

Zawartość:

- szablony w C++
- szablony w C#

Szablony C++:

Zobaczmy kod:

```
int add(int a, int b){
    return a + b;
}

double add(double a, double b){
    return a + b;
}

int main(){
    int r1 = add(1,2);
    double r2 = add(1.1, 2.3);

    return 0;
}
```

Widzimy tutaj dwie wersje funkcji add. A można to skrócić:

```
template <typename T>
T add(T a, T b){
    return a + b;
}

int main(){
    int r1 = add<int>(1,2);
    double r2 = add<double>(1.1, 2.3);

    return 0;
}
```

Pokazany powyżej przypadek jest mało drastyczny, ale pokazuje całą ideę szablonów. Tworzymy szablon funkcji, a w momencie wywołania oprócz podania parametrów wskazujemy mu jakim typem lub typami ma się posługiwać bo oczywiście można ich podać więcej.

```
template <typename T, class D>
D add(T a, T b){
    return a + b;
}

int main(){
    int r1 = add<int, double>(1,2);
    double r2 = add<double, double>(1.1, 2.3);

    return 0;
}
```

Tutaj wykorzystaliśmy już dwa parametry (często nazywane są parametrami generycznymi). Słowo kluczowe **template** oraz **class** można stosować zamiennie.

Szablony więcej sensu mają w przypadku całych klas:

```
template <typename T>
class Point{
public:
    T x;
    T y;
    Point(T xp, T yp){
        x = xp;
        y = yp;
    }
};

int main(){
    Point<int> p(1,2);

    return 0;
}
```

Mamy teraz bardziej uniwersalną klasę. Zakładając, że nasza klasa przewiduje całkiem dużo logiki, zastosowanie szablonów może pozwolić nam zaoszczędzić dużo kodu (brak duplikacji).

Jeżeli ktoś używał STLa oraz zawartych tam kontenerów jak np. vector to pewnie zauważył podobny motyw.

```
class MyClass{
    //some stuff
};

int main(){
    std::vector<int> v1;
    std::vector<double> v2;
    std::vector<bool> v3;

    std::vector<MyClass> v4;

    return 0;
}
```

Nikt nie tworzył klasy vector dla każdego możliwego typu (BTW a co z naszymi własnymi typami jak ten MyClass) tylko użyto szablonu. Podejrzymy implementację:

```
template <class _Ty, class _Alloc = allocator<_Ty>>
class vector { // varying size array of values
private:
    template <class>
    friend class _Vb_val;
    friend _Tidy_guard<vector>;

    using _Alty          = _Rebind_alloc_t<_Alloc, _Ty>;
    using _Alty_traits = allocator_traits<_Alty>;
```

To tylko pierwsze kilka linii, ale widać motyw zastosowany w przykładach wyżej.

Mimo, że szablony pomagają w pisaniu kodu sprawiają, że kod C++ jest trudniejszy do czytania i analizy.

Szablony w C#:

Tutaj sytuacja jest bardzo podobna do tej z C++.

```
1 reference
static void Swap<T>(ref T a, ref T b)
{
    T c = a;
    a = b;
    b = c;
}

0 references
static void Main(string[] args)
{
    int a = 10;
    int b = 20;

    Swap<int>(ref a, ref b);
    Console.WriteLine($"{a} : {b}");
}
```

W przypadku funkcji składnia jest niemal identyczna jak w C++. Oczywiście referencje dodane są na potrzeby tego konkretnego przykładu. Nie jest to konieczny element.

```
1 reference
static T Add<T>(T a, T b)
{
    return a + b;
}

0 references
static void Main(string[] args)
{
    int a = 10;
    int b = 20;

    int result = Add<int>(a, b);
    Console.WriteLine(result);
}
```

Pytanie dlaczego środowisko podkreśla operację dodawania na czerwono? Dzieje się tak, ponieważ kompilator nie ma pewności, czy zawsze podamy tam pod T typ, który można ze sobą dodawać. Jak podamy tak obiekt jakiejś klasy, która nie ma przeciążonego operatora + to tak czy siak będziemy mieli błąd.

Intellisense w Visual Studio potrafi wskazywać takie rzeczy już w trakcie pisania.

```
0 references
static T Add<T>(T a, T b)
{
    dynamic c = a;
    dynamic d = b;
    return c + b;
}
```

Użycie dynamicznego typu pozwala obejść ten problem, jednak tracimy na bezpieczeństwie. Typy dynamiczne są ewaluowane w trakcie pracy programu co w tym przypadku skutkuje tym, że nie mamy w momencie kompilacji pewności czy funkcja zawsze się powiedzie.

Istnieje jeszcze słowo kluczowe **where**, które pozwala ograniczyć możliwe do podania wartości pod T.

```
0 references
static void MyFunc<T>(T a, T b) where T : System.Enum
{
    //some logic
}
```

W tym przypadku T musi być enumem.

```
0 references
static void MyFunc<T>(T a, T b) where T : IEnumerable<int>
{
    //some logic
}
```

W tym przypadku wymagamy by T było kolekcją liczb całkowitych.

Słowo where nie sprawdza się w każdym przypadku. W wielu trzeba po prostu improwizować.

Przykład z klasą:


```
3 references
class MyClass<T, D>
{
    public T id;
    public D value;

    1 reference
    public MyClass(T id, D value)
    {
        this.id = id;
        this.value = value;
    }
}

0 references
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        MyClass<int, double> a = new MyClass<int, double>(1,2.2);
    }
}
```

Każdy piszący w C# widział coś takiego:

```
0 references
static void Main(string[] args)
{
    List<int> ints = new List<int>();
}
```

 class System.Collections.Generic.List<T>
Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
T is int

Wygląda znajomo? Wygląda podobnie do przykładu wyżej?

Dzięki szablonom typ przechowywanej wartości w liście może być dynamicznie dobrany. Nie trzeba pisać implementacji dla każdego typu. Podobnie jak w STL-u z C++.