Young Programmer

www.inceptio.org.pl

Organizator:

fundacja INCEPTIO

Partnerzy:

Wydział Fizyki Politechnika Warszawska

Stowarzyszenie KLATRAT

Dofinansowanie:

MIASTO STOŁECZNE WARSZAWA

Zakochaj się w Warszawie

Young Programmer

www.inceptio.org.pl

Zajęcia nr 4

Programowanie obiektowe - konstruktory

dr inż. Łukasz Graczykowski
mgr inż. Leszek Kosarzewski
Wydział Fizyki Politechniki Warszawskiej

Organizator:

Partnerzy:

Dofinansowanie:

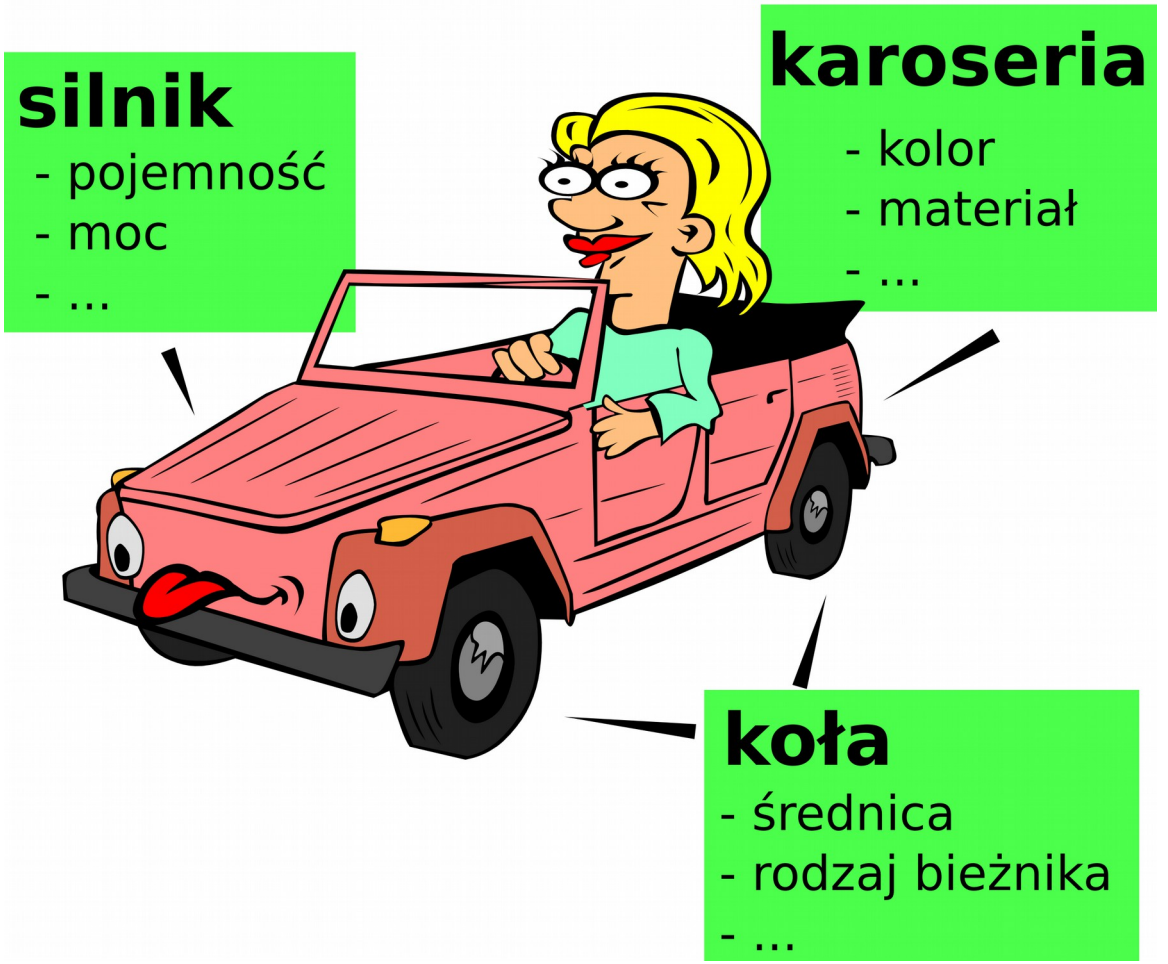
MIASTO STOŁECZNE WARSZAWA

Plan

- Klasy – przypomnienie
- Konstruktory
- Destruktor
- Przeciążanie operatorów

Idea “obiektu” - przykład

Samochód jako obiekt...
złożony z obiektów



Źródło wszystkich obrazków w tej prezentacji: http://openclipart.org

Dobra rada

ZAWSZE zaczynajcie pisanie swojego programu od NAJMNIEJSZEJ możliwej kompilującej się wersji.

Konstruktor i destruktor

- **Konstruktor** jest specjalną metodą (może ich być kilka – przeciążonych), która zostaje wywołana przy tworzeniu obiektu klasy. Na ogół służy do inicjalizacji składowych.

  Konstruktor musi nazywać się identycznie jak klasa i nie posiada typu zwracanego (nawet void!)

- **Destruktor** jest metodą (zawsze bezparametrową), która jest wywoływana automatycznie wtedy, gdy obiekt jest niszczony. Nadaje się więc doskonale do wywołania czynności finalizująco-sprzątających.

  Nazwa destruktora jest identyczna jak nazwa klasy i poprzedzona tyldą (~)

Konstruktory

Celem konstruktora jest nadanie początkowych wartości obiektu (NIE jego konstrukcja!!!).

Jeśli tworzymy nowy obiekt to musimy mu nadać wartość początkową np. **int a; a = 0;** lub **int a = 0**;

Jednak jeśli mamy konstruktor, to nie musimy tego robić!!

Konstruktor domyślny

```
class Silnik
{
int pojemnosc, moc;

public:
Silnik(){pojemnosc = 0; moc = 0;}

};


int main()
{
Silnik diesel;
return 0;
}
```

Nadajemy wartości początkowe

W tym momencie został wywołany konstruktor domyślny.
Wartości składowych obiektu diesel wynoszą:
diesel.pojemnosc = 0
diesel.moc = 0

Konstruktor główny

```
class Silnik
{
double pojemnosc, moc;

public:
Silnik(){pojemnosc = 0.; moc = 0.}
Silnik(double a, double b){pojemnosc = a; moc = b;}

};


int main()
{
Silnik diesel;
Silnik wankla(1.3, 239.);
return 0;
}
```

Nadajemy wartości początkowe, które
Są wczytywane podczas wywołania konstruktora

W tym momencie został wywołany konstruktor główny.
Wartości składowych obiektu wankla wynoszą:
wankla.pojemnosc = 1.3
wankla.moc = 239.0

Konstruktor kopiujący

```
class Silnik
{
double pojemnosc, moc;

public:
Silnik(){pojemnosc = 0.; moc = 0.}
Silnik(double a, double b){pojemnosc = a; moc = b;}
Silnik(const silnik& s){pojemnosc = s.pojemnosc; moc = s.moc;}

};


int main()
{
Silnik diesel;
Silnik wankla(1.3, 239.);
Silnik diesel2(diesel1);
return 0;
}
```

Nadajemy wartości początkowe, które są kopiowane z innego obiektu danego typu

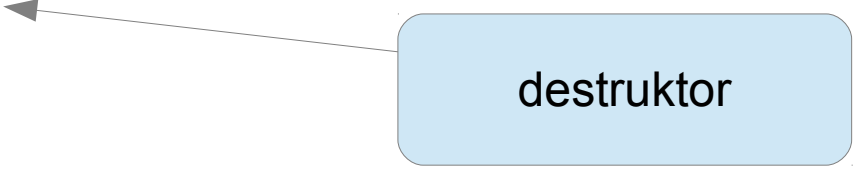
W tym momencie został wywołany konstruktor kopiujący. Wartości składowych obiektu diesel2 są takie same jak diesel

Destruktor

Destruktor nie usuwa obiektu. Jest on wywoływany w momencie usuwania obiektu i służy zwalnianiu zasobów (np. pamięci).

Destruktor nie ma określonego typu jaki zwraca, ani nie pobiera argumentów. Składa się on z wężyka '~' i nazwy klasy np.

```
class Silnik
{
    double pojemnosc, moc;

    public:
    Silnik(){pojemnosc = 0.; moc = 0.}
    Silnik(double a, double b){pojemnosc = a; moc = b;}
    Silnik(const silnik& s){pojemnosc = s.pojemnosc; moc = s.moc;}
    ~Silnik(){cout<<"Silnik: Uzycie destruktora"<<endl;}
};
```



Rozbudowujemy kod

```
#include <iostream>
#include <string>

class Silnik
{
double pojemnosc;
double moc;

public:
Silnik() {pojemnosc = 0; moc = 0;}
~Silnik() {cout<<"Uzycie destruktora"<<endl;}
Silnik(double Moc, double Poj): moc(Moc),pojemnosc(Poj) {}

void setMoc(double M) { moc = M; };
double getMoc() { return moc; };
};

using namespace std;

int main()
{
//Silnik testowy_silnik;
Silnik testowy_silnik(100.0,2.-); //Tworzymy silnik o mocy 100 KM i poj. 2 l.
cout << "Moc silnika: " << testowy_silnik.getMoc() << " KM!" << endl;

return 0;
}
```

Lista inicjalizacyjna konstruktora (zastępuje '=')

WAŻNY ŚREDNIK!

Kompilujemy i poprawiamy ewentualne błędy...

Podział na jednostki kompilacji

silnik.hpp

```
class Silnik
{
double moc;
double pojemnosc;

public:
Silnik() {moc = 0; pojemnosc =
0;}
~Silnik() {}
Silnik(double Moc, double
Pojemnosc) {moc = Moc;
pojemnosc = Pojemnosc;}

void setMoc(double Moc) { moc
= Moc; }
double getMoc() { return
moc; }
void setPojemnosc(double Moc)
{ moc = Moc; }
double getPojemnosc() { return
pojemnosc; }

};
```

main.cpp

```
#include "samochod.hpp"

using namespace std;

int main()
{
Silnik testSilnik;
testSilnik.setMoc(100.0);

cout<<"Moc testowego silnika: "<<testSilnik.getMoc()<<endl;
cout<<"Pojemnosc testowego silnika: "<<testSilnik.getPojemnosc()<<endl;

return 0;
}
```

Podział na jednostki kompilacji

silnik.hpp

```
class Silnik
{
double moc;
double pojemnosc;

public:
Silnik();
~Silnik();
Silnik(double Moc, double
Pojemnosc);

void setMoc(double Moc);
double getMoc();
void setPojemnosc(double Moc);
double getPojemnosc();

};
```

Deklaracje w pliku nagłówkowym .hpp

silnik.cpp

Definicje w pliku .cpp

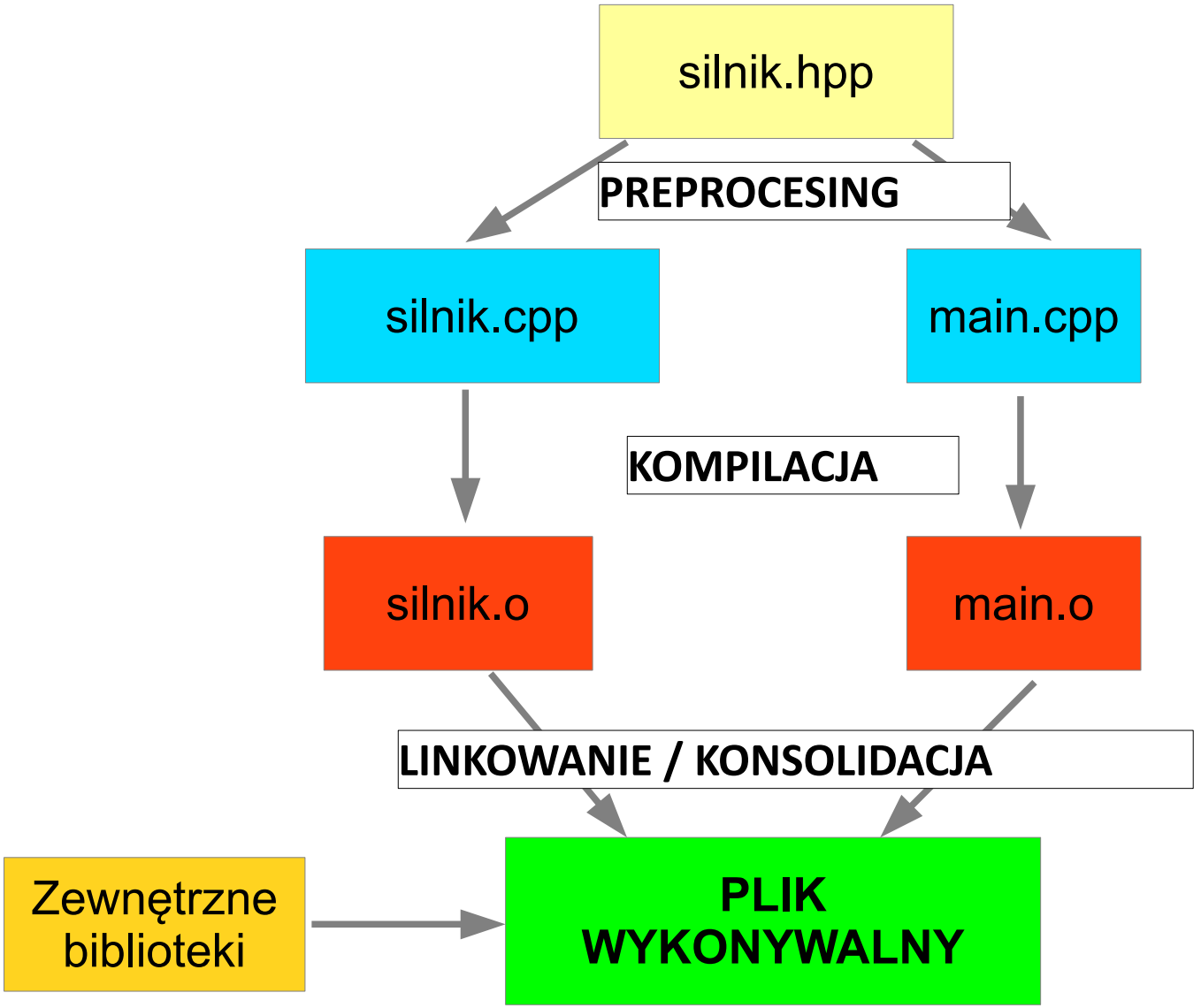
```
#include "silnik.hpp"

using namespace std;

Silnik::Silnik() {moc = 0; pojemnosc = 0;}
Silnik::~Silnik() {}
Silnik::Silnik(double Moc, double Pojemnosc) {moc = Moc; pojemnosc =
Pojemnosc;}

void Silnik::setMoc(double Moc) { moc = Moc; }
double Silnik::getMoc() { return moc; }
void Silnik::setPojemnosc(double Moc) { moc = Moc; }
double Silnik::getPojemnosc() { return pojemnosc; }
```

main.cpp

Użycie w pliku głównym

```
#include "silnik.hpp"

using namespace std;

int main()
{
Silnik testSilnik;
testSilnik.setMoc(100.0);

cout<<"Moc testowego silnika: "<<testSilnik.getMoc()<<endl;
cout<<"Pojemnosc testowego silnika: "<<testSilnik.getPojemnosc()<<endl;

return 0;
}
```

Program złożony z wielu plików



Makefile

Makefile

```
all:
    g++ -c silnik.cpp -o silnik.o
    g++ -c main.cpp -o main.o
    g++ silnik.o main.o -o silnik
clean:
    rm *.o
```

I teraz wystarczy wpisać w konsoli (w katalogu z kodem i Makefilem):

make

aby wszystkie pliki się skompilowały!

Przeciążanie operatorów

Przykład: Chcemy sprawdzić czy dwa silniki są tymi samymi.
Załóżmy, że dwa silniki są jednakowe jeśli ich silniki mają jednakową moc i pojemność.
Skorzystamy z przeciążonego operatora porównania “==”.

Przeciążony operator porównania '=='

silnik.hpp

```
class Silnik
{
double moc;
double pojemnosc;
public:
Silnik();
~Silnik();
Silnik(double Moc, double Pojemnosc);

void setMoc(double Moc);
double getMoc();
void setPojemnosc(double Moc);
double getPojemnosc();

bool operator==(const Samochod& sil);

};
```

silnik.cpp

```
#include "samochod.hpp"

//...

bool Silnik::operator==(const Silnik& sil)
{
if ( this->moc == sil.moc )
      return true;
else
      return false;
}
```

main.cpp

```
#include "silnik.hpp"

using namespace std;

int main()
{
Silnik testSilnik;
Silnik testSilnik2(100.0,2.0);


std::cout << (testSilnik2 == testSilnik ? "Silniki sa jednakowe!" : "Silniki sie roznia!") << std::endl;

return 0;
}
```

Sprawdźcie co się stanie jeśli dodacie tylko te linijki do maina (bez przeciążania operatora).

Jednolinijkowy zapis komendy if

Dla dociekliwych

- Dziedziczenie i polimorfizm: http://guidecpp.cal.pl/cplus,polimorph
- Przeciążanie operatorów: http://4programmers.net/C/Przeładowywanie_operatorów
- Typy wyliczeniowe: http://edu.pjwstk.edu.pl/wyklady/pro/scb/PRG2CPP_files/node21.html
- Unie: http://cpp0x.pl/kursy/Kurs-C++/Unia-w-C++/314
- Pola bitowe: http://edu.pjwstk.edu.pl/wyklady/pro/scb/PRG2CPP_files/node98.html

Young Programmer

www.inceptio.org.pl

Zajęcia nr 4

Programowanie obiektowe - konstruktory

Zadania

Organizator:

fundacja INCEPTIO

Partnerzy:

Wydział Fizyki Politechnika Warszawska

Stowarzyszenie KLATRAT

Dofinansowanie:

Miasto Stołeczne Warszawa

Zakochaj się w Warszawie

Zadanie 1: Składamy silnik

Skopiuj z prezentacji i skompiluj końcowe kody dla przykładu z silnikiem korzystajacego z klasy.

W programie dla klasy stwórz petlę `do{...}while(znak!='k')`, w której wprowadzając znak z klawiatury będzie można wybrać:

```
'm' - ustawić moc silnika
'p' - ustawić pojemność silnika
'k' - zakończyć program
```

Przetestuj program, a następnie dopisz konstruktory, które ustawiać będą domyślne wartości dla tworzonych obiektów przy uruchamianiu programu.

Zadanie 1.a. Wielomian

Modyfikujemy klasę **wielomian**.

a) Składowe prywatne klasy: **double a, b** oraz **string nazwa**. Pozostałe elementy klasy są publiczne.

b) Tworzymy konstruktor domyślny nadający wartości początkowe **a(0.)**, **b(0.)**, **nazwa("")**.

c) Tworzymy konstruktor główny **wielomian(double wsp_a, double wsp_b, string name)**.

d) Tworzymy zewnętrzną funkcję **void wypisz(wielomian)**, która za pomocą komendy cout wypisuje na ekran nazwę wielomianu (nazwa) i równanie liniowe **a**x + **b** (na ekranie widzimy np.: wielomian first: 2.5x + 5.0). Funkcja ma być zaprzyjaźniona z klasą **wielomian**.

e) Stwórz destruktor. W klamrach stwórz komendę cout<<”Destruktor dziala”<<endl; aby było wiadomo kiedy się wykonuje.

f) W funkcji main() piszemy:
Tworzymy obiekt wielomian first za pomocą konstruktora domyślnego.
Za pomocą funkcji wypisz(wielomian) wypisujemy wartosć obiektu first na ekran.
Tworzymy dwa obiekty klasy wielomian za pomocą konstruktora głównego i nadajemy im jakieś wartości, które różnią się od tych w konstruktorze domyślnym.
Za pomocą funkcji wypisz(wielomian) wypisujemy wartości nowych obiektów na ekran.

Zadanie 1.b. Wielomian

Tworzymy **operator+** i **operator!**, które są zaprzyjaźnione z klasą wielomian.
a) Tworzymy operator **wielomian operator+(wielomian w1, wielomian w2)**, który znajduje się poza klasą wielomian. Operator ma tworzyć nowy obiekt **w3** klasy wielomian, który będzie wynikiem dodawania wielomianów **w1** i **w2**. Wypisujemy komendą cout nazwy dodawanych wielomianów (np. cout<<"dodawanie: "<<w1.nazwa<<" + "<<w2.nazwa<<endl;). Zwracamy wielomian **w3**.

b) Tworzymy operator **wielomian operator!(wielomian &)**, który znajduje się poza klasą wielomian. Operator ma tworzyć nowy obiekt w3 klasy wielomian, którego współrzędne będą miały wartości pobranej referencji, pomnożone przez -1. Wypisywać nazwę pobranego wielomianu i zwracać wynik. UWAGA! Tym razem pobieramy wskaźnik do obiektu klasy wielomian.

c) W funkcji main() dopisujemy komendy:
Przypisujemy obiektowi first wynik działania operatora dodawania na nowych obiektach stworzonych konstruktorem głównym.
Wywołujemy funkcję wypisz dla obiektu first.
Deklarujemy obiekt wielomian, któremu przypisujemy wartości będące wynikiem operacji operatora! na obiekcie first.
Wywołujemy funkcje wypisz dla obiektu ujemny.

Zadanie 3a:

Napisz program, który pobierał będzie z pliku tekstowego dane osobowe, wyświetlał je na ekranie i pozwalał na manipulowanie (zmianę, usuwanie) nimi. Reprezentacją osoby w programie ma być odpowiednio zaprojektowana klasa. “Osoby” mogą być przechowywane w tablicy, wektorze lub liście.

Zadanie 3b:

Prosty sposób na pobranie danych z pliku zawierającego dane w formie dwóch kolumn liczb zmiennoprzecinkowych:

dane.txt

expab (USD)	expba (USD)
2.29191e+11	1.74616e+11
2.78e+08	1.053e+09
3.09e+08	5.72e+08
4.474e+09	4.423e+09
6.69e+08	1.36e+09
2.353e+09	1.104e+09

```
#include <cstdlib>
#include <fstream>

//...

double col_one, col_two; //zmienne na dane z kolumn

ifstream input("dane.txt"); //strumien wejsciowy

input.ignore(1000, '\n'); //ignoruj pierwsza linie
while (!input.eof())
{
   input >> col_one >> col_two;
   cout << col_one << '\t' << col_two << endl;
}

//...
```

Zadanie 4a: Prosta gra RPG

Napiszmy prymitywną grę. W głównej funkcji stworzymy dwie postaci (bohatera i potworka). W grze postaci te będą walczyć. Będziemy również monitorować stan obu postac, oraz nadawać nowe parametry istniejącym postaciom. Do wszystkiego wykorzystamy klasę z odpowiednimi metodami.

Zadanie 4b: Tworzenie postaci

Tworzymy klasę **postac**.
a) Składowe **private: int zycie, obrazenia, l_ciosow** oraz **string imie**.
b) Metody **public: UstawPostac(int życie, int obrazenia, int l_ciosow, string imię); UsunPostac()**
d) Stwórz zewnętrzną funkcję zaprzyjaźnioną **void Walcz(postac &a, postac &b)**. Będzie ona odpowiedzialna za rozstrzygnięcie potyczki. Korzystając z referencji działamy bezpośrednio na naszych postaciach!

Działanie funkcji **Walcz**:
– życie postaci **a** ma być równe różnicy jej życia oraz obrażeń zadanych przez postać **b** pomnożonych przez liczbę ciosów postaci **b**.
– życie postaci **b** ma być równe różnicy jej życia oraz obrażeń zadanych przez postać **a** pomnożonych przez liczbę ciosów postaci **a**.
– wstawiamy warunek **if(a.zycie <= 0)** to ustawiamy wartość życia tej postaci na 0 i wypisujemy na ekran, że postać **a** poległa (z wykorzystaniem komendy **cout**).
– wstawiamy warunek **if(b.zycie <= 0)** to ustawiamy wartość życia tej postaci na 0 i wypisujemy na ekran, że postać **b** poległa (z wykorzystaniem komendy **cout**).

Zadanie 4c: Interakcja z graczem

a) Tworzymy funkcję **void WypiszPostac()**

Działanie funkcji:

- Na ekran wypisujemy **imie** postaci **a**.
- Wypisujemy, ile zostało jej życia.
- Wypisujemy jakie zadanie obrażenia i ile zadaje ciosów, oraz łączną wartość obrażeń na turę.

b) Przeciążamy funkcję **UstawPostac**: tworzymy nową funkcję **void UstawPostac(**), która nie przyjmuje żadnych argumentów

Działanie funkcji:

- Na ekranie wypisujemy polecenie dla użytkownika, aby podał parametry postaci.
- Niech poda **imie** postaci. (**cin >> a.imie;**)
- Niech poda początkową wartość życia.
- Niech poda jakie zadaje obrażenia i ile zadaje ciosów na turę.

Zadanie 4d: Fragment maina

```
int main()
{
	postac bohater;
	bohater.UstawPostac(100,30,2,"Zenon Waleczny"); //tworzymy bohatera
	postac mobek;
	mobek.UstawPostac(50,5,1,"bazyliszek"); //tworzymy potwora
	Walcz(bohater,mobek); //tu rozgrywa sie bitwa

	//czesc dla zadnia 4c --------->
	bohater.WypiszPostac(); // wypisujemy na ekran parametry bohatera
	mobek.WypiszPostac(); // wypisujemy na ekran parametry potwora
	Mobek.UstawPostac(); // nadajemy nowe parametry potworka
	Walcz(bohater,mobek);
	bohater.WypiszPostac();
	mobek.WypiszPostac();
	//czesc dla zadania 4c <----------

	return 0;
}
```

Zadanie 4e: Dodajemy konstruktory

Modyfikujemy klasę **postac**.
a) Składowe **private: int zycie, obrazenia, l_ciosow** oraz **string imie**.
b) Konstruktory **public: postac(); postac(int życie, int obrazenia, int l_ciosow, string imię);**
c) Destruktor **public: ~postac(){}**
d) Stwórz przeciążony **void operator-(postac &a, postac &b)**. Będzie on odpowiedzialny za rozstrzygnięcie potyczki. Korzystając z referencji działamy bezpośrednio na naszych postaciach!

WAŻNE!!! Zaprzyjaźnij operator z klasą postac!!

Działanie operatora:
– życie postaci **a** ma być równe różnicy jej życia oraz obrażeń zadanych przez postać **b** pomnożonych przez liczbę ciosów postaci **b**.
– życie postaci **b** ma być równe różnicy jej życia oraz obrażeń zadanych przez postać **a** pomnożonych przez liczbę ciosów postaci **a**.
– wstawiamy warunek **if(a.zycie <= 0)** to ustawiamy wartość życia tej postaci na 0 i wypisujemy na ekran, że postać **a** poległa (z wykorzystaniem komendy **cout**).
– wstawiamy warunek **if(b.zycie <= 0)** to ustawiamy wartość życia tej postaci na 0 i wypisujemy na ekran, że postać **b** poległa (z wykorzystaniem komendy **cout**).

Zadanie 4f: Interakcja z graczem

Przeciążamy operator wyjścia << oraz operator wejścia >>
WAŻNE!!! Zaprzyjaźnij operatory z klasą **postac**!!
a) Stwórz przeciążony operator wyjścia **ostream & operator<<(ostream &ekran, postac &a);** Operator pobiera dwie referencje, jedną do zmiennej typu ostream (dokładnie przezwisko **ekran** do obiektu **cout**) a drugą typu postac.
WYTŁUMACZENIE: (**typ**: ostream, gdyż będziemy zwracać obiekt z klasy ostream; **rodzaj operatora<<:** & referencja, gdyż dzięki referencji do operatora z klasy ostream możemy stworzyć bardziej rozbudowany przeciążony operator).

Działanie operatora:

- Na ekran wypisujemy **imie** postaci **a**.
  WAŻNE!!! robimy to tak: **ekran << a <<"\n";**
- Wypisujemy, ile zostało jej życia.
- Wypisujemy jakie zadanie obrażenia i ile zadaje ciosów, oraz łączną wartość obrażeń na turę.
- Funkcja zwraca wartość ekran: **return ekran;**

b) Stwórz przeciążony operator wejscia **istream & operator>>(istream &klawiatura, postac &a);** Operator pobiera dwie referencje, jedną do zmiennej typu istream (dokładnie przezwisko **klawiatura** do obiektu **cin**), a drugą typu postac.
WYTŁUMACZENIE: (**typ**: istream, gdyż będziemy zwracać obiekt z klasy istream; **rodzaj operatora**>>: & referencja, gdyż dzięki referencji do operatora z klasy istream możemy stworzyć bardziej rozbudowany przeciążony operator).
Działanie operatora:

- Na ekranie wypisujemy polecenie dla użytkownika, aby podał parametry postaci.
- Niech poda **imie** postaci.
  WAŻNE!!! robimy to tak: **klawiatura >> a.imie;**
- Niech poda początkową wartość życia.
- Niech poda jakie zadaje obrażenia i ile zadaje ciosów na turę.
- Funkcja zwraca wartość klawiatura: **return klawiatura;**

Zadanie 4g: Rozbudowa programu

Puść wodze fantazji!
a) Stwórz kolejne przeciążone operatory np. operator++, operator+, operator-- i nadaj nowe opcje grze np. leczenie bohatera, zdobywanie poziomu postaci (dodatkowy parametr w klasie odpowiadający za doświadczenie??) i wiele innych!
b) Wprowadź więcej tur podczas walki, np. aby walka rozgrywała się do momentu spadku **zycia** postaci do poziomu <=0 . Możesz to zrealizować wstawiając do operator- pętlę do-while.
c) Wstaw całą grę w pętlę while(koniec), gdzie **bool koniec = true**. Jeśli w czasie gry chcesz ją zakończyć, to zmień wartość koniec na false.
d) Stwórz w pętli interfejs użytkownika korzystając z switch!
e) Możesz stworzyć funkcje zewnętrzne realizujące zadania, których nie może wykonać przeciążony operator (zadania wymagające pobrania wielu elementów).
f) Temat może zostać rozwinięty o tematykę dziedziczenia. Można stworzyć nowe klasy dziedziczące po klasie postać. Sugerowane nowe klasy: bohater i potwor. Co więcej po klasie bohater mogą dziedziczyć nowe klasy np. wojownik, mag, druid, kaplan.

NIECH PRZYGODA TRWA!!