Dr. hab. Inż. Sergiy Fialko,

https://torus.uck.pk.edu.pl/~fialko.sergiy/
sergiy.fialko@pk.edu.pl

Literatura

1. **B. Stroustrup. The C++ Programming Language (Fourth Edition). May 2013, Addison Wesley. Reading Mass. USA. May 2013.**

2. **Herbert Schildt. The Complete Reference C++. 2020, Tata McGraw Hill.**
https://www.allabout-engineering.com/download-the-complete-reference-c-by-herbert-schildt/

3. C++ language documentation. MSDN.
https://docs.microsoft.com/en-us/cpp/cpp/?view=msvc-160

number of lines in code (nlc)	Small nlc < 10 000	Medium 10 000 < nlc < 100 000	Large nlc > 100 000
programming language	assembler, FORTRAN	Structured languages C, Pascal	Object-oriented languages C++, Java

Program, napisany w języku strukturalnym (na przykład, w C), jest tworzony poprzez definiowanie funkcji, z których każda może operować na dowolnym typie danych, wykorzystywanych przez program.

W każdym języku obiektowym definiują się dane oraz procedury, które mogą operować wyłącznie na tych danych. Typ danych określa, jakiego rodzaju operacje mogą być dokonywane.

Cechy języków obiektowych: hermetyzacja (enkapsulacja), polimorfizm, dziedziczenie.

- **Enkapsulacja (Hermetyzacja)**

To mechanizm, łączący w całość kod i dane, na których on operuje, zabezpieczający je przed zewnętrzną ingerencja i niepoprawnym użyciem.

• Kod i dane są połączone w *obiekcie*, który i realizuje hermetyzacje.

• Znajdujące się wewnątrz obiektu kod i dane mogą być prywatne lub publiczne.

• Prywatny kod i dane są dostępne tylko w obszarze obiektu. Zewnątrz obiektu składniki prywatne są ukryte – ani prywatny kod, ani prywatne dane nie mogą być wykorzystywane przez fragmenty programu, znajdujące się poza obiektem.

• Publiczny kod i dane są dostępne dla innych elementów programu, znajdujących się jak wewnątrz obiektu, tak i poza nim. Najczęściej składowe publiczne obiektu są wykorzystywane do tworzenia interfejsu do prywatnych elementów obiektu.

Example 1 (W1).

Powstaje pytanie, po co tak bardzo komplikować dostęp do składowych obiektu, wprowadzając specyfikator dostępu private?

Rozważmy duży i złożony program zawierający setki tysięcy linii kodu. Teraz spróbujemy sprawdzić, gdzie zmieniają się wartości zmiennych *ii*, *jj* obiektu MY_CLASS.

Jeśli jest to zmienna publiczna, musisz przeszukać olbrzymi kod dla wszystkich fragmentów typu *id_object.ii* = … .

Jeśli jest to zmienna prywatna, jej wartość można zmienić tylko przez wywołanie funkcji typu *Set_j*. Ustawiamy punkt przerwania w funkcji *Set_j* i uruchamiamy zadanie pod debug'erem. Teraz kontrolujemy wszystkie wywołania funkcji *Set_j* z olbrzymiej części kodu, która nie jest związana z obsługą naszego obiektu. **(W1)**

Polimorfizm – “jeden interfejs, wielu realizacji”.

Nadaje możliwość jednemu interfejsowi kontrolować dostęp do ogólnej klasy akcji. O tym, jaka konkretnie akcja pozostanie wybrana, decyduje sytuacja.

Różnica pomiędzy statycznym i dynamicznym polimorfizmem.

- Statyczny polimorfizm jest związany z przeciążeniem (przeładowaniem) funkcji i operatorów i odbywa się na etapie kompilacji.

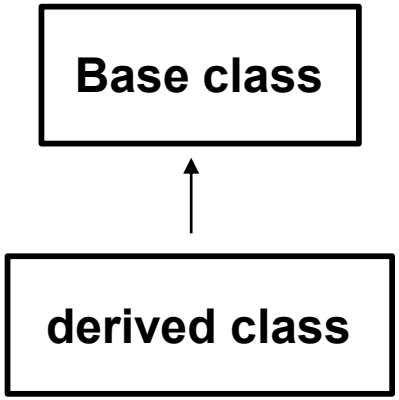
- Polimorfizm dynamiczny opiera się na wykorzystaniu funkcji wirtualnych i jest implementowany w czasie wykonywania. Decyzja o to, która z wielu funkcji wirtualnych powinna zostać wywołana, jest podejmowana na etapie wykonywania zadania.

Dziedziczenie - to proces, w którym jeden obiekt może nabyć właściwości innego obiektu.

Jeśli dane i działania na nich mogą być zorganizowane hierarchicznie, bardzo wygodnie jest na każdym poziomie hierarchii utworzyć tylko obsługę danych, które należą do tego poziomu.

Taka technika programowania ułatwia zrozumienie programu i skraca czas programowania.

Example W2.



Prosty przykład programu na C++

```
# include <iostream>   //C++ header (stdio.h)
using namespace std;

int main()
{
        int i;
        cout  << “hello, C++” << “\n”;   //single line comment
        /* You can use C style comments as well*/

        //input number using >> operator
        cin >> i;   //formatting is produced as default

        //Output using << operator
        cout  <<  ”i = ” << i << ”i *i = ” << i*i << ”\n ” ;

        return 0;
}
```

```
# include <iostream>
```

Jest to anałog C - #include <stdio.h>. Standard C++ używa stylu, który nie wymaga *.h*. Nagłówki używane w C++ mogą nie być plikami nagłówkowymi, są to konstrukcje abstrakcyjne, które gwarantują, że zostaną zadeklarowane odpowiednie prototypy i definicje wymagane przez biblioteki C++. Na przykład
<vector> <fstream> <string>

```
using namespace std;
```

Zostanie użyta przestrzeń nazw *std*. Przestrzeń nazw tworzy deklaratywny obszar, w którym można umieścić wiele elementów programu. Przestrzenie nazw pomagają nam w organizacji dużych programów.

Instrukcja *using* udostępnia przestrzeń nazw *std*, w której zadeklarowana jest standardowa biblioteka C++ (CRT).

C++ ma operator wyjścia << oraz operator wejścia >> . Kompilator C++ w zależności od kontekstu może odróżnić operator wstawiania (wyjścia) << od operatora binarnego przesunięcia w lewo << oraz operator ekstrakcji (wejścia) >> od operatora binarnego przesunięcia w prawo >> .

Insertion to stream cout: cout << a;

Extraction from stream cin: cin >> b;

Standardowe strumieni związane z urządzeniami I/O:

cin – urządzenie wejściowe połączone ze strumieniem wejściowym (kławiatura → sterownik OS → strumień *cin*).

cout – urządzenie wyjściowe połączone ze strumieniem wyjściowym (api → strumień *cout* → sterownik OS → monitor).

cerr – urządzenie wyjściowe połączone ze strumieniem wyjściowym przeznaczone dla wyprowadzenia błędów.

Instrukcja *cout* << *object*; Jeśli *object* należy do typów danych predefiniowanych, to nie jest wymagane jawne użycie specyfikacji formatu, ponieważ działa jako formatowanie domyślne. Jeśli *object* – obiekt klasy (struktury, unii), to taka klasa (struktura, unia) powinna zawierać przeciążenie operatora <<.

Dokładnie tak samo wygląda sytuacja z instrukcją cin >> object;

Nadaje to możliwość jedynego interfejsu I/O dla dowolnych typów danych.

W przypadku typów predefiniowanych dla tego, żeby zmienić formatowanie domyślne, trzeba użyć tak zwane manipulatory albo składowe klasy *ios*. Rozważymy to później.

Examples:

```
double a = 0.3;
int i = 15;
char ch = 'a';
char str[] = “abcde”;

cout << “ a =  “ << a << “ i = “ << i << “ ch = ” << ch << “ str: “ << str << endl;

double b;
cout << “input date: “ << endl;
cin >> b;
.....
```

Example W3

Po wprowadzeniu wiersza tekstowego operator >> zatrzymuje się po wykryciu pierwszej spacji. Wprowadzone zostanie tylko pierwsze słowo, pozostała część wiersza tekstowego pozostanie w strumieniu *cin*. Działa to dokładnie tak samo jak wprowadzanie wiersza tekstowego za pomocą funkcji CRT scanf (“%s”, str);

Deklaracja zmiennych lokalnych

W C zmienne lokalne muszą być zadeklarowane na początku bloku. W C++ zmienne lokalne mogą być deklarowane w dowolnym miejscu bloku, zwykle w tych miejscach kodu, w których jest to wygodne dla programisty.

Brak domyślnej konwersji do typu *int*

W języku C, jeśli funkcje zwracają zmienne, których typ nie jest jawnie zadeklarowany, domyślnie typ tej zmiennej jest traktowany jako int. W języku C++ typ zwracany przez funkcje musi być zadeklarowany jawnie.

Typ bool

W C++ jest wbudowany typ logiczny - *bool*. Obiekt typu *bool* może przechowywać tylko wartości *true* i *false*, które są słowami kluczowymi zdefiniowanymi w C++. Wartości inne niż 0 są konwertowane na prawdę, a zero na fałsz. I odwrotnie, prawda jest zamieniana na 1, a fałsz na 0.

Przestrzeni nazw

Przestrzeń nazw to jest obszar dekłaratywny. Przestrzenie nazw służą do identyfikowania nazw i unikania kolizji. Elementy zadeklarowane w jednej przestrzeni nazw są oddzielone od elementów zadeklarowanych w innej.

```
namespace MY_NAMESPACE_1
{
    int i;
    void fun(int j, char *str);
};

namespace MY_NAMESPACE_2
{
    int i;
    void fun(int j, char *str);
};

void MY_NAMESPACE_1::fun(int j, char *str)
{
    cout << “j =  “ << j << “ str:  “ << str << endl;
}

void MY_NAMESPACE_2::fun(int j, char *str)
{
    cout << “j*j =  “ << j*j << “ str:  “ << str << “ !!!” << endl;
}
```

```
int i;                                  //belongs to the global namespace
void fun(int j, char *str)
{
   cout << “j*j*j = “ << j*j*j << “str: “ << “[ “ << str << “] “ << endl;
}

int main()
{
   MY_NAMESPACE_1::i = 1;          //belongs to namespace  MY_NAMESPACE_1
   MY_NAMESPACE_2::i = 2;          // belongs to namespace MY_NAMESPACE_2
.....................................................................................................................
```

// Są to różne zmienne - znajdują się w różnych adresach pamięci
// W tym celu nie dochodzi do kolizji nazw; :: - operator rozpoznawania zakresu (operator
// zakresu).

MY_NAMESPACE_1::fun(10, “abcd”); //call the function from scope MY_NAMESPACE_1

MY_NAMESPACE_2::fun(15, “efgk”); // call the function from scope MY_NAMESPACE_2

fun(20, “tunm”);   // call the function from global namespace
:: fun(20, “tunm”); // call the function from global namespace

```
#include <iostream>

using namespace std;  // przestrzeń nazw std jest dostępna w całym pliku
. . . . . .

void fun(list of arguments)
{
      using namespace MY_NAMESPACE_1;  //przestrzeń nazw
                                       //MY_NAMESPACE_1 jest dostępna
                                       //w bloku funkcji
       . . . . . .
}

void fun_second()
{
      MY_NAMESPACE_2::i = 10;      //udostępnia MY_NAMESPACE_1::i
                                   //tylko w tym miejscu kodu
      . . . . . .
}
```

Header:

```
#pragma once

MY_NAMESPACE_A {
    class MY_CLASS
    {
        public:
            void Set(int ii);
        private:
            int i;
    };
};
```

Source: //definicja funkcji-metody klasy, umieszczonej w przestrzeni nazw

```
#include <iostream>
#include “Header.h”
using namespace std;

void MY_NAMESPACE_A :: MY_CLASS :: Set(int ii) {
    i = ii;
}
```

Example W1 (Odkomentuj #define USE_NAMESPACE_EXAMPLE)