Programowanie obiektowe

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

e-mail: pszwed@agh.edu.pl

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

Aktualizacja:16.12.2019

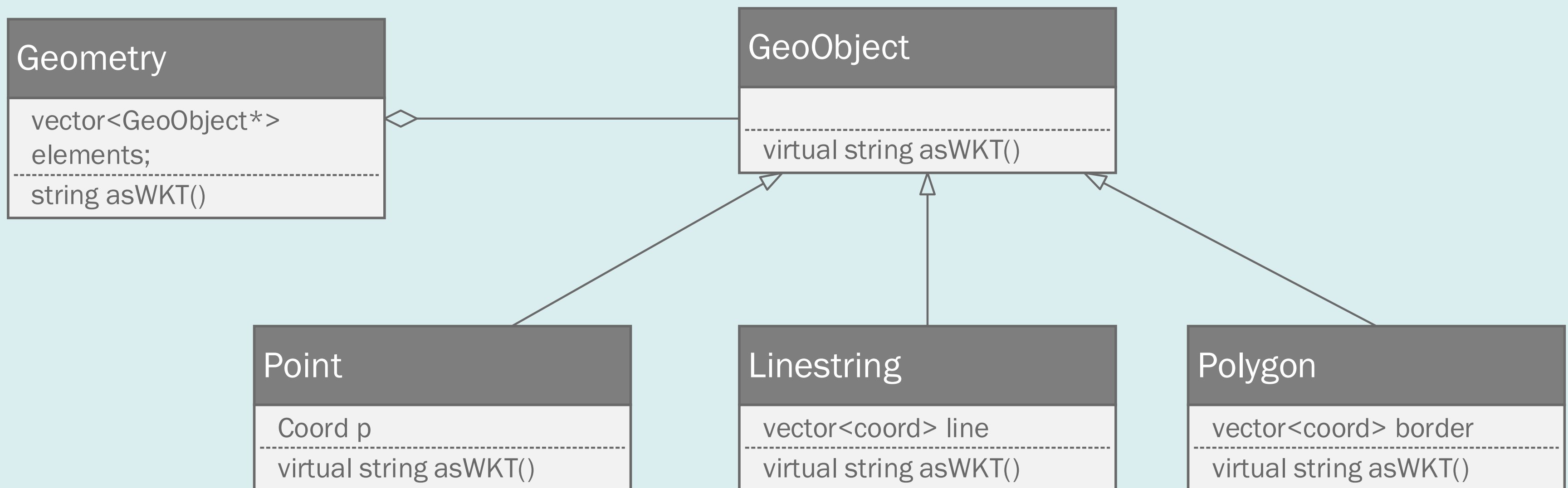
3. Funkcje wirtualne, konstruktory i destruktory

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**.



Point

Point
Coord p
virtual string asWKT()

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

POINT(20.114787585423528 50.07897111470772)

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 | Clear Map | Map It!

Mapa

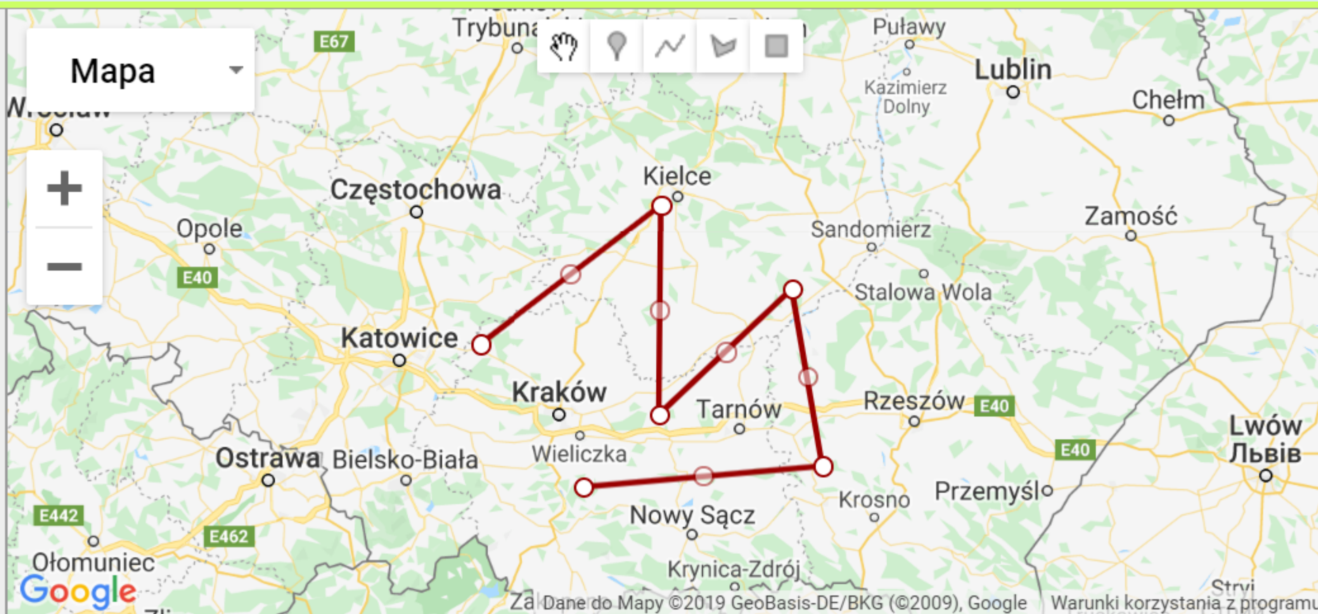
Dane do Mapy ©2019 GeoBasis-DE/BKG (©2009), Google | Warunki korzystania z programu

http://arthur-e.github.io/Wicket/sandbox-gmaps3.html

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.

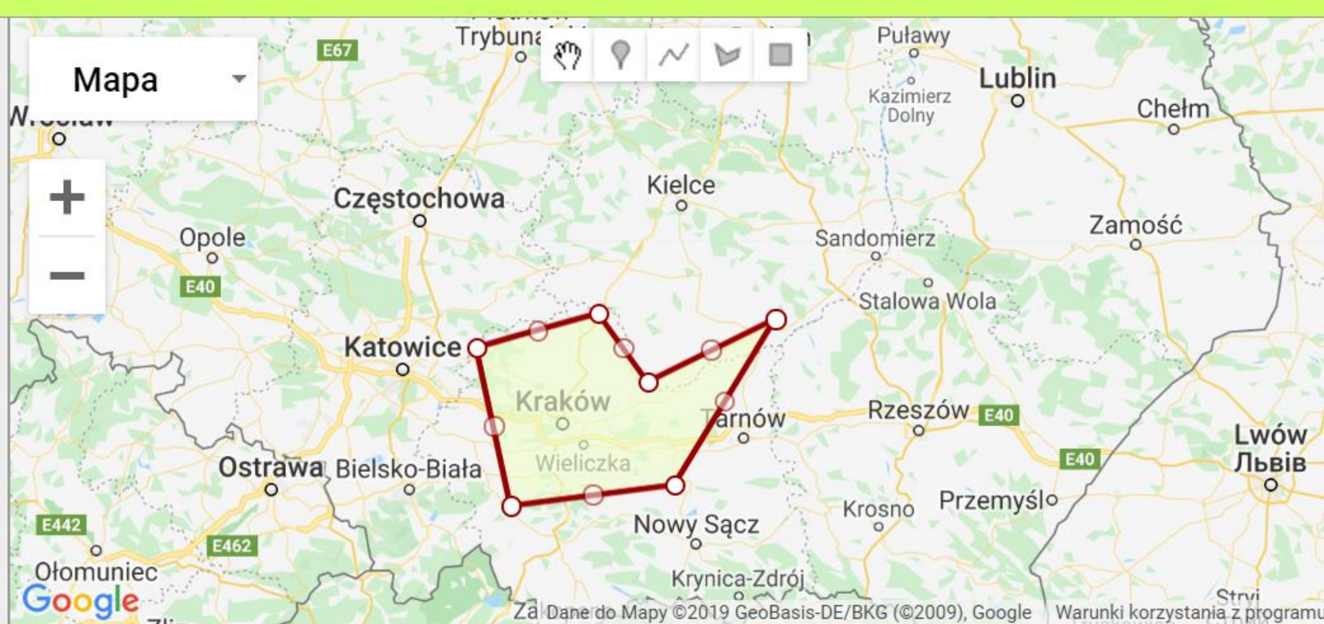
```
LINESTRING(19.488566882298528
50.325095827083295,20.532268054173528
50.83436351970718,20.521281726048528
50.06486850297297,21.290324694798528
50.52806702224989,21.466105944798528
49.87407715763925,20.081828601048528
49.7961304385344)
```

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

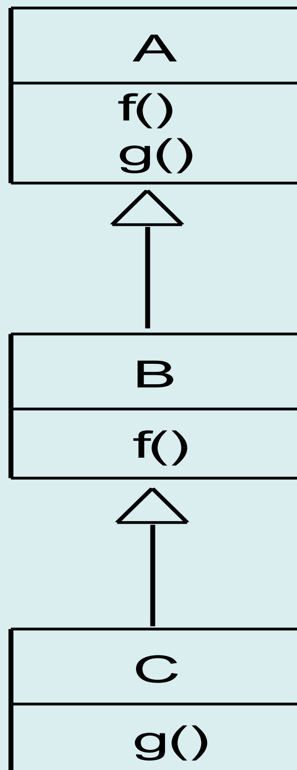
Aby osiągnąć efekt polimorficznego wywołania funkcji:

- Funkcja musi być zdeklarowana 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



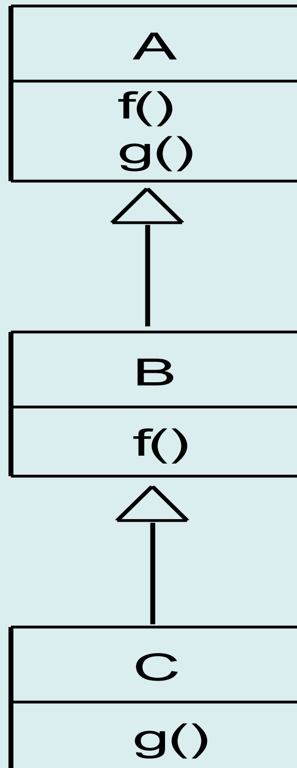

```
class A
{
public:
    A(){}
    virtual void f(){
        printf("A::f");
    }
    virtual void g(){}
};
```

```
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.

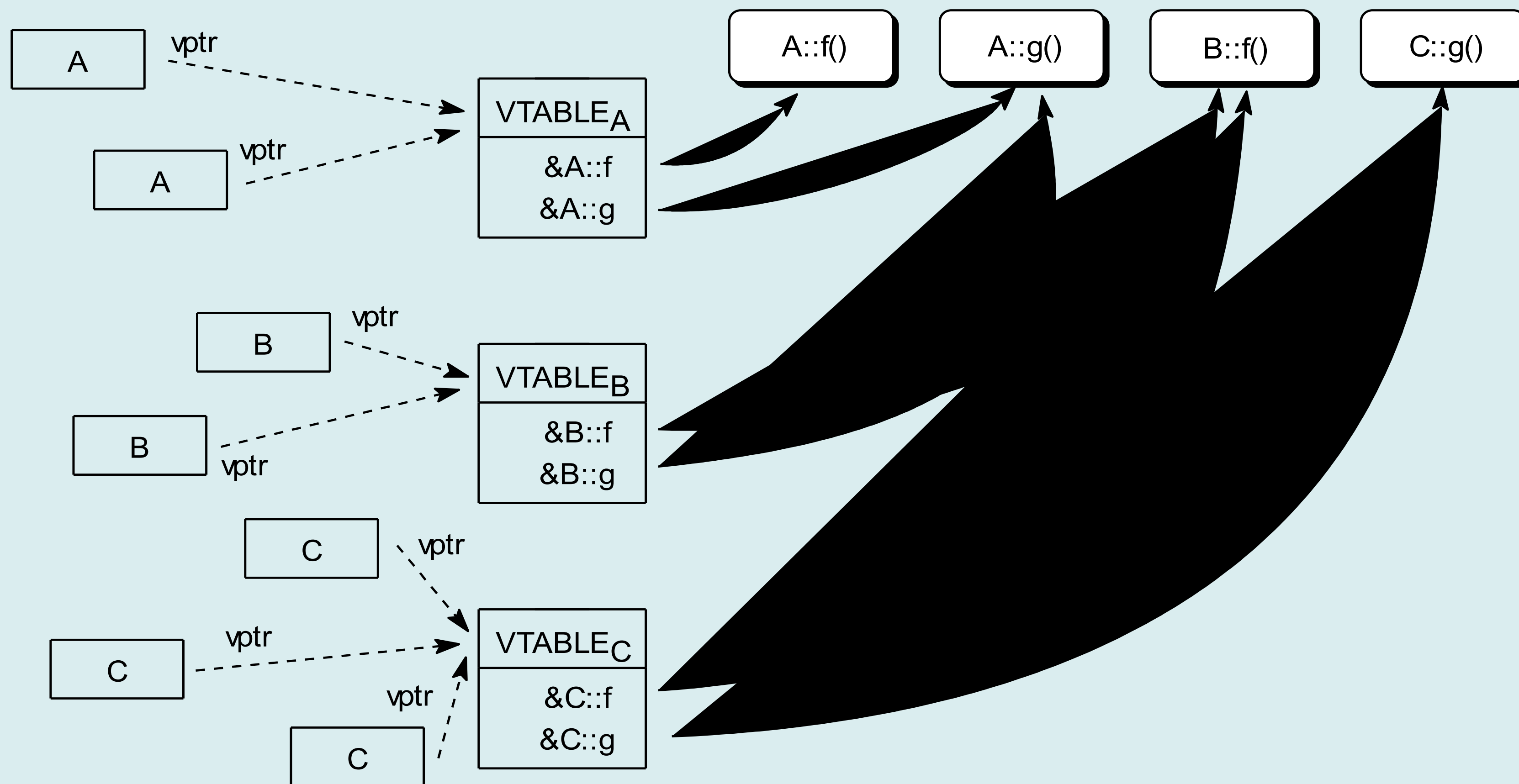
Wywołana zostanie, funkcja która została zdefiniowana
na poziomie klasy lub odziedziczona.

Dziedziczenie metod

- W przypadku zwykłych funkcji kompilator w momencie kompilacji jest w stanie określić, która funkcja zostanie wywołana. 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.

Implementacja funkcji wirtualnych



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.

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 przedefiniowane,
  - które *mogą* zostać przedefiniowane.

Funkcje których nie ma

```
class Connection
{
public:
    // musi zostać przedefiniowana
    virtual int readByte()=0;
    // może zostać przedefiniowana
    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 przedefiniować `readByte()`,
- mogą przedefiniować `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 jakiekolwiek 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.

Konstruktory i destruktory

Konstruktory i destruktory

- **Konstruktor** jest specjalną funkcją odpowiedzialną za prawidłową inicjalizację obiektu – nadanie polom danych poprawnych wartości początkowych.
- Zadaniem **destruktora** jest przeprowadzenie wymaganych operacji przy usuwaniu obiektu – np.: zwolnienie przydzielonej pamięci na stercie, zamknięcie plików, zwolnienie innych przydzielonych zasobów systemowych.

Konstruktor

- Konstruktor jest specjalną funkcją wołaną w momencie tworzenia obiektu
- Konstruktor nie może zwracać wartości.
- Możliwe jest zdefiniowanie kilku różnych konstruktorów klasy różniących się parametrami.
- Konstruktor ma taką samą nazwę, jak nazwa klasy.

Przydział pamięci dla obiektu i wywołanie konstruktora należy traktować rozłącznie. Często pamięć jest przydzielana wcześniej, natomiast moment wywołania konstruktora wynika z logiki przetwarzania

Konstruktor

Konstruktory są wołane przy tworzeniu obiektów

Globalnych	Przed rozpoczęciem wykonania programu, np.: przed wejściem do funkcji `main`
Lokalnych	Deklarowanych wewnątrz funkcji lub instrukcji blokowej. Konstruktor jest wołany w momencie osiągnięcia odpowiedniej instrukcji.
Dynamicznie tworzonych	Obiekty tworzone są dynamicznie w wyniku wywołania operatora new. Operator new przydziela pamięć obiektu na stercie. Jeżeli alokacja pamięci przebiegła pomyślnie, woła natychmiast konstruktor.
Obiektów tymczasowych	Obiekty tymczasowe mogą być tworzone jako rezultaty wywołania funkcji zwracających obiekty lub w niektórych przypadkach rzutowania.
Pól składowych klas	Jeżeli składowymi obiektu są podobiekty innych klas, wówczas podczas tworzenia obiektu nadrzędnego wołane będą konstruktory komponentów.
Podobiektów klas bazowych	W podczas konstrukcji obiektu klasy potomnej wołane są konstruktory inicjujące komponenty klas bazowych.

Konstruktor

Operacje wykonywane przez konstruktor:

1. Woła konstruktor klasy bazowej
2. Woła konstruktory komponentów danych (atrybutów) w kolejności deklaracji.
3. Jeżeli klasa definiuje lub dziedziczy funkcje wirtualne, inicjalizuje wskaźnik vptr obiektu wskazujący VTABLE klasy.
4. Wykonuje kod zdefiniowany w ciele konstruktora.

Oznacza to, że w trakcie wykonania kodu konstruktora dysponujemy zestawem funkcji wirtualnych odpowiadających danemu poziomowi hierarchii.

Przykład

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

class B : public A
{

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

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

void main()
{
    C c;
}
// wypisze A::f B::f C::f
```

Destruktor

- Destruktor jest funkcją wywoływaną przy usuwaniu obiektu.
  Zadeklarowany jest jako:

```
~class-name()
```

- W klasie może być zdefiniowany dokładnie jeden destruktor.
- Destruktor nie może mieć argumentów
- Destruktor nie może zwracać wartości.

Destruktor

Destruktory są wołane dla następujących typów obiektów

Globalnych	Po zakończeniu programu, np.: po wyjściu z funkcji main
Lokalnych	Destruktor jest wołany w momencie osiągnięcia końca bloku instrukcji.
Dynamicznie tworzonych	Dynamicznie tworzone obiekty zwalniane są za pomocą operatora delete. Operator ten wywołuje destruktor oraz zwalnia pamięć obiektu.
Obiektów tymczasowych	
Jawnie	W niektórych przypadkach destruktory mogą być wywołane jawnie. Dotyczy to głównie obiektów, dla których pamięć przydzielana jest w niestandardowy sposób oraz szczególnych przypadków dziedziczenia wielobazowego.

Kolejność wykonania destruktorów

Destruktory wołane są w kolejności odwrotnej do konstruktorów:

1. Wpierw wykonywane jest ciało destruktora klasy
2. Następnie wywoływane są destruktory obiektów składowych klasy (komponentów) w kolejności odwrotnej do kolejności deklaracji. Dotyczy to komponentów, które nie są zadeklarowane jako `static`. Statyczne komponenty są usuwane jak obiekty globalne.
3. Dalej wołane są destruktory klas bazowych

Wywołanie funkcji wirtualnych w destruktorze

Zasady wywołania funkcji wirtualnych w destruktorach są analogiczne, jak w przypadku konstruktorów.

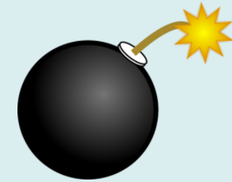
```
class A
{
public:
    ~A(){f();}
    virtual void f(){
        printf("~A::f ");
    }
};
class B : public A
{
public:
    ~B(){
        printf("~B ");
        f();
    }
};
```

```
class C : public B
{
public:
    ~C(){f();}
    void f(){
        printf("~C::f ");
    }
};
void main()
{
    C c;
}
// ~C::f ~B~A::f ~A::f
```

Wirtualne destruktory

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

Geometry::~Geometry(){
    for(int i=0;i<elements.size();i++){
        delete elements[i];
    }
}
```



Kiedy usuwamy obiekt za pomocą `delete elements[i]` – wołamy destruktor klasy `GeoObject`, a nie rzeczywistego obiektu wskazywanego przez wskaźnik.

```
class Linestring:public GeoObject{
public:
    vector<Coord> line;
    Linestring(const vector<double>&v);
    // . . .
};
```

To nie zostanie zwolniione

Wirtualne destruktory

Rozwiązanie – w klasie bazowej należy zadeklarować pusty wirtualny destruktor

```
class GeoObject{
public:
    virtual ~GeoObject(){}
    virtual string asWKT(bool printHeader = true)const{
        return "";
    }
};
```

Dzięki temu wskaźnik do kodu destruktora stanie się składnikiem tablicy VTABLE i wywołanie destruktora za pośrednictwem operatora delete wywoła zawsze właściwy destruktor klasy potomnej (który na końcu wywoła destruktor klasy bazowej)..

Przykład

```
class A
{
public:
    virtual ~A(){
        printf("~A ");
    }
};

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

```
int main()
{
    A*a=new B;
    delete a;
// dla
// virtual ~A(){printf("~A ");}
//  wypisze ~B ~A
// dla ~A(){printf("~A ");}
// wypisze ~A
}
```

Operator przypisania i konstruktor kopiujący

Operator przypisania

- Podczas **przypisania** danemu obiektowi nadaje się wartość drugiego obiektu.
- Standardowa implementacja przypisania polega na skopiowaniu *kolejnych pól* obiektu. To działanie może zostać przedefiniowane poprzez dostarczenie własnego operatora przypisania postaci:

```
X&X::operator=(const X&);
```

- Podczas przypisania poprzednia zawartość obiektu zostaje zastąpiona nową. Dla kontenerów (listy, tablice) wiąże się to ze zwolnieniem pamięci i przydzieleniem nowej.

Konstruktor kopiujący

- Podczas **inicjalizacji** z użyciem konstruktora kopiującego obiekt jest inicjowany wartością innego obiektu.
- Podobnie, jak w przypadku operatora przypisania, standardowa implementacja polega na skopiowaniu kolejnych pól drugiego obiekt.
- W przypadku klas alokujących pamięć konieczna jest odrębna implementacja konstruktor kopiującego:

```
X::X(const X&);
```

Kiedy wołany jest konstruktor kopiujący?

- Jawna inicjalizacja obiektu

```
class A {...};
A a1;
A a2=a1; /* wołany jest konstruktor
kopiujący, a nie operator przypisania! */
A a3(a1);
```

Kiedy wołany jest konstruktor kopiujący?

- Inicjalizacja związana z przesyłaniem argumentów do funkcji.
- Argumenty do funkcji mogą być przesyłane przez wartość lub referencję (adres). W pierwszym przypadku na stosie przydziela się miejsce na nowy obiekt, a następnie automatycznie wołany jest konstruktor kopiujący, który inicjuje formalny parametr funkcji wartością argumentu wywołania

```
class A {...};
void foo(A a)
{ // tu zostanie stworzona kopia argumentu
}
A a1;
foo(a1);
```

Kiedy wołany jest konstruktor kopiujący?

- Inicjalizacja związana z przesyłaniem rezultatów wywołania funkcji.
- Funkcje mogą zwracać obiekty przez wartość lub referencje. W przypadku zwracanej referencji obiekt nie może być obiektem automatycznym. W przypadku zwracanej wartości pamięć obiektu jest przydzielana na stosie.

Przykład

```
class Ret{
public:
    Ret(const Ret&o){
        printf("Ret::copy constuctor\n");}
    Ret(){printf("Ret::constuctor\n");}
};

Ret foo()
{
    Ret r;
    return r;  // tu konstr. kopiujący
}

int main()
{
    Ret r2;
    r2=foo();
}
```

Rezultat

```
Ret::constuctor
Ret::copy constuctor
```