Język C++

Programowanie obiektowe

Cechy programowania obiektowego

- abstrakcyjne typy danych
- hermetyczność obiektów (kapsułkowanie)
- dziedziczenie
- polimorfizm

Programowanie proceduralne vs. programowanie obiektowe

Podejście proceduralne

```
struct Punkt
{
    int x, y;
};

void Narysuj (struct Punkt P)
{
    // ciało funkcji
}
```

Podejście obiektowe

```
class Punkt
{
    int x, y;

public:
    void Narysuj ()
    {
        // ciało funkcji
    }
};
```

Klasa

- Podstawowe pojęcie w C++
- Stanowi obiektowy typ danych
- Definiuje przedmiot, pojęcie, zjawisko
- Dane charakteryzujące przedmiot
- Funkcje określające jego właściwości
- Do tworzenia klasy służą słowa kluczowe:
  class, struct i union

```
class NazwaKlasy
{
    // ciało definicji klasy
}
```

Klasa (2)

Ciało definicji klasy może zawierać:

- deklaracje danych składowych
- deklaracje funkcji składowych (metod)
- definicje funkcji składowych

Klasy można zagnieżdżać:

- wg obecnie obowiązującego standardu klasy zagnieżdżone mają charakter lokalny
- w starszych wersjach kompilatorów klasy zagnieżdżone są traktowane na równi z innymi

Sekcje definicji klasy

Składowe klasy rozmieszczone są w sekcjach definicji klasy

```
class Klasa
{
public:
    // funkcje i dane składowe publiczne
protected:
    // funkcje i dane składowe zabezpieczone
private:
    // funkcje i dane składowe prywatne
};
```

Sekcje definicji klasy

Sekcja prywatna (`private`)

- Domyślna
- Składowe prywatne są widoczne tylko w obrębie funkcji składowych danej klasy
- Składowe prywatne są stosowane gdy zmiana wartości składowej może być ryzykowna lub gdy składowa ma charakter wewnętrzny (pomocniczy) i dostęp do niej z zewnątrz nie ma sensu

Sekcja zabezpieczona (`protected`)

- Składowe zabezpieczone są widoczne tylko w obrębie funkcji składowych danej klasy i klas wyprowadzonych (pochodnych)

Sekcja publiczna (`public`)

- Dostęp do składowych publicznych jest możliwy zarówno z wnętrza jak i spoza klasy

Odwołania do składowych klasy

Notacja kropkowa (-> w przypadku dostępu przez wskaźnik)

```
class A
{
public:
    int X;
    void Metoda() {}

protected:
    int y;

private:
    int Z;
};
```

```
void main()
{
    A obiekt;
    A *wsk = new A();
    obiekt.X = 1;     // OK
// obiekt.y = 1;     // błąd
// obiekt.z = 1;     // błąd
    obiekt.Metoda(); // OK

    wsk->X = 1;     // OK
// wsk->y = 1;     // błąd
// wsk->z = 1;     // błąd
    wsk->Metoda(); // OK

    delete wsk;
}
```

Klasy: *class*, *struct* i *union*

- W C++ struktury i unie są klasami
- W klasach tworzonych słowem *class* składowe domyślnie są prywatne
- W klasach tworzonych słowem *struct* lub *union* składowe domyślnie są publicznie
- W przypadku unii istnieje szereg ograniczeń nie występujących w innych typach klas np. unie nie mogą brać udziału w dziedziczeniu

Klasy zagnieżdżone - przykład

```
class X
{
    class N1 {int n;};

public:
    class N2 {int n;};

    N1 Metoda(N2);
};

void funkcja()
{
    N1 a;     // błąd: nie ma klasy N1 w zasięgu
    N2 b;     // błąd: nie ma klasy N2 w zasięgu
    X::N1 c;  // błąd: N1 prywatne w X
    X::N2 d;  // OK
}
```

Zmienna *this*

- W każdym obiekcie dostępna jest zmienna *this* wskazująca na ten obiekt.
- Za jej pomocą można zwracać w funkcjach składowych bieżący obiekt (**this*).
- Wszystkie odwołania do składowych z wnętrza obiektu są domyślnie poprzedzane *this->*.

Zmienna *this* - przykład

```
class A
{
    int i;

public:
    void Ustaw(int x)
    {
        i = x;  // this->i = x;
    }
};
```

```
class A
{
    int i;

public:
    void Ustaw(int i)
    {
        this->i = i;
    }
};
```

Deklaracje i definicje składowych

- Ciała funkcji składowych można umieszczać w klasie lub poza klasą.
- Funkcje zdefiniowane w klasie mają domyślnie atrybut *inline*, czyli przy kompilacji ich wywołanie jest zastępowane ciałem (pod warunkiem, że ciało funkcji spełnia pewne warunki np. nie zawiera pętli).
- Funkcjom składowym definiowanym poza klasą też można nadać atrybut *inline*.

Deklaracje i definicje składowych Przykład

```
class A
{
    int x, y;

public:
    int Fun1(int z)
    {
        x = y = z;
    }
    void Fun2();
    void Fun3();
};

inline void A::Fun2()
{
    x = y;
}

void A::Fun3()
{
    for (...)
    { ... }
}
```

Składowe statyczne

- Składowe statyczne (`static`) są wspólne dla wszystkich obiektów danej klasy i istnieją nawet wtedy gdy nie istnieje żaden obiekt klasy.
- Statyczne funkcje składowe mogą odwoływać się tylko do składowych statycznych, nie widzą wskaźnika `this`, nie mogą być wirtualne.

Składowe statyczne
Przykład

```
class A
{
    static int licznik; // licznik obiektów
public:
    static int PodajIle()
    {
        return licznik;
    }
};

int A::licznik = 0;  // konieczna definicja !

int main()
{
    cout << A::PodajIle();

    return 0;
}
```

Konstruktory

- Podczas tworzenia obiektu danej klasy wywoływana jest funkcja zwana konstruktorem.
- Konstruktor jest specjalną, najczęściej przeciążoną funkcją wywoływaną zawsze gdy tworzony jest obiekt klasy.
- Deklaracja konstruktora nie może zawierać typu zwrotnego.
- Konstruktor nie jest wywoływany jawnie.
- Wybór konstruktora przy tworzeniu obiektu dokonywany jest na postawie przekazanych parametrów.
- Nazwa konstruktora musi być taka sama jak nazwa klasy.
- Konstruktory nie są dziedziczone i nie mogą być wirtualne.
- Typowo konstruktory są publiczne (jeśli konstruktor jest prywatny, to pozwala on na tworzenie obiektów tylko z poziomu metod jego klasy i funkcji zaprzyjaźnionych z klasą!) .

Szczególne typy konstruktorów

- Domyślny (bezargumentowy) `Klasa()` - tworzony automatycznie przez kompilator gdy nie zdefiniowano jawnie żadnego konstruktora.
- Kopiujący `Klasa(const Klasa&)` - używany zawsze gdy tworzona jest kopia obiektu, domyślnie tworzony jest konstruktor kopiujący, który kopiuje składową po składowej, należy go definiować gdy domyślny działałby nieprawidłowo.

Konstruktor kopiujący - Przykład

```
class String
{
private:
    char *txt;

public:
    String()
    {
        txt = new char[1];
        strcpy(txt, "");
    }
    String(char* t)
    {
        txt = new char[strlen(t)+1];
        strcpy(txt, t);
    }
    String(const String &s)
    {
        txt = new char[strlen(s.txt)+1];
        strcpy(txt, s.txt);
    }
    ~String()
    {
        delete [] txt;
    }
};

//...
    String s1("Ala");
    String s2(s1);      //kopiujący
    String s3 = s1;     //kopiujący
//...
```

Dla tej klasy domyślny konstruktor kopiujący działałby nieprawidłowo, gdyż obiekt źródłowy i jego kopia współdzieliłyby obszar pamięci (ze względu na kopiowanie wskaźnika, a nie zawartości wskazywanego obszaru pamięci)

```
String(const String &s)
{
    txt = s.txt;    // domyślny, błędny!
}
```

Destruktor

- Wywoływany niejawnie, gdy obiekt jest niszczony
- Domyślnie tworzony jest destruktor, który nic nie robi
- Należy definiować destruktor, gdy konieczne jest dokonanie porządków po obiekcie np. zwolnienie przydzielonej dynamicznie pamięci
- Nazwa destruktora to nazwa klasy poprzedzona tyldą
- Destruktor nie ma parametrów i typu zwrotnego
- Typowo publiczny

```
class Complex
{
    int re, im;

public:
    Complex (int x, int y) : re(x), im(y)
    {}

    Complex ()
    {
        re = im = 0;
    }

    Complex (const Complex &c)
    {
        re = c.re;
        im = c.im;
    }

    ~Complex ()
    {}
};
```

Lista inicjalizacyjna

- Wygodny sposób nadawania początkowych wartości składowym w konstruktorach
- Jedyny sposób inicjalizacji składowych stałych lub będących obiektami
- Jedyny sposób podania wywołania konstruktora klasy bazowej w konstruktorze klasy pochodnej

```
class Complex
{
    int re, im;

public:
    Complex (int x, int y) : re(x), im(y) {}
};
```

Konstruktory obiektów zawierających inne obiekty

- Obiekty składowe tworzone są przez ich konstruktory zanim zacznie działać konstruktor obiektu otaczającego
- Muszą korzystać z listy inicjalizacyjnej, w przeciwnym wypadku dla obiektów składowych wywołane zostaną ich domyślne bezargumentowe konstruktory
- Destruktory wołane są w odwrotnej kolejności

```
class procesor
{
    char typ[10];
    double zegar;

public:
    procesor(char *p_typ, double p_f)
        : zegar(p_f)
    {
        strcpy(typ, p_typ);
    };
};

class plyta
{
    char typ[10];
    procesor proc;

public:
    plyta(char * p_typ, char *p_proc_typ, double p_f)
        : proc(p_proc_typ, p_f)
    {
        strcpy(typ, p_typ);
    };
};

void main()
{
    plyta pl("P7H55-M LE", "Core i3", 3300);
    // ...
}
```

Obiekty automatyczne, statyczne i dynamiczne

```
class Complex
{
    int re, im;

public:
    Complex (int x, int y) : re(x), im(y)
    {}
    Complex ()
    {
        re = im = 0;
    }
    Complex (const Complex &c)
    {
        re = c.re;
        im = c.im;
    }
};
```

```
Complex c0(3, 6); // statyczny

int main()
{
    // automatyczne
    Complex c1(4, -1);
    Complex c2;
    Complex c3 = c1;  // kopiujący <=> c3(c1)

    // dynamiczne
    Complex *p1, *p2, *p3;
    p1 = new Complex(4, 7);
    p2 = new Complex();
    p3 = new Complex(*p1);

    delete p1; delete p2; delete p3;

    return 0;
}
```

Funkcje i klasy zaprzyjaźnione

- Funkcje zaprzyjaźnione z klasą to funkcje, które posiadają dostęp do składowych prywatnych i zabezpieczonych danej klasy, same nie będąc składowymi tej klasy
- O tym, że dana funkcja jest zaprzyjaźniona z daną klasą informuje deklaracja friend w ciele tej klasy
- Funkcjami zaprzyjaźnionymi mogą być metody innej klasy
- Ta sama funkcja może być zaprzyjaźniona z wieloma klasami
- W przypadku gdy wszystkie metody jednej klasy mają być zaprzyjaźnione z drugą klasą, można uczynić całą pierwszą klasę klasą zaprzyjaźnioną z drugą
- Zaprzyjaźnienie nie jest symetryczne
- Zaprzyjaźnienie nie jest przechodnie
- Zaprzyjaźnienie nie jest dziedziczone

```
class A
{
    int i, j;
    friend void pokaz(A&);
    friend void C::f1();
    friend class B;
};

class B
{
    void zerujA(A &a)
    {
        a.i = a.j = 0;
    }
};

void pokaz(A &a)
{
    cout << a.i << " " << a.j;
}
```

Tablice

- Deklaracja tablic jednowymiarowych:

```
typ_elementu nazwa[rozmiar];
```

- Deklaracja tablic dwuwymiarowych:

```
typ_elementu nazwa[rozmiar1][rozmiar2];
```

- Tablice w C/C++ są indeksowane od 0
- Nawigacja po elementach tablicy:
  - poprzez indeks tablicy
  - za pomocą arytmetyki na wskaźnikach
- Tablice mogą zawierać obiekty lub wskaźniki do obiektów

```
class Rozdzial
{
public:
    char Tytul[40];
    long int Str_p, Str_k;
};

Rozdzial spis_tresci[10];

Rozdzial *spis_tresci;
spis_tresci = new Rozdzial[4];
```

Funkcje operatorowe

- Pozwalają na przeciążenie istniejących operatorów, tak aby uzyskały znaczenie dla definiowanych przez programistę nowych typów obiektowych
- Można przeciążyć m.in.:
  - `+, -, *, /, !, =, <, >, ++, --, (), [], new, delete`
- Nie można zmienić priorytetów operatorów, np. `a+b*c ≡ a+(b*c)`
- Nie można zmienić wymaganej liczby argumentów operatorów, np.
  - `/` zawsze 2-argumentowe,
  - `!` zawsze 1-argumentowa
- Mogą być definiowane jako metody klasy lub funkcje zewnętrzne (najczęściej zaprzyjaźnione)
- Nazwa funkcji operatorowej jest konkatenacją słowa kluczowego operator i symbolu operatora np. `operator=`, `operator++, ...`
- Przynajmniej jeden argument wywołania operatora musi być typu zdefiniowanego przez użytkownika, np. `operator+(int, float)` jest niemożliwe
- Operatory są dziedziczone

Funkcje operatorowe (2)

```
class Complex
{
    int re, im;

public:
    Complex() {};
    Complex(int x, int y) : re(x), im(y) {}
    Complex& operator++()  // przedrostkowy
    { re++; return *this; }
    Complex& operator++(int i)  // przyrostkowy
    { im++; return *this; }
    Complex& operator=(const Complex &c)
    {
        re = c.re;
        im = c.im;
        return *this;
    }

    operator int() { return re; }

    // operator wywołania funkcji
    void operator()(int x) { re += x; }
    void operator()(int x, int y)
    { re += x; im += y; }

    friend Complex operator+(Complex c1, Complex c2);
    friend ostream& operator<<(ostream &o, Complex c);
};
```

```
Complex operator+(Complex c1, Complex c2)
{
    return Complex(c1.re + c2.re, c1.im + c2.im);
}

ostream& operator<<(ostream &o, Complex c)
{
    o << "[" << c.re << ";" << c.im << "]";
    return o;
}

void main()
{
    Complex z1 = Complex (1, 2);
    Complex z2 = Complex (2, 3);
    Complex z3;
    z3 = z1 + z2;
    cout << z3;    // [3;5]
}
```

Funkcje operatorowe (3)

```
class String
{
    char *txt;
public:
    // konstruktory itp. ...

    char& operator[](int idx)
    {
        return *(txt + idx);
    }
};
```

Operator indeksowania z referencją zwraca l-wartość.
Bez referencji zwracałby znak bez możliwości jego zmiany.

Dziedziczenie

- Pozwala na współdzielenie kodu przez klasy
- Zapewnia wspólny interfejs dla kilku klas
- Ułatwia wyrażanie wspólnych cech klas

```
class NazwaKlasy : [public|private|protected]
KlasaBazowa1, ...
{
    // składowe klasy
};
```

- Klasa może posiadać jedną lub więcej klas bazowych
- Dla każdej z klas bazowych specyfikuje się tryb dziedziczenia od którego zależy dostępność odziedziczonych składowych
- W przypadku klas potomnych tworzonych za pomocą słowa `class` domyślnym trybem dziedziczenia jest `private`, w przypadku słowa `struct` - `public`

Dziedziczenie (2)

- Dostępność składowych w klasie pochodnej w zależności od trybu dziedziczenia:

Tryb dziedziczenia

	private	protected	public
private	-	private	private
protected	-	protected	protected
public	-	protected	public

Składowe

- Dostęp do klasy bazowej w zależności od trybu dziedziczenia:

```
class Y1 : public X {};
```

- każda funkcja może (niejawnie) dokonać konwersji `Y1*` do `X*` tam gdzie jest to potrzebne

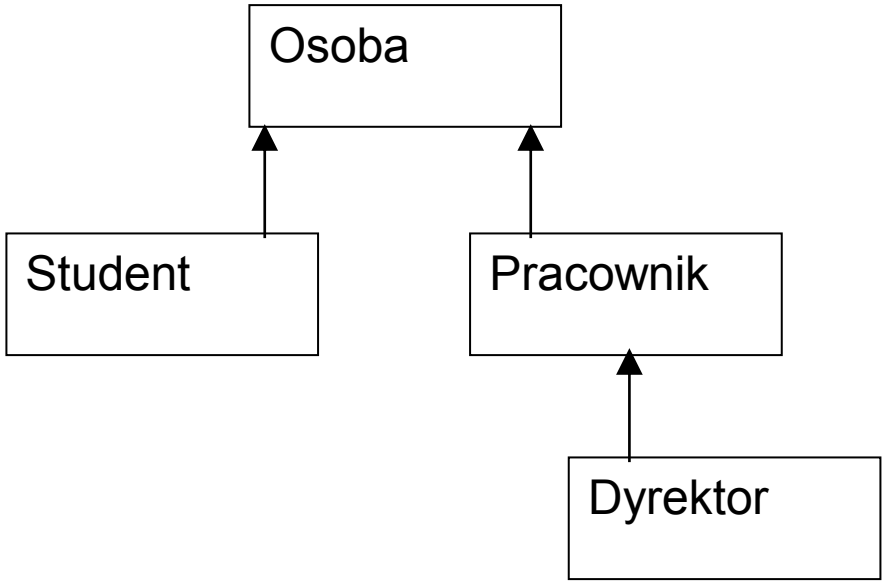
```
class Y2 : protected X {};
```

- tylko metody i funkcje zaprzyjaźnione `Y2` oraz metody i funkcje zaprzyjaźnione klas pochodnych klasy `Y2` mogą (niejawnie) dokonać konwersji `Y1*` do `X*` tam gdzie jest to potrzebne

```
class Y3 : private X {};
```

- tylko metody i funkcje zaprzyjaźnione `Y3` mogą (niejawnie) dokonać konwersji `Y1*` do `X*` tam gdzie jest to potrzebne

Hierarchie dziedziczenia




```
class Osoba
{
public:
    String Imie, Nazwisko;
};

class Student : public Osoba
{
public:
    float Srednia;
};
```

Konstruktory klas pochodnych

- Wywołanie konstruktora klasy pochodnej zawsze jest poprzedzone wywołaniem konstruktora klasy bazowej (w celu inicjalizacji odziedziczonych składowych)
- W przypadku braku jawnej specyfikacji wywołania konstruktora klasy bazowej, wywołany zostanie konstruktor bezargumentowy (w przypadku jego braku wystąpi błąd kompilacji)
- Konstruktor klasy bazowej można wywołać z konstruktora klasy pochodnej tylko za pomocą listy inicjalizacyjnej

```
class Osoba
{
private:
    String Imie, Nazwisko;

public:
    Osoba( char *i, char *n )
        : Imie( i ), Nazwisko( n ) {}
};

class Pracownik : public Osoba
{
private:
    int Staz;

public:
    Pracownik( char *i, char *n, int s )
        : Osoba( i, n ), Staz( s ) {}
};
```

Destruktory wołane są w odwrotnej kolejności niż konstruktory

Dziedziczenie wielobazowe

```
class jablko
{
public:
    char gatunek[20];
    jablko(char *p_gatunek)
    {
        strcpy(gatunek, p_gatunek);
    };
};

class placek
{
public:
    float waga;
    placek(float p_waga): waga(p_waga){};
};

class jablecznik: public jablko, public placek
{
public:
    float cena;
    jablecznik(float p_cena, char *p_gatunek, float p_waga)
        :cena(p_cena), jablko(p_gatunek), placek(p_waga){};
};

void main()
{
    jablecznik j1(45, "jonagold", 1.2);
    cout << j1.cena << " " << j1.gatunek << " " << j1.waga << endl;
}
```

Dziedziczenie wielobazowe (2)

Odwołanie się do składników w nadklasie

```
jablko::gatunek
placek::waga
```

```
class jablko
{
public:
    char gatunek[20];
    float waga;
};

class placek
{
public:
    float waga;
};

class jablecznik: public jablko, public placek
{
public:
    float cena;
};

void main()
{
    jablecznik j1(45, "jonagold", 1.2, 1.9);
    cout << j1.cena << " " << j1.gatunek << " " << j1.waga << endl;
}
```

```
[C++ Error] ... Member is ambiguous: 'jablko::waga' and 'placek::waga,

j1.jablko::waga   // brak niejednoznaczności
```

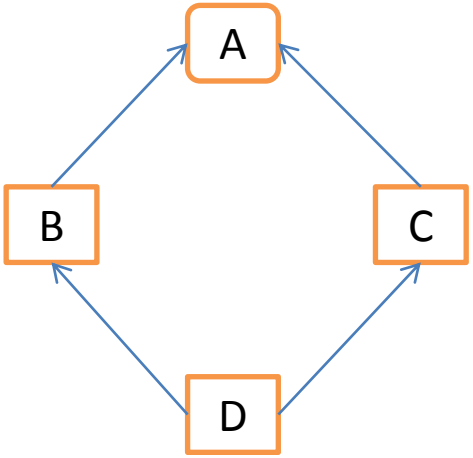
Zadanie

Korzystając z metody klasy jabłecznik zlikwidować niejednoznaczność odwołania do składnika waga.

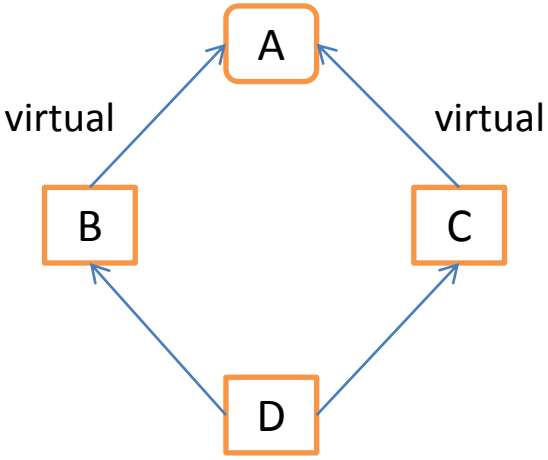
```
class jablecznik: public jablko, public placek
{
public:
    float cena;
    jablecznik(float p_cena, char *p_gatunek,float p_waga1, float p_waga2)
        :cena(p_cena),jablko(p_gatunek, p_waga1),placek(p_waga2){};
    float waga(){
        return this->jablko::waga + this->placek::waga;
    };
};
```

Dziedziczenie wielobazowe może również doprowadzić do odziedziczenia składowych pewnej klasy bazowej wielokrotnie (kilkoma drogami).
Rozwiązaniem powyższego problemu może być tzw. dziedziczenie wirtualne.

Dziedziczenie wirtualne

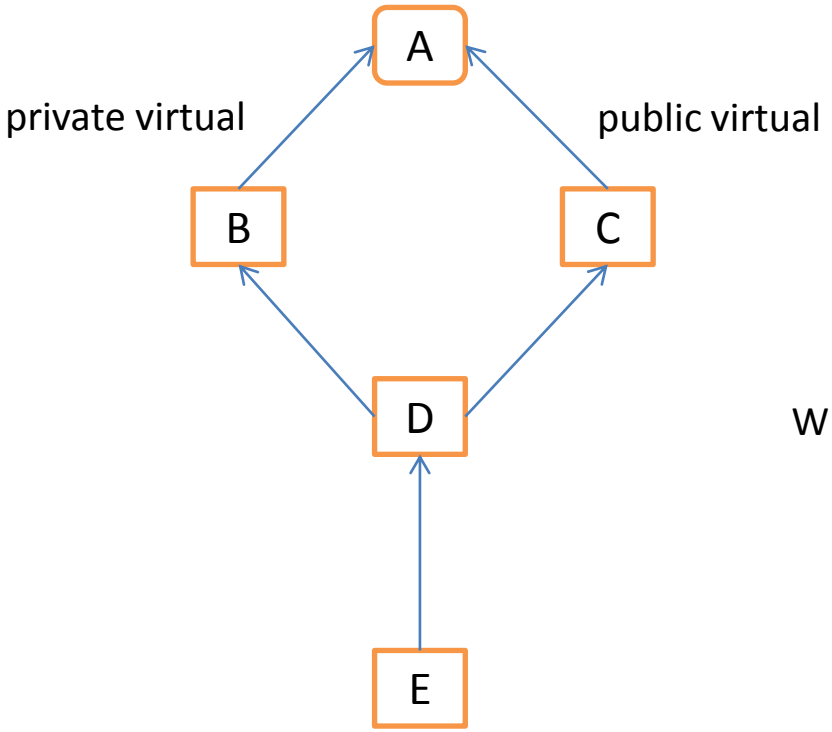


Obiekty klasy D posiadają zduplikowane składniki pochodzące z klasy A (odziedziczone dwukrotnie) -> niejednoznaczność odwołań



Zestaw składników z klasy A w obiektach klasy D występuje tylko raz

Dziedziczenie wirtualne (2)



W klasie D dziedziczenie a A jest ***public***

Konstruowanie obiektów z hierarchiami wirtualnymi

Najpierw uruchamiane są konstruktory klas wirtualnych, w kolejności od lewej do prawej na liście dziedziczenia

```
class A
{
public:
    int a;
    A(){ a = 0;};
    A(int p): a(p){};
};

class B: public virtual A
{
public:
    int b;
    B(){ b = 0;};
    B(int p1, int p2): A(p1), b(p2){};
};

class C: public virtual A
{
public:
    int c;
    C(){ c = 0;};
    C(int p1, int p2): A(p1), c(p2){};
};
```

```
class D: public B, public C
{
public:
    int d;
    D(){ d = 0;};
    D(int p1, int p2, int p3)
        :A(p1), B(p1, p2), d(p3){};
};

class E: public D
{
public:
    int e;
    E(){ e = 0;};
    E(int p1, int p2, int p3, int p4, int p5)
        :A(p1), D(p2, p3, p4), e(p5){};
};

void main()
{
    A a1(10);
    E e1;
    e1 = E();
    E e2(1, 2, 3, 4, 5);
}
```

Zadanie: Określić wartości składowych powyższych klas dla obiektów e1 i e2.

Zgodność obiektów, referencji i wskaźników

- Wszędzie tam gdzie spodziewany jest obiekt publicznej klasy bazowej może wystąpić obiekt klasy pochodnej (przy dziedziczeniu publicznym)
- Automatyczne konwersje dokonywane są dla obiektów, referencji i wskaźników

```
class A
{
public:
    int a;
};

class B : public A
{
public:
    int b;
};

void main()
{
    A a1;
    A *a2 = new B();
    B b1;
    A &a3 = b1;

    delete a2;
}
```

Zgodność obiektów, referencji i wskaźników (2)

```
class osoba
{
public:
    void przedstaw()
    { cout << "osoba" << endl; };
};

class student: public osoba
{
public:
    void przedstaw()
    { cout << "student zaoczny" << endl; };
};

class pracownik: public osoba
{
public:
    void przedstaw()
    { cout << "pracownik II" << endl; };
};

class dyrektor: public pracownik
{
public:
    void przedstaw()
    { cout << "dyrektor firmy" << endl; };
};
```

```
void main()
{
    osoba o1;
    student s1;
    pracownik p1;
    dyrektor d1;
    o1.przedstaw(); //"osoba"
    s1.przedstaw(); //"student zaoczny"
    p1.przedstaw(); //"pracownik II"
    d1.przedstaw(); //"dyrektor firmy"
    osoba *wo2;
    wo2 = &o1; wo2->przedstaw(); //"osoba"
    wo2 = &s1; wo2->przedstaw(); //"osoba"
    wo2 = &p1; wo2->przedstaw(); //"osoba"
    wo2 = &d1; wo2->przedstaw(); //"osoba"
}
```

Polimorfizm

- Polimorfizm pozwala na zapewnienie jednolitego interfejsu do obiektów klas z gałęzi dziedziczenia z zapewnieniem wywoływania metod specyficznych dla konkretnego obiektu
- Polimorfizm działa w oparciu o funkcje wirtualne
- Jeśli funkcja jest wirtualna, to przy jej wywołaniu poprzez wskaźnik na obiekt, brany jest pod uwagę faktyczny typ obiektu a nie typ wskaźnika (późne wiązanie metod - na etapie uruchomienia, a nie kompilacji)
- W celu poprawnego korzystania z polimorfizmu, konieczne może być uczynienie destruktorów wirtualnymi

Polimorfizm - przykład

```
class Osoba
{
private:
    String imie, nazwisko;

public:
    Osoba( char *i, char *n )
        :imie( i ), nazwisko( n ){}
    virtual void print( void )
    {
        cout  << imie << " " << nazwisko;
    }
    virtual ~Osoba() {}
};

class Pracownik : public Osoba
{
private:
    int staz;

public:
    Pracownik( char *i, char *n, int s )
        : Osoba( i, n ), staz( s ){}
    virtual void print( void )
    {
        Osoba::print();
        cout << " staz:" << staz << "\n";
    }
    virtual ~Pracownik() {}
};
```

```
void main()
{
    Osoba *o = new Pracownik("Jan", "Kowal", 20);
    o->print();          // Pracownik::print()
    delete o;
}
```

Polimorfizm (2)

Dwa sposoby deklarowania funkcji wirtualnej

```
class C
{
    virtual void fun ()
    { /*ciało funkcji*/ }
};
```

Jeśli w podklasie nie przedefiniujemy funkcji *fun* wówczas zostanie użyta ta z nadklasy

```
class C
{
    virtual void fun() = 0;
};
```

fun nie jest przedefiniowana w podklasie -> podklasa jest klasą abstrakcyjną -> nie można tworzyć jej wystąpień

Destruktory wirtualne

```
class osoba
{
public:
    void virtual przedstaw()
    { cout << "osoba" << endl; };
    virtual ~osoba()    // destruktor wituralny
    { cout << "destruktor osoby" << endl; }
};

class student: public osoba
{
    char* nazwisko;
public:
    student(char* p_nazw)
    {
        nazwisko  =  new char[strlen(p_nazw)+1];
        strcpy(nazwisko, p_nazw);
    }
    ~student()  // destruktor wirtualny
    {
        cout << "destruktor studenta" << endl;
        delete nazwisko;
    }
    void przedstaw()
    { cout << "student zaoczny" << endl; };
};
```

```
class pracownik: public osoba
{
    char* stanowisko;
public:
    pracownik(char* p_stan)
    {
        stanowisko  =  new char[strlen(p_stan)+1];
        strcpy(stanowisko, p_stan);
    }
    ~pracownik()  // destruktor wirtualny
    {
        cout << "destruktor pracownika" << endl;
        delete stanowisko;
    }
    void przedstaw(){cout << "pracownik II" << endl;};
};
```

```
void main()
{
    osoba* stud  =  new student("Jezierski");
    osoba* prac  =  new pracownik("Konopka");
    delete stud;
    delete prac;
    getch();
}
```

Wynik:

```
destruktor studenta
destruktor osoby
destruktor pracownika
destruktor osoby
```

Destruktory wirtualne (2)

- Gdyby destruktor nie był wirtualny

```
class osoba
{
public:
    void virtual przedstaw()
    { cout << "osoba" << endl; };
    ~osoba()
    { cout << "destruktor osoby" << endl; }
};
```

```
void main()
{
    osoba* stud  =  new student("Jezierski");
    osoba* prac  =  new pracownik("Konopka");
    delete stud;
    delete prac;
    getch();
}
```

Wynik:

```
destruktor osoby
destruktor osoby
```

Klasy abstrakcyjne

- Klasa abstrakcyjna to klasa posiadająca jedną lub więcej czystą funkcję wirtualną
- Kompilator nie pozwala na tworzenie obiektów klas abstrakcyjnych
- Klasy abstrakcyjne występują jako węzły wewnętrzne w hierarchiach dziedziczenia
- Wszystkie czysto wirtualne metody dziedziczone muszą być zdefiniowane w podklasie

```
class Figura
{
public:
    virtual void rysuj() = 0;
    virtual void obrot(int kat) = 0;
};

class Okrag : public Figura
{
    int x, y, r;
public:
    virtual void rysuj()
    { /* cialo */ }
    virtual void obrot(int kat)
    { /* cialo */ }
    // ...
};

class Trojkat : public Figura
{
    // ... rysuj(), obrot(int), ...
};
```

Wzorce (szablony) funkcji

Chcemy zdefiniować funkcję zwracającą większa liczbę z dwóch podanych liczb typu `int`

```
int max(int x, int y)
{
    if (x > y) return x;
    else return y;
}
```

Dla porównywania liczb typu `float` należy zdefiniować odrębną funkcję

```
float max(float x, float y)
{
    return (x > y) ? x : y;
}
```

Można problem wyznaczania maksimum rozwiązać „zbiorczo” za pomocą wzorca

```
template <class typ>
typ max(typ x, typ y)
{
    return (x > y) ? x : y;
}
```

```
void main()
{
    int x = 2, y = 4;
    float a = 1.22, b = 4.55;
    cout << max(x, y) << endl;
    cout << max(b, a) << endl;
    getch();
}
```

Kompilator generuje funkcję wzorcową na podstawie typów argumentów wejściowych tej funkcji

Wzorce (szablony) funkcji (2)

Funkcja zamieniająca wartości dwóch liczby typu int -> wersja z wykorzystaniem wskaźników.

```
void zamien (int* a, int* b)
{
    int pomoc;
    pomoc = *b;
    *b = *a;
    *a = pomoc;
}
```

Wzorzec generujący funkcję zamieniającą dwie liczby dowolnego (ale identycznego) typu.

```
template <class typ_liczb>
void zamien (typ_liczb* a, typ_liczb* b)
{
    typ_liczb pomoc;
    pomoc = *b;
    *b = *a;
    *a = pomoc;
}
```

Wzorce (szablony) klas

- Nazwa wzorca klasy musi być unikalna w całym programie
- Wzorzec klasy musi być zdefiniowany w zakresie globalnym
- Nie można zagnieżdżać definicji wzorców

```
template <class typ>
class nazwa_klasy
{
    typ zmienna1, zmienna2;
    nazwa_klasy (...);  //konstruktor
    ~nazwa_klasy (...); //destruktor
    typ metoda1 (...);
    void metoda2(typ zmienna3);
    ...
};
```

Zadanie

- Zdefiniować wzorzec klasy o nazwie tablica umożliwiającej przechowywanie tablicy liczb dowolnego typu i tablicy znakowych. Rozmiar tablicy jest podawany w momencie tworzenia obiektu tej klasy. Każda tablica przechowuje również swój zadany rozmiar. Zdefiniować również konstruktor, destruktor i metodę wyświetlającą zawartość tablicy.

```
template <class typ>
class tablica
{
    typ* dane;
    int rozmiar;
public:
    tablica(int p_rozmiar) : rozmiar(p_rozmiar)
    { dane = new typ [p_rozmiar]; };
    ~tablica()
    { delete [] dane; };
    void wyswietl()
    {
        for (int i=0; i < rozmiar; i++)
            cout << "dane[" << i << "]=" << *(dane+i) << endl;
    };
};
```

```
void main()
{
    tablica<int>* tI = new tablica<int>(4);
    tablica<char>* tC = new tablica<char>(6);
    tablica<int> tI1(5);  //obiekt statyczny
    tablica<char> tC1(7); //obiekt statyczny
    tI->wyswietl();
    tC->wyswietl();
    tI1.wyswietl();
    tC1.wyswietl();

    delete tI;
    delete tC;
}
```

Wzorce (szablony) klas (2)

Definiowanie funkcji składowych wzorca klasy

- wewnątrz definicji wzorca
- na zewnątrz definicji wzorca

Funkcje takie definiuje się jak wzorzec funkcji

```
template <class typ>
class tablica
{
    typ* dane;
    int rozmiar;
public:
    tablica(int p_rozmiar);
    ~tablica();
    void wyswietl();
};
```

```
template <class typ>
tablica<typ>::tablica(int p_rozmiar)
{
    rozmiar = p_rozmiar;
    dane = new typ [p_rozmiar];
};

template <class typ>
tablica<typ>::~tablica()
{
    delete [] dane;
};

template <class typ>
void tablica<typ>::wyswietl()
{
    for (int i=0; i < rozmiar; i++)
        cout << "dane[" << i << "]=" << *(dane+i) << endl;
};
```

Obsługa wyjątków

```
// instrukcje programu
try
{ /*
  fragment programu, który może generować
  wyjątki, np. wywoływane funkcje biblioteczne
  */
}
catch (typ1)
{
    // obsługa wyjątku zgłoszonego obiektem typu typ1
}
catch (typ2)
{
    // obsługa wyjątku zgłoszonego obiektem typu typ2
}
catch (...)
{
    // obsługa dowolnego wyjątku
    // musi być ostatnim blokiem obsługi
}
// dalszy ciąg programu
```

Po przechwyceniu wyjątku (blok `catch`) następuje jego obsługa.

Po obsłużeniu wyjątku wykonują się instrukcje umieszczone za blokiem `catch`.

„…” w bloku `catch` oznaczają obsługę dowolnego wyjątku.

Obsługa wyjątków - przykład

```
class wyjatek
{
public:
    char w;
    wyjatek(char c):w(c){};
};

void generuj_wyjatek()
{
    int exc = random(10);
    wyjatek wyj('w');
    switch (exc)
    {
    case 0:
        cout << "wyjatek INT\n";
        throw (int) 1;
        break;
    case 2:
        cout << "wyjatek FLOAT\n";
        throw (float) 2.333;
        break;
    case 4:
        cout << "wyjatek CHAR\n";
        throw 'x';
        break;
    case 6:
        cout << "wyjatek WYJ\n";
        throw wyj;
        break;
    }
}
```

```
void main()
{
    randomize();
    cout << "sekcja try\n";
    try
    { //blok generujacy wyjatki
        generuj_wyjatek();
    }
    catch (int)
    { cout << "zgloszony INT\n"; }
    catch (float)
    { cout << "zgloszony FLOAT\n"; }
    catch (char)
    { cout << "zgloszony CHAR\n"; }
    catch (wyjatek& w1)
    { cout << "zgloszony WYJ w=" << w1.w << endl; }

    cout << "koniec obslugi wyjatkow\n";
    getch();
}
```