

Podstawy programowania 2

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>

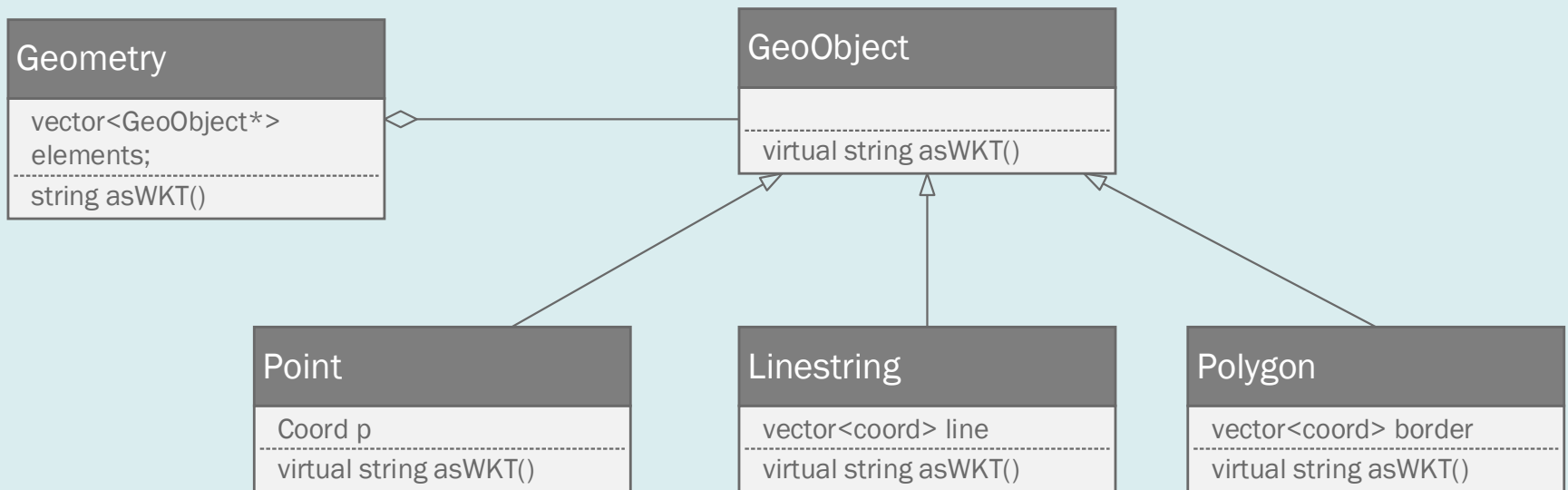
Aktualizacja: 24.03.2021

4. Funkcje wirtualne

Polimorfizm

Konstruując hierarchie klas bardzo często definiujemy metody, które mają **taki sam interfejs** (nazwę, parametry i typ zwracanych wartości) ale mają **różną implementację**, zależną od typu obiektu.

Takie funkcje nazywamy **polimorficznymi**.



Klasami potomnymi **GeoObject** są **Point**, **Linestring** i **Polygon**. Każda z klas implementuje funkcję `asWKT()`, ale w różny sposób. **Jest to funkcja polimorficzna.**
Geometry zawiera obiekty klas potomnych **GeoObject**

Point

Point

Coord p

virtual string asWKT()

- Dane – to para współrzędnych
- asWKT() – formatuje dane w formacie Well-Known Text

POINT(20.114787585423528
50.07897111470772)

Mapa

Wicket is a lightweight Javascript library that reads and writes Well-Known Text (WKT) strings. It can also be extended to parse and to create geometric objects from various mapping frameworks, such as Leaflet, the ESRI ArcGIS JavaScript API, and the Google Maps API.

POINT(20.114787585423528
50.07897111470772)

Format for URLs

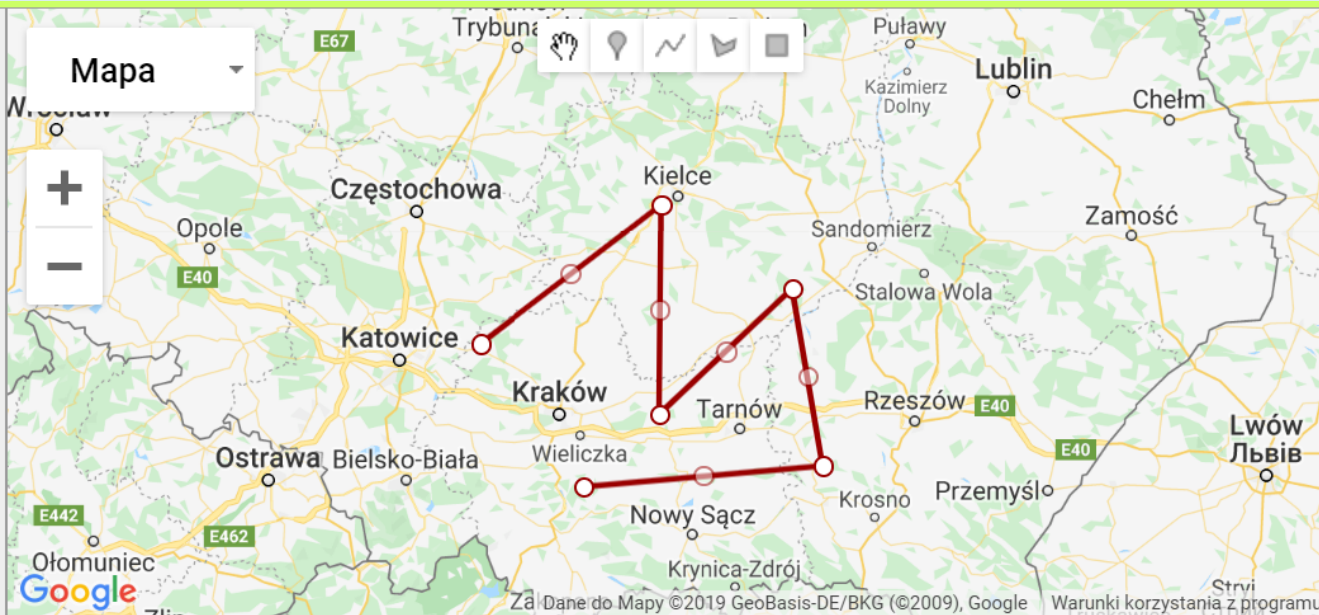
Za Dane do Mapy ©2019 GeoBasis-DE/BKG ©2009, Google Warunki korzystania z programu

Linestring

Linestring

```
vector<coord> line  
-----  
virtual string asWKT()
```

Linestring to ciąg punktów tworzących linię łamaną



Wicket is a lightweight Javascript library that reads and writes **Well-Known Text (WKT)** strings. It can also be extended to parse and to create geometric objects from various mapping frameworks, such as **Leaflet**, the ESRI ArcGIS JavaScript API, and the Google Maps API.

```
LINSTRING(19.488566882298528  
50.325095827083295,20.532268054173528  
50.83436351970718,20.521281726048528  
50.06486850297297,21.290324694798528  
50.52806702224989,21.466105944798528  
49.87407715763925,20.081828601048528  
49.7961304385344)
```

Format for URLs

Clear Map

Map It!

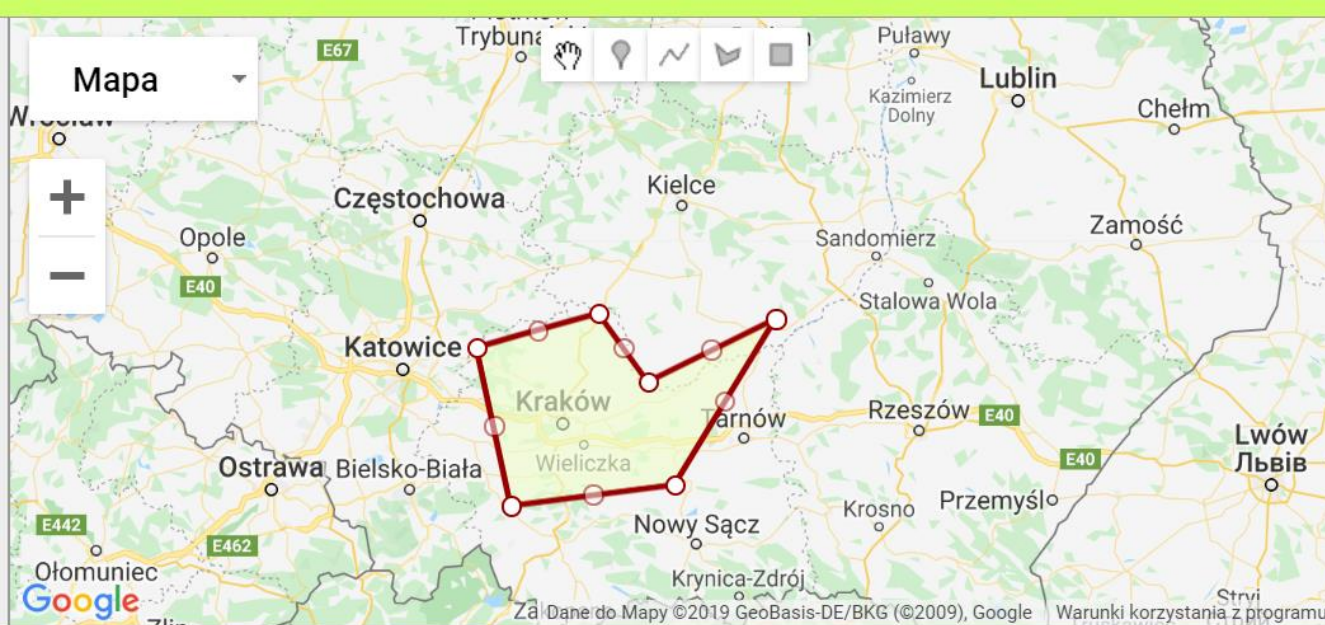
Polygon

Polygon

vector<coord> border

virtual string asWKT()

Polygon to wielobok.



Wicket is a lightweight Javascript library that reads and writes **Well-Known Text (WKT)** strings. It can also be extended to parse and to create geometric objects from various mapping frameworks, such as **Leaflet**, the **ESRI ArcGIS JavaScript API**, and the **Google Maps API**.

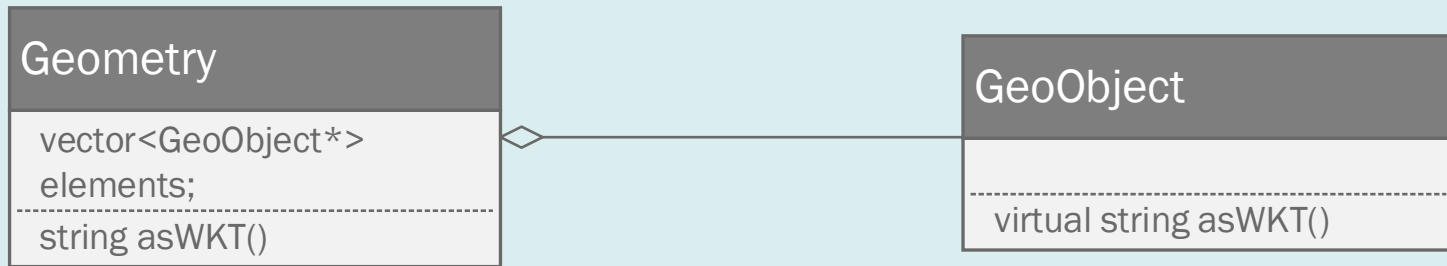
```
POLYGON((19.444621569798528  
50.34613319838949,19.642375476048528  
49.76065857218804,20.587199694798528  
49.83866240567945,21.169475085423528  
50.45118035404765,20.433391101048528  
50.21976919878196,20.147746569798528  
50.47216185412133,19.444621569798528  
50.34613319838949))
```

Format for URLs

Clear Map

Map It!

Geometry



- **GeoObject** jest sztucznym korzeniem hierarchii wprowadzonym po to aby zapewnić jednolity interfejs. Jego funkcja `asWKT()` jest pusta.
- **Geometry** zawiera tablicę (`vector` z biblioteki standardowej) wskaźników typu `GeoObject*`. Zakładamy, że te wskaźniki wskazują jednak rzeczywiste elementy typu `Point`, `Linestring` lub `Polygon`.

Geometry

```
class Geometry{
public:
    vector<GeoObject*> elements;
    string asWKT();
};

string Geometry::asWKT(){
    string r;
    for(int i=0;i<elements.size();i++){
        r.append(elements[i]->asWKT());
    }
    r.append(" ");
    return r;
}
```

Aby utworzyć tekstową reprezentacją WKT funkcja `Geometry::asWKT()` iteruje przez wszystkie wskaźniki w tablicy `elements` i woła dla wskazywanych obiektów funkcję `asWKT()`.

W zależności od wskazywanego obiektu będzie wywołana polimorficzna funkcja, która zwróci tekst POINT, LINESTRING lub POLYGON.

Funkcje polimorficzne

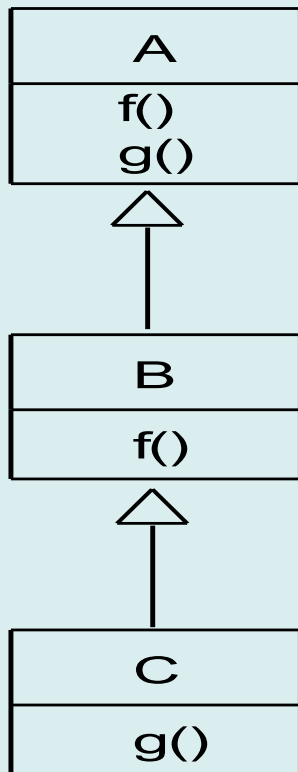
Aby osiągnąć efekt polimorficznego wywołania funkcji:

- Funkcja musi być zadeklarowana w klasie bazowej jako wirtualna
- Musi być wywołana za pośrednictwem wskaźnika lub referencji

W C++ występują trzy rodzaje metod:

1. **Statyczne** – nie są polimorficzne
2. Zadeklarowane **bez** słowa kluczowego **virtual** – nie są polimorficzne
3. Zadeklarowane **z użyciem virtual** – zachowują się polimorficznie, jeżeli są wołane za pośrednictwem wskaźnika lub referencji

Dziedziczenie metod

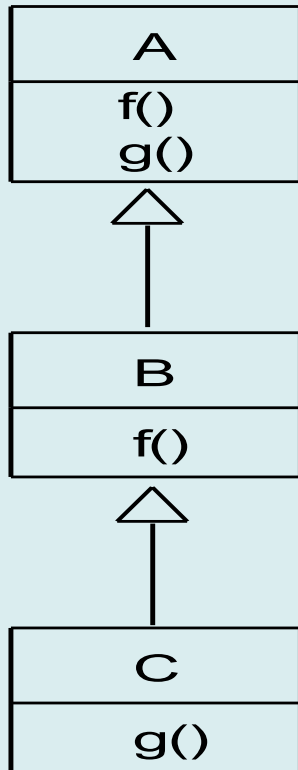


```
class A
{
public:
    A(){}
    virtual void f(){
        printf("A::f");
    }
    virtual void g(){}
};
```

```
class B : public A
{
public:
    B():A(){}
    void f(){
        printf("B::f ");
    }
};
```

```
class C : public B
{
public:
    C():B(){}
    void g(){
        printf("C::g ");
    }
};
```

Dziedziczenie metod



```
void call_g(A&a){
    a.g();
}

void call_f(A&a){
    a.f();
}

int main(){
    A a; B b; C c;
    call_f(a); // A::f()
    call_f(b); // B::f()
    call_f(c); // B::f()
    call_g(a); // A::g()
    call_g(b); // A::g()
    call_g(c); // C::g()
}
```

Wywołanie poprzez referencję typu bazowego A. Jeżeli parametrem będzie obiekt klasy B lub C nastąpi przekształcenie w referencję typu A.

Wywołana zostanie, funkcja która została zdefiniowana w danej klasie lub odziedziczona po klasie bazowej.

Dziedziczenie metod

- W przypadku zwykłych funkcji kompilator jest w stanie określić, która funkcja zostanie wywołana. Wywołanie zostaje powiązane w trakcie kompilacji (**statycznie**) z kodem funkcji. Kryterium jest sposób dostępu – obiekt, referencja lub wskaźnik określonego typu.
- W przypadku funkcji wirtualnych wybór funkcji do wykonania dokonywany jest **dynamicznie** w trakcie działania programu na podstawie typu rzeczywistego obiektu wskazywanego przez wskaźnik lub referencję.

Wskaźniki do funkcji - przypomnienie

Wskaźniki do funkcji (1)

- Po skompilowaniu każdej funkcji przydziela się pewien obszar w pamięci. Podczas wywołania funkcji – po przeprowadzeniu niezbędnych inicjalizacji – program dokonuje skoku do instrukcji mieszczącej się pod adresem początkowym bloku kodu.
- Adres tego obszaru może zostać pobrany i wykorzystany do wywołania funkcji.
- Wskaźnik do funkcji jest zmienną, która zawiera adres funkcji. Posługując się wskaźnikiem można tę funkcję wywołać.
- Typową praktyką przy projektowaniu bibliotek w C/C++ jest możliwość przekazania wskaźnika do funkcji, która, na przykład, będzie odpowiedzialna za: wyświetlanie pewnych informacji, porównywanie elementów, zapis i odczyt danych.

Wskaźniki do funkcji (2)

- Kompilator języków C/C++ zwraca uwagę na zgodność typów. W przypadku wskaźników do funkcji typ określony jest przez typ **zwracanej wartości** i typy **argumentów**.
- Deklaracje wskaźników do funkcji jest kłopotliwa. Najlepiej posłużyć się prostym przepisem:

jeżeli funkcja jest zadeklarowana jako

```
return-type function(arg-list)
```

wówczas

```
return-type (*function-pointer) (arg-list)
```

deklaruje wskaźnik o nazwie **function-pointer** do funkcji zwracającej `return-type` i biorącej za argumenty `arg-list`.

- W przypadku bardziej złożonych definicji najlepiej przeprowadzić deklarację dwuetapową wykorzystując `typedef`:

```
typedef return-type (*fp-type) (arg-list);
```

```
fp-type function-pointer;
```

Wskaźniki do funkcji (3)

Przykład

```
void foo(int a)
{
    printf("%d", a);
}

typedef void (*VOID_INT_FP) (int);

int main()
{
    VOID_INT_FP myptr = foo;
    if(myptr) myptr (7);
    return 0;
}
```

Analogicznie, jak dla tablic, identyfikator funkcji jest niemodyfikowalnym wskaźnikiem do funkcji!

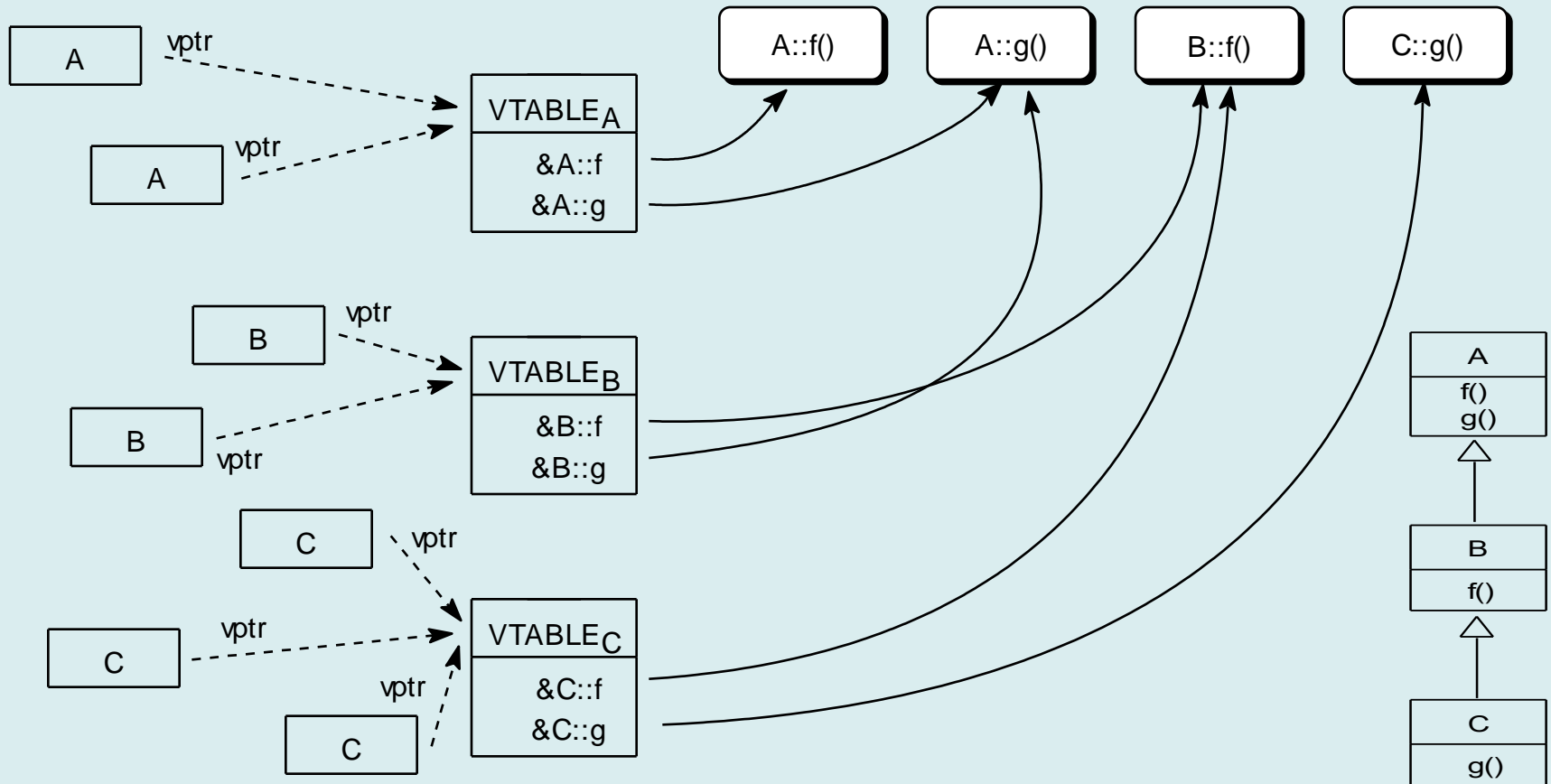
Z użyciem wskaźników do funkcji wiążą się analogiczne problemy, jak ze wskaźnikami do danych:

- Mogą mieć wartość nieokreśloną (wywołanie spowoduje zapewne błąd wykonania)
- Można testować, czy nie mają wartości zerowej i wywoływać funkcję opcjonalnie

Implementacja funkcji wirtualnych

- Dla każdej klasy zawierającej funkcje zadeklarowane jako wirtualne kompilator generuje pojedynczy obiekt będący tablicą wskaźników do funkcji zadeklarowanych jako wirtualne. Tablica ta nazywana jest **VTABLE**. Wskaźniki do funkcji są umieszczone zawsze w stałej kolejności.
- Do każdego obiektu klasy dodawane jest ukryte pole **vptr**, które jest wskaźnikiem na VTABLE odpowiedniego typu.
- Jeżeli w klasie bazowej zdefiniowane są funkcje wirtualne, wówczas dla każdej klasy potomnej generowane są analogiczne struktury danych, nawet jeżeli nie redefiniuje funkcji wirtualnych klasy bazowej.

Implementacja funkcji wirtualnych



Kompilator dodaje do wywołania funkcji wirtualnej, np.: `ptr->f()`, dodatkowy kod, który:

- odczytuje wskaźnik `vptr` wskazywanego obiektu,
- na jego podstawie odnajduje odpowiednią tablicę `VTABLE`
- wywołuje funkcję, której adres jest umieszczony w `VTABLE` pod adresem odpowiadającym funkcji `f()`.

Funkcje których nie ma

- Bardzo często projektując hierarchię klas definiuje się pewien interfejs, który powinien zostać zachowany w klasie potomnej. Dzięki temu można tworzyć funkcje, które będą działały niezależnie od rzeczywistego obiektu.
- W specyfikacji interfejsu można wyróżnić dwa rodzaje funkcji:
 - które *muszą* zostać zdefiniowane,
 - które *mogą* zostać zdefiniowane.

Funkcje których nie ma

```
class Connection
{
public:
    // musi zostać zdefiniowana
    virtual int readByte()=0;
    // może zostać zdefiniowana
    virtual void showProgress(){}
};

void readInt(Connection&c,int&target)
{
    char buf[4];
    for(int i=0;i<4;i++) buf[i]= c.readByte();
    target = *(int*)buf;
    c.showProgress();
}
```

Klasy dziedziczące po Connection

- muszą zdefiniować readByte(),
- mogą zdefiniować showProgress().

Klasy abstrakcyjne

- W powyższym przykładzie funkcja `readByte()` jest *czystą funkcją wirtualną* (ang. *pure virtual function*). Nie ma ona implementacji. W `VTABLE` klasy na jej miejscu wstawiony jest zerowy wskaźnik.
- Klasa, w której zdefiniowane są jakiegokolwiek czyste funkcje wirtualne nazywana jest klasą **abstrakcyjną**.
- Klasy abstrakcyjne definiuje się wyłącznie jako dodatkowe konstrukcje ułatwiające wykorzystanie polimorfizmu. Nie jest możliwe utworzenie obiektu klasy abstrakcyjnej. Zawsze konieczne jest zdefiniowanie klasy potomnej.

Klasy finalne

Deklarując klasę z użyciem słowa kluczowego final można zabronić dalszego dziedziczenia.

```
class Funkcja{
public:
    virtual double evaluate(double x)const=0;
};

class Sinus final:public Funkcja {
public:
    double evaluate(double x)const{
        return sin(x);
    }
};

class LepszySinus: public Sinus{
public:
    double evaluate(double x)const{
        std::cout<<"obliczam sin("<<x<<"");
        return Sinus::evaluate(x);
    }
};
```

error: cannot derive from 'final' base 'Sinus' in derived type 'LepszySinus'

19 | class LepszySinus: public Sinus{

Modyfikator override

Modyfikator override informuje kompilator, że intencją programisty jest przedefiniowanie istniejącej metody. Jeżeli takiej nie ma – sygnalizowany jest błąd.

```
class Funkcja{
public:
    virtual double evaluate(double x) const=0;
};

class ArcusSinus final :public Funkcja {
public:
    virtual double evaluate(double x) const{
        if(x<-1.0 || x>1.0) return NAN;
        return acos(x);
    }
    virtual Funkcja*getInverse()const override;
};

class Sinus final:public Funkcja {
public:
    double evaluate(double x) const{
        return sin(x);
    }
    virtual Funkcja*getInverse()const override;
};
```

```
error: 'virtual Funkcja*
ArcusSinus::getInverse() const'
marked 'override', but does not
override
```

```
38 | virtual
Funkcja*getInverse()const
override;
```

```
error: 'virtual Funkcja*
Sinus::getInverse() const'
marked 'override', but does not
override
```

```
45 | virtual
Funkcja*getInverse()const
override;
```

Przykład: żółw

```
class Turtle{
public:
    double x,y;
    double angle;
    bool isPenDown;

    Turtle(){
        x=y=0;
        angle=0;
        isPenDown=true;
    }
    void forward(double distance);
    void rotate(double angle);
    void changePenState(bool putDown);
};
```

Stan żółwia:

- x,y – współrzędne położenia
- angle – kąt obrotu w stopniach
- isPenDown – czy pisak jest opuszczony (czyli przesunięcie narysuje kreskę)

Metody

- forward() – przesunięcie do przodu
- rotate() – zmiana kąta
- changePenState() – podniesienie lub opuszczenie pisaka

Metody żółwia

```
void Turtle::forward(double distance){
    if(isPenDown){
        cout<<"<line x1=\"\"<<x<<"\" y1=\"\"<<y<<"\" ";
    }
    x+=distance*cos(angle*M_PI/180);
    y+=distance*sin(angle*M_PI/180);
    if(isPenDown) {
        cout<<"x2=\"\"<<x<<"\" y2=\"\"<<y<<"\" "
            "style=\"stroke:rgb(255,0,0);stroke-width:2\" />\n";
    }
}

void Turtle::rotate(double angle){
    this->angle+=angle;
}

void Turtle::changePenState(bool putDown){
    isPenDown = putDown;
}
```

Żółw generuje obiekty <line> w formacie SVG (Scalable Vector Graphics)

Test żółwia

```
void test(){
    cout<<"<!DOCTYPE html>\n<html>\n<body>\n";
    cout<<"<svg height=\"300\" width=\"300\">\n";
    Turtle t;
    int n= 32;
    t.changePenState(false);
    t.rotate(45);
    t.forward(200);

    bool isPenDown = true;
    t.changePenState(isPenDown);
    for(int i=0;i<n;i++){
        t.forward(10);
        t.rotate(360.0/n);
        isPenDown=!isPenDown;
        t.changePenState(isPenDown);
    }
    cout<<"</svg>\n";
    cout<<"</body>\n</html>";
}
```

Program przesuwaa żółwia na środek i rysuje okrąg linią przerywaną (pętla for)

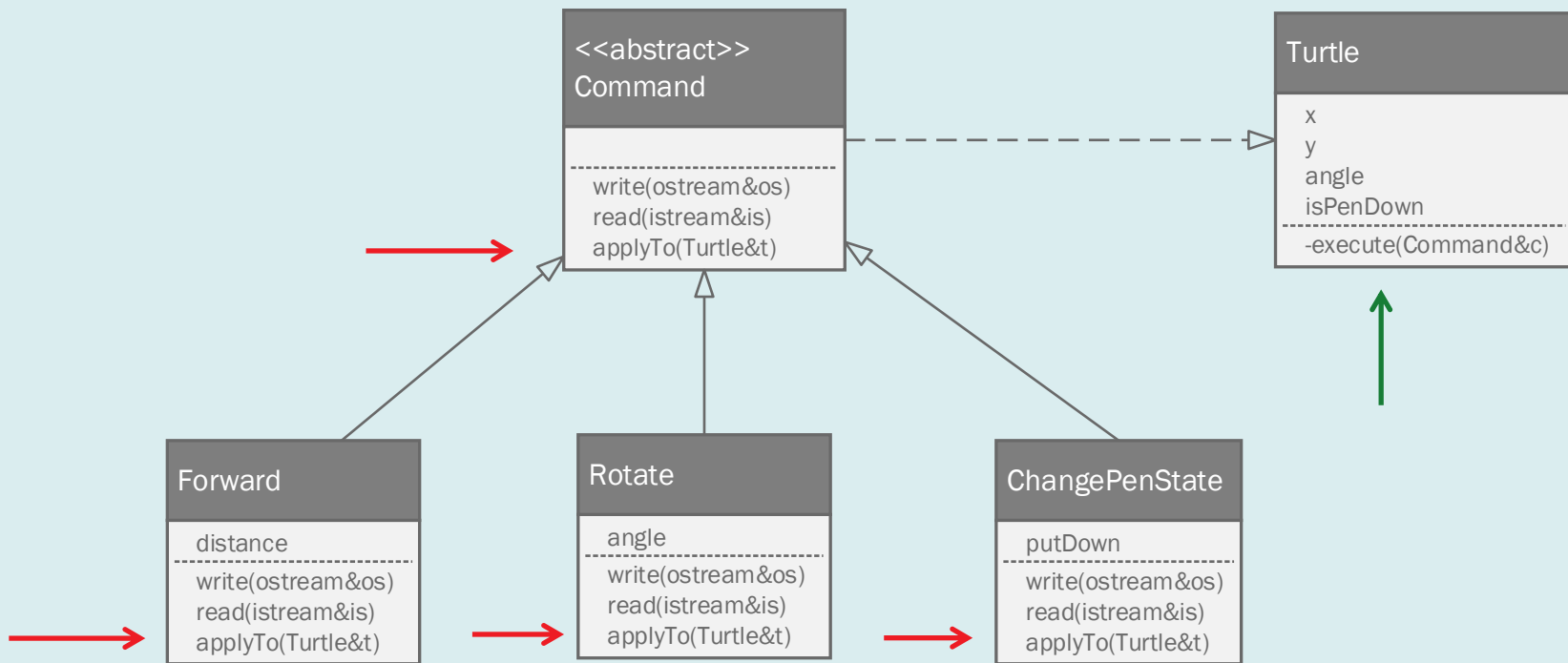
Wynik

```
<!DOCTYPE html>
<html>
<body>
<svg height="300" width="300">
<line x1="141.421" y1="141.421" x2="148.492" y2="148.492" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="154.048" y1="156.807" x2="157.875" y2="166.046" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="159.826" y1="175.854" x2="159.826" y2="185.854" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="157.875" y1="195.662" x2="154.048" y2="204.9" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="148.492" y1="213.215" x2="141.421" y2="220.909" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="133.107" y1="225.842" x2="123.868" y2="231.62" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="114.06" y1="231.62" x2="104.06" y2="231.62" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="94.2522" y1="229.669" x2="85.0134" y2="220.909" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="76.6987" y1="220.286" x2="69.6276" y2="185.854" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="64.0719" y1="204.9" x2="60.2451" y2="166.046" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="58.2942" y1="185.854" x2="58.2942" y2="148.492" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="60.2451" y1="166.046" x2="64.0719" y2="135.866" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="69.6276" y1="148.492" x2="76.6987" y2="130.088" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="85.0134" y1="135.866" x2="94.2522" y2="130.088" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="104.06" y1="130.088" x2="114.06" y2="132.039" style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="123.868" y1="132.039" x2="133.107" y2="132.039" style="stroke:rgb(255,0,0);stroke-width:2" />
</svg>
</body>
</html>
```



Refaktoryzacja

Program (ciąg poleceń) dla żółwia jest napisany w języku C++. Co należałoby zrobić, aby polecenia dla żółwia zapisywać w innym formacie?



- Zapisujemy polecenia jako obiekty. Każdy obiekt przechowuje argument polecenia.
- Polimorficzna metoda `Command::applyTo(Turtle&t)` wykonuje polecenie na żółwiu (zmienia jego stan).
- Żółw też może wykonać dowolne polecenie za pomocą `execute(Command&c)`

Klasa Command

```
class Command{
public:
    virtual void write(ostream&os)const=0;
    virtual void read(istream&is)=0;
    virtual void applyTo(Turtle&turtle)const=0;
    virtual ~Command(){}
};
```

- `write(ostream&os)` – zapisuje polecenie do strumienia wyjściowego. Strumieniem wyjściowym może być `cout` (standardowe wyjście) lub strumień skojarzony z plikiem (`ofstream`)
- `read(istream&is)` – odczytuje polecenie (a właściwie jego argument ze strumienia wejściowego: `cin` (standardowe wejście) lub `ifstream` (plik))
- `applyTo(Turtle&turtle)` – wykonuje polecenie wołając odpowiednią metodę `Turtle`
- Wirtualny destruktor `~Command()` – będzie wyjaśnione później...

Komenda Forward

```
class Forward : public Command{
    double distance;
public:
    Forward(double distance=0){
        this->distance=distance;
    }

    void write(ostream&os)const{
        os<<"FD"<<" "<<distance<<endl;
    }

    void read(istream&is){
        is>>distance;
    }

    void applyTo(Turtle&t)const{
        t.forward(distance);
    }
};
```

Zapisuje nazwę polecenia i argument. Np. wynikiem będzie: FD 100

Czyta tylko argument (distance)

Wywołuje metodę Turtle::forward()

Komenda Rotate

```
class Rotate : public Command{
    double angle;
public:
    Rotate(double angle=0){
        this->angle=angle;
    }
    void write(ostream&os)const{
        os<<"RL"<<" "<<angle<<endl;
    }
    void read(istream&is){
        is>>angle;
    }
    void applyTo(Turtle&t)const{
        t.rotate(angle);
    }
};
```

Komenda ChangePenState

```
class ChangePenState: public Command{
    bool putDown;
public:
    ChangePenState(bool putDown=true){
        this->putDown=putDown;
    }
    void write(ostream&os)const{
        os<<"PD"<<" "<<putDown<<endl;
    }
    void read(istream&is){
        is>>putDown;
    }
    void applyTo(Turtle&t)const{
        t.changePenState(putDown);
    }
};
```


Metoda Turtle::execute()

```
void Turtle::execute(const Command&c){  
    c.applyTo(*this);  
}
```

Żółw też ma możliwość wykonywania poleceń.

```
[1] Turtle t;  
[2] Rotate r(45);  
[3] t.execute(r);  
[4] t.execute(Forward(100));
```

Polecenia można przekazywać, jak w [3] – jako obiekty lub tworzyć w locie, jak [4].

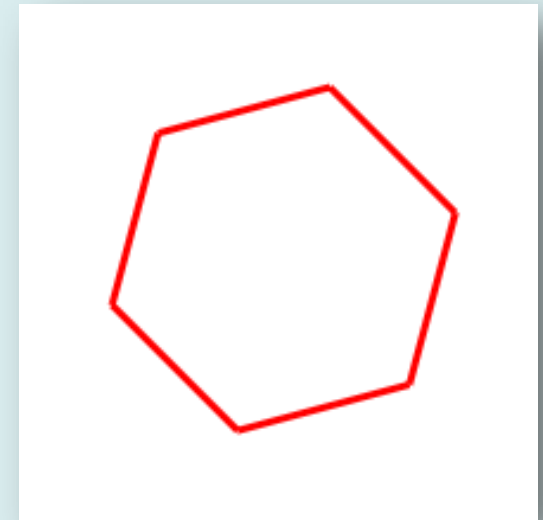
1. Obiekt, np. Rotate r(45) jest zamieniany na referencję typu Command przez pobranie adresu i **rzutowanie w górę**
2. Wołana jest **polimorficzna** metoda Command::applyTo(Turtle&t), do której przekazywana jest referencja do obiektu żółwia *this
3. Dla obiektu typu Rotate nastąpi wywołanie Rotate::applyTo(Turtle&t), a w niej t.rotate(angle)

Przykład

```
void drawHexagon(){
    int n= 6;
    Turtle t;
    t.execute(ChangePenState(false));
    t.execute(Rotate(45));
    t.execute(Forward(200));
    t.execute(ChangePenState(true));

    for(int i=0;i<n;i++){
        Forward f(50);
        Rotate r(360.0/n);
        t.execute(f);
        t.execute(r);
    }
}
```

```
<!DOCTYPE html>
<html>
<body>
<svg height="300" width="300">
<line x1="141.421" y1="141.421" x2="176.777" y2="176.777"
style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="176.777" y1="176.777" x2="163.836" y2="225.073"
style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="163.836" y1="225.073" x2="115.539" y2="238.014"
style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="115.539" y1="238.014" x2="80.1841" y2="202.659"
style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="80.1841" y1="202.659" x2="93.1251" y2="154.362"
style="stroke:rgb(255,0,0);stroke-width:2" />
<line x1="93.1251" y1="154.362" x2="141.421" y2="141.421"
style="stroke:rgb(255,0,0);stroke-width:2" />
</svg>
</body>
</html>
```



Program dla żółwia

```
int n= 16;
int p= 2;
Command*cmds [p+3*n];
cmds[0] = new Rotate(45);
cmds[1] = new Forward(100);

bool isPenDown = true;
for(int i=0;i<n;i++){
    cmds[p+3*i] = new Forward(10);
    cmds[p+3*i+1] = new Rotate(360.0/n);
    isPenDown=!isPenDown;
    cmds[p+3*i+2] = new ChangePenState(isPenDown);
}

Turtle t;
for(int i=0;i<sizeof(cmds)/sizeof(cmds[0]);i++){
    //t.execute(*cmds[i]);
    cmds[i]->applyTo(t);
}
```

Tablica wskaźników. Microsoft:
zamiast deklaracji VLA może być
konieczne obliczenie rozmiaru.

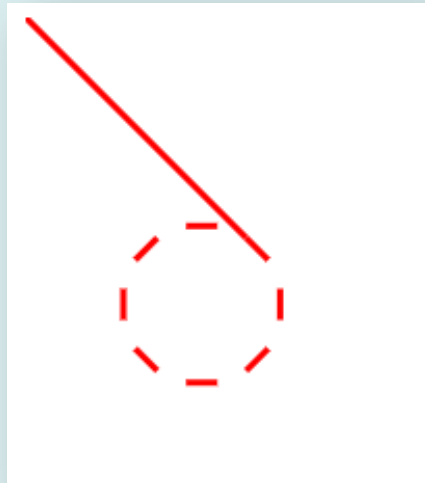
Wykonanie

Program dla żółwia

Zapis do pliku

```
ofstream of("commands.txt");  
for(int i=0;i<sizeof(cmds)/sizeof(cmds[0]);i++){  
    cmds[i]->write(of);  
}
```

```
RL 45  
FD 100  
FD 10  
RL 22.5  
PD 0  
FD 10  
RL 22.5  
PD 1  
FD 10  
RL 22.5  
PD 0  
FD 10  
...
```



Nasz program nie podnosił pisaka na początku, stąd linia od punktu (0,0) do pierwszego punktu okręgu.

Wyrażenie `sizeof(cmds)/sizeof(cmds[0])` pozwala wyznaczyć liczbę elementów tablicy (jeżeli jej deklaracja jest widoczna). Jeżeli tablica jest przekazana do funkcji zwróci $8/8=1$

Program dla żółwia

Tablica zawiera obiekty utworzone za pomocą operatora `new`, dla których pamięć jest przydzielona na stercie. Należy je usunąć...

```
for(int i=0;i<sizeof(cmds)/sizeof(cmds[0]);i++){  
    delete cmds[i];  
}
```

Odczyt poleceń z pliku

```
Command*next(istream&is){
    if(!is)return 0;
    string s;
    is>>s;
    Command *c=0;
    if(s=="FD"){
        c=new Forward();
    }
    if(s=="RL"){
        c=new Rotate();
    }
    if(s=="PD"){
        c=new ChangePenState();
    }
    if(!c)return c;
    c->read(is);
    return c;
}
```

1. Funkcja czyta symbol polecenia.
2. Następnie w zależności od symbolu tworzy obiekt odpowiedniego typu.
3. Obiekt odczytuje swoją zawartość ze strumienia:
`c->read(is)`

Funkcja zwraca 0 (NULL) jeżeli strumień zakończy się lub symbol polecenia nie zostanie rozpoznany.

Odczyt i wykonanie

```
void read_execute(){
    ifstream is("commands.txt");
    vector<Command*> cmds;
    for(;;){
        Command*c = next(is);
        if(!c)break;
        cmds.push_back(c);
    }
    Turtle t;
    for(int i=0;i<cmds.size();i++){
        t.execute(*cmds[i]);
    }
    for(int i=0;i<cmds.size();i++){
        delete cmds[i];
    }
}
```

Funkcja:

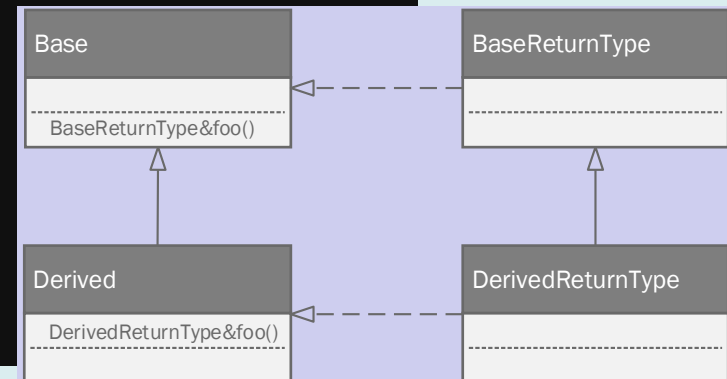
1. Otwiera strumień plikowy
2. W pętli nieskończonej odczytuje polecenia; przerywa gdy zwrócony zostanie zerowy wskaźnik.
3. Polecenia dodawane są do wektora – kontenera standardowej biblioteki C++.
4. Następnie żółw wykonuje polecenia.
5. Przy wyjściu z funkcji wektor cmds zniknie, ale nie obiekty Command na stercie. Musimy je usunąć,



Przedefiniowanie funkcji wirtualnych

- Użycie funkcji wirtualnych narzuca pewne ograniczenia. Wskaźniki umieszczane w VTABLE kolejnych klas hierarchii muszą być tych samych typów. Stąd nie jest możliwa zmiana deklaracji funkcji wirtualnych w klasach potomnych.
- Jedynym wyjątkiem jest typ zwracanej wartości. Możliwe jest zwracanie wskaźnika lub referencji do pewnej klasy potomnej typu zwracanego w bazowej funkcji wirtualnej.

```
class BaseReturnType {};  
class DerivedReturnType : public BaseReturnType {};  
  
class Base{  
    virtual BaseReturnType& foo();  
};  
class Derived : public Base{  
    DerivedReturnType& foo();  
};
```



Derived:: foo() zwraca wartość typu DerivedReturnType&. Referencja tego typu może być automatycznie zrzutowana w górę (*upcast*) do referencji typu BaseReturnType&.

Przedefiniowanie funkcji wirtualnych

Typowe zastosowanie:

- Base \equiv BaseReturnType
- Derived \equiv DerivedReturnType

```
class Object{
    virtual Object*clone()const;
};

class MyObject : public Object{
    MyObject*clone()const{
        return new MyObject(*this);
    }
};
```

Ponieważ klasa MyObject dziedziczy po Object, wskaźnik MyObject* może zostać automatycznie zrzutowany w górę – do klasy bazowej.

Funkcja clone() zwraca kopię danego obiektu, dla której pamięć przydzielono na stacku.

MyObject(*this) to wywołanie konstruktora kopiującego. Konstruktor kopiujący może być automatycznie generowany przez kompilator lub zaimplementowany jawnie

Funkcje wirtualne i dziedziczenie wielobazowe

Klasa dziedzicząca po kilku klasach bazowych może przedefiniowywać funkcje wirtualne występujące w różnych hierarchiach.

```
class A1{
public:
    virtual void f(){
        cout<<"A1::f"<<endl;}
    virtual void g(){
        cout<<"A1::g"<<endl;}
};
```

```
class A2{
public:
    virtual void g(){
        cout<<"A2::g"<<endl;}
};
```

```
class B : public A1, public A2{
public:
    void f(){
        cout<<"B::f"<<endl;}
    void g(){
        cout<<"B::g"<<endl;}
};
```

```
int main(){
    A1*b1=new B();
    b1->f(); // wypisze B::f
    b1->g(); // wypisze B::g
    A2*b2=new B();
    b2->g(); // wypisze B::g
}
```

Interfejs

- Pod pojęciem interfejsu rozumiana jest klasa abstrakcyjna, która zawiera wyłącznie czyste funkcje wirtualne.
- Taką klasą była np. Command

```
class Command{
public:
    virtual void write(ostream&os)const=0;
    virtual void read(istream&is)=0;
    virtual void applyTo(Turtle&turtle)const=0;
    virtual ~Command(){}
};
```

- W innych językach, np. Java, C# wprowadzono słowo kluczowe `interface` do oznaczenia takich klas.
- Klasa może implementować kilka interfejsów (dziedziczenie wielobazowe).
- Interfejs nie powinien mieć stanu (deklarować atrybutów)

Przykład

Deklarujemy dwa interfejsy:

- IStack – dostęp LIFO:
 - push() – umieszcza wartość na stosie
 - pop() – usuwa i zwraca ostatni element
 - empty() – czy stos jest pusty
- IRandomAccess – dostęp swobodny:
 - get(int i) – zwraca i-ty element
 - size() – zwraca liczbę elementów

```
class IStack{
public:
    virtual bool push(int i)=0;
    virtual int pop()=0;
    virtual bool empty()const=0;
    //nie zaszkodzi
    virtual ~IStack(){}
};
```

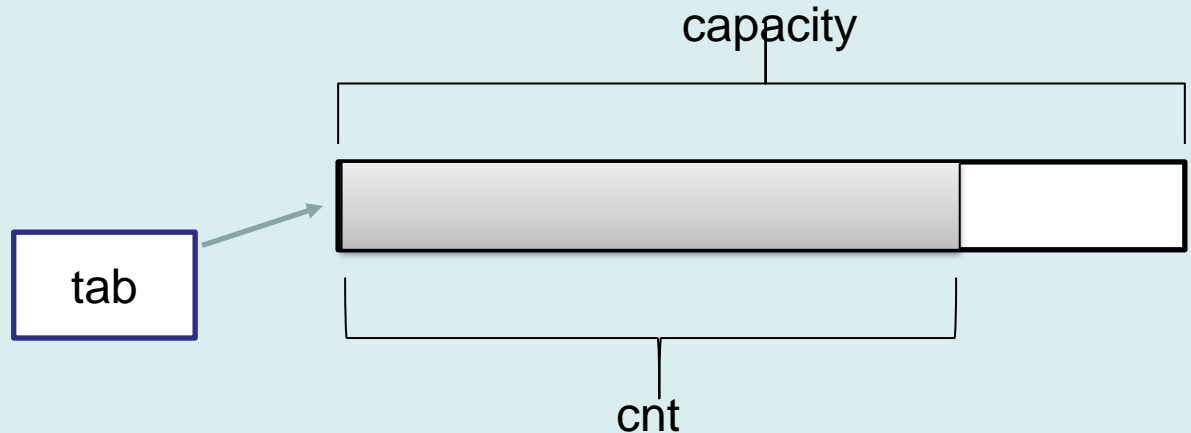
```
class IRandomAccess{
public:
    virtual int size()const=0;
    virtual int get(int i)const=0;
    //nie zaszkodzi
    virtual ~IRandomAccess(){}
};
```

Implementacja

```
class MyStack: public IStack, public IRandomAccess{
    int*tab;
    int capacity;
    int cnt;
public:
    MyStack(int _size):tab(new int[_size]),
        capacity(_size),cnt(0){}
    ~MyStack(){if (tab) delete []tab;}
    bool push(int i){
        if (cnt == capacity) return false;
        tab[cnt++]=i;
        return true;
    }
    int pop(){return tab[--cnt];}
    bool empty()const{return cnt==0;}
    int size()const{return cnt;}
    int get(int i)const{return tab[i];}
};
```

Implementacja

```
int*tab;  
int capacity;  
int cnt;
```



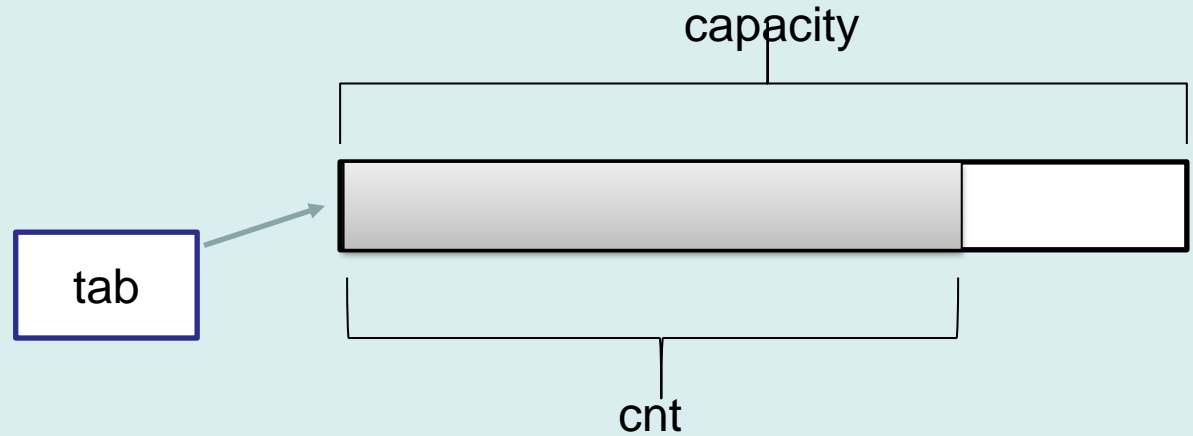
tab – wskaźnik do tablicy, dla której pamięć będzie przydzielona na stackie
capacity – pojemność (rozmiar) tablicy
cnt – licznik elementów w tablicy

```
MyStack(int _size):tab(new int[_size]),  
             capacity(_size),cnt(0){}  
  
~MyStack(){if (tab) delete []tab;}
```

Konstruktor – tworzy tablicę o zadanym rozmiarze
Destruktor – zwalnia pamięć

Implementacja

push() – dodaje element i zwiększa licznik
pop() – zmniejsza licznik i zwraca ostatni element
empty() – porównuje licznik do 0



```
bool push(int i){
    if (cnt == capacity) return false;
    tab[cnt++]=i;
    return true;
}
int pop(){return tab[--cnt];}
bool empty()const{return cnt==0;}

int size()const{return cnt;}
int get(int i)const{return tab[i];}
```

Funkcje korzystające z interfejsów:

- `fillStack()` i `emptyStack()` „widzi” tylko stos
- `iterate()` realizuje dostęp swobodny `IRandomAccess`

```
void fillStack(IStack&s){
    for(int i=0;i<10;i++)s.push(i);
}

void emptyStack(IStack&s){
    while(!s.empty())cout<<s.pop()<<" ";
}

void iterate(IRandomAccess&s){
    for(int i=0;i<s.size();i++){
        cout<<s.get(i)<<" ";
    }
}
```


Wywołanie

```
int main(){
    MyStack ms(100);
    fillStack(ms);

    iterate(ms);
    cout<<endl;
    emptyStack(ms);
}
```

0	1	2	3	4	5	6	7	8	9
9	8	7	6	5	4	3	2	1	0

- Zaletą interfejsów jest separacja abstrakcyjnej specyfikacji od konkretnej implementacji.
- Jeżeli w miejsce klasy MyStack wstawimy klasę implementującą te same interfejsy IStack i IRandomAccess, wynik nie zmieni się.

Dziedziczymy po klasie `vector<int>`

```
class StackAsVector:
    public vector<int>,
    public IStack,
    public IRandomAccess{
public:
    bool push(int i){
        push_back(i);
        return true;
    }
    int pop(){
        int r = back();
        pop_back();
        return r;
    }
    bool empty()const{return vector<int>::empty();}
    int size()const{return vector<int>::size();}
    int get(int i)const{return at(i);}
};
```

Klasa `StackAsVector` dziedziczy po kontenerze standardowej biblioteki `std::vector<int>`

Metody obu interfejsów są implementowane z wykorzystaniem funkcji kontenera.

Metody `empty()` i `size()` pokrywają się, ale nie mogą zostać odziedziczone (nie są wirtualne w kontenerze `vector`).

vector<int> jako komponent

```
class StackUsingVector:
    public IStack, public IRandomAccess{
    vector<int> tab;
public:
    bool push(int i){
        tab.push_back(i);
        return true;
    }
    int pop(){
        int r = tab.back();
        tab.pop_back();
        return r;
    }
    bool empty()const{return tab.empty();}
    int size()const{return tab.size();}
    int get(int i)const{return tab[i];}
};
```

Klasa StackUsingVector wykorzystuje kontener `std::vector<int>` jako komponent

Większość metod implementowana jest przez delegację: `empty()` woła `tab.empty()`, `size()` woła `tab.size()`, `get()` woła operator `[]`, itd.