Niepolimorficzna i polimorficzna Czarna Dziura

```
...
void CzarnaDziura::Zniszcz (CelestialBody* pBody)
{
   delete pBody;
}
```

// Założenie: klasy `Star` i `Planet` dziedziczą z `CelestialBody`

```
Star * pStar = new Star (1, 2);
Planet * pPlanet = new Planet (3, 4);

czarnaDziurka.Zniszcz (pStar);    // Mniam!
czarnaDziurka.Zniszcz (pPlanet);    // Mniam, mniam!
...
```

Wynik:

```
Destroying CelestialBody...     // destruktor klasy CelestialBody
Destroying CelestialBody...     // destruktor klasy CelestialBody
```

Metoda `Zniszcz` nie może samodzielnie rozstrzygnąć, czy przekazano jej obiekt klasy `Star`, `Planet` czy `CelestialBody`. Kompilator wybiera destruktor klasy `CelestialBody` na podstawie deklaracji wskaźnika `pBody`! Takie dziedziczenie prowadzi do utraty informacji i rzadko kiedy jest użyteczne!

Rozwiązanie: wirtualny destruktor:

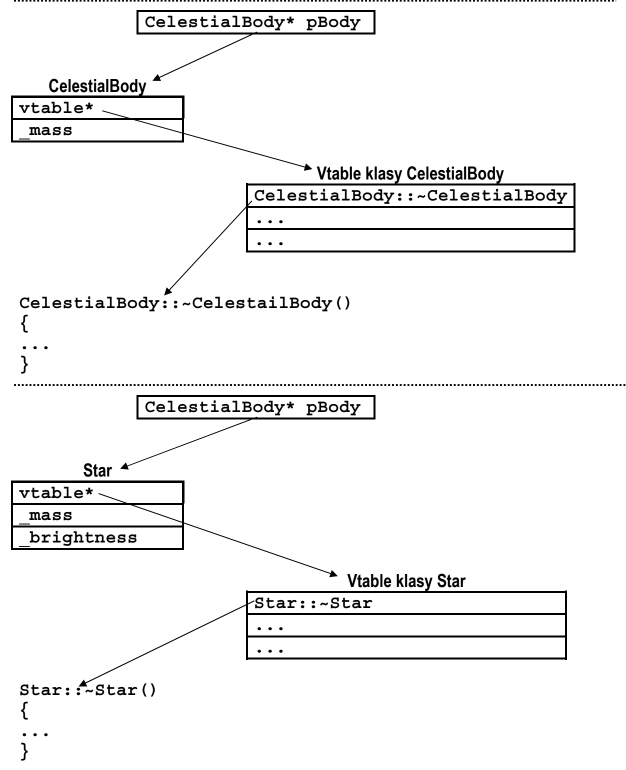
```
virtual void CelestialBody::~CelestialBody();

Destroying a Star...     // destruktor klasy Star
Destroying a Planet...   // destruktor klasy Planet
```

Informacja o tym, którą metodę wirtualną należy wywołać w kontekście danego obiektu zawarta jest wewnątrz tego obiektu!

TO JEST ISTOTA POLIMORFIZMU!
ADRES WŁAŚCIWEJ METODY WYZNACZANY JEST <u>DYNAMICZNIE</u>
NA PODSTAWIE INFORMACJI ZAWARTYCH W OBIEKCIE!

Tablica wirtualna (`vtable`)



Właściwości polimorfizmu

Koszt:

- **każdy obiekt wirtualny zawiera dodatkową ukrytą zmienną wskaźnikową**
- **metody wirtualne są wywoływane pośrednio, poprzez `vtable`.**
- **metoda wirtualna nie może być funkcją wklejaną (*inline*)**

Ale:

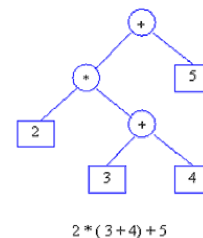
- **metody niewirtualne klas wirtualnych wywoływane są bezpośrednio, jak każde inne metody; w jednej klasie można więc definiować obok siebie metody wirtualne, zwykłe metody funkcyjne i metody „inline”.**
- **ukryty wskaźnik obciąża tylko obiekty klas wirtualnych**

Ponadto:

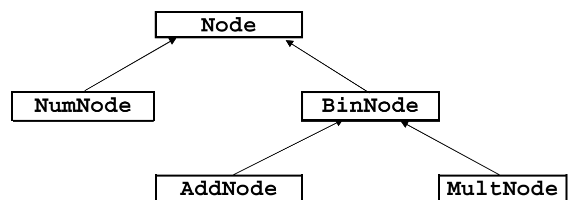
- **Klasy wirtualne niemal zawsze wymagają, by ich <u>destruktor</u> też był wirtualny.**
- **Polimorfizm wymaga dostępu do oryginalnego obiektu, zawierającego oryginalny wskaźnik do vtable. Przekazanie obiektu klasy polimorficznej przez wartość z reguły jest błędem!**

Obiekty klas polimorficznych
przekazuj przez wskaźnik lub referencję!

Drzewo arytmetyczne



Hierarchia klas drzewa arytmetycznego



- **Linią przerywaną zaznaczono klasy abstrakcyjne.**
- **Strzałki wskazują klasy bazowe**

Polimorficzna implementacja drzewa arytmetycznego

Abstrakcyjna klasa bazowa

```
class Node
{
public:
    virtual ~Node () {}
    virtual double Calc () const = 0;
};
```

- **„Definicja" = *0* oznacza czystą funkcję wirtualną.**
- **Klasa zawierająca czystą funkcję wirtualną nazywa się klasą abstrakcyjną.**
- **Klasy abstrakcyjne służą jako klasy bazowe w hierarchii dziedziczenia (zapewniają wspólny interfejs wszystkim członkom hierarchii).**

Wirtualna klasa pochodna – węzeł liczbowy

```
class NumNode: public Node
{
public:
    NumNode (double num) : _num ( num ) {}
    double Calc () const
    {
       cout << "Numeric node " << _num << endl;
       return _num;
    }

private:
    const double _num; // wartość liczby, przechowywanej w węźle
};
```

- **To już nie jest klas abstrakcyjna, bo zdefiniowano w niej własną implementację czystej metody wirtualnej `Calc()` klasy podstawowej.**

Abstrakcyjna kasa pochodna – węzły operatorów dwuargumentowych

```
class BinNode: public Node
{
public:
    BinNode (Node * pLeft, Node * pRight)
      : _pLeft (pLeft), _pRight (pRight) {}
    ~BinNode ();
protected:
    Node * const _pLeft;   // lewy potomek
    Node * const _pRight;  // prawy potomek
};

BinNode::~BinNode ()
{
    delete _pLeft;    // tu wywoła się destruktor lewego potomka
    delete _pRight;   // a tu – destruktor prawego potomka
}
```

- **Powyższa klasa jest abstrakcyjna, bo nie zdefiniowaliśmy w niej w sposób jawny definicji czystej metody wirtualnej `Calc()`.**
- **W klasach, zaprojektowanych jako klasy podstawowe w hierarchiach dziedziczenia, zamiast składowych prywatnych najczęściej stosuje się składowe chronione (`protected`)**

Specyfikator dostępu	**Kto ma dostęp do takiej składowej?**
`public`	Każdy
`protected`	Dana klasa, jej przyjaciele i klasy pochodne
`private`	Wyłącznie dana klasa i jej przyjaciele

Węzły dodawania i mnożenia

```
class AddNode: public BinNode
{
public:
    AddNode (Node * pLeft, Node * pRight)
        : BinNode (pLeft, pRight) {}
    double Calc () const;
};

double AddNode::Calc () const
{
    cout << "Dodawanie\n";
    return _pLeft->Calc () + _pRight->Calc ();
}

class MultNode: public BinNode
{
public:
    MultNode (Node * pLeft, Node * pRight)
        : BinNode (pLeft, pRight) {}
    double Calc () const;
};

double MultNode::Calc () const
{
    cout << "Mnozenie\n";
    return _pLeft->Calc () * _pRight->Calc ();
}
```

Program testowy

```
int main ()
{
    // ( 20.0 + (-10.0) ) * 0.1
    Node * pNode1 = new NumNode (20.0);
    Node * pNode2 = new NumNode (-10.0);
    Node * pNode3 = new AddNode (pNode1, pNode2);
    Node * pNode4 = new NumNode (0.1);
    Node * pNode5 = new MultNode (pNode3, pNode4);
    cout << "Wyznaczamy wartosc drzewa\n";
    // prosimy korzeń drzewa o wyznaczenie wartości całego drzewa
    double x = pNode5->Calc ();
    cout << x << endl;
    delete pNode5; // niszczymy korzeń wraz ze wszystkimi potomkami
}
```


Funkcje jako argumenty innych funkcji

- **Zmienne adresowe mogą zawierać adresy funkcji.**
- **Tak jak każda nazwa tablicy reprezentuje adres swojego pierwszego elementu, każda nazwa funkcji reprezentuje swój adres.**
- **Pracę ze wskaźnikami do funkcji najczęściej rozpoczynamy od deklaracji `typedef`:**

```
typedef double (*pFun)(double x);
```

Powyższa instrukcja deklaruje identyfikator `pFun` jako alternatywną nazwę typu „wskaźnik do funkcji pobierającej jako argument kopię obiektu typu `double` oraz przekazującej jako swoją wartość kopię obiektu typu `double`".
Porównaj tę deklarację z deklaracją funkcji `sin`:

```
double sin (double x);
```

Jest jasne, że `*pFun` powinno być tego samego typu, co `sin`.

- **Definicja i inicjacja wskaźnika do funkcji lub tablicy takich wskaźników nie nastręcza trudności:**

```
pFun fun1 = sin;
pFun tabFun[2] = {sin, cos};
cout << fun1(1.0) << "\t" << tabFun[1](3.14) << "\n";
if (tabFun[1] == cos)
   tabFun[1] = acos;
```

- **Wskaźnikiem do funkcji można posługiwać się bez operatora wyłuskania (`*`), np. wyrażenie**

`(*fun1)(1.0)` **jest równoważne wyrażeniu** `fun1(1.0)`

Przykład

```
typedef std::complex<double> zesp;   // typ liczb zespolonych
typedef zesp (*zespFun)(zesp const & z);// funkcje zespolone
Integrator::Integrator (BasePath & kontur, zespFun f);
```

jest czytelniejsze, niż:

```
Integrator::Integrator (BasePath & kontur,
                        std::complex<double> (*f) ( std::complex<double> const & z ) );
```

Domyślne argumenty funkcji

- **Domyślne wartości argumentów (dowolnej) funkcji w C++ zapisujemy w deklaracji funkcji, stosując składnię inicjacji z operatorem =.**

```
CirclePath::CirclePath (int n, double radius, double x0 = 0.0, double y0 = 0.0);
```

- **Wywołanie**

  ```
  CirclePath cp (100, 2.0, 1.0);
  ```

  zostanie przez kompilator automatycznie zamienione na

  ```
  CirclePath cp (100, 2.0, 1.0, 0.0);
  ```

- **Wartości domyślne można nadać tylko ostatnim argumentom funkcji.**
- **Oczywiście informacja o wartościach domyślnych potrzebna jest w deklaracji, a nie w implementacji funkcji (konsolidator i tak nic nie wie o wartościach domyślnych).**
- **Domyślne argumenty funkcji:**
  - **Prowadzą do zwięźlejszego zapisu kodu**
  - **Umożliwiają połączenie kodu kilku funkcji (np. konstruktorów) w jedną.**

Przykład

Jeden konstruktor

```
complex::complex (double re, double im = 0.0);
```

zastępuje dwa konstruktory:

```
complex::complex (double re, double im);
complex::complex (double re);
```

co bardzo ułatwia pielęgnację kodu.