



Programowanie obiektowe

Paweł Mikołajczak, 2019

4. Funkcje zaprzyjaźnione

Web site: informatyka.umcs.lublin.pl

4. *Funkcje zaprzyjaźnione*

- *Funkcje i klasy zaprzyjaźnione*
- *Funkcja niezależna zaprzyjaźniona z klasą*
- *Funkcja składowa zaprzyjaźniona z inną klasą*
- *Klasy zaprzyjaźnione*

Funkcje i klasy zaprzyjaźnione.

W języku C++ to **klasa** jest chroniona przed nieuprawnionym dostępem a nie **obiekt**. W takiej sytuacji funkcja składowa konkretnej klasy może używać wszystkich składowych prywatnych dowolnego obiektu tej samej klasy.

W językach czysto obiektowych mamy do czynienia z mechanizmem odbioru **komunikatu** przez **obiekt**. Wywołując funkcję obsługującą dany obiekt, wysyłamy odpowiedni komunikat, obiekt pełni rolę odbiorcy komunikatu. Treścią komunikatu są wartości zmiennych, adresy, itp. przekazywane jako argumenty aktualne wywoływanej funkcji. Jeżeli mamy klasę o nazwie **punkt**, oraz funkcję składową klasy **punkt**, której prototyp ma postać **int ustaw(int, int)**, to deklaracja obiektu **a** może mieć postać:

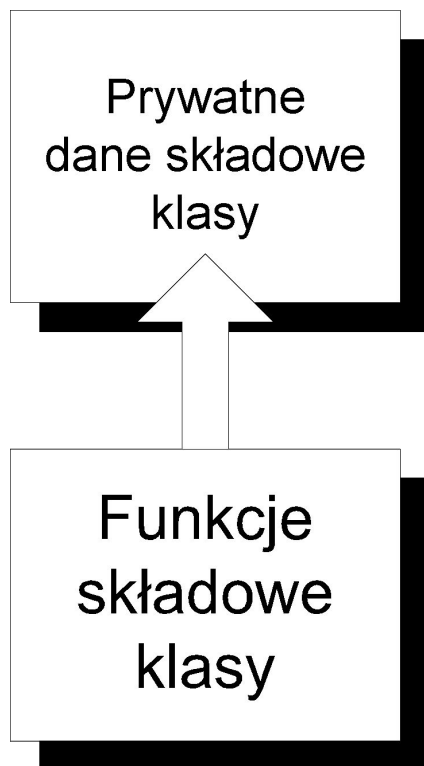
punkt a;

Dostęp do publicznej składowej funkcji klasy uzyskujemy przy użyciu operatora dostępu, np.

a.ustaw(5, 5);

Funkcje i klasy zaprzyjaźnione.

W ten sposób wywołujemy funkcję składową **ustaw()** klasy, do której należy obiekt **a**, to znaczy klasy **punkt**. Z formalnego punktu widzenia, to wywołanie możemy traktować zaadresowany do obiektu **a** komunikat o nazwie **ustaw()**, którego treścią są wartości dwóch zmiennych (5 i 5).



Do danych prywatnych klasy dostęp mają **jedynie** funkcje składowe (metody) klasy.

Funkcje i klasy zaprzyjaźnione

Z drugiej strony wiemy, że do składowych prywatnych utworzonego obiektu **nie mają** dostępu funkcje **niezależne** ani **funkcje innej klasy**. W wielu jednak przypadkach istnieje potrzeba dostępu do chronionych danych obiektu.

Język C++ znany ze swojej elastyczności ma specjalny mechanizm pozwalający na dostęp, w sposób kontrolowany, do formalnie niedostępnych danych.

Możemy, wykorzystując odpowiednie konstrukcje języka C++ spowodować, że „zwykła” funkcja będzie mogła wykonywać operacje na obiekcie w taki sposób, jak czyni to funkcja składowa klasy. W tym przypadku, obiekt **a** może być traktowany jako jeden z argumentów formalnych funkcji. Wtedy wywołanie może mieć postać:

ustaw(a, 5, 5);

W tym wywołaniu, obiekt **a** jest traktowany tak samo jak pozostałe argumenty funkcji.

Tym specjalnym mechanizmem w języku C++ jest mechanizm **deklaracji zaprzyjaźnienia**. Deklaracja zaprzyjaźnienia (*ang. **friend***) pozwala zadeklarować w danej klasie funkcje, które będą miały dostęp do składowych prywatnych, ale nie są funkcjami składowymi klasy.

Funkcje i klasy zaprzyjaźnione

Wyróżniamy następujące sytuacje:

- Funkcja **niezależna** zaprzyjaźniona z **klasą X**
- Funkcja składowa **klasy Y** zaprzyjaźniona **z klasą X**
- Wszystkie funkcje klasy Y są zaprzyjaźnione z klasą X
(***klasa zaprzyjaźniona***)

Funkcje zaprzyjaźnione mają dostęp do wszystkich składowych klasy zadeklarowanych jako **private** lub **protected**. Z formalnego punktu widzenia, funkcje zaprzyjaźnione łamią zasadę hermetyzacji (ukrywania) danych, ale czasami ich użycie może być korzystne.

Funkcja niezależna zaprzyjaźniona z klasą

Funkcja zaprzyjaźniona z klasą jest zdefiniowana **na zewnątrz** jej zasięgu i ma dostęp do składowych **prywatnych** klasy.

Aby zadeklarować funkcję jako zaprzyjaźnioną (mającą dostęp do danych prywatnych) z jakąś klasą, prototyp funkcji w jej definicji należy poprzedzić słowem kluczowym **friend**.

Mimo, że prototyp funkcji umieszczony jest w definicji klasy, **nie** jest ona jej funkcją składową.

Etykiety definiujące sposób dostępu do składowych, takie jak **private**, **public** i **protected** nie mają nic wspólnego z definicją funkcji zaprzyjaźnionych.

Dobrym zwyczajem programistycznym jest umieszczanie prototypów wszystkich funkcji zaprzyjaźnionych z daną klasą na jej początku, ale nie jest to konieczne.

Funkcja niezależna zaprzyjaźniona z klasą

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

class punkt
{
private:
    int x;
    int y;
public:
    punkt ( int xx = 0, int yy = 0 )
        {x = xx ;    y = yy ; }
    friend void pokaz (punkt );
};
```

```
void pokaz (punkt p1)
{
    cout << "Pozycja X = " << p1.x << endl;
    cout << "Pozycja Y = " << p1.y << endl;
}

int main()
{
    punkt a1( 2, 4 );
    pokaz ( a1 );
    getch();
    return 0;
}
```

Wydruk z programu ma postać:
Pozycja X = 2
Pozycja Y = 4

Funkcja niezależna zaprzyjaźniona z klasą

W pokazanym przykładzie mamy prostą klasę **punkt**:

```
class punkt
{
    int x, y;
public:
    punkt ( int xx = 0, int yy = 0 )
        {x = xx ;    y = yy ; }
    friend void pokaz (punkt );
};
```

Dwie dane składowe **x** i **y** są prywatne, dostęp do nich możliwy jest jedynie poprzez funkcje składowe klasy **punkt**. Klasa posiada konstruktor. Jeżeli chcemy, aby funkcja zewnętrzna miała dostęp do danych składowych klasy **punkt**, musimy jawnie zadeklarować taką funkcję jako **zaprzyjaźnioną**.

W naszym przykładzie tworzymy funkcję zaprzyjaźnioną o nazwie **pokaz()**:

```
friend void pokaz (punkt );
```

Funkcja niezależna zaprzyjaźniona z klasą

Dzięki specyfikacji **friend** funkcja **pokaz()** staje się funkcją zaprzyjaźnioną, ma dostęp do prywatnych danych klasy **punkt**. Funkcja **pokaz()** wyświetla aktualne wartości danych prywatnych **x** i **y**. Funkcja **pokaz()** jest samodzielną funkcją w stylu języka C – nie jest ona funkcją składową klasy **punkt**:

```
void pokaz (punkt p1)
{ cout << "Pozycja X = " << p1.x << endl;
  cout << "Pozycja Y = " << p1.y << endl;
}
```

Funkcja niezależna zaprzyjaźniona z klasą

W programie testującym:

```
int main()  
{ punkt a1( 2, 4 );  
  pokaz ( a1 );  
  return 0;  
}
```

tworzymy obiekt **a1** klasy **punkt** i inicjujemy go wartościami 2 i 4. Aby wyświetlić na ekranie monitora wartości **x** i **y** musimy wywołać funkcję **pokaz()**. Ponieważ jest to funkcja zaprzyjaźniona, możemy wywoływać tę funkcję tak, jak zwykłą funkcję w języku C:

```
pokaz ( a1 );
```

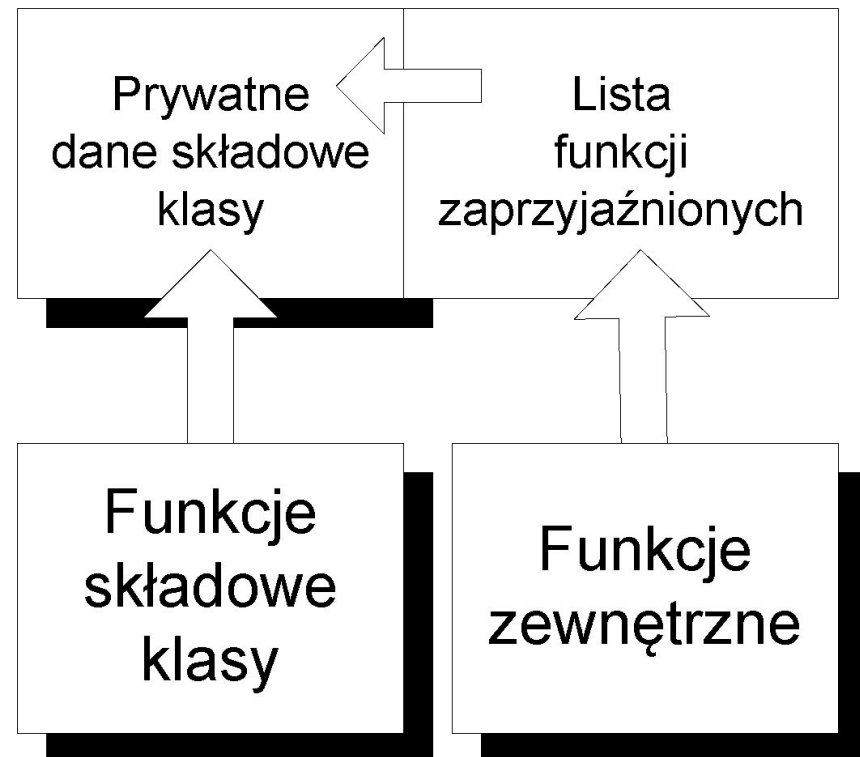
W naszym przykładzie funkcja **pokaz()** pobiera jako argument **a1**. Pamiętajmy, że gdyby funkcja **pokaz()** była funkcją składową klasy **punkt**, to wywołanie jej miałoby postać:

```
a1.pokaz();
```

Funkcje i klasy zaprzyjaźnione.

W naszym przykładzie, funkcja **pokaz()** nie jest składową klasy **punkt**. Ze względu na deklarację zaprzyjaźnienia **ma ona pełny dostęp do prywatnych składowych** tej klasy. Funkcja zaprzyjaźniona **pokaz()** jest wywoływana bez użycia operatora dostępu (operator kropki). Nie jest funkcją składową, nie może być poprzedzana nazwą obiektu.

Funkcja zaprzyjaźniona z klasą definiowana jest na zewnątrz jej zasięgu, a mimo tego ma dostęp do jej składowych prywatnych



funkcje zaprzyjaźnione

W kolejnym prostym przykładzie w klasie **Liczby**

```
class Liczby {  
    int a,b,c;  
public:  
    void set(int aa, int bb, int cc){  
        a =aa; b = bb; c = cc; };  
    friend int suma(Liczby x);  
    friend int iloczyn(Liczby x);  
};
```

tworzymy dwie funkcje zaprzyjaźnione. Inicjalizacja danych wykonana jest za pomocą funkcji dostępowej **set()**.

funkcje zaprzyjaźnione

```
#include <iostream>
using namespace std;
class Liczby{
    int a,b,c;
public:
    void set(int aa, int bb, int cc){
        a =aa; b = bb; c = cc; };
    friend int suma(Liczby x);
    friend int iloczyn(Liczby x);
};
int suma(Liczby x){
    return x.a + x.b + x.c;
};
int iloczyn(Liczby x){
    return x.a * x.b * x.c;
};
int main()
{ Liczby n;
  n.set(2,4,6);
  cout << "suma = " << suma(n)<<endl;
  cout << "iloczyn = " << iloczyn(n)<<endl;
  return 0;
}
```

Wynik:

suma = 12
iloczyn = 48

funkcje zaprzyjaźnione


W kolejnym przykładzie program po wpisaniu aktualnej daty porównuje ją z naszą datą urodzenia (dzień i miesiąc) aby stwierdzić czy mamy urodziny. W klasie

```
class urodziny { //sprawdzamy czy dzisiaj mamy urodziny
    int dzien, miesiac;
    void sprawdz_date();
public:
    friend bool rowny(urodziny d1, urodziny m1);
    urodziny (int dd, int mm);
};
```

Mamy funkcję zaprzyjaźnioną **rowny()**, która jest typu **bool**. Inicjalizacji obiektu dokonamy z pomocą konstruktora dwuparametrowego (dzień i miesiąc). Metod **sprawdz_date()** sprawdza czy wprowadzane dane nie wykraczają poza zakres (np. czy miesiąc ma numer >12).

funkcje zaprzyjaźnione

```
#include <iostream>
using namespace std;
class urodziny{ //sprawdzamy czy dzisiaj mamy urodziny
    int dzien, miesiac;
    void sprawdz_date();
public:
    friend bool rowny(urodziny d1, urodziny m1);
    urodziny (int dd, int mm);
};
urodziny :: urodziny(int dd, int mm):dzien(dd),miesiac(mm){
    void sprawdz_date();
};
void urodziny :: sprawdz_date(){
    if ((miesiac<1) || (miesiac>12) || (dzien<1) || (dzien >31) )
        cout << "zla data" <<endl;
    exit(1);
};
bool rowny(urodziny d1, urodziny m1){
    return (d1.dzien == m1.dzien
        && d1.miesiac == m1.miesiac );
};
```



```
int main()
{ //dzisiaj - dzisiejsza data, data_urodzenia - nasza data urodzenia
    urodziny dzisiaj(8,2), data_urodzenia(5,1);
    if (rowny(dzisiaj,data_urodzenia)) cout <<"mamy urodziny"<<endl;
    else
        cout << "nie mamy urodzin"<<endl;
    return 0;
}

//-----
```

Wynik:

nie mamy urodzin

funkcje zaprzyjaźnione

Kolejny przykład jest bardziej skomplikowany, mamy dwie klasy, funkcja zaprzyjaźniona otrzymuje parametry wskaźnikowe. W programie po wprowadzeniu liczby oraz zadanego przedziału, sprawdzamy czy nasza liczba jest w zadanym przedziale. Zapis *class przedzial:* jest tzw. deklaracją zapowiadającą, klasa *liczba* potrzebuje wskaźnika do obiektu klasy *przedzial*.

```
class przedzial; //deklaracja zapowiadajaca
```

```
class liczba {
    int x;
    friend void test(liczba *, przedzial *);
public:
    liczba( int x = 0) : x(x) { }
};

class przedzial {
    int pocz, kon;
    friend void test(liczba *, przedzial *);
public:
    przedzial(int po = 0, int ko = 0 ) : pocz(po),kon(ko){ }
};
```

funkcje zaprzyjaźnione

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

```
class przedzial; //deklaracja zapowiadajaca
```

```
class liczba {
    int x;
    friend void test(liczba *, przedzial *);
public:
    liczba( int x = 0 ) : x(x) { }
};

class przedzial {
    int pocz, kon;
    friend void test(liczba *, przedzial *);
public:
    przedzial(int po = 0, int ko = 0 ) : pocz(po),kon(ko){ }
};

void test(liczba *li, przedzial *pr) {
    if ((li->x >= pr->pocz) && (li->x <= pr->kon))
        cout << "liczba w zakresie" << endl;
    else
        cout << "liczba poza zakresem" << endl;
};
```



```
int main()
{ liczba p(7);
  przedzial p1(0,10), p2(8,20);
  test(&p,&p1);
  return 0;
}
//-----
```

Wynik:

liczba w zakresie



funkcje zaprzyjaźnione

W kolejnym przykładzie wykorzystujemy funkcję zaprzyjaźnioną *ustaw* do obsługi punktu. Mając dane współrzędne punktu, na podstawie wprowadzonego przesunięcia obliczymy nowe współrzędne punktu. Funkcja *ustaw* otrzymuje referencję do obiektu *klasy punkt* oraz dwie dane typu *int* (przesunięcie x oraz przesunięcie y)

```
class punkt {  
    int x,y;  
    public:  
    punkt(int xx, int yy) {x = xx; y = yy;}  
    friend void ustaw(punkt &, int, int);  
    int get_x() {return x;}  
    int get_y() {return y;}  
};
```

Definicja funkcji *ustaw()* jest standardowa:

```
void ustaw(punkt &p, int dx, int dy){  
    p.x += dx; p.y += dy;  
};
```

funkcje zaprzyjaźnione

```
#include <iostream>
```

```
using namespace std;
```

```
class punkt {  
    int x,y;  
public:  
    punkt(int xx, int yy) {x = xx; y = yy;}  
    friend void ustaw(punkt &, int, int);  
    int get_x() {return x;}  
    int get_y() {return y;}  
};
```

```
void ustaw(punkt &p, int dx, int dy){  
    p.x += dx; p.y += dy;  
};
```

```
int main()  
{ punkt p(2,3);  
  ustaw (p,5,5);  
  cout <<"nowa pozycja x = " << p.get_x() << endl;  
  cout <<"nowa pozycja y = " << p.get_y() << endl;  
  return 0;  
}
```

Wynik:

nowa pozycja x = 7
nowa pozycja y = 8

funkcje zaprzyjaźnione

Kolejne dwa programy ilustrują definicje funkcji zaprzyjaźnionych do obsługi obiektów.

W pierwszym przykładzie :

```
float promien ( punkt n)
{ return sqrt(n.x * n.x + n.y * n.y);
}
```

funkcja zaprzyjaźniona `promien()` jako argument otrzymuje obiekt.

W drugim przykładzie funkcja zaprzyjaźniona `promien()` otrzymuje jako argument obiekt oraz pola klasy (`int a` oraz `int b`):

```
float promien ( punkt n, float a, float b)
{ n.x = a;
  n.y = b;
  return sqrt(n.x * n.x + n.y * n.y);
}
```

Funkcja niezależna zaprzyjaźniona z klasą

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

class punkt
{ float x, y;
  public:
    void ustaw(float a, float b);
    friend float promien(punkt n);
};

void punkt :: ustaw(float a, float b)
{ x = a;  y = b;
}
```

```
float promien ( punkt n)
{ return sqrt(n.x * n.x + n.y * n.y);
}

int main()
{ punkt p1;
  p1.ustaw(1.0, 1.0);
  cout << "Odleglosc = " << promien(p1) << endl;
  getch();
  return 0;
}
```

Wydruk z programu ma postać:
Odleglosc = 1.41421

Funkcja niezależna zaprzyjaźniona z klasą

```
#include <iostream>
#include <math>
#include <conio>
using namespace std;
class punkt
{ float x, y;
  friend float promien(punkt n, float a, float b);
};
float promien ( punkt n, float a, float b)
{ n.x = a;
  n.y = b;
  return sqrt(n.x * n.x + n.y * n.y);
}

int main()
{ punkt p1;
  cout << "odleglosc = "
        << promien(p1, 1.0, 1.0) << endl;
  getche();
  return 0;
}
```

Wydruk z programu ma postać:
Odleglosc = 1.41421

Składowe **x** i **y** są danymi prywatnymi klasy (gdy nie podamy specyfikatora **private**, składowe domyślnie są zadeklarowane jako prywatne).

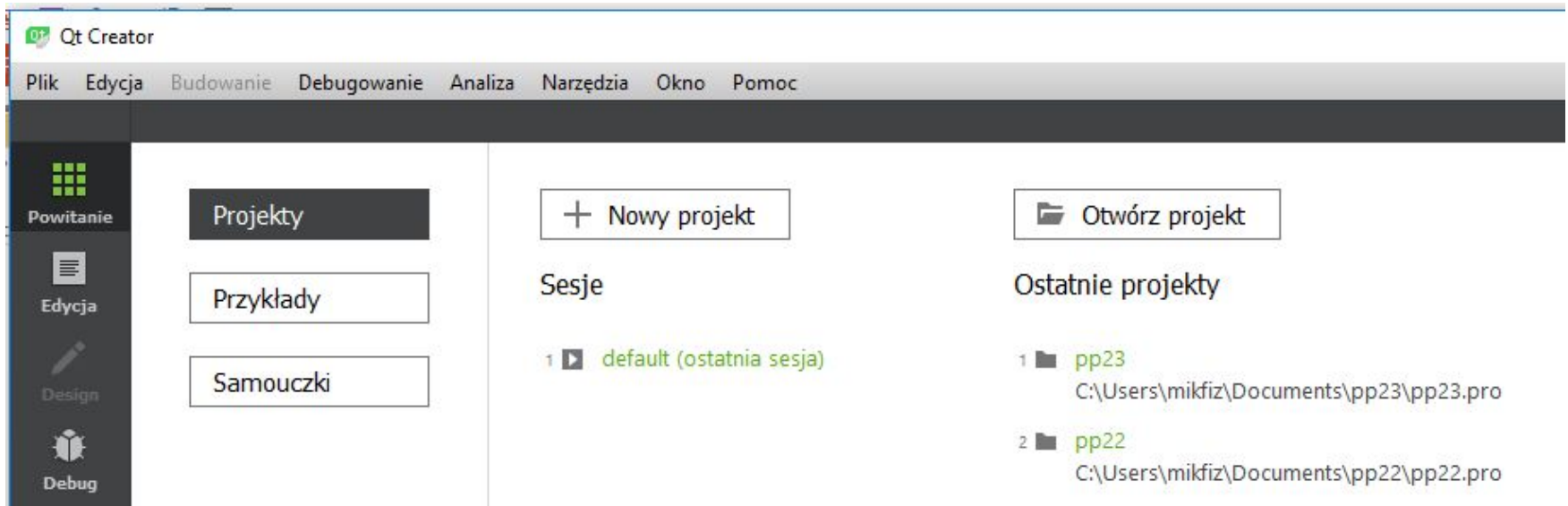
Ponieważ klasa nie posiada żadnych funkcji składowych, nie ma potrzeby stosowania specyfikatora **public** (funkcje zaprzyjaźnione nie podlegają specyfikacji dostępu, możemy je umieszczać w dowolnej sekcji zarówno prywatnej jak i publicznej).

Funkcja zaprzyjaźniona otrzymuje potrzebne argumenty i ustawia zmienne **x** i **y**, a następnie oblicza odległość punktu od początku układu współrzędnych.

Programy wieloplikowe

W aplikacjach mamy do czynienia z wieloma klasami, funkcjami, nie jest rozsądne trzymać te wszystkie kody w jednym pliku. Rekomenduje się tworzenie programów wieloplikowych. Mamy do tego odpowiednie pakiety. Takim przykładem jest pakiet Qt, gdzie stosunkowo prosto tworzymy programy wieloplikowe.

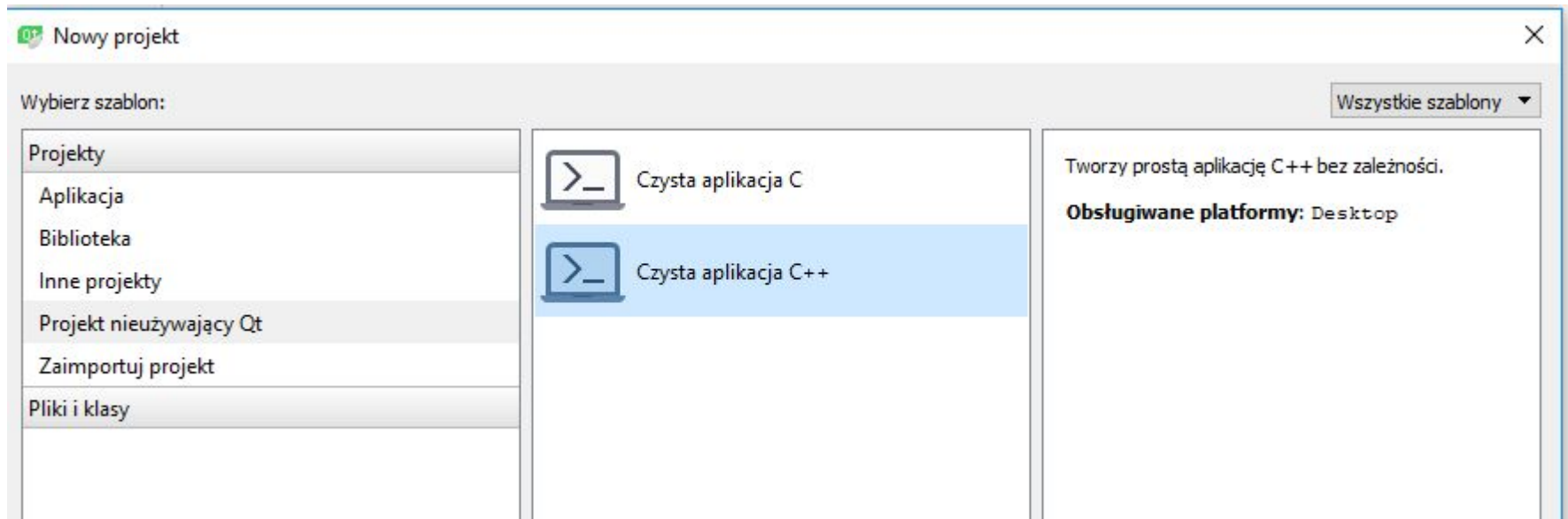
Uruchamiamy pakiet **Qt Creator**, pojawia się aplikacja:



Wybieramy opcję **+ Nowy projekt**, otwiera się kolejne okno.

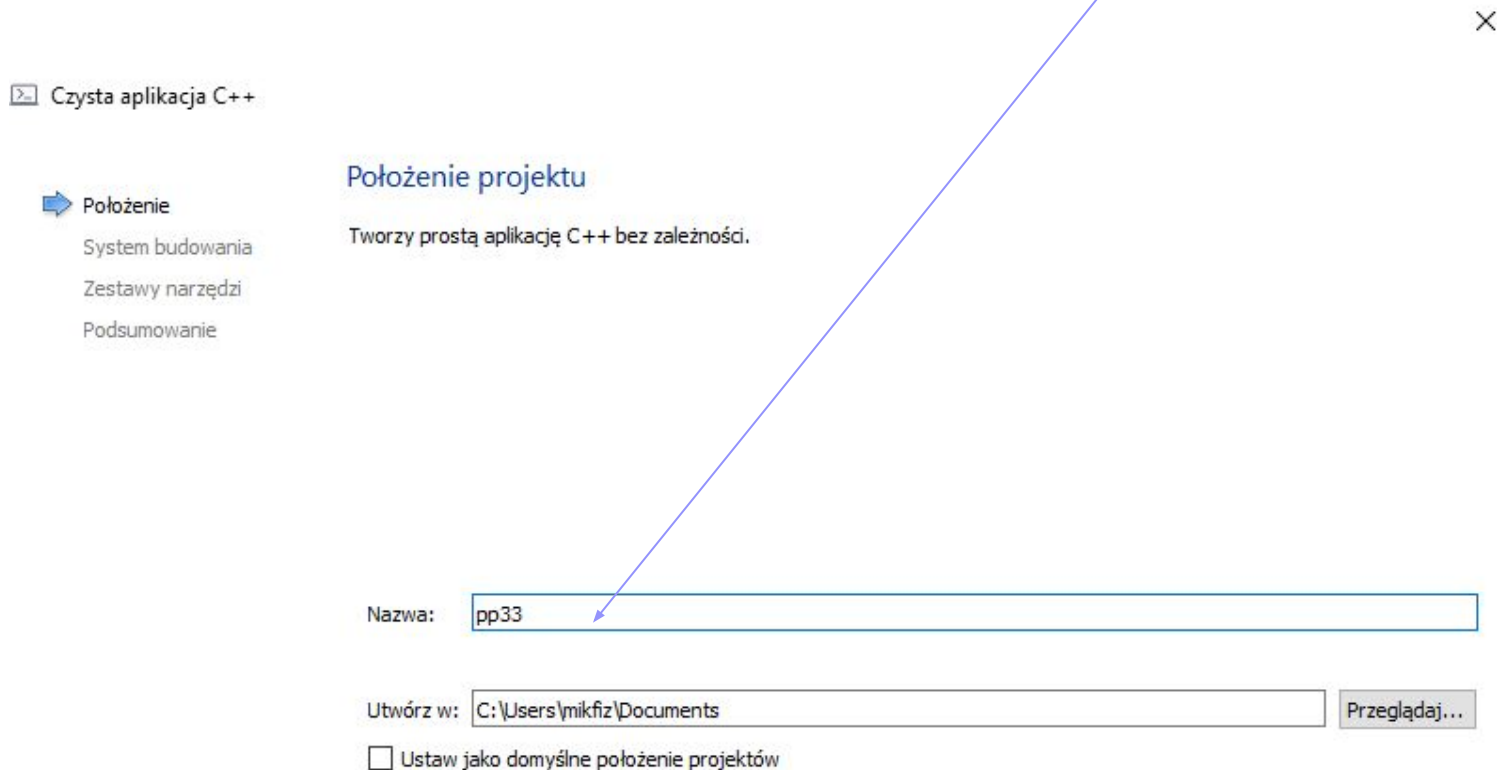
Programy wieloplikowe

W tym oknie wybieramy Projekt nieużywający Qt, oraz Czysta aplikacja C++



Programy wieloplikowe

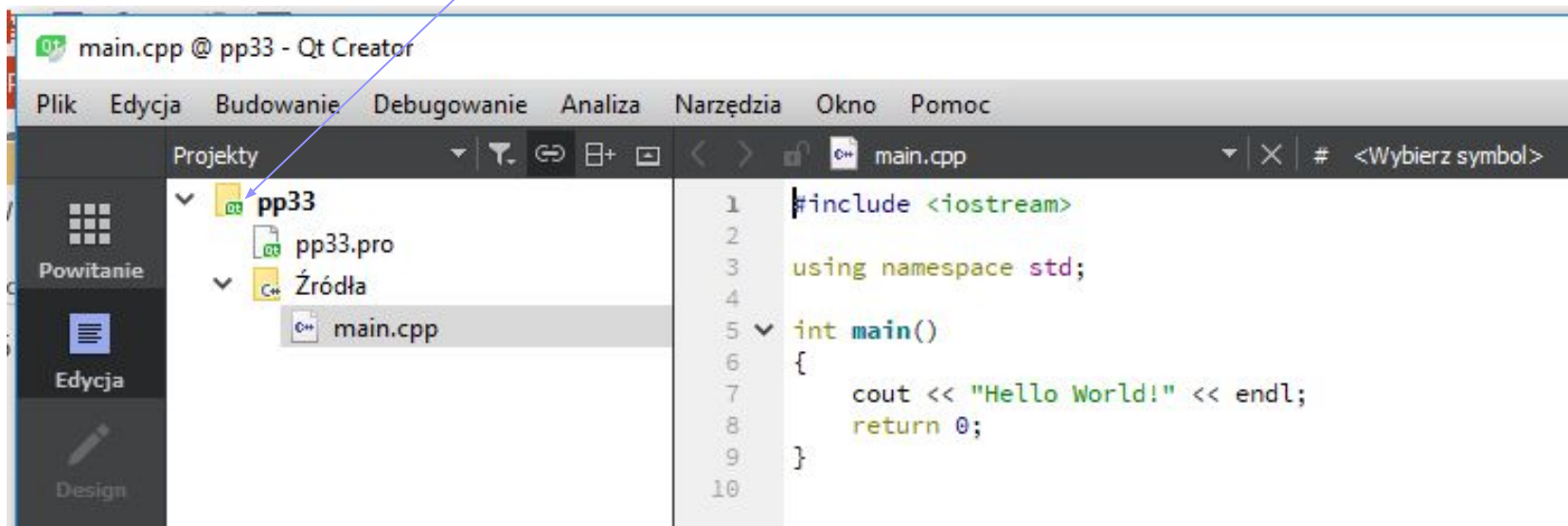
W następnym oknie *nazywamy* nasz projekt (tutaj pp33)



i zatwierdzamy kolejne okna bez zmian. Na końcu pokazuje się edytor tekstu.

Programy wieloplikowe

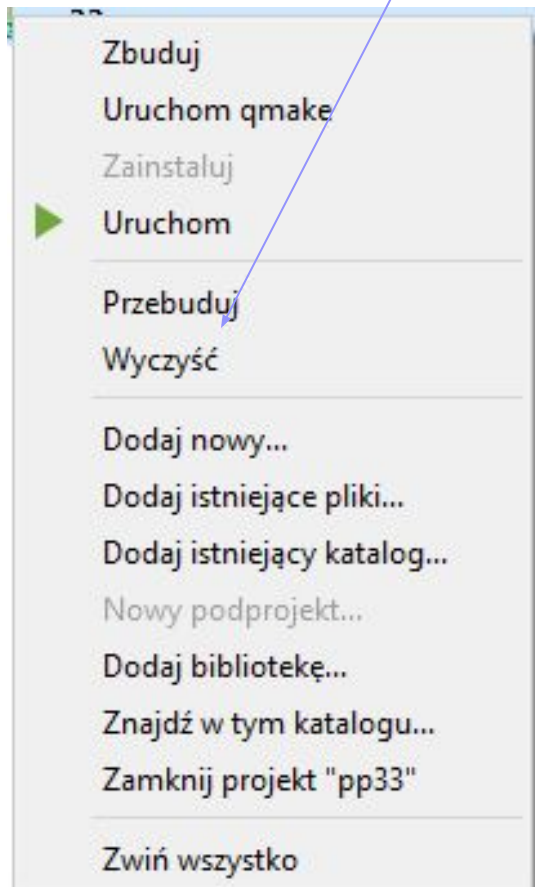
W oknie edytora tekstu, myszą wskazujemy ikonę z nazwą projektu (tutaj pp33) i naciskamy prawy przycisk myszy.



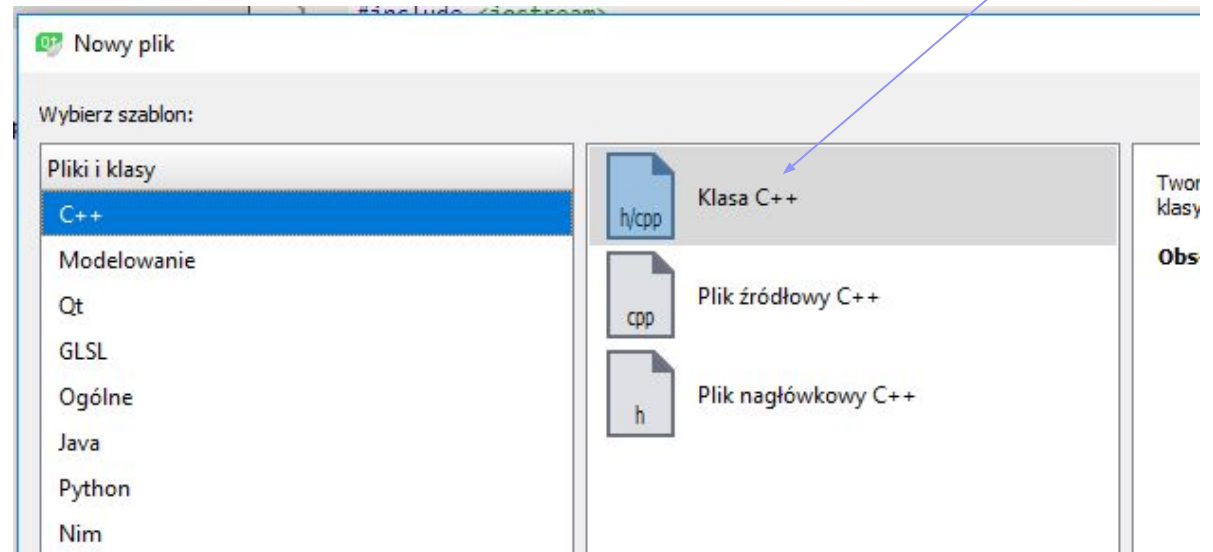
Rozwija się menu wyboru:

Programy wieloplikowe

Wybieramy opcję **Dodaj nowy ...**



Rozwija się kolejne menu, wybieramy opcję **Klasa C++**



Programy wieloplikowe

Pojawia się kolejne okno, w okienku **Nazwa klasy** wpisujemy nazwę klasy (lub funkcji). W naszym projekcie piszemy nazwę **punkt**, klikamy opcję **dalej**.

Klasa C++

Szczegóły
Podsumowanie

Zdefiniuj klasę

Nazwa klasy: punkt

Klasa bazowa: <Własna>

☐ Dołącz QObject

☐ Dołącz QWidget

☐ Dołącz QMainWindow

☐ Dołącz QDeclarativeItem - Qt Quick 1

☐ Dołącz QQuickItem - Qt Quick 2

☐ Dołącz QSharedData

Plik nagłówkowy: punkt.h

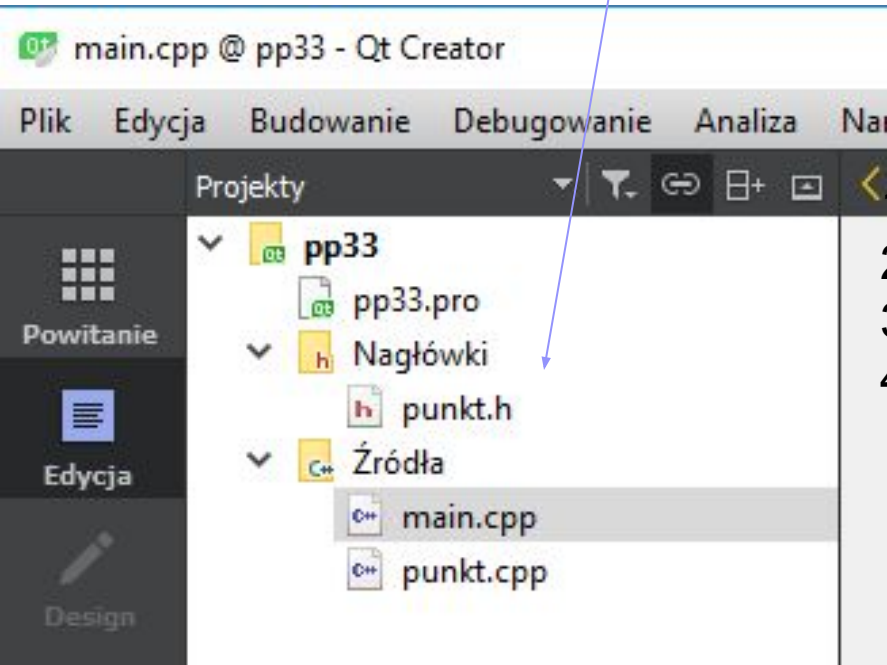
Plik źródłowy: punkt.cpp

Ścieżka: C:\Users\mikfiz\Documents\pp33 Przeglądaj...

Dalej Anuluj

Programy wieloplikowe

Na końcu (w końcu) pojawia się edytor projektu. Qt tworzy zestaw plików. Najważniejsze są cztery pliki (widoczne na zrzucie ekranu).



Te pliki (w naszym przykładzie to:

1. pp33.pro – plik pakietu Qt
2. punkt.h – plik nagłówkowy (deklaracje
3. punkt.cpp – plik z definicjami
4. main.cpp – plik funkcji main(), testowanie

Musimy opracować trzy pliki: punkt.h, punkt.cpp, main.cpp

projekt wieloplikowy (.pro)

Plik systemowy (w naszym przypadku o nazwie pp33.pro), może mieć postać:

-----plik pp33.pro-----

TEMPLATE = app

CONFIG += console c++11

CONFIG -= app_bundle

CONFIG -= qt

SOURCES += main.cpp \
punkt.cpp

HEADERS += \
punkt.h

W tym pliku radzimy nic nie zmieniać!

projekt wieloplikowy (.h)

Kolejny plik, o nazwie **punkt.h** jest plikiem nagłówkowym. W naszym przykładzie może mieć postać:

----- plik punkt.h -----

```
#ifndef PUNKT_H  
#define PUNKT_H
```

```
class punkt  
{ int x,y;  
  public:  
    punkt();  
    punkt (int, int);  
    void pokaz();  
};
```

```
#endif // PUNKT_H  
-----
```

Dyrektywy preprocesora:

```
#ifndef PUNKT_H  
#define PUNKT_H  
  
.....  
.....  
.....  
#endif // PUNKT_H
```

Zapobiegają wielokrotnemu
włączaniu tego pliku do programu.
Radzimy stosować tą konwencję.

W tym pliku możemy umieścić deklaracje klas, metod , funkcji, itp..

projekt wieloplikowy (.cpp)

Kolejny plik o nazwie punkt.cpp może mieć postać:

----- plik punkt.cpp -----

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

```
punkt::punkt()
{ x = 0 ;
  y = 0 ;
};
```

```
punkt::punkt(int xx, int yy)
{ x = xx ;
  y = yy ;
}
```

```
void punkt::pokaz(){
  cout << "x = " << x << "   y = " << y << endl;
}
```

W tym pliku zazwyczaj umieszczamy definicje klas, metod, funkcji .

projekt wieloplikowy (main.cpp)

Ostatnim plikiem jest plik o nazwie main.cpp. Jest to po prostu plik wykonawczy, który tworzy naszą aplikację. W naszym przypadku może mieć postać:

----- plik main.cpp -----

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

int main()
{ punkt p1(2,4);
  cout << "utworzony obiekt klasy punkt :"<<endl;
  p1.pokaz();
  return 0;
}
```

Wynik:

utworzony obiekt klasy punkt :
x = 2 y = 4

W funkcji **main()** testujemy naszą klasę. Realizujemy proste zadanie. Klasa o nazwie **punkt** pozwala utworzyć obiekt **punkt** (dwuwymiarowy), zainicjalizować go dwoma wartościami (x i y) a następnie wyświetlić współrzędne tego punktu.

Referencje- przypomnienie

Pamiętamy, że w języku C++ argumenty w wywoływanej funkcji przekazywane są przez **wartość**, **wskaźnik** lub przez **referencję**.

Kiedy argument przekazywany jest przez wartość, tworzona jest kopia wartości argumentu i ta kopia przekazywana jest do wywoływanej funkcji. Możemy dokonywać dowolnych zmian na kopii, oryginalna wartość jest bezpieczna – **nie ulega zmianom**. Przekazywanie argumentów przez wartość jest bezpieczne, ale bardzo **wolne**. Przekazywanie parametrów funkcji przez referencje jest korzystne ze względu na wydajność – jest po prostu procesem bardzo szybkim. Za każdym razem, gdy przekazywany jest obiekt do funkcji przez wartość, tworzona jest kopia tego obiektu. Za każdym razem, gdy zwracany jest obiekt z funkcji tworzona jest kolejna kopia. Obiekty kopiowane są na stos. Wymaga to sporej ilości czasu i pamięci. W przypadku zdefiniowanych przez programistę dużych obiektów, ten koszt staje się bardzo duży. Rozmiar obiektu umieszczonego na stosie jest sumą rozmiarów wszystkich jego zmiennych składowych.

Stosowanie referencji jest korzystne, ponieważ eliminuje koszty związane z kopiowaniem dużych ilości danych.

Referencje, funkcje zaprzyjaźnione

```
#include <iostream.h>
#include <conio.h>
class Demo_1
{ friend void ustawA ( Demo_1 &, int );    // funkcja zaprzyjazniona
public:
    Demo_1( ) { a = 0; }                  // konstruktor
    void pokaz () const { cout << a << endl; }
private:
    int a;
};
void ustawA ( Demo_1 &x, int ile )
{ x.a = ile; }
int main()
{ Demo_1 zm;
  cout << "zm.a po utworzeniu obiektu: ";
  zm.pokaz();
  cout << "zm.a po wywołaniu funkcji friend ustawA: ";
  ustawA ( zm, 10);
  zm.pokaz();
  getch();
  return 0;
}
```

Po uruchomieniu otrzymujemy następujący wynik:

zm.a po utworzeniu obiektu : 0

zm.a po wywołaniu funkcji friend ustawA: 10

Referencje, funkcje zaprzyjaźnione

W trakcie wykonywania programu zostaje utworzony obiekt **zm** i danej **a** jest przypisana wartość **0**. Następnie wywołana zostaje funkcja zaprzyjaźniona **ustawA()** i danej **a** zostaje przypisana wartość **10**.

Funkcja składowa **pokaz()**:

```
void pokaz () const { cout << a << endl; }
```

została zadeklarowana jako **const**, co oznacza, że nie może ona modyfikować danych obiektów. Funkcja deklarowana jest jako **const** zarówno w prototypie jak i w definicji przez wstawienie słowa kluczowego po liście jej parametrów i, w przypadku definicji przed nawiasem klamrowym rozpoczynającym ciało funkcji. Programista może określić, które obiekty wymagają modyfikacji, a które pod żadnym pozorem nie mogą być zmieniane. Gdy obiekt nie może być zmieniony, programista może posłużyć się słowem kluczowym **const**. Wszystkie próby późniejszej modyfikacji takiego obiektu znajdowane są już w trakcie kompilacji programu.

Referencje, funkcje zaprzyjaźnione (1)

W kolejnym przykładzie, pokażemy jak napisać niezależną funkcję **pokaz()**, zaprzyjaźnioną z klasą **punkt**, wypisującą na ekranie współrzędne punktu.

Deklaracja wyjściowa klasy **punkt** ma postać:

```
class punkt
{
    int x, y;
    public:
        punkt (int xx = 0, int yy = 0)
        { x = xx ;  y = yy ;
        }
};
```

Aby mieć dostęp do prywatnych danych klasy **punkt**, co jest nam potrzebne, aby wyświetlić współrzędne punktu **x** i **y**, musimy dysponować zewnętrzną funkcją **pokaz()**, zaprzyjaźnioną z klasą **punkt**. Argumenty do tej funkcji muszą być przekazane jawnie.

Referencje, funkcje zaprzyjaźnione

Schemat zależności w przykładowym programie, funkcja jest zaprzyjaźnioną z klasą **punkt**, wypisuje na ekranie współrzędne punktu.

```
// plik „punkt1.h”
#ifndef _PUNKT1_H
#define _PUNKT1_H
class punkt
{ int x, y;
public:
friend void pokaz ( const punkt & ) ;
punkt (int xx = 0, int yy = 0)
    { x = xx ;   y = yy;
    }
};
#endif
```

```
// plik „punkt1.cpp”
#include "punkt1.h"
#include <iostream.h>

void pokaz (const punkt &p)
{
    cout << " współrzędne punktu: "
          << p.x << " " << p.y << "\n";
}
```

```
//program testujący
#include <iostream.h>
#include <conio.h>
#include <punkt1.h>

int main()
{
    // ciało funkcji testującej
}
```

Referencje, funkcje zaprzyjaźnione (2)

Prototyp funkcji **pokaz()** może mieć postać:

```
void pokaz ( punkt );
```

gdy chcemy przekazywać argumenty przez wartość,
lub:

```
void pokaz (punkt & );
```

gdy chcemy przekazać argumenty przez referencję.

Możemy usprawnić funkcję **pokaz()** wiedząc, że ta funkcja nie zmienia współrzędnych i zastosować modyfikator **const**:

```
void pokaz ( const punkt & );
```

Modyfikacja pierwotnej klasy **punkt** może mieć postać pokazaną na wydruku.

Referencje, funkcje zaprzyjaźnione (3)

//Dostęp do składowych prywatnych, funkcje zaprzyjaźnione

// deklaracja klasy punkt, plik „punkt1.h”

#ifndef _PUNKT1_H

#define _PUNKT1_H

class punkt

{ int x, y;

public:

friend void pokaz (const punkt &) ;

punkt (int xx = 0, int yy = 0)

{ x = xx ; y = yy;

}

};

#endif

Referencje, funkcje zaprzyjaźnione (4)

Funkcja służąca do wyświetlenia współrzędnych ma postać:

friend void pokaz(const punkt &);

Funkcja **pokaz()** jest zaprzyjaźniona z klasą **punkt** i ma dostęp do prywatnych danych **x** i **y**. Jeżeli powstanie obiekt klasy **punkt** o nazwie **p** to możemy uzyskać dostęp do jego składowych klasycznie:

p.x i p.y

Definicję funkcji **pokaz()** umieszczamy w oddzielnym pliku:

```
//definicja klasy punkt , plik „punkt1.cpp”
#include "punkt1.h"
#include <iostream.h>
void pokaz (const punkt &p)
{ cout << " współrzędne punktu: " << p.x << " " << p.y << "\n";
}
```

Funkcje zaprzyjaźnione, obiekty dynamiczne (1)

W kolejnym programie przypominaemy także tworzenie **obektów dynamicznych**.

W języku C++ dla każdego programu przydzielany jest pewien obszar pamięci dla alokacji obiektów tworzonych dynamicznie. Obszar ten jest zorganizowany w postaci sterty (czasem – kopca?) (ang. heap). Na sterce alokowane są obiekty dynamiczne.

Obiekty dynamiczne tworzone są przy pomocy operatora `new`.

Operator **`new`** alokuje (przydziela) pamięć na sterce. Gdy obiekt nie jest już potrzebny należy go zniszczyć przy pomocy operatora **`delete`**. Aby można było zastosować operator **`new`** należy najpierw zadeklarować zmienną wskaźnikową, dla przykładu:

```
int *wsk;
```

Tworzenie zmiennej dynamicznej ma postać:

```
wsk = new typ;
```

lub

```
wsk = new typ (wartość);
```

gdzie typ oznacza typ zmiennej (np. int, float, itp.)

Funkcje zaprzyjaźnione, obiekty dynamiczne (2)

Możemy deklarację zmiennej wskaźnikowej połączyć z tworzeniem zmiennej dynamicznej:

```
typ * wsk = new typ;
```

Tak złożona instrukcja może mieć przykładową postać:

```
int *wsk = new int;
```

W pokazanym programie utworzono zmienną dynamiczną w następujący sposób:

```
cout << "zmienna dynamiczna : " << endl;  
punkt *wsk;  
wsk = new punkt (20, 40);
```

Funkcje zaprzyjaźnione, obiekty dynamiczne (3)

```
#include <iostream.h>           //jeden plik 2 wersja
#include <conio.h>
class punkt
{ int x,y;
public:
    friend void pokaz(const punkt &);
    punkt(int xx=0, int yy=0)
    {x = xx;  y = yy;
    }
};
void pokaz(const punkt &p)
{ cout << "wspolrzedne punktu: "<<p.x<< " " << p.y << "\n"; }
int main()
{ cout << "zmienna automatyczna : " << endl;
  punkt p1(10,20);
  pokaz(p1);
  cout << "zmienna dynamiczna : " << endl;
  punkt *wsk;
  wsk = new punkt (20,40);
  pokaz(*wsk);
  getch();
  return 0;
}
```

Wynik:
zmienna automatyczna :
wspolrzedne punktu: 10 20
zmienna dynamiczna :
wspolrzedne punktu: 20 40

Funkcja składowa zaprzyjaźniona z inną klasą (1)

Funkcja zaprzyjaźniona danej klasy może być też funkcją składową zupełnie innej klasy.

Taka funkcja ma dostęp do prywatnych danych swojej klasy i do danych klasy, z którą się przyjaźni. Kolejny przykład ilustruje to zagadnienie.

Mamy dwie klasy - klasę **prostokat** i klasę **punkt**. Klasa **prostokat** definiuje prostokąt przy pomocy współrzędnych dwóch punktów – lewego dolnego rogu prostokąta i prawego górnego rogu prostokąta. Klasa **punkt** opisuje punkt na podstawie jego współrzędnych kartezjańskich.

Mając dany punkt i dany prostokąt należy określić czy punkt znajduje się wewnątrz prostokąta czy też leży poza nim.

W programie deklarujemy dwie klasy – **punkt** i **prostokat** z konstruktorami. Funkcja **miejsce()** jest funkcją składową klasy **prostakat** i jest zaprzyjaźniona z klasą **punkt**.

W programie testującym wywołujemy funkcję **miejsce()**, aby ustalić położenie punktu względem prostokąta.

Funkcja składowa zaprzyjaźniona z inną klasą (2)

```
#include <iostream>
#include <conio>
using namespace std;
class punkt;    //deklaracja zapowiadająca

class prostokat
{ int xp, yp, xk, yk;
public:
    prostokat( int xpo, int ypo, int xko, int yko);
    void miejsce ( punkt &p);
};

class punkt
{ int x1, y1;
public:
    punkt (int ax, int ay);
    friend void prostokat::miejsce ( punkt &p);
};

prostokat::prostokat( int xpo, int ypo, int xko,
                    int yko)
{ xp = xpo; yp = ypo;
  xk = xko; yk = yko;
}
```

```
punkt :: punkt (int ax, int ay)
{ x1 = ax; y1 = ay;
}

void prostokat :: miejsce( punkt &pz)
{ if ( (pz.x1 >= xp) && (pz.x1 <= xk)
      &&
      (pz.y1 >= yp) && (pz.y1 <= yk)
    )
    cout << "punkt lezy w polu" << endl;
  else
    cout << "punkt lezy poza polem" << endl;
}

int main()
{
    prostokat pr( 0, 0, 100, 100);
    punkt pu ( 10, 10);
    pr.miejsce(pu);
    getch();
    return 0;
}
```

Wydruk z programu ma postać:

punkt lezy w polu

Funkcja składowa zaprzyjaźniona z inną klasą (3)

Funkcja **miejsce()** jest zwykłą funkcją składową klasy **prostokat**. W deklaracji funkcji składowej **miejsce()** argumentem jest obiekt klasy **punkt**, dlatego konieczna jest **deklaracja zapowiadająca** klasę **punkt**.

Deklaracja klasy **punkt** ma postać:

```
class punkt
{ int x1, y1;
public:
    punkt (int ax, int ay);
    friend void prostokat::miejsce ( punkt &p);
};
```

Deklaracja funkcji zaprzyjaźnionej ma postać:

```
friend void prostokat::miejsce ( punkt &p);
```

W deklaracji zaprzyjaźnionej funkcji **miejsce()** musimy podać nazwę klasy, w której ta funkcja jest funkcją składową, w naszym przypadku jest to klasa **prostokat**.

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor

Założmy, że są zdefiniowane dwie klasy: **wektor** i **macierz**. Każda z nich ukrywa swoje dane i dostarcza odpowiedni zbiór operacji do działania na obiektach swojego typu.

Należy zdefiniować funkcję mnożącą macierz przez wektor.

Ustalmy konkretne warunki. Niech wektor ma cztery elementy indeksowane 0,1,2,3. Wektor zapamiętywany będzie w postaci tablicy jednowymiarowej. Macierz ma rozmiar 4x4 i będzie zapamiętywana w postaci tablicy dwuwymiarowej. Funkcja obliczająca iloczyn musi korzystać z danych pochodzących z dwóch klas, jest więc oczywiste, że musi być z nimi **zaprzyjaźniona**. W tym przypadku konieczne jest także użycie **deklaracji referencyjnej** (zwaną także deklaracją zapowiadającą, albo referencją zapowiadającą), w przypadku deklaracji klasy **wekt** musi wystąpić deklaracja klasy **macierz** i podobnie w deklaracji klasy **macierz** musi wystąpić deklaracja klasy **wekt**. Jest to konieczne, gdyż w klasie **wekt** w deklaracji **iloczyn()** istnieje odwołanie do niezadeklarowanej jeszcze klasy **macierz**. Deklaracje poszczególnych klas i ich definicje zapisujemy w oddzielnych plikach.

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor (1)

Wydruk a. iloczyn macierzy przez wektor, funkcja zaprzyjaźniona

//plik wektor1.h, deklaracja klasy wekt

```
#ifndef _WEKTOR1_H
```

```
#define _WEKTOR1_H
```

```
class macierz ;           // deklaracja zapowiadajaca
```

```
class wekt
```

```
{ double v[4];           //wektor o 4 składowych
```

```
public:
```

```
    wekt(double v1=0, double v2=0, double v3=0, double v4=0)
```

```
    { v[0] = v1; v[1] = v2; v[2] = v3; v[3] = v4; }
```

```
    friend wekt iloczyn (const macierz &, const wekt &) ;
```

```
    void pokaz();
```

```
};
```

```
#endif
```

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor (2)

Wydruk b. iloczyn macierzy przez wektor, funkcja zaprzyjaźniona

```
//plik macierz.h, deklaracja klasy macierz
```

```
#ifndef _MACIERZ_H  
#define _MACIERZ_H
```

```
class wekt;
```

```
class macierz {  
    double mac[4][4] ;           //macierz 4x4  
public:  
    macierz();                   //konstruktor z inicjacja na 0  
    macierz(double t[4][4]);     //konstruktor, dane z tablicy  
    friend wekt iloczyn (const macierz &, const wekt &);  
};
```

```
#endif
```

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor (3)

Wydruk c. iloczyn macierzy przez wektor, funkcja zaprzyjaźniona

```
//plik pokaz.cpp, definicja składowej pokaz()
```

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

```
#include "wektor1.h,,
```

```
void wekt :: pokaz()
```

```
{
```

```
    int i;
```

```
    for (i=0; i < 4; i++)    cout << v[i] << " ";
```

```
    cout << "\n";
```

```
}
```

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor (4)

Wydruk d. iloczyn macierzy przez wektor, funkcja zaprzyjaźniona

//plik konmac.cpp, definicja konstruktora klasy macierz

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

```
#include "macierz.h,,
```

```
Macierz :: macierz (double t [4][4])
```

```
{
```

```
    int i;
```

```
    int j;
```

```
    for (i=0; i<4; i++)
```

```
        for (j=0; j<4; j++)
```

```
            mac[i][j] = t[i][j];
```

```
}
```

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor (5)

Wydruk e. iloczyn macierzy przez wektor, funkcja zaprzyjaźniona

```
//plik iloczyn.cpp
//definicja funkcji iloczyn
#include "wektor1.h"
#include "macierz.h,,

wekt iloczyn (const macierz & m, const wekt & x)
{ int i,j;
  double suma;
  wekt wynik;
  for (i=0; i<4; i++)
    { for (j=0, suma=0; j<4; j++)
        suma += m.mac[i][j] * x.v[j];
      wynik.v[i] = suma;
    };
  return wynik;
}
```

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor

Możemy zacząć testować mnożenie macierzy przez wektor. Funkcję `iloczyn()` możemy wykorzystać do realizacji przekształceń w przestrzeni 3D.

W grafice komputerowej wykorzystuje się tzw. współrzędne jednorodne. W tych współrzędnych punkt (x,y,z) reprezentowany jest jako punkt w przestrzeni 4-wymiarowej $(x,y,z,1)$. Poszczególne przekształcenia, takie jak przesunięcie, skalowanie czy obroty reprezentowane są macierzami 4x4. Aby otrzymać położenie nowego punktu P' należy punkt początkowy P pomnożyć przez macierz przekształcenia M :

$$\mathbf{P}' = \mathbf{M} * \mathbf{P}$$

Przesunięcie:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Obrót wokół osi Z:

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor

Te transformacje można łatwo zweryfikować: wynikiem obrotu o 90 stopni jednostkowego wektora osi x $[1\ 0\ 0\ 1]^T$ powinien być jednostkowy wektor $[0\ 1\ 0\ 1]^T$ osi y.

Ogólnie mamy do czynienia z mnożeniem macierzy przez wektor:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Mnożenie macierzy przez wektor, testy

```
#include <iostream.h>
#include <conio.h>
#include "wektor1.h"
#include "macierz.h"
int main()
{ wekt w (1, 0, 0, 1);
  wekt wynik;
  wynik = w;
  cout << "punkt początkowy: ";
  wynik.pokaz();
  double trans[4][4] = { 1, 0, 0, 2,
                        0, 1, 0, 3,
                        0, 0, 1, 1,
                        0, 0, 0, 1 };

  macierz a = trans;
  wynik = iloczyn (a, w);
  cout << " po translacji: ";
  wynik.pokaz ();
  double skal[4][4] = { 2, 0, 0, 0,
                        0, 2, 0, 0,
                        0, 0, 2, 0,
                        0, 0, 0, 1 };
```

```
    macierz b = skal;
    wynik = iloczyn (b, w);
    cout << " po skalowaniu: ";
    wynik.pokaz ();
    double obrZ[4][4] = { 0, -1, 0, 0,
                          1, 0, 0, 0,
                          0, 0, 1, 0,
                          0, 0, 0, 1 };

    macierz c = obrZ;
    wynik = iloczyn (c, w);
    cout << "po obrocie o  $\pi/2$  : ";
    wynik.pokaz ();
    getch();
    return 0;
}
```

Wydruk z programu ma postać:

```
punkt początkowy : 1 0 0 1
    po translacji : 3 3 1 1
    po skalowaniu : 2 0 0 1
    po obrocie o  $\pi/2$  : 0 1 0 1
```

Funkcje zaprzyjaźnione, mnożenie macierzy przez wektor

XXX

```
//plik wektor1.h,  
//deklaracja klasy wekt  
#ifndef _WEKTOR1_H  
#define _WEKTOR1_H  
class macierz ;  
class wekt  
{  
    //.....  
};  
#endif
```

```
//plik pokaz.cpp,  
//definicja składowej pokaz()  
#include <iostream.h>  
#include "wektor1.h"  
void wekt::pokaz()  
{ int i;  
  for (i=0; i < 4; i++)   cout << v[i] << " ";  
  cout << "\n";  
}
```

```
//plik macierz.h,  
// deklaracja klasy macierz  
#ifndef _MACIERZ_H  
#define _MACIERZ_H  
class wekt;  
class macierz  
{  
    //.....  
};  
#endif
```

```
//plik konmac.cpp,  
//definicja konstruktora klasy macierz  
#include <iostream.h>  
#include "macierz.h"  
macierz::macierz (double t [4][4])  
{ int i;  
  int j;  
  for (i=0; i<4; i++)  
    for (j=0; j<4; j++)  
      mac[i][j] = t[i][j];  
}
```

```
//plik iloczyn.cpp , definicja funkcji iloczyn  
#include "wektor1.h"  
#include "macierz.h"  
wekt iloczyn (const macierz & m, const wekt & x)  
{ int i,j;  
  double suma;  
  wekt wynik;  
  for (i=0; i<4; i++)  
    { for (j=0, suma=0; j<4; j++)   suma += m.mac[i][j] * x.v[j];  
      wynik.v[i] = suma;  
    };  
  return wynik;  
}
```

```
#include <iostream.h>  
#include <conio.h>  
#include "wektor1.h"  
#include "macierz.h"  
int main()  
{  
    //.....  
}
```

Struktura projektu mnożenia
macierzy przez wektor – 6 plików

Klasy zaprzyjaźnione

Każda klasa może mieć wiele funkcji zaprzyjaźnionych, można nawet uczynić **wszystkie** funkcje składowe jednej klasy **zaprzyjaźnione** z inną klasą. Możemy przy pomocy słowa kluczowego **friend** uczynić daną klasę zaprzyjaźnioną z inną klasą. Jeżeli zadeklarujemy, że **cała klasa A jest uznawana za przyjaciela klasy B** to ten fakt zapisujemy w następujący sposób:

```
class B
{
friend class A ;
// .....
};
```

Zapis:

```
friend class A ;
```

oznacza, że wszystkie funkcje składowe klasy **A** mają dostęp do danych prywatnych i chronionych klasy **B**. Jeżeli klasa **A** ma być zaprzyjaźniona z klasą **B**, to deklaracja klasy **A** musi poprzedzać deklarację klasy **B**.

Klasy zaprzyjaźnione

```
#include <iostream>
#include <conio>
using namespace std;
class Dane
{
    int x1, x2;
public:
    Dane (int a, int b) { x1 = a; x2 = b; }
    friend class Test;
};
class Test
{
public:
    int min(Dane a)
        { return a.x1 < a.x2 ? a.x1 : a.x2; }
    int iloczyn (Dane a)
        { return a.x1 * a.x2 ; }
};
```

```
int main()
{
    Dane liczby(10, 20);
    Test t;
    cout << "mniejsza to: " << t.min(liczby)
          << endl;
    cout << "    iloczyn = " << t.iloczyn(liczby)
          << endl;

    getch();
    return 0;
}
```

Wydruk z programu ma postać:

mniejsza to: 10
 iloczyn = 200

Klasy zaprzyjaźnione

Klasa zaprzyjaźniona to klasa której metody mają dostęp do prywatnych oraz chronionych metod i danych innej klasy.

W naszym przykładzie *klasa prostokat* jest przyjacielem *klasy kwadrat*, umożliwiając członkom funkcji *prostokat* dostęp do prywatnych i chronionych członków *kwadrat*. Konkretniej, *prostokat* uzyskuje dostęp do zmiennej składowej *kwadrat :: bok*, która opisuje bok kwadratu.


W tym przykładzie jest coś nowego: na początku programu jest pusta deklaracja *klasy kwadrat*. Jest to konieczne, ponieważ *klasa prostokat* używa *kwadrat* (jako parametru w przeliczeniu elementu), a *kwadrat* używa *prostokat* (deklarując go jako znajomego).

W naszym przykładzie *prostokat* jest uważany za klasę przyjaciela po *kwadrat*, ale *kwadrat* nie jest uważany za przyjaciela przez *prostokat*. Dlatego funkcje członkowskie *prostokat* mogą uzyskać dostęp do chronionych i prywatnych członków *kwadrat*, ale nie na odwrót. Oczywiście *kwadrat* może zostać również uznany za przyjaciela *prostokat*, jeśli zajdzie taka potrzeba. musimy przyznać taki dostęp.

Klasy zaprzyjaźnione

```
// friend class
#include <iostream>
using namespace std;
class kwadrat; //deklaracja zapowiadajaca
class prostokat {
    int w, h;
public:
    prostokat(){w=0; h=0;};
    prostokat (int a, int b){w=a; h = b;};
    int pole () {return (w * h);}
    void convert (kwadrat a);
};

class kwadrat {
    friend class prostokat;
private:
    int bok;
public:
    kwadrat (int a) : bok(a) {}
};
```



```
void prostokat::convert (kwadrat a) {
    w = a.bok;    h = a.bok;
}

int main () {
    prostokat pr(4,5), pr1;
    cout <<"pole prostokata = "<< pr.pole()<<endl;
    kwadrat kw (4);
    pr.convert(kw);
    cout <<"pole kwadratu  = " <<pr.pole()<<endl;
    kwadrat kw1 (12);
    pr1.convert(kw1);
    cout <<"inny kwadrat, pole = "<< pr1.pole()<<endl;

    return 0;
}
```

Wynik:

```
pole prostokata = 20
pole kwadratu   = 16
inny kwadrat, pole = 144
```

Z1. Należy opracować *klasę prostokąt*, która obliczy pole powierzchni prostokąta.

Klasa prostokąt posiada funkcję zaprzyjaźnioną, dzięki której jesteśmy w stanie pobrać wartości boków prostokąta z klawiatury.

Z2. W początku układu współrzędnych umieszczamy okrąg o promieniu *rk*. Na tej samej płaszczyźnie umieszczamy drugi okrąg o promieniu *r* oraz współrzędnych ich środka okręgu (*x* i *y*).

Napisać program testujący czy te dwa okręgi się przecinają. Utworzyć dwie klasy *kolo* i *kolo_centra*. Wykorzystać technikę klas zaprzyjaźnionych



Wykład 4

KONIEC