# Funkcje wirtualne

- **W C++ polimorfizm jest zrealizowany w dwa sposoby: na etapie kompilacji i na etapie wykonania. Na etapie kompilacji polimorfizm jest zrealizowany poprzez przeciążenie funkcji i operatorów. To jest polimorfizm statyczny. Na etapie wykonania (polimorfizm dynamiczny) polimorfizm jest zrealizowany poprzez funkcje wirtualne.**

- **Wskaźniki do klas pochodnych są podstawą dla funkcji wirtualnych i polimorfizmu dynamicznego.**

- **Wskaźnik do klasy bazowej może występować jako wskaźnik do obiektu dowolnej klasy pochodnej**

- **Podany poniżej przykład ilustruje fragment poprawnego kodu. Błąd o niespójności typów się nie generuje, kiedy wskaźnik typu klasy bazowej wskazuje do obiektu klasy pochodnej.**

```
class  base
{
……..
};

class derived : public base
{
……..
};

void main()
{
        base *ptr;
        base ob_base;
        derived ob_derived;

        ptr = &ob_base;     //wskaźnik ptr wskazuje na obiekt klasy bazowej

        ptr = &ob_derived;   //wskaźnik wskazuje na obiekt klasy pochodnej
}
```

- **Jeśli wskaźnik typu klasy bazowej wskazuje do obiektu klasy pochodnej, to:**

  - **przez ten wskaźnik możemy mieć dostęp tylko do tych składowych klasy pochodnej, które dziedziczą się z klasy bazowej – klasa bazowa nic nie wie o składowych, które powstali w klasie pochodnej**

  - **arytmetyka wskaźników jest skojarzona z typem danych, do którego ten wskaźnik pozostał zadeklarowany. Z tego wynika:**

```
derived ob_derived[10]
base *ptr = &ob_derived[0];
ptr++;     //nie wskazuje na następny obiekt klasy pochodnej !
```

  - **Wskaźnik klasy pochodnej nie można wykorzystać dla dostępu do obiektów klasy bazowej**

- **Funkcja wirtualna to składową zadeklarowana w klasie bazowej i zdefiniowana w klasie pochodnej.**

- **Aby utworzyć funkcję wirtualną, należy jej deklaracje w klasie bazowej poprzedzić kluczowym słowem *virtual* . Klasa pochodna, która dziedziczy z klasy bazowej funkcją wirtualną, definiuje ją zgodnie ze swoimi potrzebami.**

- **W klasie bazowej funkcje wirtualne definiują postać interfejsu. Każde (ponowne) definiowanie funkcji wirtualnej w klasie pochodnej wyznacza specyficzną realizacje tej funkcji dla podanej klasy pochodnej. W taki sposób powstaje konkretna metoda.**

- **Przy definiowani funkcji wirtualnej w klasie pochodnej kluczowe słowo *virtual* nie jest potrzebne.**

- **Funkcje wirtualne mogą być wywołane jako zwykłe funkcje – metody klasy. Przecież polimorfizm dynamiczny występuje, jeśli funkcje wirtualne są wywoływane przez wskaźnik do typu klasy bazowej, który wskazuje na obiekt klasy pochodnej.**

- **Gdy wskaźnik klasy bazowej jest wykorzystywany do wskazania obiektu, zawierającego funkcję wirtualną, C++ decyduje o wyborze wersji funkcji na podstawie typu obiektu wskazywanego przez ten wskaźnik. Wybór wersji odbywa się na etapie wykonywania programu – będzie wywoływana ta realizacja funkcji wirtualnej, która należy obiektowi klasy, wskazanej przez wskaźnik.**

- **Przykład W12_1**

- **Ponowne definiowanie funkcji wirtualnej w klasie pochodenej różni się od przeciążenia funkcji:**

  - **prototyp funkcji (nazwa, ilość i typy argumentów formalnych i typ wartości zwracanej), redefiniowanej w klasie pochodnej, musi dokładnie odpowiadać prototypowi, występującemu w klasie bazowej. (Przy przeciążeni funkcji każda realizacje się różni listą argumentów formalnych). Jeśli prototyp funkcji wirtualnej pozostanie zmieniony w klasie pochodnej, to taka funkcja będzie potraktowana jako funkcja przeciążona, a nie wirtualna.**

  - **funkcje wirtualne muszą być niestatycznymi składowymi klasy, do której należą (funkcji przeciążone – nie).**

- **Wybór realizacji funkcji wirtualnych jest dokonywany przy wykonani programu (dla funkcji przeciążonych – na etapie kompilacji).**

- **Konstruktory nie mogą być funkcjami wirtualnymi, a desktruktory – mogą**

- **Przykład W12.4**

- **Dla tego, żeby podkreślić różnice pomiędzy funkcjami wirtualnymi i funkcjami przeciążonymi, redefiniowanie funkcji wirtualnych jest określone jako *przesłanianie (overriding)*.**

- **Funkcje wirtualne mogą być wywołane przez użycie referencji do klasy bazowej (przykład W12_1).**

- **Funkcja wirtualna zachowuje atrybut *virtual* przy dziedziczeni. Oznacza to, że przy kolejnym dziedziczeni w kolejnej klasie pochodnej funkcja wirtualna może być także przesłoniętą.**

- **Przykład W12_2.**

- **Jeśli funkcja wirtualna nie pozostała przesłonięta w klasie pochodnej, wtedy obiekty tej klasy, odwołujące się do takiej funkcji, będą wykorzystywali funkcję, zdefiniowaną w klasie bazowej.**

## Funkcje abstrakcyjne (czysto wirtualne funkcji – pure virtual functions)

- **Często się okazuje, że w klasie bazowej nie da się stworzyć przydatną definicje funkcji wirtualnej – po prostu klasa bazowa nie dysponuje dostatnią informacją. Wtedy klasa bazowa zawiera tylko deklaracje (prototyp) funkcji wirtualnej bez jej definicji – wyznacza interfejs:**

```
virtual typ_zwrac nazwa_funkcji (lista parametrów) = 0;
```

- **Składnia …= 0 informuje kompilator o to, że definicji funkcji nie istnieje w klasie bazowej.**

- **Każda klasa pochodna powinna mieć swoją własną definicje każdej funkcji abstrakcyjnej – wyznacza swoją własną realizacje.**

- **Klas, który zawiera chociażby jedną funkcje abstrakcyjną, nazywamy klasą abstrakcyjną (abstract class). Niewolno tworzyć obiektów klas abstrakcyjnych – takie klasy nie są pełne. Przecież można tworzyć wskaźniki do klas abstrakcyjnych i referencji.**

- **Ważnym stosowaniem klas abstrakcyjnych i funkcji wirtualnych jest użycie ich w bibliotekach klas. Na przykład, klasa abstrakcyjna Figura zawiera wspólne dane dla różnych płaskich obiektów geometrycznych, ale nic nie wie o szczegółach realizacji – obliczeni pola, zapisu danych na dysk, odczyt z dysku, wyświetlani danych na monitorze. Ale jest możliwe rozszerzyć funkcjonalność tej klasy – stworzyć klasę pochodną, i w niej zdefiniować te realizacje metod wirtualnych, które są potrzebne dla każdego podanego obiektu.**

- **Wczesne wiązanie** odnosi się do wydarzenia, które ma miejsce przy kompilacji (obiekt i wywołanie funkcji są powiązane na etapie kompilacji). Przykłady: wywołanie funkcji zwyczajnej, wywołanie funkcji i operatorów przeciążonych. Główną zaletą jest **wysoka wydajność** (dla tego że wszystkie informacje do wywołania funkcji są określane na etapie kompilacji). Wada – **niewielka elastyczność**.

- **Późnie wiązanie** dotyczy funkcji, wywołanie których jest determinowane przy wykonani programu. Przykład – funkcje wirtualne, wywołanie których odbywa się przez wskaźnik lub referencje do klasy bazowej, a decyzje o to, która realizacje pozostanie wywołana, będzie podjęta przy wykonani programu na podstawie typu wskazywanego obiektu. Główna zaleta – **elastyczność**, która nadaje możliwość reagować na różne zdarzenia zachodzące dopiero w trakcie wykonania programu. **Może wydłużyć czas wykonania** (dla tego że wywołanie funkcji nie jest zdeterminowane aż do momentu jej wykonania).

# Szablony

- **Szablony nadają możliwość tworzenia ogólnych funkcji i ogólnych klas. W ogólnych funkcjach (klasach) typ danych, na których operuje funkcja (klasa), jest określany jako parametr. Oznacza to, że można utworzyć funkcje (klasę) przystosowaną do pracy z kilkoma typami danych bez jawnego oprogramowania nowej wersji funkcji (klasy) dla każdego z poszczególnych typów danych.**

## Funkcje ogólne

- **Definiuje zestaw operacji, przeznaczonych do wykonywania na różnych typach danych**

- **Typ danych jest przekazywany jako parametr**

- **Wielu algorytmów mają dokładnie taką samą logikę dla różnych typów danych. Na przykład: przeszukiwanie obiektu w tablice (struktury danych), sortowanie obiektów w tablice (strukturze danych), dodawanie obiektu do tablicy (do struktury danych), usunięcie obiektu z tablicy (z struktury danych) ….**

- **Zastosowanie funkcji ogólnych (klas ogólnych) jest bardzo skuteczne, kiedy algorytmy przetwarzania danych nie zależą od typów danych. Wtedy można rozdzielić kod na procedury, które nie zależą od typu danych (odnoszą się do algorytmów), i na procedury, które odnoszą się do konkretnego typu danych (zależą od typu danych).**

- **Ta cześć kodu, która nie zależy od typu danych, może być skutecznie napisana w postaci funkcji ogólnych (klas ogólnych).**

- **Przy tworzeni funkcji ogólnej jest tworzona funkcja, która potrafi sama się przeciążyć.**

- **Dla tworzenia funkcji ogólnych posługując słowem kluczowym *template*. Przy tym powstaje szkielet funkcji, w który w trakcie kompilacji będą podstawione odpowiednie typy danych – kompilator sam generuje każdą konkretną realizacje dla podstawionego typu danych**

```
template  <class Ttype>  typ_zwrac  nazwa_funkcji(lista_parametrów)
{
   //ciało funkcji
}
```

- **Ttype – typ danych, wykorzystywany przez funkcje. Nazwa ta może być używana wewnątrz definicji funkcji. To i jest symbol – szablon. Wszystkie działania ogólnego algorytmu przy jego realizacji w tej funkcji będą wykonywane nad tym symbolem, a podczas tworzenia specyficznej wersji funkcji kompilator zamieni ten symbol-szablon na rzeczywisty (podstawiony) typ danych. Słowo *class* może być zamienione na słowo *typename* :**

```
template  <typename Ttype>  typ_zwrac  nazwa_funkcji(lista_parametrów)
{
   //ciało funkcji
}
```

- **Przykład: W13_0**

- **Funkcje ogólna (definicja tej funkcji pozostała poprzedzoną kluczowym słowem *template*) jest także nazywana szablonem funkcji. Dla każdego podstawionego typu danych kompilator tworzy konkretną *specjalizacje* tej funkcji – mówimy, że kompilator stworzył *funkcje wygenerowaną* (generated function). Proces generowania funkcji nazywają *tworzeniem egzemplarza* (instantiation).**

- **Słowo kluczowe template może być w innej linii:**

```
template <class T>
void findmydata(T ob, size_t  len)
{
……..
}
```

**Pomiędzy linią template <class T> i linią void findmydata……. nie mogą się pojawić żadne instrukcje.**

- **W szablonie można wykorzystać więcej niż jeden typ ogólny:**

```
template <class T1, class T2>
void myfunc(T1 a, T2 b)
{
……..
}
```

- **Przy tworzeni szablonu funkcji polecamy kompilatorowi wygenerowanie tyle różnych wersji tej funkcji, ile jest potrzebne do obsługi wszystkich różnych wywołań funkcji występujących w programie.**

- **Funkcje-szablony są podobne do funkcji przeciążonych, przecież są bardziej ograniczone. Przy przeciążeni funkcji można wykonywać dowolne zmiany w instrukcjach kodu, funkcja-szablon musi wykonywać te same**

**instrukcji kodu dla każdej wersji.**

- **Mimo że funkcja-szablon sama w miarze potrzeby dokonuje przeciążenie, jednak można wykonać przeciążenie jawne:**

```
template <class T> void swapargs(T  &a,  T  &b)
{
      T    tmp;
      tmp = a;      // class T  powinien mieć przeciążenie  operatora =
      a = b;
      b = tmp;
}

//Tu funkcje-szablon pozostanie redefiniowaną
void swapargs(double &a, double &b)
{
            double t = a;
            a = (a+b)*(a+b);
            b = t;
}

void main()
{
      double a = 10.0, b = 20.0;
      coord c1(0, 0), c2(1,1);
      swapargs(c1, c2);          //wywołanie pierwotną swapargs
      swapargs(a, b);            //wywołanie przeciążoną swapargs
}
```

- **Podany przykład nie jest typowym. Jeśli logika programu jest taka, że każdą wersja funkcji powinna wykonywać swój własny algorytm, bardziej naturalnym jest użycie funkcji przeciążonych. Jeśli dla każdej wersji algorytm jest taki samy, a różnica występuje tylko w typie danych, naturalnie używać szablony.**

# Klasy ogólne

- **Klasa zawiera definicje wykorzystanych przez nią algorytmów, jednak typ danych, na których klasa operuje, zostanie przekazany do niej jako parametr dopiero w chwili tworzenia obiektu.**

- **Klasy ogólne są przydatne kiedy jeden zbiór algorytmów trzeba zastosowywać dla różnych typów danych.**

- **Deklaracja klasy-szablonu**

```
temlate <class Ttype> class  nazwa_clasy
{
};
```

- **Ttype – nazwa typu, który zostanie określony przy tworzeni egzemplarza klasy.**

- **Tworzenie egzemplarza klasy:**

```
nazwa_clasy  <typ>  obiekt;
```

- **typ – to ten typ danych, na którym ma operować klasa. Funkcje – składowe klasy ogólnej automatycznie stają funkcjami ogólnymi, nie ma konieczności poprzedzać ich nazwę słowem kluczowym *template* przy deklaracji ich prototypów.**

- **Przykład: W14_0**

- **Szablony klas nadają możliwość stworzyć ogólną postać algorytmu, który dalej można użyć dla obsługi danych dowolnego typu. To pozbawia programistę od napisania wielokrotnych realizacji tego samego algorytmu dla różnych typów danych.**

- **Klasa – szablon może mieć wielu typów danych:**

```
template  <class  T1, class T2>  class   myclass
{
      ……..……
};
```

- **W klasie-szablonie można wykorzystać domyślne typy danych, związanych z typami danych standardowych:**

```
tempalte  <class T1, class T2 = int>  class   my_cl
{
……..
};

void main()
{
      my_cl< coord, double >  aa(……..);   //T1 ← coord;  T2 ← double
      my_cl <coord>   bb(……….);            //T1 ← coord;  T2 ← int
}
```

- **Przykład 14_1.**

- **Słowo kluczowe *typename* ma dwa zastosowania:**
  - **służy zamiennikiem słowa *class***
  - **poinformowanie kompilatora, że nazwa użyta w deklaracji szablonu dotyczy nazwy typu, a nie nazwy obiektu:**

```
typename X::Name someObj;
```

**X::Name jest traktowane jako nazwa typu.**

- **Słowo kluczowe *export* może poprzedzić deklaracje *template*. Nadaje możliwość umieścić deklarowanie szablonu w jednym pliku, a definicje – w drugim. UWAGA! Kompilator Visual Studio 2005 nie wspiera słowa kluczowego *export*. Dla programów złożonych trzeba umieszczać deklaracje i definicje szablonu w pliku *.h, a w tych plikach, gdzie trzeba ten szablon udostępnić, inkludować plik nagłówkowy.**