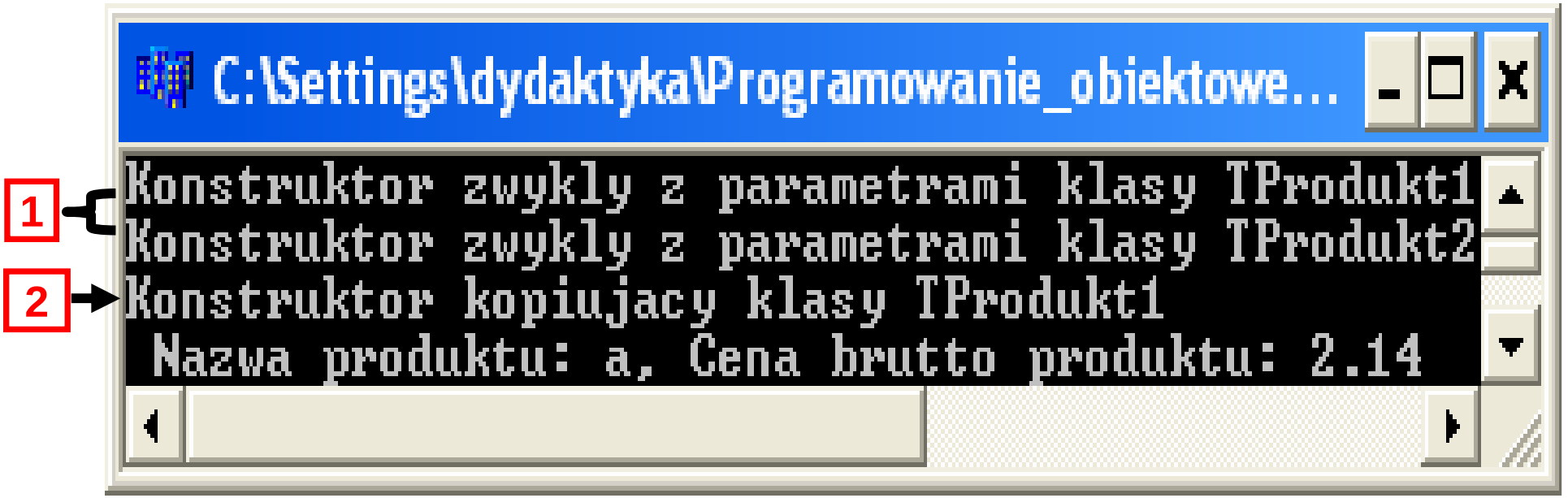

MAIN.CPP
lab4_1.bpf
ZAKUP.CPP
MAIN.CPP
PRODUKT1.CPP
#include "Zakup.h"
void main()
{
TProdukt2* p1=new TProdukt2("zeszyt",1.0,7);
TProdukt1* p2=new TProdukt1("zeszyt", 1.0);
TProdukt1* p3=new TProdukt1("zeszyt", 1.0);
cout<<*p1<<*p2<<*p3<<endl;
TZakup* z1=new TZakup(p1,4);
TZakup* z2=new TZakup(p2,3);
TZakup* z3=new TZakup(p3,10);
cout<<"1: "<<*z1<<endl;
cout<<"2: "<<*z2<<endl;
cout<<"3: "<<*z3<<endl;
cout<<"Korekta zakupow\n"<<endl;
if(*z1==*z2)
{*z1+=*z2;}
cout<<"1: "<<*z1<<endl;
if(*z1==*z3)
{*z1+=*z3;}
cout<<"1: "<<*z1<<endl;
if(*z2==*z3)
{*z2+=*z3;}
cout<<"2: "<<*z2<<endl;
delete p1; delete p2; delete p3;
delete z1; delete z2; delete z3;
cin.get();
}
// 1
// 2, 3
// 2, 4
// 6
// 6
// 2, 4
// 6
// 5
W klasie TZakup wywoływana jest zawsze funkcja zaprzyjaźniona z klasy TProdukt1
23: 20
Modified
Insert


C:\Settings\dydaktyka\Programowanie_obiektowe\w4...
1
2, 3
2, 4
6
5
Konstruktor zwykly z parametrami klasy TProdukt1
Konstruktor zwykly z parametrami klasy TProdukt2
Konstruktor zwykly z parametrami klasy TProdukt1
Konstruktor zwykly z parametrami klasy TProdukt1
Nazwa produktu: zeszyt, Cena brutto produktu: 1.07
Nazwa produktu: zeszyt, Cena produktu: 1
Nazwa produktu: zeszyt, Cena produktu: 1
Konstruktor zwykly z parametrami klasy TZakup
Konstruktor zwykly z parametrami klasy TZakup
Konstruktor zwykly z parametrami klasy TZakup
1: Nazwa produktu: zeszyt, Cena produktu: 1
Ilosc produktu: 4, Wartosc zakupu: 4
2: Nazwa produktu: zeszyt, Cena produktu: 1
Ilosc produktu: 3, Wartosc zakupu: 3
3: Nazwa produktu: zeszyt, Cena produktu: 1
Ilosc produktu: 10, Wartosc zakupu: 10
Korekta zakupow
Operator== klasy TProdukt1
1: Nazwa produktu: zeszyt, Cena produktu: 1
Ilosc produktu: 7, Wartosc zakupu: 7
Operator== klasy TProdukt1
1: Nazwa produktu: zeszyt, Cena produktu: 1
Ilosc produktu: 17, Wartosc zakupu: 17
Operator== klasy TProdukt1
2: Nazwa produktu: zeszyt, Cena produktu: 1
Ilosc produktu: 13, Wartosc zakupu: 13
Destruktor klasy TProdukt2
Destruktor klasy TProdukt1
Destruktor klasy TProdukt1
Destruktor klasy TProdukt1
Destruktor klasy TZakup
Destruktor klasy TZakup
Destruktor klasy TZakup

1.4. Wywołanie konstruktora kopiującego

1. *w przypadku, gdy w klasie* **B** *nie ma jawnego konstruktora kopiującego*: wywoływany jest konstruktor kopiujący domniemany, który wywołuje konstruktor kopiujący z **A** (jawny lub domniemany, inicjujący pola w **B** odziedziczone od **A**) i inicjuje pozostałe pola w **B**

```
TProdukt2  k1("a", 2, 7);   //1
TProdukt2  k2=k1;   cout<<k2<<endl; //2
```



2. *w przypadku, gdy w klasie* **B** *jest jawny konstruktor kopiujący z listą argumentów*: wywołuje on dowolny konstruktor klasy bazowej **A,** wymieniony w liście argumentów (kopiujący lub zwykły) np.:

- ***konstruktor kopiujący z klasy A*:**

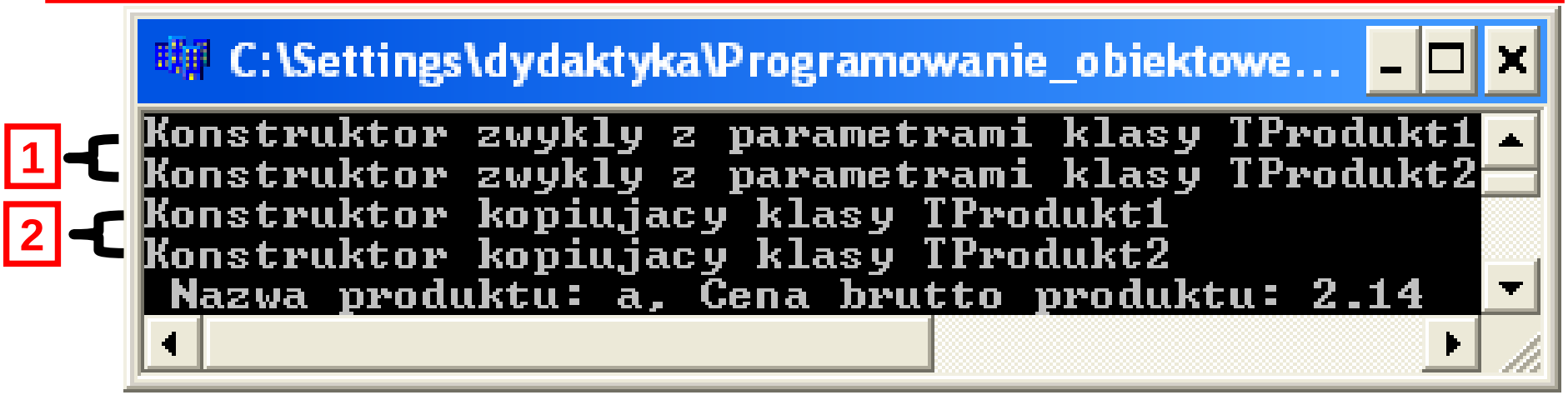
B(B&b) : A(b)

Zalecana definicja

(wywołanie konstruktora kopiującego z A, do którego zostanie przekazana część obiektu klasy B odziedziczona od klasy A);

```
TProdukt2::TProdukt2(TProdukt2& p):TProdukt1(p), podatek(p.podatek)
 { cout<<"Konstruktor kopiujacy klasy TProdukt2"<<endl; }
```

```
TProdukt2  k1("a", 2, 7);  //1
TProdukt2  k2=k1;   cout<<k2<<endl;  //2
```



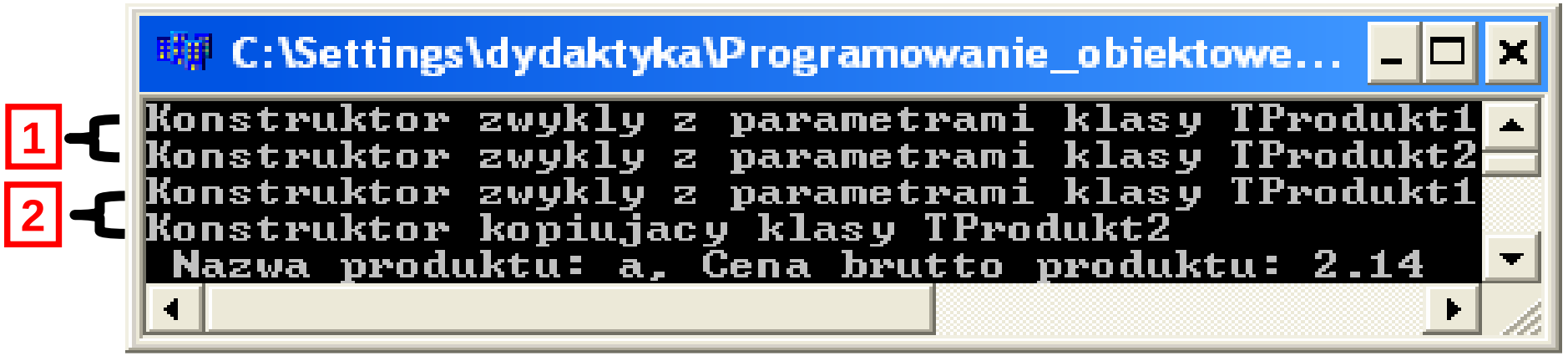
- *konstruktor zwykły z klasy **A***:

  B(**B**&b) : **A**(b.skladowa_1(), b.skladowa_2())

  (wywołanie zwykłego konstruktora z klasy bazowej **A**, do którego przekazuje się dane odziedziczone przez obiekt klasy **B** od klasy **A**)

```
TProdukt2::TProdukt2(TProdukt2& p):TProdukt1(p.nazwa,p.cena),
                                                    podatek(p.podatek)
 { cout<<"Konstruktor kopiujacy klasy TProdukt2"<<endl; }
```

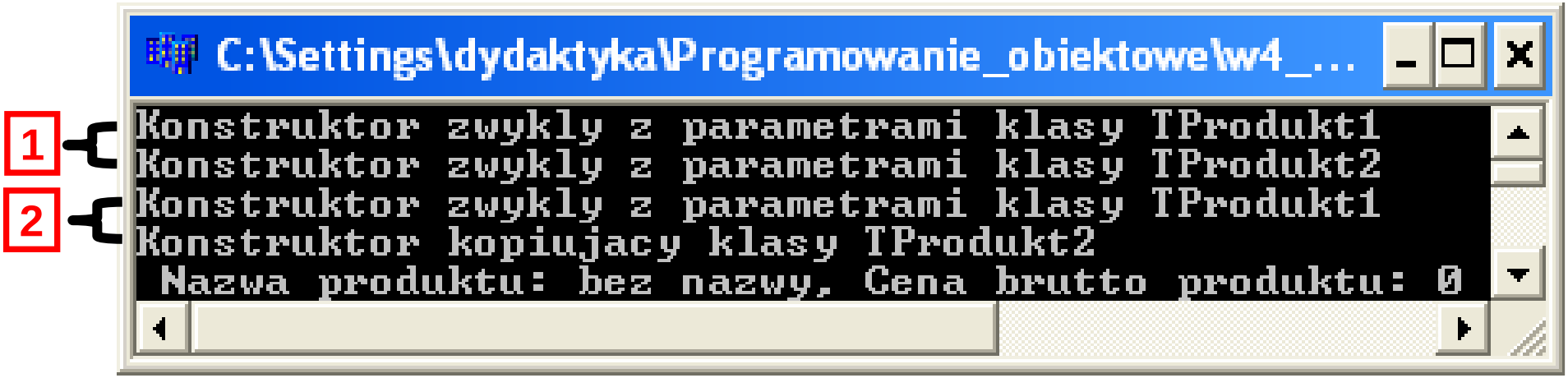
```
TProdukt2  k1("a", 2, 7);  //1
TProdukt2  k2=k1;   cout<<k2<<endl;  //2
```



3. *w przypadku, gdy w klasie* **B** *jest jawny konstruktor kopiujący, lecz nie zastosowano listy argumentów:*
wywołuje on konstruktor bezargumentowy klasy **A**
(domyślny, jawny –również z parametrami domniemanymi)

```
TProdukt2::TProdukt2(TProdukt2& p)
 { cout<<"Konstruktor kopiujacy klasy TProdukt2"<<endl; }
```

```
TProdukt2  k1("a", 2, 7);  //1
TProdukt2  k2=k1;   cout<<k2<<endl;  //2
```



1.5. Własności

- nie można zmienić pól zdefiniowanych w klasie bazowej
- można przedefiniować metody w klasie pochodnej
- metody dziedziczone działają na obiekcie klasy pochodnej **B** tak, jak gdyby był obiektem klasy bazowej **A,**
- dziedziczeniu nie podlegają: konstruktory, destruktory, operator = oraz zaprzyjaźnienia z klasami i funkcjami
- istnieją standardowe konwersje typu z klasy pochodnej do klasy bazowej dla **obiektów, wskaźników i referencji**, dostępne poza zakresem klasy pochodnej w przypadku publicznej klasy bazowej:

klasa_bazowa ← **klasa_pochodna**

klasa_bazowa * ← **klasa_pochodna ***

klasa_bazowa & ← **klasa_pochodna &**

W wyrażeniu

obiekt_klasy_bazowej = obiekt_klasy_pochodnej

kopiowane z obiektu klasy pochodnej do obiektu klasy bazowej tylko pola wspólne w obu klasach w wyniku dziedziczenia

```
TProdukt1  p1("a", 1), p2, *p3;
TProdukt2  k1("b", 2), k2, *k3;
p2 = p1;                cout<<p2<<endl;        //na ekranie dane p1
k2 = k1;                cout<<k2<<endl;        //na ekranie dane k1
k3 = &k2;               cout<<*k3<<endl;       //na ekranie dane k2
TProdukt2 & k4 = k2;  cout<<k4<<endl;    //na ekranie dane k2
```

Kopiowane są tylko składowe nazwa i cena od obiektu klasy TProdukt2

```
//na ekranie dane z obiektu k1 dziedziczone od klasy TProdukt1 skopiowane
//do obiektu p2
p2 = k1;                cout<<p2<<endl;
```

Zmienna wskaźnikowa p3 jest adresem części typu klasy TProdukt1 w obiekcie k1 klasy TProdukt2

p3 = &k1; **cout<<*p3<<endl;** //na ekranie dane k1 dziedziczone

p3 = k3; **cout<<*p3<<endl;** //na ekranie dane k3 dziedziczone

Zmienna referencyjna p4 jest częścią typu klasy TProdukt1 w obiekcie k2 klasy TProdukt2

//na ekranie dane k2 dziedziczone

TProdukt1 & p4 = k2; cout<<p4<<endl;

Odwrotne przypisania są błędne, lecz możliwe jest rzutowanie wskaźników lub referencji.

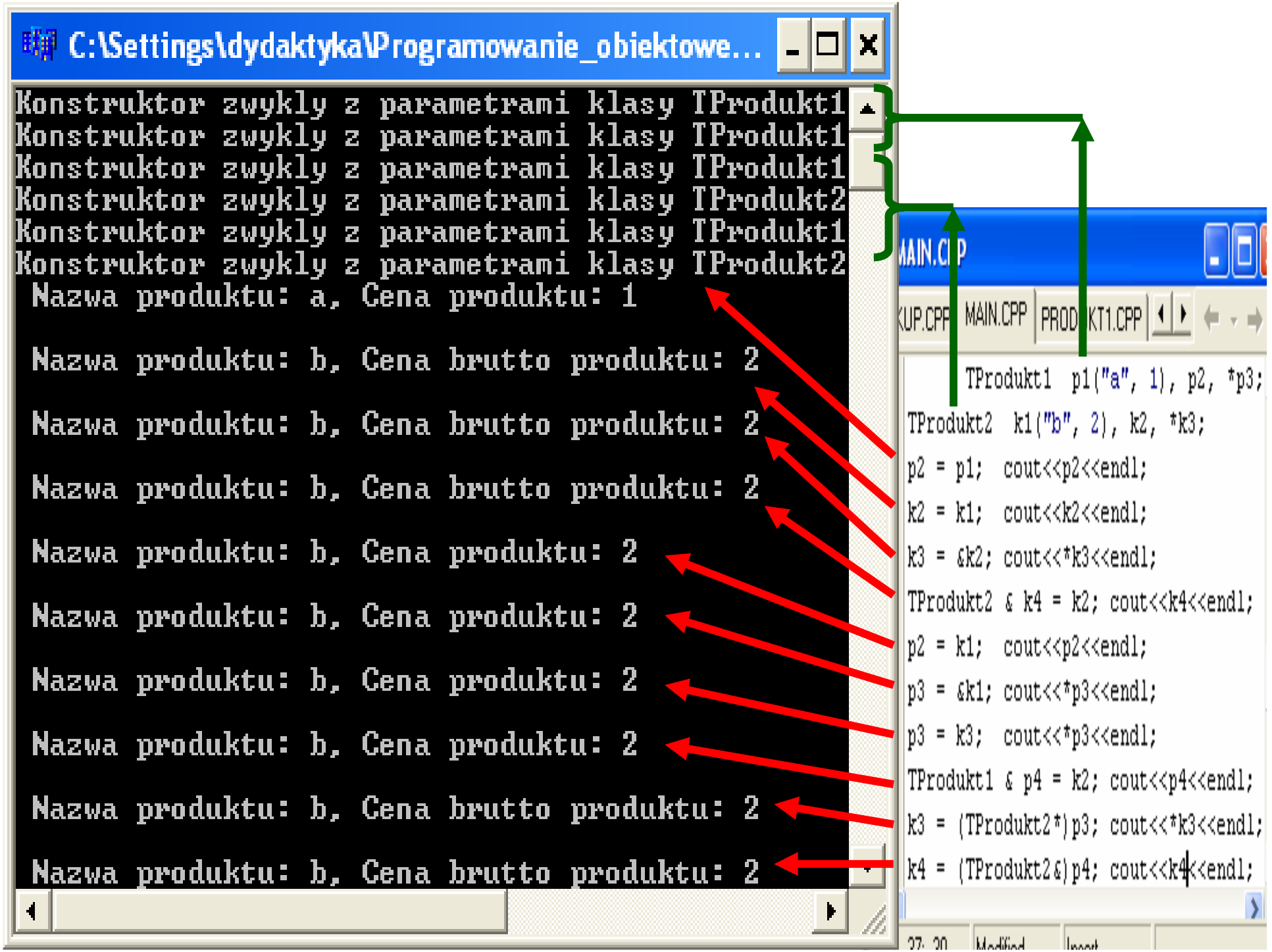
Można to robić, jeśli rzutowane wskaźniki i referencje wskazują na obiekty typów, do których są rzutowane

//na ekranie dane wskazywane przez p3 interpretowane jako dane wskazane przez //wskaźnik k3 na obiekt typu TProdukt2- poprawnie, ponieważ p3 zawiera wskaźnik k3

k3 = (TProdukt2*)p3; cout<<*k3<<endl;

//na ekranie dane p4 interpretowane jako referencja k4 do obiektu typu TProdukt2; //poprawnie, ponieważ p4 jest referencją do obiektu k2

k4 = (TProdukt2&)p4; cout<<*k4<<endl;


C:\Settings\dydaktyka\Programowanie_obiektowe...
Konstruktor zwykly z parametrami klasy TProdukt1
Konstruktor zwykly z parametrami klasy TProdukt1
Konstruktor zwykly z parametrami klasy TProdukt1
Konstruktor zwykly z parametrami klasy TProdukt2
Konstruktor zwykly z parametrami klasy TProdukt1
Konstruktor zwykly z parametrami klasy TProdukt2
Nazwa produktu: a, Cena produktu: 1
Nazwa produktu: b, Cena brutto produktu: 2
Nazwa produktu: b, Cena brutto produktu: 2
Nazwa produktu: b, Cena brutto produktu: 2
Nazwa produktu: b, Cena produktu: 2
Nazwa produktu: b, Cena produktu: 2
Nazwa produktu: b, Cena produktu: 2
Nazwa produktu: b, Cena produktu: 2
Nazwa produktu: b, Cena brutto produktu: 2
Nazwa produktu: b, Cena brutto produktu: 2
MAIN.CPP
MAIN.CPP
TProdukt1 p1("a", 1), p2, *p3;
TProdukt2 k1("b", 2), k2, *k3;
p2 = p1; cout<<p2<<endl;
k2 = k1; cout<<k2<<endl;
k3 = &k2; cout<<*k3<<endl;
TProdukt2 & k4 = k2; cout<<k4<<endl;
p2 = k1; cout<<p2<<endl;
p3 = &k1; cout<<*p3<<endl;
p3 = k3; cout<<*p3<<endl;
TProdukt1 & p4 = k2; cout<<p4<<endl;
k3 = (TProdukt2*)p3; cout<<*k3<<endl;
k4 = (TProdukt2&)p4; cout<<k4<<endl;

Dziedziczenie jednobazowe, poliformizm

1. **Dziedziczenie jednobazowe**
2. **Polimorfizm – część 1**

Metody wirtualne, polimorfizm

Kompilator ustala adresy wszystkich wywołanych metod bez słowa **virtual** na etapie kompilacji (**statyczne wiązanie funkcji**) następująco:

a) poszukiwania metody rozpoczyna na poziomie rozpoznanej klasy-jeśli nie znajdzie, kontynuuje poszukiwania na poziomie klas bazowych i dalej w klasach poprzedników klas bazowych

b) kieruje się zasadami konwersji typów obiektów, wskaźników i referencji (zawsze od typu pochodnego do typu poprzedników)

Uwaga: W przypadku użycia wskaźnika lub parametru referencyjnego typ tych parametrów może być różny od wskazywanego obiektu. Jednak kompilator przy ustalaniu adresów wywołanych metod na rzecz wskazanych obiektów kieruje typem wskaźnika lub zmiennej referencyjnej, a nie typem rzeczywistego obiektu.

```
class a
{.....
   public : void fun ( )
      { /*  kod */}
};
class b : public a
{.....
  public :  void fun ( )
        { /*  inny kod */}
};
a* A;
b* B = new  b;
A = B;          //wskaźnik A wskazuje na obiekt klasy pochodnej b
A→fun();        //jednak wywołano metodę z klasy a, która przetwarza
                //dane w obszarze wskazywanym przez B
B→fun();        //tutaj zostanie wywołana metoda przedefiniowana klasy b
```

Jeżeli w klasie bazowej *a* zadeklarujemy metodę wirtualną przy użyciu słowa kluczowego **virtual**, to metoda *fun* będzie wywołana z tej klasy, z której pochodzi obiekt wywołujący metodę - kompilator nie ustala na etapie kompilacji adresu metody, lecz w czasie działania programu.

Jest to dynamiczne wiązanie funkcji i zwane jest *polimorfizmem.*

```
class a
{.....
   virtual public : void fun ( ) { /*  kod */}
};

class b : public a
{.................
  public :  void fun ( ) { /*  inny kod */}
};
a * A;
b * B = new b;
A = B;              // wskaźnik A wskazuje na obiekt klasy pochodnej b
A→fun();            // wywołano metodę z klasy b, która przetwarza dane
                    // w obszarze wskazywanym przez B
B→fun();            // tutaj zostanie wywołana metoda przedefiniowana klasy b
```

Reguły stosowania metod wirtualnych:

1. Słowo kluczowe **virtual** jest używane tylko raz dla metody i nie powinno być używane dla metod przedefiniowanych w klasach pochodnych
2. Przedefiniowanie metody wirtualnej jest możliwe przy zachowaniu identycznego nagłówka metody w klasie pochodnej
3. Metoda zadeklarowana w funkcji bazowej jako wirtualna nie musi być przedefiniowana w klasach pochodnych
4. Funkcja wirtualna może być przeciążona, każda funkcja przeciążona może być, ale nie musi, funkcją wirtualną
5. Konstruktor nie może być wirtualny, natomiast destruktor może nim być
6. Z samej zasady wynika, że **wiązanie dynamiczne** jest używane tylko dla hierarchii klas; często, aby wykorzystać je dla całej biblioteki klas, wprowadza się dziedziczenie wszystkich klas z biblioteki po jednej klasie bazowej.

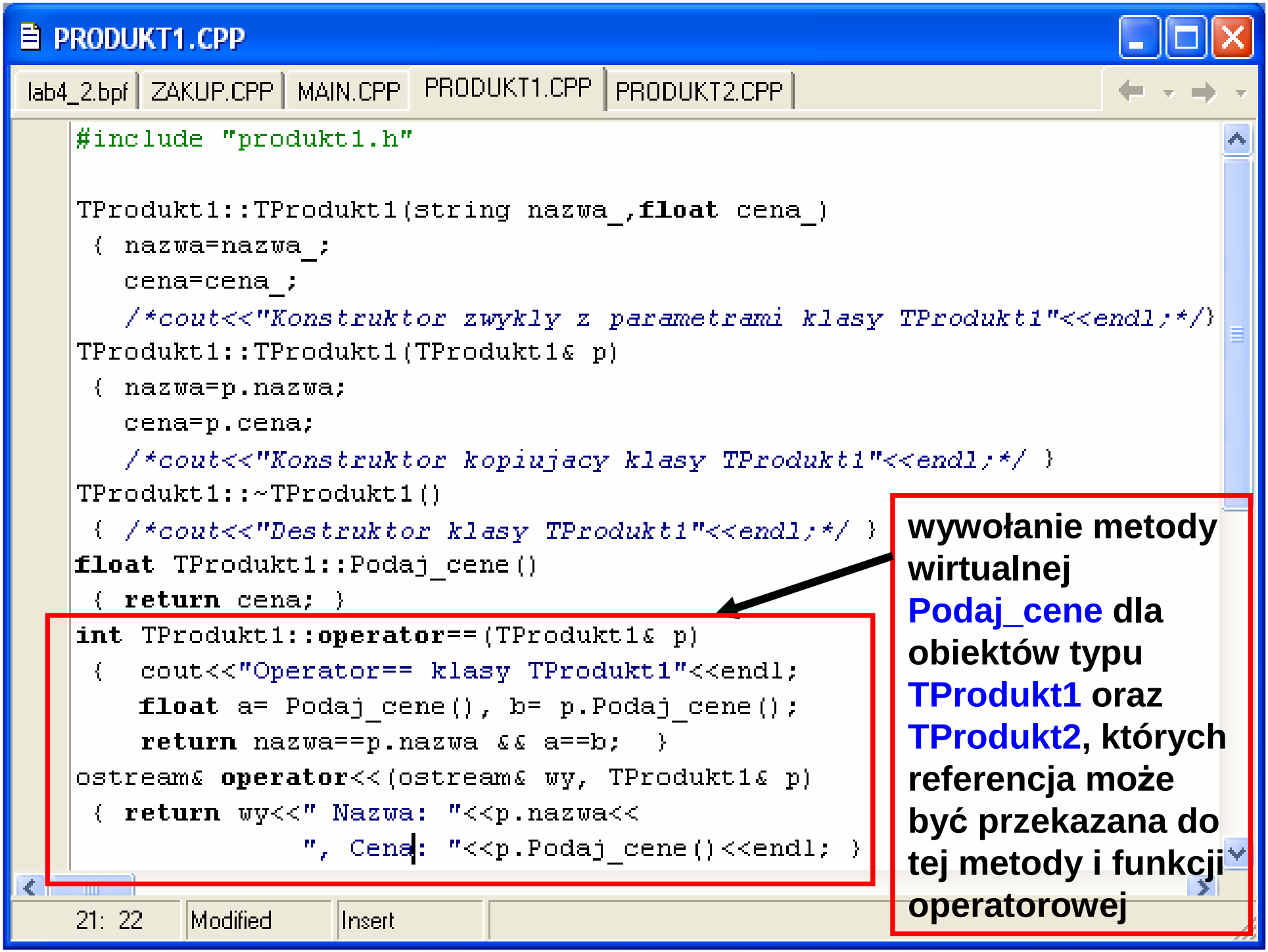
C:\Settings\dydaktyka\Programowanie_obiektowe\w4_2\PRODUKT1.h

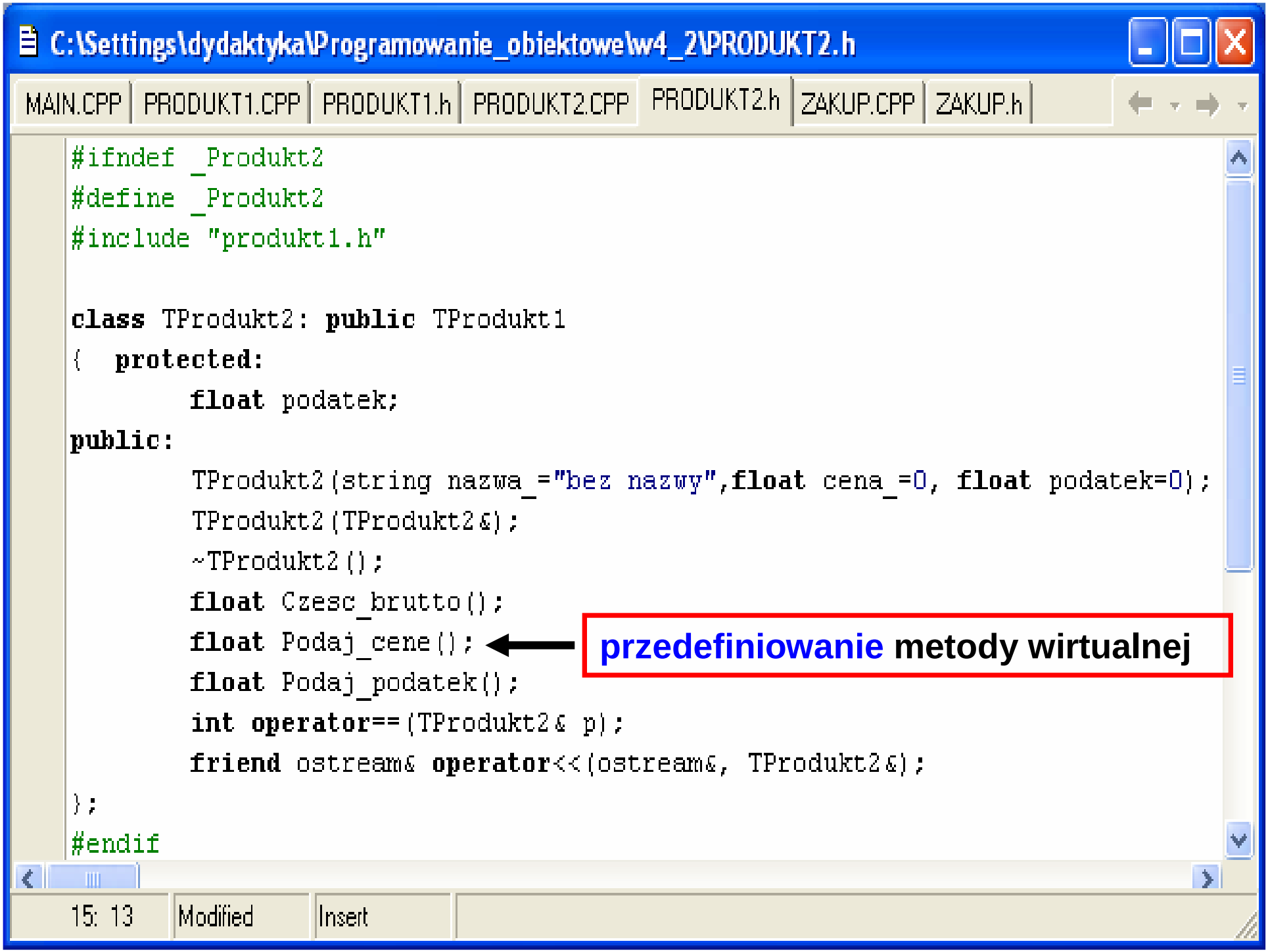
ZAKUP.CPP | ZAKUP.h | MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PRODL

```
#ifndef _Produkt1
#define _Produkt1
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
class TProdukt1
{protected:
        string nazwa;
        float cena;
public:
        TProdukt1(string nazwa_="bez nazwy",float cena_=0);
        TProdukt1(TProdukt1&);
        ~TProdukt1();
        virtual float Podaj_cene();
        int operator==(TProdukt1&);
        friend ostream& operator<<(ostream&, TProdukt1&);
};
#endif
```

metoda wirtualna

14: 17 | Modified | Insert


PRODUKT1.CPP
lab4_2.bpf
ZAKUP.CPP
MAIN.CPP
PRODUKT1.CPP
PRODUKT2.CPP
#include "produkt1.h"
TProdukt1::TProdukt1(string nazwa_,float cena_)
{ nazwa=nazwa_;
cena=cena_;
/*cout<<"Konstruktor zwykly z parametrami klasy TProdukt1"<<endl;*/}
TProdukt1::TProdukt1(TProdukt1& p)
{ nazwa=p.nazwa;
cena=p.cena;
/*cout<<"Konstruktor kopiujacy klasy TProdukt1"<<endl;*/ }
TProdukt1::~TProdukt1()
{ /*cout<<"Destruktor klasy TProdukt1"<<endl;*/ }
float TProdukt1::Podaj_cene()
{ return cena; }
int TProdukt1::operator==(TProdukt1& p)
{ cout<<"Operator== klasy TProdukt1"<<endl;
float a= Podaj_cene(), b= p.Podaj_cene();
return nazwa==p.nazwa && a==b; }
ostream& operator<<(ostream& wy, TProdukt1& p)
{ return wy<<" Nazwa: "<<p.nazwa<<
", Cena: "<<p.Podaj_cene()<<endl; }
21: 22
Modified
Insert
wywołanie metody wirtualnej Podaj_cene dla obiektów typu TProdukt1 oraz TProdukt2, których referencja może być przekazana do tej metody i funkcji operatorowej


C:\Settings\dydaktyka\Programowanie_obiektowe\w4_2\PRODUKT2.h
MAIN.CPP
PRODUKT1.CPP
PRODUKT1.h
PRODUKT2.CPP
PRODUKT2.h
ZAKUP.CPP
ZAKUP.h
#ifndef _Produkt2
#define _Produkt2
#include "produkt1.h"
class TProdukt2: public TProdukt1
{ protected:
float podatek;
public:
TProdukt2(string nazwa_="bez nazwy",float cena_=0, float podatek=0);
TProdukt2(TProdukt2&);
~TProdukt2();
float Czesc_brutto();
float Podaj_cene();
float Podaj_podatek();
int operator==(TProdukt2& p);
friend ostream& operator<<(ostream&, TProdukt2&);
};
#endif
przedefiniowanie metody wirtualnej
15: 13
Modified
Insert

PRODUKT2.CPP

lab4_2.bpf | ZAKUP.CPP | MAIN.CPP | PRODUKT1.CPP | PRODUKT2.CPP | PRODUKT2.h

```
#include "produkt2.h"

TProdukt2::TProdukt2(string nazwa_, float cena_, float podatek_):
                              TProdukt1(nazwa_,cena_), podatek(podatek_)
 {/*cout<<"Konstruktor zwykly z parametrami klasy TProdukt2"<<endl;*/ }
TProdukt2::TProdukt2(TProdukt2& p):TProdukt1(p), podatek(p.podatek)
 { /*cout<<"Konstruktor kopiujacy klasy TProdukt2"<<endl;*/ }
TProdukt2::~TProdukt2()
 { /*cout<<"Destruktor klasy TProdukt2"<<endl;*/}
float TProdukt2::Podaj_podatek()
 { return podatek; }
float TProdukt2::Czesc_brutto()
 { return cena*podatek/100; }
float TProdukt2::Podaj_cene()
 { return TProdukt1::Podaj_cene() + Czesc_brutto();}
int TProdukt2::operator==(TProdukt2& p)
 {  cout<<"Operator== klasy TProdukt2"<<endl;
    return TProdukt1::operator==(p)&&
           Podaj_podatek()==p.Podaj_podatek();  }
ostream& operator<<(ostream& wy, TProdukt2& p)
 { return wy<<" Nazwa: "<<p.nazwa<<
              ", Cena brutto: "<<p.Podaj_cene()<<
              ", Podatek: "<<p.Podaj_podatek()<<endl;}
```

wywołanie metody wirtualnej Podaj_cene dla obiektów typu TProdukt2, którego referencja może być przekazana do funkcji operatorowej

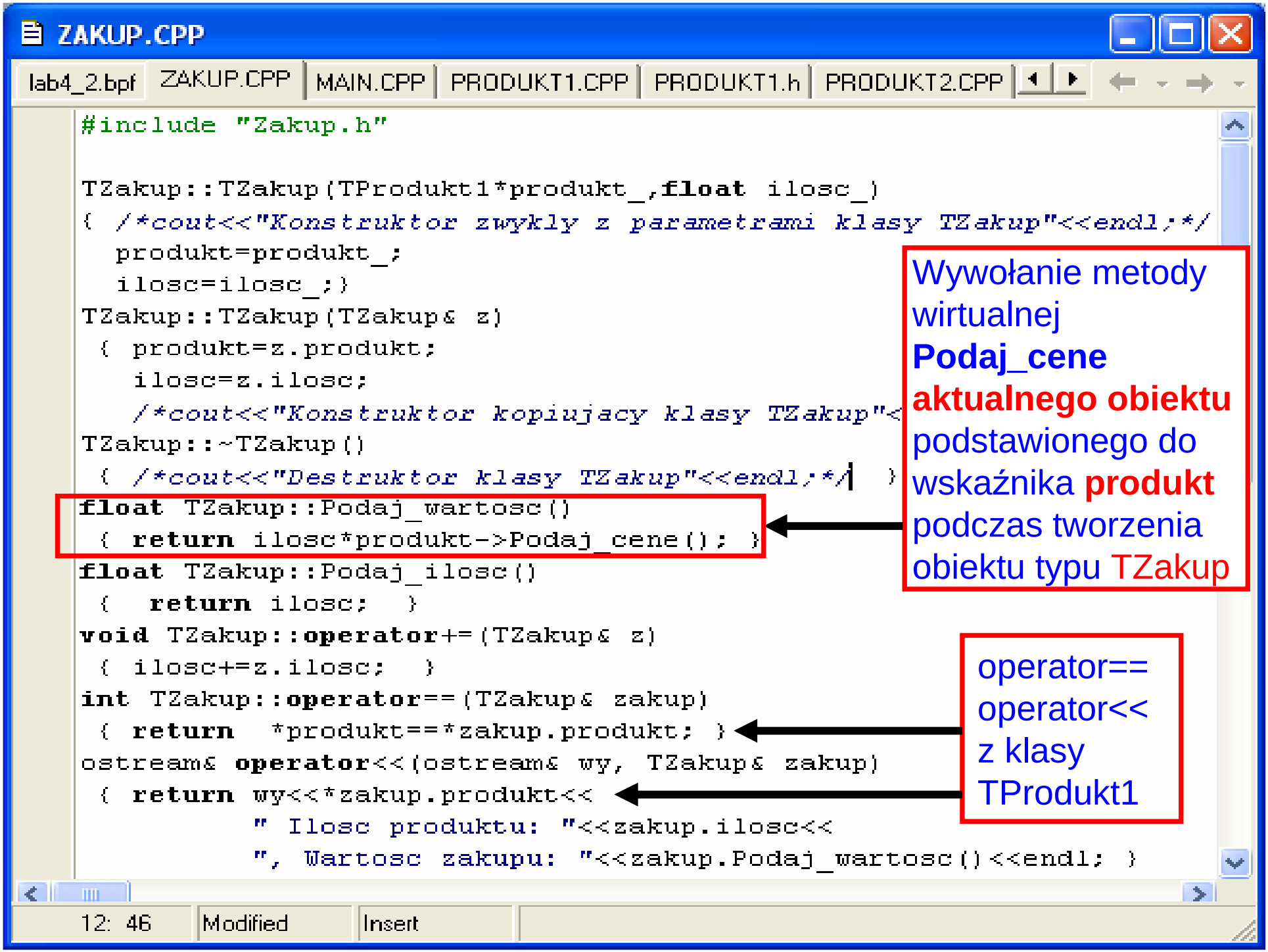
22: 50 | Modified | Insert

C:\Settings\dydaktyka\Programowanie_obiektowe\w4_2\ZAKUP.h

ZAKUP.CPP | ZAKUP.h | MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PI

```
#ifndef _Zakup
#define _Zakup
#include "produkt2.h"
class TZakup
 {
        protected:
        TProdukt1* produkt;
        float ilosc;
public:
        TZakup(TProdukt1*produkt_=NULL,float ilosc_=0);
        TZakup(TZakup&);
        ~TZakup();
        float Podaj_wartosc();
        float Podaj_ilosc();
        void operator+=(TZakup&);
        int operator==(TZakup&);
        friend ostream& operator<<(ostream&, TZakup&);
};
#endif
```

1: 1 | Insert


ZAKUP.CPP
lab4_2.bpf
ZAKUP.CPP
MAIN.CPP
PRODUKT1.CPP
PRODUKT1.h
PRODUKT2.CPP
#include "Zakup.h"
TZakup::TZakup(TProdukt1*produkt_,float ilosc_)
{ /*cout<<"Konstruktor zwykly z parametrami klasy TZakup"<<endl;*/
produkt=produkt_;
ilosc=ilosc_;}
TZakup::TZakup(TZakup& z)
{ produkt=z.produkt;
ilosc=z.ilosc;
/*cout<<"Konstruktor kopiujacy klasy TZakup"<
TZakup::~TZakup()
{ /*cout<<"Destruktor klasy TZakup"<<endl;*/ }
float TZakup::Podaj_wartosc()
{ return ilosc*produkt->Podaj_cene(); }
float TZakup::Podaj_ilosc()
{ return ilosc; }
void TZakup::operator+=(TZakup& z)
{ ilosc+=z.ilosc; }
int TZakup::operator==(TZakup& zakup)
{ return *produkt==*zakup.produkt; }
ostream& operator<<(ostream& wy, TZakup& zakup)
{ return wy<<*zakup.produkt<<
" Ilosc produktu: "<<zakup.ilosc<<
", Wartosc zakupu: "<<zakup.Podaj_wartosc()<<endl; }
Wywołanie metody wirtualnej Podaj_cene aktualnego obiektu podstawionego do wskaźnika produkt podczas tworzenia obiektu typu TZakup
operator== operator<< z klasy TProdukt1
12: 46
Modified
Insert

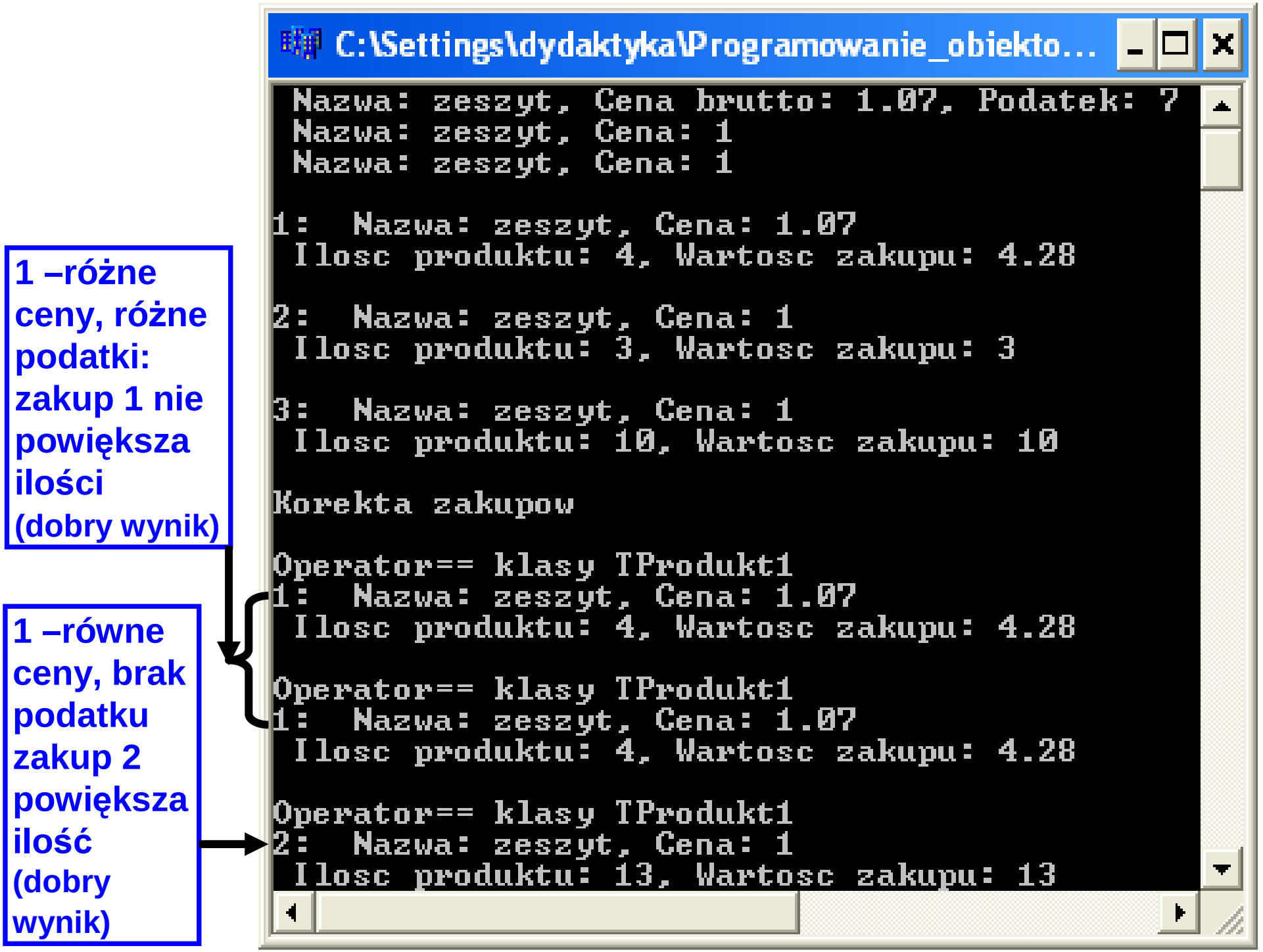
MAIN.CPP

MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PRODUKT2

```
#include "Zakup.h"
void main()
{
 TProdukt2* p1=new TProdukt2("zeszyt",1.0,7);
 TProdukt1* p2=new TProdukt1("zeszyt", 1.0);
 TProdukt1* p3=new TProdukt1("zeszyt", 1.0);
 cout<<*p1<<*p2<<*p3<<endl;
 TZakup* z1=new TZakup(p1,4);
 TZakup* z2=new TZakup(p2,3);
 TZakup* z3=new TZakup(p3,10);
 cout<<"1: "<<*z1<<endl;
 cout<<"2: "<<*z2<<endl;
 cout<<"3: "<<*z3<<endl;
 cout<<"Korekta zakupow\n"<<endl;
  if(*z1==*z2)
    {*z1+=*z2;}
  cout<<"1: "<<*z1<<endl;
 if(*z1==*z3)
    {*z1+=*z3;}
 cout<<"1: "<<*z1<<endl;
 if(*z2==*z3)
    {*z2+=*z3;}
    cout<<"2: "<<*z2<<endl;
 delete p1; delete p2; delete p3;
 delete z1; delete z2; delete z3;
 cin.get();
}
```

//1 porównanie cen aktualnych obiektów w operatorze klasy TProdukt1 za pomocą wirtualnej metody Podaj_cene – ceny brutto lub netto mogą być równe!

21: 37 | Modified | Insert


C:\Settings\dydaktyka\Programowanie_obiekto...
Nazwa: zeszyt, Cena brutto: 1.07, Podatek: 7
Nazwa: zeszyt, Cena: 1
Nazwa: zeszyt, Cena: 1
1: Nazwa: zeszyt, Cena: 1.07
Ilosc produktu: 4, Wartosc zakupu: 4.28
2: Nazwa: zeszyt, Cena: 1
Ilosc produktu: 3, Wartosc zakupu: 3
3: Nazwa: zeszyt, Cena: 1
Ilosc produktu: 10, Wartosc zakupu: 10
Korekta zakupow
Operator== klasy TProdukt1
1: Nazwa: zeszyt, Cena: 1.07
Ilosc produktu: 4, Wartosc zakupu: 4.28
Operator== klasy TProdukt1
1: Nazwa: zeszyt, Cena: 1.07
Ilosc produktu: 4, Wartosc zakupu: 4.28
Operator== klasy TProdukt1
2: Nazwa: zeszyt, Cena: 1
Ilosc produktu: 13, Wartosc zakupu: 13
1 –różne ceny, różne podatki: zakup 1 nie powiększa ilości (dobry wynik)
1 –równe ceny, brak podatku zakup 2 powiększa ilość (dobry wynik)

MAIN.CPP

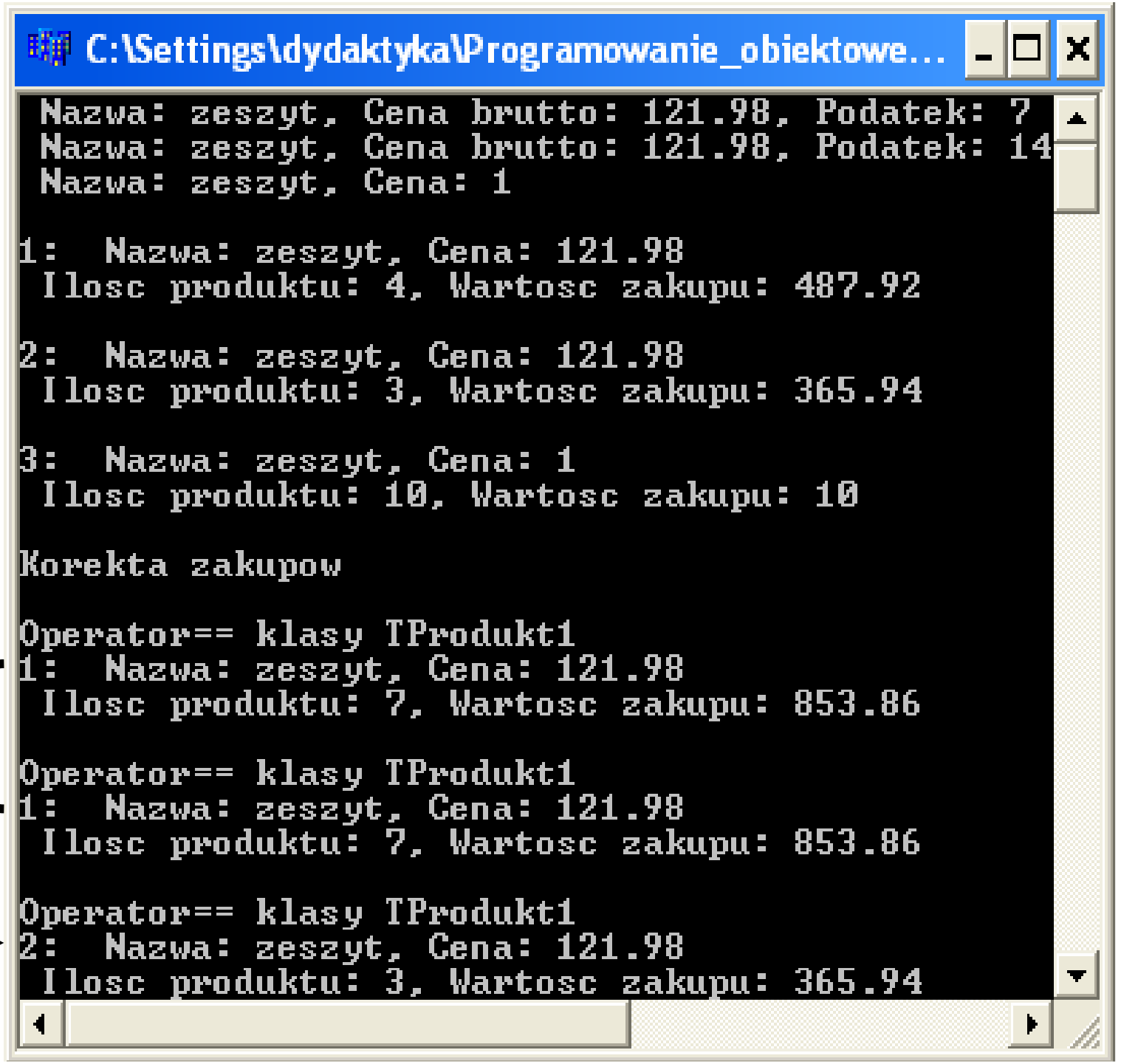
MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PRODUKT2.CP

```
#include "Zakup.h"
void main()
{
 TProdukt2* p1=new TProdukt2("zeszyt",114,7);
 TProdukt2* p2=new TProdukt2("zeszyt", 107,14);
 TProdukt1* p3=new TProdukt1("zeszyt", 1.0);
 cout<<*p1<<*p2<<*p3<<endl;
 TZakup* z1=new TZakup(p1,4);
 TZakup* z2=new TZakup(p2,3);
 TZakup* z3=new TZakup(p3,10);
 cout<<"1: "<<*z1<<endl;
 cout<<"2: "<<*z2<<endl;
 cout<<"3: "<<*z3<<endl;
 cout<<"Korekta zakupow\n"<<endl;
  if(*z1==*z2)
   {*z1+=*z2;}
  cout<<"1: "<<*z1<<endl;
 if(*z1==*z3)
   {*z1+=*z3;}
 cout<<"1: "<<*z1<<endl;
 if(*z2==*z3)
   {*z2+=*z3;}
   cout<<"2: "<<*z2<<endl;
 delete p1; delete p2; delete p3;
 delete z1; delete z2; delete z3;
 cin.get();
}
```

5: 11 | Modified | Insert

1 – równe ceny, różne podatki: zakup 1 powiększa ilości (zły wynik)

1 – różne ceny, różne podatki: zakup 2 nie powiększa ilości (dobry wynik)



Dziedziczenie jednobazowe, poliformizm

1. Dziedziczenie jednobazowe
2. Polimiorfizm – część 1
3. Polimorfizm – część 2

C:\Settings\dydaktyka\Programowanie_obiektowe\w4_3\PRODUKT1.h

lab4_3.bpf | ZAKUP.CPP | MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PROD

```
#ifndef _Produkt1
#define _Produkt1
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
class TProdukt1
{protected:
        string nazwa;
        float cena;
public:
        TProdukt1(string nazwa_="bez nazwy",float cena_=0);
        TProdukt1(TProdukt1&);
        ~TProdukt1();
        virtual float Podaj_cene();
        virtual float Podaj_podatek();
        int operator==(TProdukt1&);
        friend ostream& operator<<(ostream&, TProdukt1&);
};
#endif
```

Tylko jeden operator==, w którym wywołane są metody wirtualne obiektów typu TProdukt1 lub TProdukt2

15: 23 | Modified | Insert

PRODUKT1.CPP

lab4_3.bpf | ZAKUP.CPP | MAIN.CPP | PRODUKT1.CPP | PRODUKT2.CPP

```
#include "produkt1.h"

TProdukt1::TProdukt1(string nazwa_,float cena_)
 { nazwa=nazwa_;
   cena=cena_;
   /*cout<<"Konstruktor zwykly z parametrami klasy TProdukt1"<<endl;*/}
TProdukt1::TProdukt1(TProdukt1& p)
 { nazwa=p.nazwa;
   cena=p.cena;
   /*cout<<"Konstruktor kopiujacy klasy TProdukt1"<<endl;*/}
TProdukt1::~TProdukt1()
 { /*cout<<"Destruktor klasy TProdukt1"<<endl;*/ }
float TProdukt1::Podaj_cene()
 { return cena; }
float TProdukt1::Podaj_podatek()
 { return -1; }
int TProdukt1::operator==(TProdukt1& p)
 {  /*cout<<"Operator== klasy TProdukt1"<<endl;*/
    float a= Podaj_cene(), b= p.Podaj_cene();
    return nazwa==p.nazwa && a==b &&
           Podaj_podatek()==p.Podaj_podatek();  }
ostream& operator<<(ostream& wy, TProdukt1& p)
 { return wy<<" Nazwa: "<<p.nazwa<<
             ", Cena: "<<p.Podaj_cene()<<endl; }
```

Tylko jeden operator==, w którym wywołane są metody wirtualne obiektów typu TProdukt1 lub TProdukt2: : Podaj_cene() oraz Podaj_podatek()

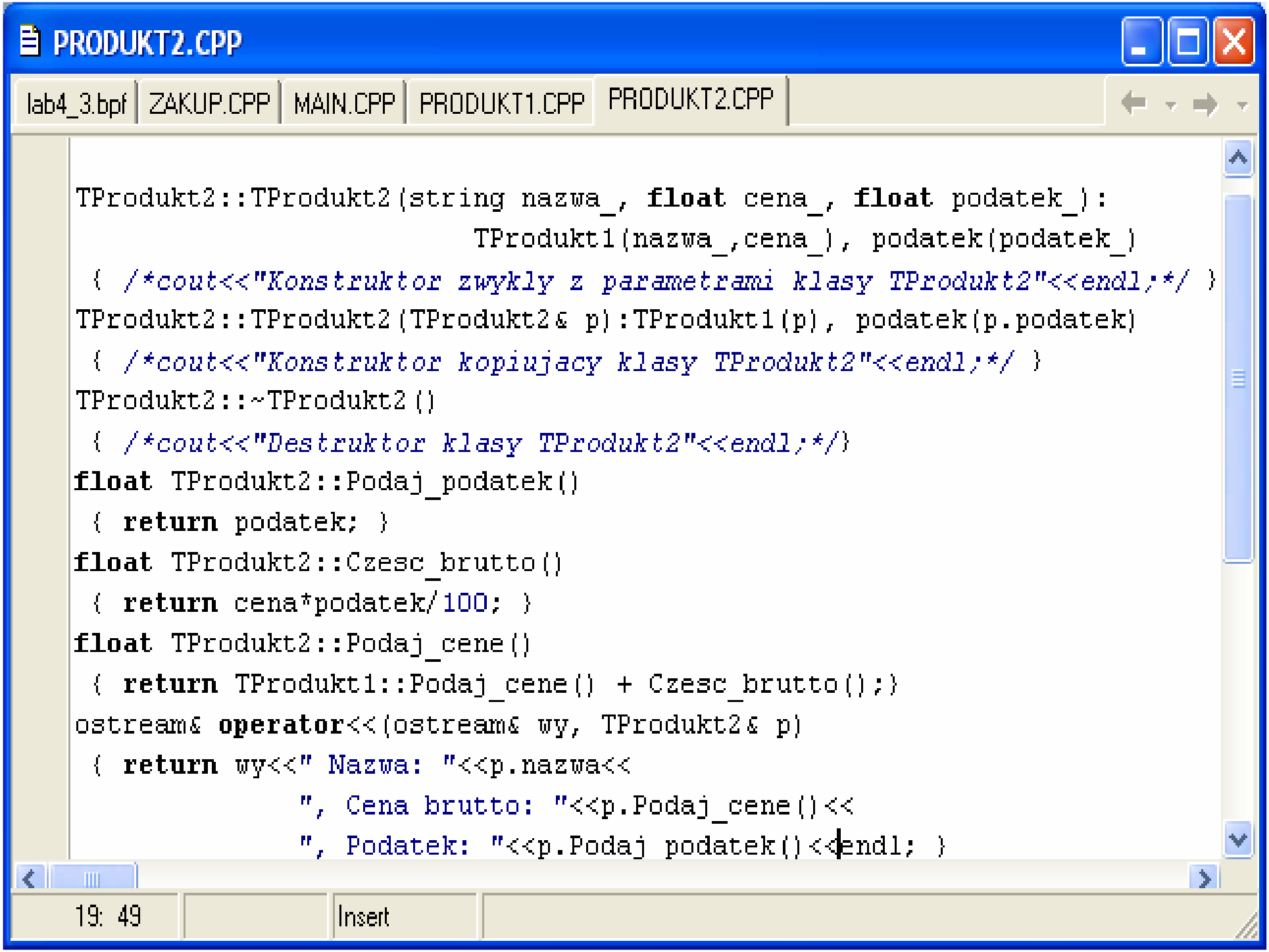
24: 22 | Modified | Insert

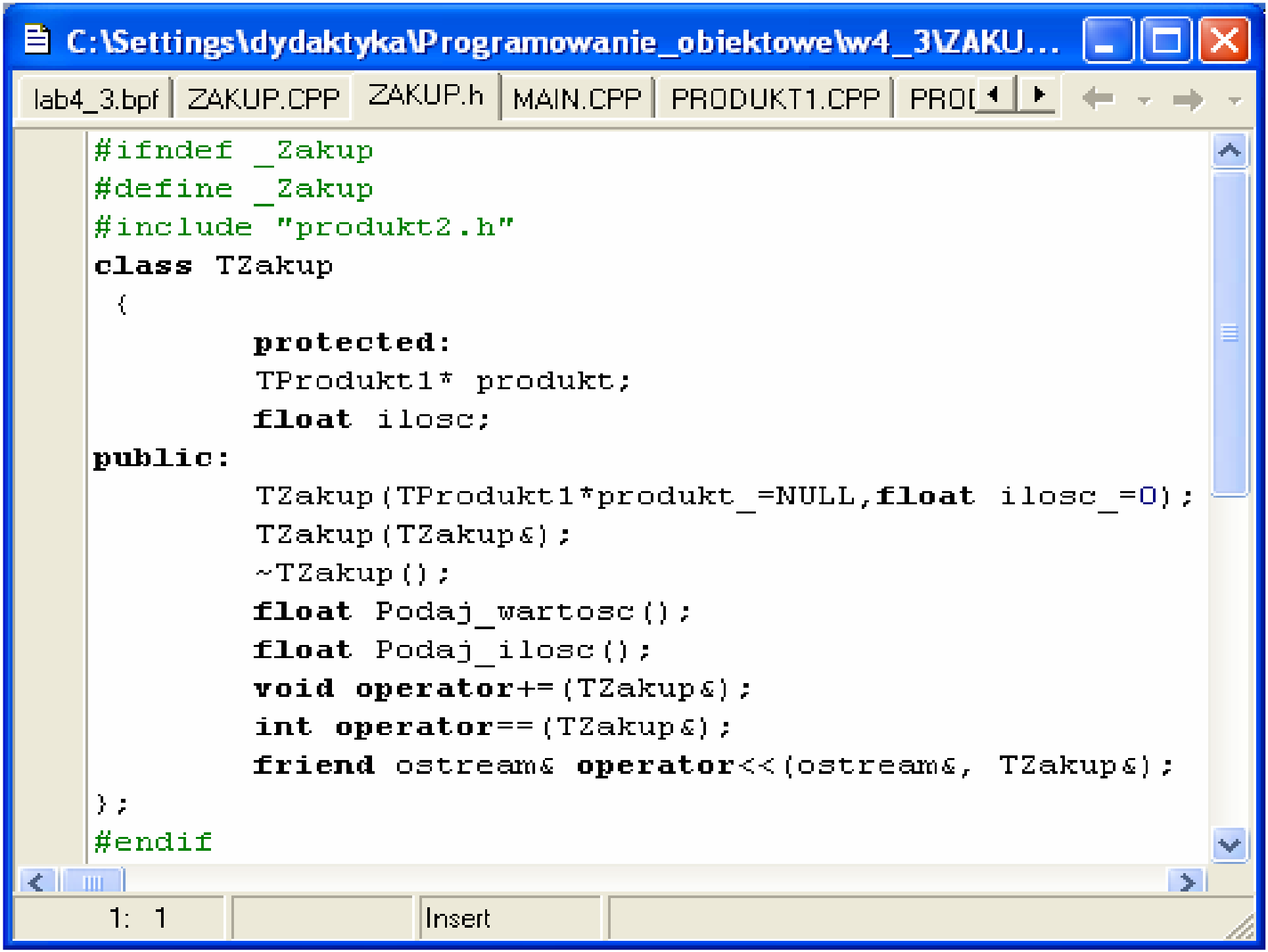
C:\Settings\dydaktyka\Programowanie_obiektowe\w4_3\PRODUKT2.h

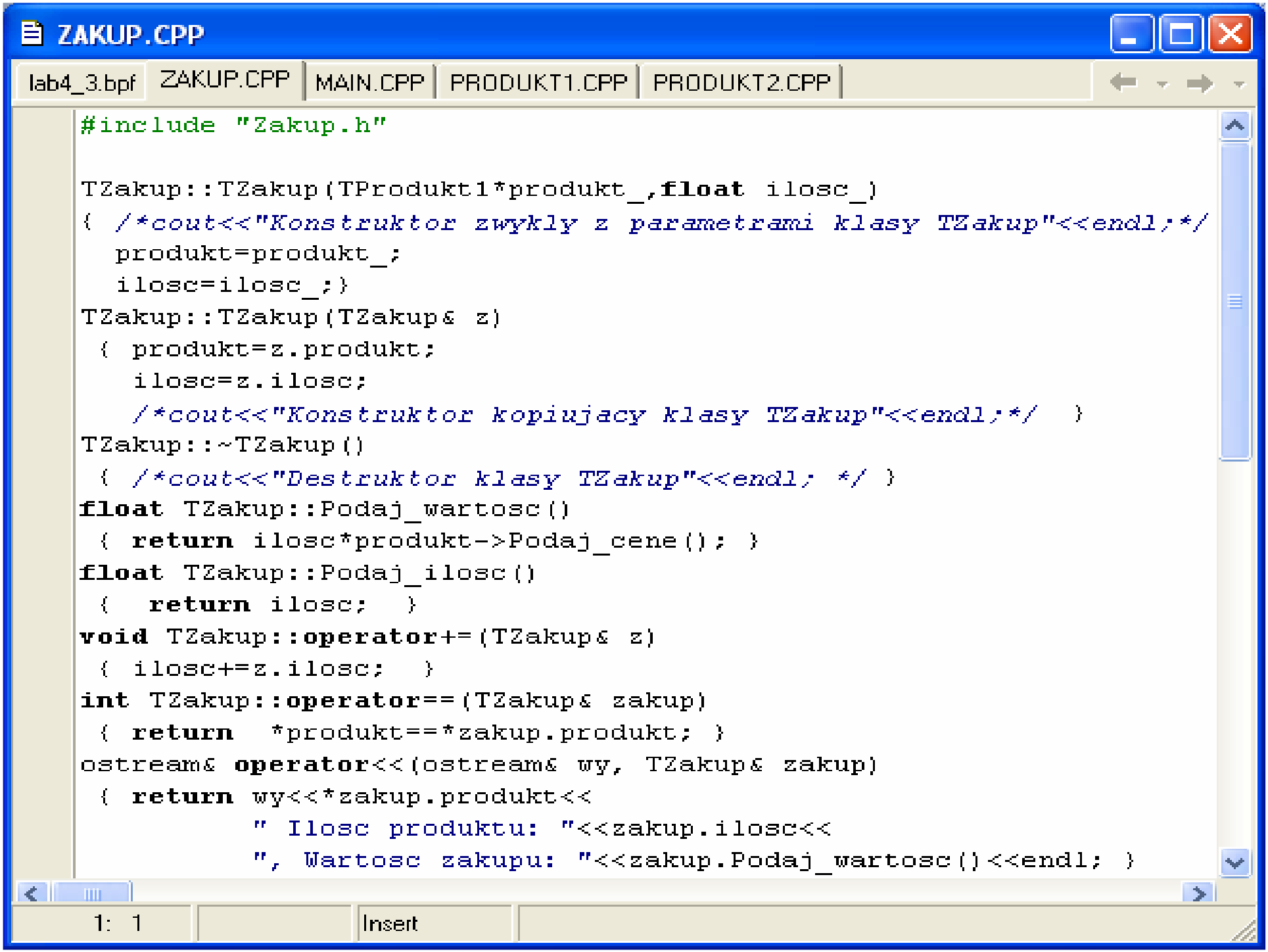
lab4_3.bpf | ZAKUP.CPP | MAIN.CPP | PRODUKT1.CPP | PRODUKT1.h | PRODUKT2.CPP | PRODUKT2.h

```
#ifndef _Produkt2
#define _Produkt2
#include "produkt1.h"

class TProdukt2: public TProdukt1
{  protected:
        float podatek;
public:
        TProdukt2(string nazwa_="bez nazwy",float cena_=0, float podatek=0);
        TProdukt2(TProdukt2&);
        ~TProdukt2();
        float Czesc_brutto();
        float Podaj_cene();
        float Podaj_podatek();
        friend ostream& operator<<(ostream&, TProdukt2&);
};
#endif
```

14: 31 | Modified | Insert


PRODUKT2.CPP
lab4_3.bpf
ZAKUP.CPP
MAIN.CPP
PRODUKT1.CPP
PRODUKT2.CPP
TProdukt2::TProdukt2(string nazwa_, float cena_, float podatek_):
TProdukt1(nazwa_,cena_), podatek(podatek_)
{ /*cout<<"Konstruktor zwykly z parametrami klasy TProdukt2"<<endl;*/ }
TProdukt2::TProdukt2(TProdukt2& p):TProdukt1(p), podatek(p.podatek)
{ /*cout<<"Konstruktor kopiujacy klasy TProdukt2"<<endl;*/ }
TProdukt2::~TProdukt2()
{ /*cout<<"Destruktor klasy TProdukt2"<<endl;*/}
float TProdukt2::Podaj_podatek()
{ return podatek; }
float TProdukt2::Czesc_brutto()
{ return cena*podatek/100; }
float TProdukt2::Podaj_cene()
{ return TProdukt1::Podaj_cene() + Czesc_brutto();}
ostream& operator<<(ostream& wy, TProdukt2& p)
{ return wy<<" Nazwa: "<<p.nazwa<<
", Cena brutto: "<<p.Podaj_cene()<<
", Podatek: "<<p.Podaj podatek()<<endl; }
19: 49
Insert


C:\Settings\dydaktyka\Programowanie_obiektowe\w4_3\ZAKU...
lab4_3.bpf
ZAKUP.CPP
ZAKUP.h
MAIN.CPP
PRODUKT1.CPP
#ifndef _Zakup
#define _Zakup
#include "produkt2.h"
class TZakup
{
protected:
TProdukt1* produkt;
float ilosc;
public:
TZakup(TProdukt1*produkt_=NULL,float ilosc_=0);
TZakup(TZakup&);
~TZakup();
float Podaj_wartosc();
float Podaj_ilosc();
void operator+=(TZakup&);
int operator==(TZakup&);
friend ostream& operator<<(ostream&, TZakup&);
};
#endif
1: 1
Insert


ZAKUP.CPP
lab4_3.bpf
ZAKUP.CPP
MAIN.CPP
PRODUKT1.CPP
PRODUKT2.CPP
#include "Zakup.h"
TZakup::TZakup(TProdukt1*produkt_,float ilosc_)
{ /*cout<<"Konstruktor zwykly z parametrami klasy TZakup"<<endl;*/
produkt=produkt_;
ilosc=ilosc_;}
TZakup::TZakup(TZakup& z)
{ produkt=z.produkt;
ilosc=z.ilosc;
/*cout<<"Konstruktor kopiujacy klasy TZakup"<<endl;*/ }
TZakup::~TZakup()
{ /*cout<<"Destruktor klasy TZakup"<<endl; */ }
float TZakup::Podaj_wartosc()
{ return ilosc*produkt->Podaj_cene(); }
float TZakup::Podaj_ilosc()
{ return ilosc; }
void TZakup::operator+=(TZakup& z)
{ ilosc+=z.ilosc; }
int TZakup::operator==(TZakup& zakup)
{ return *produkt==*zakup.produkt; }
ostream& operator<<(ostream& wy, TZakup& zakup)
{ return wy<<*zakup.produkt<<
" Ilosc produktu: "<<zakup.ilosc<<
", Wartosc zakupu: "<<zakup.Podaj_wartosc()<<endl; }
1: 1
Insert

MAIN.CPP

lab4_3.bpf | ZAKUP.h | MAIN.CPP | PRODUKT1.CPP | PRODU

```
#include "Zakup.h"
void main()
{
 TProdukt2* p1=new TProdukt2("zeszyt",114,7);
 TProdukt2* p2=new TProdukt2("zeszyt", 107,14);
 TProdukt1* p3=new TProdukt1("zeszyt", 1.0);
 cout<<*p1<<*p2<<*p3<<endl;
 TZakup* z1=new TZakup(p1,4);
 TZakup* z2=new TZakup(p2,3);
 TZakup* z3=new TZakup(p3,10);
 cout<<"1: "<<*z1<<endl;
 cout<<"2: "<<*z2<<endl;
 cout<<"3: "<<*z3<<endl;
 cout<<"Korekta zakupow\n"<<endl;
  if(*z1==*z2)
   {*z1+=*z2;}
  cout<<"1: "<<*z1<<endl;
 if(*z1==*z3)
   {*z1+=*z3;}
 cout<<"1: "<<*z1<<endl;
 if(*z2==*z3)
   {*z2+=*z3;}
   cout<<"2: "<<*z2<<endl;
 delete p1; delete p2; delete p3;
 delete z1; delete z2; delete z3;
 cin.get();
}
```

23: 20 | Insert

1 – równe ceny, różne podatki: zakup 1 nie powiększa ilości (dobry wynik)

1 – różne ceny, różne podatki zakup 2 nie powiększa ilości (dobry wynik)