Wykład 6

dr inż. Agnieszka Rybarczyk

Instytut Informatyki
Politechnika Poznańska
ul. Piotrowo 2
tel. 61-6653029

e-mail: agnieszka.rybarczyk@cs.put.poznan.pl
http://www.cs.put.poznan.pl/arybarczyk

Wzorce (szablony) funkcji

- **Poniższe funkcje wykonują tą samą pracę, z tą różnicą, że wykonują ją na innych typach obiektów (powielanie kodu).**

```
int getMaxVal (int a, int b) {
    return (a > b ? a : b);
}
//dla porównywania liczb typu float trzeba zdefiniować
//odrębną funkcję
float getMaxVal (float a, float b) {
    return (a > b ? a : b);
}

long getMaxVal (long a, long b) {
    return (a > b ? a : b);
}

double getMaxVal (double a, double b) {
    return (a > b ? a : b);
}
```

Wzorce (szablony) funkcji

- Możliwa jest automatyzacja generowania zestawu takich funkcji za pomocą makrodefinicji (makra):

```
#define maxVal(a, b) (((a)>(b)) ? (a) : (b))
// …
int main(){
    int m = 4, k = 6;
    float x = 4.7, y = 4.1;
    char c = 'c', g = 'A';
    cout << "Wieksza liczba calkowita to " << maxVal(m, k) << endl;
    cout << "Wieksza liczba rzeczywista to " << maxVal(x, y) << endl;
    cout << "Wiekszy znak to " << maxVal(c, g) << endl;
    return 0;
}
```

  - Preprocesor zastępuje wszystkie wystąpienia nazwy makra jego rozwinięciem z definicji, wstawiając konkretne wartości parametrów (mechaniczne zamienianie jednego stringu na drugi).
  - Nie sprawdza typów argumentów.
  - Powoduje nieoczekiwane efekty uboczne – kompilator nie gwarantuje, kiedy dokona postinkrementacji w wyrażeniu.
  - Stosowany głównie w działaniach.

Wzorce (szablony) funkcji

- Szablon jest mechanizmem do tworzenia rodziny bardzo podobnych funkcji, identycznych w działaniu, ale różniących się tylko typem argumentów.
- Rozwiązanie problemu powielania kodu w przypadku funkcji różniących się jedynie typem obiektów (nieformalny zapis):

```
typ getMaxVal (typ a, typ b) {
    return (a > b ? a : b);
}
```

  - Jest to szablon, według którego można definiować potrzebną funkcję
  - Wystarczy *typ* podmienić na typ nas interesujący.
- Wzorzec funkcji definiujemy jako globalny, tzn. na zewnątrz wszystkich funkcji klas. Podobnie jak zwykłą funkcję globalną.
- Nie można zagnieżdżać definicji wzorców (szablonów).
- Nazwa wzorca musi być unikalna w całym programie.

Wzorce (szablony) funkcji

- **Definicja szablonu funkcji:**

```
template <class typ>      //lista parametrów wzorca
typ getMaxVal (typ a, typ b) {
    return (a > b ? a : b);
}
```

  - W nawiasach <> jest lista parametrów (może ich być wiele), lista nie może być pusta.
  - Czytamy: szablon w którym zastępowanym tekstem będzie nazwa typu *typ.*
  - Zastępowany tekst to parametr szablonu. Parametrem szablonu funkcji może być jedynie nazwa typu (wbudowanego lub zdefiniowanego przez użytkownika).
  - Słowo *class* w definicji potwierdza tendencję w C++ aby typy definiowane przez użytkownika były traktowane tak samo jak typy wbudowane.
  - Słowo kluczowe *template* oznacza, że nie jest to zwykły kod, tylko szablon.

Wzorce (szablony) funkcji

- Jeśli tworząc szablon użyjemy dla parametru nazwy, która w programie oznacza coś innego to kolizji nie będzie, nazwa zostanie zasłonięta.

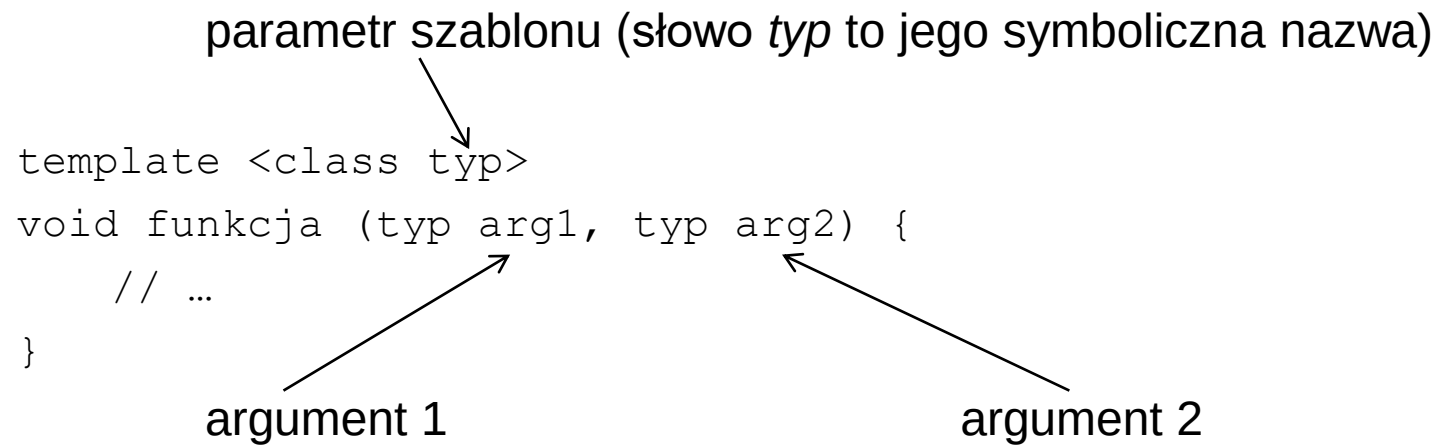
- **Wywołanie funkcji w programie:**

```
int main() {
    int a = 6, b = 0, c;
    double x = 2.7, y = 2.1;
    c = getMaxVal(a, b);
    cout << "wiekszy int: " << c << endl;
    cout << "wiekszy double: " << getMaxVal(x, y) << endl;
    cout << "wiekszy znak (kod ascii): " << getMaxVal('A', 'Z');
    return 0;
}
```

- Definicje powyższych funkcji nie istnieją w programie.
- Kompilator gdy napotka w programie wywołanie takiej funkcji, wtedy korzystając z szablonu zdefiniuje/wygeneruje ją.
- Uzyskaliśmy oszczędność kodu.

Wzorce (szablony) funkcji

- **Definicja szablonu funkcji:**



Schemat szablonu o jednym parametrze. Szablon będzie produkować funkcje szablonowe o dwóch argumentach.

- Parametr szablonu musi wystąpić jako typ argumentu funkcji definiowanej szablonem.
- Dzięki temu wywołanie funkcji szablonowej wygląda jak wywołanie zwykłej funkcji.

Wzorce (szablony) funkcji

- **Deklaracja szablonu (zapowiedź definicji)**

```
template <class typ1>
void funkcja(typ1 arg1, typ1 arg2);
```

  - Symboliczna nazwa typu w deklaracji i definicji nie musi być identyczna (jest umowna dla kompilatora).
- **Wywołanie funkcji szablonowej:**
  - Sygnałem dla kompilatora, żeby wygenerował daną funkcję szablonową jest miejsce w programie, gdzie taką funkcję wywołujemy lub gdy pytamy o jej adres.
  - Kompilator sprawdza typ argumentów wywołania danej funkcji i ten typ wstawia do szablonu (jako jego parametr aktualny). W ten sposób produkuje żądaną funkcję.
  - Przy wybieraniu który wariant funkcji szablonowej wygenerować kompilator przygląda się tylko typom argumentów umieszczonych w wywołaniu. Natomiast typ tego czemu przypisujemy wynik działania funkcji jest bez znaczenia.

Wzorce (szablony) funkcji

- **Czy dany szablon można użyć dla dowolnego typu?**
  - Nie można. Funkcja szablonowa nie musi być poprawna dla wszystkich typów. Nie można oczekiwać, że będzie można jej użyć dla każdego typu parametru.
  - Szablon funkcji można stosować dla tych typów danych, dla których poprawne są wszystkie operacje zdefiniowane na obiektach tych typów w ciele szablonu.
  - Definiując szablon programista odpowiada za to co znajduje się w jego ciele i czy ma to sens w przypadku konkretnych typów, np.:

```
template <class typ> int funkcja (typ a, typ b) {
    if (a > b) return a;
    else return b;
}
```

Wzorce (szablony) funkcji

```
class osoba {
    public:
    int wiek;
    char nazwisko[20];
    osoba (int w = 0): wiek(w) {}
};
int main() {
    osoba ka(23), ed(34);
    int res = funkcja(ka, ed); //błąd, kompilator nie wie jak porównywać osoby
    return 0;
}
//po przeciążeniu operatora > powyższe będzie poprawne
class osoba {
    public:
    // …
     int operator>(osoba b){
        if (wiek > b.wiek) return 1;
        else return 0;
    }
};
```

Wzorce (szablony) funkcji

- **Szablony funkcji, podsumowanie:**
  - Ideą szablonu jest zdefiniowanie funkcji przy użyciu typów ogólnych, które później zastępowane są typami konkretnymi. Można w ten sposób zdefiniować dowolną funkcję.
  - Umożliwia programowanie ogólne *(generic programming),* gdyż pozwala reprezentować rodzinę funkcji działających dla dowolnej liczby typów danych.
  - Definicja szablonu funkcji jest tylko informacją dla kompilatora, nie jest tworzony żaden obiekt w pamięci.
  - Wywołanie szablonu funkcji następuje w momencie natknięcia się w kodzie programu na pierwsze użycie funkcji odpowiadającej funkcji szablonowej lub użycie adresu takiej funkcji. W takiej sytuacji kompilator automatycznie generuje kod funkcji.
  - Proces generacji kodu funkcji na podstawie definicji szablonu i jego użycia nosi nazwę **konkretyzacji szablonu**.
  - Konkretna postać kodu funkcji (wygenerowana na podstawie wzorca i jego użycia) to **specjalizacja.**

Wzorce (szablony) funkcji

- **Wzorzec (szablon) z wieloma parametrami**
  - Funkcja szablonowa może mieć więcej argumentów.

```
template <class typA, class typB, class typC>
void fun(typA a, typA b, typB c, typB d, typC e);
```

  - Część z nich może być typów określonych jako parametry szablonu (**argumenty parametryzowane**), natomiast inne typów ustalonych **(argumenty nieparametryzowane** np. *int*).

```
template <class t_a, class t_b>
void fun(t_a cecha, t_b cena, int ilosc)
template <class A, class B>
A fun2(A a, B b, A c, int k, A *d, B &r, char z) {
 …
 //można definiować zmienne automatyczne typu parametru szablonu
 A temp;
 …
}
```

  - Na liście parametrów szablonu każdy typ może wystąpić tylko raz (jeśli funkcja szablonowa ma mieć jakieś dwa argumenty tego samego, parametryzowanego typu, to na liście parametrów szablonu nie powtarza się dwa razy tej samej nazwy typu).

Wzorce (szablony) funkcji

- Każdy z parametrów szablonu musi być poprzedzony słowem kluczowym *class*.

```
template <class tA, class tB, class tC>
```

…

nie można opuścić słów kluczowych *class*

```
template <class tA, tB, tC>
```

…

podobnie jak w definicji zwykłej funkcji

```
void fun (int a, b, c)
```

- Nie można zdefiniować dwóch lub więcej szablonów funkcji różniących się między sobą jedynie typem wyznaczanej wartości (**typ zwracanej wartości przez funkcję nie jest brany pod uwagę przez kompilator przy generowaniu funkcji z szablonu**).

Wzorce (szablony) funkcji

- Przykład szablonu z dwoma parametrami:

```
/*parametrami szablonu są 2 typy (typ1 i typ2). Oba, według zasady, pojawiają się także jako typy argumentów funkcji szablonowej*/
template <class typ1, class typ2>
typ2 wieksza (typ1 a, typ2 b) {
 return (a > b ? a : b);
}
int main() {
    int a1 = 23;
    double a2 = 34.6;

    /*miejsce w programie, gdzie posługujemy się funkcją szablonową (z argumentami int i double). Kompilator sprawdzi czy jest definicja tej funkcji, nie znajdzie jej ale znajdzie szablon. Wygeneruje następującą funkcję: double wieksza(int, double) – typ zwracanej wartości jest double, ponieważ jako drugi argument wywołania funkcji postawiono obiekt typu double i takiego typu jest też zwracany wynik (jak drugi argument funkcji)*/
    cout << "wieksza jest " << wieksza(a1, a2) << endl;
```

Wzorce (szablony) funkcji

```
    /*Kompilator ponownie wygeneruje nam funkcję na bazie
    szablonu, tym razem z pierwszym argumentem typu double a
    drugim typu int. Typ zwracanej wartości przez funkcję będzie
    taki jak jej drugiego argumentu czyli int.

    int wieksza(double, int)

    Jeśli argument funkcji a2 typu double będzie miał większą
    wartość to funkcja zwróci tą wartość ale odetnie część
    ułamkową liczby.*/

    cout << "wieksza jest " << wieksza(a2, a1) << endl;


    return 0;
}
```

- Można byłoby ten problem rozwiązać definiując drugi szablon różniący się tylko zwracaną przez funkcję wartością (aby uniknąć odcięcia części ułamkowej). Nie można jednak tego zrobić.

Wzorce (szablony) funkcji

- **Szablony będące szczególnymi przypadkami innych szablonów.**
  - Przykładowo, czy w programie mogą być zdefiniowane oba szablony obok siebie:

```
template <class typ>
typ wieksza(typ a, typ b);
```

    oraz szablon

```
template <class typ1, class typ2>
typ2 wieksza(typ1 a, typ2 b);
```

    - Mogą, ale ten drugi nadaje się do wyprodukowania takiej funkcji jak ten pierwszy.
    - Szablon z jednym parametrem jest szczególnym przypadkiem szablonu z dwoma parametrami (ponieważ dwa parametry mogą czasami oznaczać dwa razy ten sam typ).
    - W przypadku wywołania funkcji:

```
wieksza(1, 3);
```

    Kompilator nie będzie wiedział, którego szablonu użyć (oba będą pasować) i zgłosi błąd.
  - Tworząc w programie dwa szablony (o tej samej nazwie) należy uważać, żeby jeden nie był szczególnym przypadkiem drugiego.

Wzorce (szablony) funkcji

- **Przykład** – funkcja zamieniająca wartości dwóch liczb typu *int* – wersja z wykorzystaniem wskaźników.
  - Bez użycia szablonu.

```
void zamien (int* a, int* b) {
 int tmp;
 tmp = *b;
 *b = *a;
 *a = tmp;
}
```

  - Z wykorzystaniem wzorca generującego funkcję zamieniającą dwie liczby dowolnego (ale identycznego) typu.

```
template <class typ_liczb>
void zamien (typ_liczb* a, typ_liczb* b) {
 //należy pamiętać, że typ takiej zmiennej lokalnej jest taki
 //jak parametr szablonu – uwaga np. jak się go używa w pętli for
 typ_liczb tmp;
 tmp = *b;
 *b = *a;
 *a = tmp;
}
```

Wzorce (szablony) funkcji

- **Szablony funkcji a przydomki *inline*, *static* i *extern*.**
  - Można stworzyć szablony, które będą generować rodziny funkcji *inline*, *static* czy *extern*.
  - Aby to osiągnąć należy umieścić odpowiednie słowo kluczowe (*inline*, *static* lub *extern*) przed nagłówkiem funkcji szablonowej i po liście parametrów szablonu.

```
//generuje funkcje typu inline
template <class typ>
inline typ wiekszaI(typ a, typ b);


//generuje funkcje statyczne
template <class typ>
static typ wiekszaS(typ a, typ b);


/*Słowo extern w deklaracji zwykłej funkcji oznacza, że
definicja deklarowanej funkcji jest gdzieś indziej, np. w
innym module programu*/
template <class typ>
extern typ wiekszaE(typ a, typ b);
```

Wzorce (szablony) funkcji

- **Specjalizacja szablonu funkcji.**
  - Obiekty statyczne w szablonie funkcji:
    - Każda funkcja szablonowa ma swoją własną zmienną statyczną.
    - Jeśli w ciele szablonu funkcji było ich kilka, to każda funkcja szablonowa ma swój własny zestaw.
    - Np.:

```
template <class typ>
void licz(typ a) {
     static int licznik; //obiekt statyczny
     ++licznik;
     cout  << a << " " << licznik << endl;
}
int main() {
     licz(2); //pierwsze uruchomienie dla typu int
     licz(4); //drugie uruchomienie dla typu int
     licz(3.23); //pierwsze uruchomienie dla typu double
     return 0;
}
```

Wzorce (szablony) funkcji

- **Specjalizacja szablonu funkcji cd.**
  - Przy definiowaniu szablonu funkcji należy unikać obiektów globalnych (bardzo zła praktyka).
  - Specjalizacja stosowana jest w następujących sytuacjach:
    - Mamy zdefiniowany szablon funkcji, który działa dla większości typów, dla których używamy funkcji (np. szablon wieksza).
    - Dla pewnego szczególnego typu stosowany ogólnie algorytm nie działa, albo działa niezgodnie z naszymi oczekiwaniami.
    - Wyjściem jest zdefiniowanie specjalnej wersji szablonu funkcji, która dla jakiegoś konkretnego typu (parametru szablonu) będzie się zachowywała inaczej niż dla reszty.
  - W ciele funkcji szablonowej nie ma możliwości dowiedzenia się dla jakiego typu parametru (parametrów) funkcja została wyprodukowana (np. *if (obiekt == *char) …*). Rozwiązanie – zdefiniowanie **funkcji specjalizowanej**.

Wzorce (szablony) funkcji

- **Specjalizacja szablonu funkcji cd.**
  - Przykład bez funkcji specjalizowanej:

```
template <class typ>
typ wieksza (typ a, typ b) {
 return (a > b ? a : b);
}

int main() {
     char str1[20] = "String 1";
     char str2[20] = "Napisano 2";
     /*wygenerowana z szablonu funkcja porówna adresy zmiennych
     str1 i str2, a nie przechowywany w nich tekst.*/
     cout << wieksza(str1, str2) << endl;
     return 0;
}
```

Wynikiem będzie zawsze wartość zmiennej *str2*.

Wzorce (szablony) funkcji

- Przykład z **funkcją specjalizowaną**:

```
template <class typ>
typ wieksza (typ a, typ b) {
 return (a > b ? a : b);
}

/*Funkcja specjalizowana. Jej wyjątkowość polega na tym, że
można byłoby ją wygenerować z powyższego szablonu.*/
char* wieksza (char* a, char* b) {
 return (strlen(a) > strlen(b) ? a : b);
}

int main() {
    char str1[20] = "String 1";
    char str2[20] = "Napisano 2";
    /*Kompilator wywoła funkcję specjalizowaną, mimo, że
    widzi, że jest też szablon. Funkcja porówna długości
    łańcuchów str1 i str2. Wynikiem będzie łańcuch w str2.*/
    cout << wieksza(str1, str2) << endl;
    return 0;
}
```

Wzorce (szablony) funkcji

- Można też zdefiniować **szablon specjalizowany** (w miejsce funkcji specjalizowanej):

```
template <>
char* wieksza <char*> (char* a, char* b) {
 return (strlen(a) > strlen(b) ? a : b);
}
```

  - Definicja szablonu specjalizowanego jest prawie identyczna z definicją zwykłej funkcji.
  - Konieczna jest fraza *template*<>, aby można było traktować ją jako specjalizację szablonu *wieksza*
  - Nagłówek funkcji w specjalizacji szablonu musi być identyczny z nagłówkiem w szablonie ogólnym (różnica dotyczy zmiany typu z *typ* na *char**).
  - Konstrukcję *wieksza<char*>* można traktować jako nazwę funkcji specjalizacji szablonu *wieksza*.

Wzorce (szablony) funkcji

- **Szablony funkcji a przeciążanie.**
  - W języku C++ przeciążenie (przeładowanie) nazwy funkcji oznacza, że dana nazwa oznacza nie jedną, ale kilka różnych funkcji. Łączy je identyczność nazwy, różni lista parametrów.
  - Szablon służy do generowania wielu funkcji o tych samych nazwach, ale różniących się typem argumentów (wielokrotnie przeciąża nazwę funkcji).
  - Szablony funkcji mogą istnieć, ponieważ istnieje przeciążanie w C++.
  - Istnienie funkcji specjalizowanej jest jeszcze jednym przeładowaniem tej nazwy.
  - Dwa szablony o tej samej nazwie to kolejne przeciążanie. Warunek – deklaracje szablonów muszą być inne, np.:

```
//dwie rodziny funkcji szablonowych
template <class typ> void fun(typ, int);
template <class typ> void fun(typ, float, int, char);
```

    Mogą istnieć obok siebie bezkonfliktowo, ponieważ nie będą produkowały funkcji o takich samych parametrach (taka sama kolejność, ilość i typ).

Wzorce (szablony) funkcji

- **Działanie szablonów funkcji – podsumowanie:**
  - Szablon to wzorzec, jego definicja nie generuje kodu (nie jest definicją funkcji)
  - Gdy w kodzie zostanie znaleziona nazwa odpowiadająca nazwie szablonu, to:
    - Kompilator analizuje kod wywołania, ustala typy jakie zostały użyte (parametry szablonu)
    - Dokonuje wyboru wersji definicji
    - Dokonuje definicji konkretnej wersji funkcji, dla konkretnych typów (parametrów szablonu) – **konkretyzacja** (tworzenie egzemplarzy)
    - Taki sposób tworzenia egzemplarzy nazywa się **konkretyzacją niejawną**, kompilator sam ustala typ dla którego szablon ulega konkretyzacji
  - Konkretyzacja niejawna – zachodzi wtedy, gdy jest to potrzebne, tzn. jest konieczność użycia funkcji lub pobrania jej adresu, a funkcja ta nie jest jeszcze zdefiniowana

Wzorce (szablony) klas

- **Działanie szablonów funkcji – podsumowanie cd.:**
  - Inna możliwość – **konkretyzacja jawna (jawne tworzenie egzemplarzy)**
    - Celem jest nakazanie kompilatorowi utworzenie konkretnej wersji funkcji (dla konkretnych typów)
    - Składnia jest następująca: podajemy typy w < >, poprzedzamy deklaracje słowem kluczowym *template*, np. `template <> unsigned suma<unsigned>( unsigned, unsigned)`
    - po takiej definicji kompilator nie przeprowadza żadnej dedukcji typu argumentów funkcji – zakłada że są wskazanego typu
    - UWAGA – nie należy w jednym programie dokonywać jawnego tworzenia egzemplarza i jawnej specjalizacji dla tego samego typu danych, czyli np.:

```
template <> unsigned suma<unsigned>( unsigned, unsigned)
unsigned suma( unsigned, unsigned)
```

Wzorce (szablony) funkcji

- **Dopasowanie w wypadku funkcji szablonowych**
  - Proceder, gdy kompilator poszukuje funkcji najlepiej nadającej się do danego wywołania nazywamy dopasowywaniem.

1. Szukanie dopasowania dokładnego (funkcji o dokładnie takiej samej nazwie i typie parametrów). Jeśli znajdzie, to dopasowanie udało się, jeśli nie to przechodzi do punktu 2.
2. Kompilator szuka szablonu, który mógłby wyprodukować funkcję o argumentach identycznych, jak występujące w wywołaniu (musi być zgodność wszystkich argumentów na liście funkcji – tych, które były parametrami szablonu i tych zwykłych np. *int*). Jeśli znajdzie, to dopasowanie się udało, jeśli nie to przechodzi do punktu 3.
3. Kompilator kontynuuje proces dopasowywania, patrząc tylko na zwykłe, nie-szablonowe funkcje i kolejno rozważa:
   - Dopasowanie dosłowne z trywialną konwersją

     czyli np. takie zamiany argumentów jak *float -> double*
   - Dopasowanie z użyciem konwersji zdefiniowanej przez użytkownika (np. operator konwersji zdefiniowany dla klasy)

Wzorce (szablony) funkcji

- **Dopasowanie w wypadku funkcji szablonowych cd.**
  - Dopasowanie do funkcji z wielokropkiem (funkcje o zmiennej liczbie argumentów).

  Jeśli ten etap się nie powiedzie to kompilator sygnalizuje błąd.

  - Przykład:

```
//deklaracja szablonu
template <class typ>
void fun(typ, int, typ);
```

Kompilator natrafia na takie wywołanie funkcji:

```
char c1, c2;
fun(c1, 4, c2);
```

Dopasowanie realizowane jest w następujący sposób:

- Sprawdź czy w programie istnieje deklaracja funkcji o nazwie *fun*, o dokładnie odpowiadających typach i kolejności argumentów, czyli:

```
//typ wyniku nie ma znaczenia przy dopasowywaniu
…fun(char, int, char);
```

Tak zdefiniowanej funkcji kompilator nie znajdzie w programie.

Wzorce (szablony) funkcji

- **Dopasowanie w wypadku funkcji szablonowych cd.**
  - Kompilator sprawdza tylko szablony o nazwie *fun*. Jeśli jest ich więcej to próbuje po kolei każdy z nich dopasować aż trafi na właściwy.
  - Dopasowanie funkcji specjalizowanej
    - Funkcja specjalizowana - funkcja, której deklaracja jest taka, że mogłaby zostać wygenerowana przez jakiś szablon.
      - **Dopasowanie (dokładne) do takiej funkcji zadziała w Etapie 1**.
      - Jeśli w Etapie 1 i 2 (do szablonu) nie udało się dokonać dopasowania to następuje przejście do **Etapu 3 – funkcja specjalizowana brana jest znów pod uwagę**.

Wzorce (szablony) funkcji

- **Rzutowaniem można zmniejszyć liczbę potrzebnych funkcji szablonowych.**
  - Przykładowo, w programie jest taki szablon:

    ```
    template <class typ> void funkcja(typ k);
    ```

    poniższe wywołanie w programie:

    ```
    funkcja(3.14);
    ```

    spowoduje wygenerowanie następującej funkcji szablonowej:

    ```
    void funkcja (double);
    ```

    Natomiast poniższe wywołanie w programie:

    ```
    funkcja(6);
    ```

    Spowoduje, że kompilator nie skorzysta z już wygenerowanej funkcji szablonowej, ponieważ musiałby dokonać konwersji *6 -> 6.0* (czyli *int -> double*), a tego dla funkcji szablonowych nie robi. Wyprodukuje więc nową funkcję szablonową:

    ```
    void funkcja (int);
    ```

    - Stworzy nową instancję funkcji z szablonu dla każdego innego typu np. *short, char etc.*
    - Powoduje to rozrastanie się programu w pamięci.

Wzorce (szablony) funkcji

- **Rzutowaniem można zmniejszyć liczbę potrzebnych funkcji szablonowych cd.**
  - Problem ten można rozwiązać za pomocą rzutowania (konwersji), np.:

```
template <class typ> void funkcja(typ k);
//…
int main() {
 int i = 8;
 double pi = 3.14;
 //kompilator wygeneruje funkcja(double)
 funkcja(pi);
 /* kompilator wykorzysta funkcja(double), oszczędzamy generowania funkcja(int)*/
 funkcja(double(i));
 return 0;
 }
```

  - Mechanizm rzutowania w takiej sytuacji wykorzystujemy tylko wtedy, gdy nam się to opłaca, czyli gdy działanie funkcji *funkcja(double)* nas zadowala.

Wzorce (szablony) funkcji

- Przykład:

```
//wartość mniejsza z dwóch
template <class T> T minimum (T Pierwsza, T Druga) {
 return Pierwsza < Druga ? Pierwsza : Druga;
};
//wartość większa z dwóch
template <class T> bool greater1 (T Pierwsza, int Druga) {
 return Pierwsza > Druga;
};
int main() {
    int a = 5;
    double x = -112.8;
    cout << minimum(a, -17) << endl;         // wersja int
    cout << minimum(x, a);                     // błąd
    cout << minimum(x, (double)a) << endl;   // wersja double

    cout << greater1(a, 5) << endl;  // wersja int, int
    cout << greater1(x, a) << endl;  // wersja double, int
    cout << greater1(a, x) << endl;  // wersja int, int
}
```

Wzorce (szablony) klas

- Prosty przykład – koszyk na liczby.

```
class koszyk_int {
     int liczba;
public:
     int get() {return liczba;}
     void put(int a) {liczba = a;}
};

class koszyk_char {
     char liczba;
public:
     char get() {return liczba;}
     void put(char a) {liczba = a;}
};
```

- W podobny sposób trzeba byłoby zdefiniować kolejne klasy dla innych typów: *long, double, float* etc.
- Definicje tych wszystkich klas mogą być zastąpione definicją szablonu (wzorca) klas.

Wzorce (szablony) klas

- Jeśli mamy zdefiniować kilka podobnych klas to możemy posłużyć się szablonem klasy.
- W momencie korzystania z takiej klasy np. tworzenia obiektu tej klasy, kompilator wygeneruje potrzebną klasę ze wzorca.
- **Definicja szablonu:**

```
//szablon klas o nazwie nazwa_klasy
template <class typ>
class nazwa_klasy {
 typ zmienna1, zmienna2;
 nazwa_klasy (...); //konstruktor
 ~nazwa_klasy (...); //destruktor
 typ metoda1(...);
 void metoda2(typ zmienna3);
 ...
};
```

Lista parametrów wzorca (zamiast słowa *class* można użyć *typename*)

  - Szablon klas, który będzie służył do generowania klas różniących się parametrem o umownej nazwie *typ*.
  - Słowo *typ* będzie oznaczać czasem np. *float, int* (jakiś typ wbudowany), czasem klasę zdefiniowaną przez użytkownika.

Wzorce (szablony) klas

- Parametr w ostrym nawiasie w definicji to parametr formalny szablonu.
- Pod ten parametr będą podstawiane różne typy, zależnie od potrzeb. Te typy będą się stawać parametrem aktualnym szablonu (takim, dla którego kompilator ma aktualnie wygenerować odpowiednią klasę).
- Klasy, które powstaną z szablonu nazywane będą klasami szablonowymi.
- **Nazwa konkretnej klasy szablonowej** to nazwa szablonu oraz stojąca bezpośrednio za nią w „ostrym” nawiasie, nazwa parametru aktualnego:

  nazwa_szablonu <parametr_aktualny>

- W celu utworzenia obiektu klasy szablonowej wystarczy napisać:

```
nazwa_szablonu <parametr_aktualny> obiekt;
//przykładowo
koszyk <int> k1;
```

- Nazwa szablonu (wzorca) musi być unikalna w całym programie.
- Wzorzec klasy musi być zdefiniowany globalnie.
- **Definicji szablonów klas nie można zagnieżdżać i przeciążać.**

Wzorce (szablony) klas

- Szablon klas, to mechanizm do automatycznego pisania klas.
- Przykład szablonu klas dla koszyka na liczby:

```
template <class typ>
class koszyk {
     typ liczba;
 public:
     typ get() {return liczba;}
     void put(typ a) {liczba = a;}
};
int main() {
    /*definiujemy obiekt klasy koszyk. Nie ma definicji takiej
    klasy, więc kompilator szuka szablonu o nazwie koszyk i jednym
    parametrze. Utworzy z szablonu koszyk klasę koszyk<int>*/
    koszyk <int> k1;
    koszyk <double> k2;
    koszyk <char> k3;
    /*kompilator sprawdza czy jest definicja klasy koszyk<int> i
    orientuje się, że jest. Jest to klasa szablonowa, którą już
    zdefiniował wcześniej na bazie szablonu. Nie generuje nowej.*/
    koszyk <int> k5;
```

Wzorce (szablony) klas

```
//definicja koszyka wskaźników do obiektów typu int
koszyk <int*> k4;


//mniej oczywiste przykłady definicji
/*definicja klasy koszyk, która zawiera wskaźniki do obiektów
zdefiniowanej przez użytkownika klasy wizytowka*/
koszyk <wizytowka*> wiz;


/*t jest 10-cio elementową tablicą koszyków wizytówek.
Powstaje tablica, której elementy są obiektami klasy koszyk.
Każdy obiekt klasy koszyk zawiera obiekty zdefiniowanej przez
użytkownika klasy wizytowka*/
koszyk <wizytowka> t[10];
```

Wzorce (szablony) klas

```
    int x = 8;

    /* praca na obiekcie klasy szablonowej odbywa się tak jak na
    obiekcie zwykłej klasy */
    k1.put(3);
    k2.put(4.56);
    k3.put('f');
    k4.put(&x);

    cout << "zawartosc koszyka int: " << k1.get() << endl;
    cout << "zawartosc koszyka double: " << k2.get() << endl;
    cout << "zawartosc koszyka char: " << k3.get() << endl;
    cout << "zawartosc koszyka *int: " << *k4.get() << endl;
    return 0;
}
```

- W ciele szablonu parametr może wystąpić dowolną ilość razy, a na liście parametrów szablonu <…> pojawia się raz.
- Używając szablonów klas zmniejszamy tylko ilość kodu źródłowego, który musielibyśmy napisać. Wersja skompilowana będzie równie duża (kompilator wygeneruje potrzebne klasy).

Wzorce (szablony) klas

- **Szablony podobne są do makrodefinicji** (definiowane dyrektywą *#define*), różnice:
  - Wywołanie makrodefinicji w programie powoduje, że preprocesor w danym miejscu w programie napisze nam żądaną definicję klasy. W technice szablonowej kompilator podejmuje decyzję czy i kiedy zdefiniować odpowiednią klasę.
  - Dla makrodefinicji definicja obiektu wygląda tak, jak dla zwykłych klas (klasę utworzył już preprocesor). W przypadku szablonów trzeba użyć nazwy klasy szablonowej przy definicji obiektu, a kompilator podejmuje decyzję czy generować klasę czy skorzystać z już wygenerowanej.
  - Od czasu gdy wymyślono mechanizm szablonu klas, posługiwanie się w tym celu makrodefinicjami jest niepotrzebne. Są one gorsze i trudniejsze w użyciu (po to wymyślono szablony).
- Klasy szablonowe powstałe z tego samego szablonu nie mają ze sobą nic wspólnego. Są to różne klasy i nie łączy ich żaden związek (np. dziedziczenie). Można to zrozumieć przez analogię do makrodefinicji.

Wzorce (szablony) klas

- **Parametrem szablonu klasy może być:**
  - Nazwa typu

```
template <class typObj> …
```

  - Stałe wyrażenie będące:
    - Wartością całkowitą.

```
//definicja szablonu, w którym drugim parametrem jest stała
template <class typObj, int rozmiar>
class schowek {
    typ tablica[rozmiar];
    //…
};
//można w programie mieć klasy o różnych, wybranych
//rozmiarach tablic
schowek<int, 30> maly;
schowek<int, 300> duzy;
```

    Wymagana jest dokładna zgodność parametrów. Drugi parametr szablonu musi być *int* – w pp. szablon nie zostanie użyty.

    - Adresem funkcji globalnej.
    - Adresem statycznego składnika klasy.

Wzorce (szablony) klas

- **Parametrem szablonu klasy może być cd.:**
  - Adresem obiektu globalnego (znanym w danym miejscu lub możliwym do zobaczenia dzięki deklaracji *extern*)

```
struct struktura{
 char* nazwa;
 struktura (char *txt) { nazwa = txt;}
 void pisz() {cout << nazwa << endl;}
};
//parametrem szablonu jest adres obiektu, tu struktury
template <struktura *adres>
class klasa {
 public:
     void pisz(){ cout << "Struktura: "; adres->pisz(); }
}; //deklaracje w programie
struktura str1("Struktura1"), str2("Struktura2"); //globalnie
//w ramach funkcji main
klasa<&str1> kl1; //deklaracja w programie
klasa<&str2> kl2;
kl1.pisz();
kl2.pisz();
```

Wzorce (szablony) klas

- **Parametrem szablonu klasy nie może być:**
  - Stała dosłowna będąca stringiem
  - Adres jakiegoś elementu tablicy
  - Adres niestatycznego (czyli zwykłego) składnika klasy
  - Typ (klasa), który jest zdefiniowany lokalnie (np. definicja klasy zagnieżdżona w środku jakiejś funkcji)
  - Stała dosłowna, gdy szablon oczekuje parametru będącego obiektem (wymagałoby to założenia obiektu chwilowego dla stałej)
- **Definiowanie funkcji składowych szablonu klas**
  - Są dwa sposoby, tak jak dla zwykłych klas:
    - Definiowanie funkcji składowych wewnątrz definicji (ciała) klasy.
    - Definiowanie funkcji składowych na zewnątrz definicji (ciała) klasy. Wewnątrz pozostawiona jest tylko jej deklaracja.
  - W przypadku definiowania funkcji na zewnątrz klasy, definicja takiej funkcji jest jakby definicją szablonu funkcji, o takich samych parametrach jak szablon klas.

Wzorce (szablony) klas

- **Definiowanie funkcji składowych szablonu klas cd.**
  - Definicja takiej funkcji:

```
template <class typ>
typ_zwracany nazwa_szablonu_klasy::nazwa_funkcji_składowej(argumenty)
{
     //ciało funkcji
}
```

  - Przykład szablonu klas dla koszyka na liczby :

```
template <class typ>
class koszyk {
     typ liczba;
 public:
     //definicja konstruktora
     koszyk(typ l = 0);
     //zwykłe funkcje
     typ get();
     void put(typ a);
};
```

Wzorce (szablony) klas

```
template <class typ>
koszyk<typ>::koszyk(typ l = 0): liczba(l){}

template <class typ>
typ koszyk<typ>::get() {return liczba;}

template <class typ>
void koszyk<typ>::put(typ a) {liczba = a;}
```

- Jeden szablon klas ma odpowiadający mu jeden szablon stosownej funkcji składowej. Jeśli z tego szablonu klasy powstanie jakaś klasa szablonowa, to ma ona tylko jeden wariant danej funkcji składowej (składnikiem wyprodukowanej klasy szablonowej nie jest szablon funkcji, tylko jakaś konkretna funkcja szablonowa).
- Definiowanie i wywołanie konstruktora
  - Nazywa się tak samo jak klasa szablonowa, kompilator z poniższej definicji wie o konstruktor której klasy chodzi:

```
szablon <parametry>::szablon(argumenty)
```

Wzorce (szablony) klas

- **Definiowanie funkcji składowych szablonu klas cd.**
  - Definiowanie i wywołanie konstruktora cd.
    - Wywołanie konstruktora, np.:

```
koszyk<int> k1(24);
```

    - Zapis *koszyk<int>* mówi kompilatorowi o jaki typ nam chodzi, Zapis *koszyk* spowodowałby, że kompilator nie wiedziałby czy chodzi o np. *koszyk<int>*, czy też np. *koszyk<long double>*
  - Podobnie wygląda to w przypadku definiowania destruktora.

```
template <class typ>
koszyk<typ>::~koszyk(){
 cout << "Jestem destruktorem" << endl;
}
```

Wzorce (szablony) klas

- Przykład:
  - Zdefiniować wzorzec klasy o nazwie *tablica* umożliwiającej przechowywanie tablicy liczb dowolnego typu i tablic znakowych. Rozmiar tablicy jest podawany w momencie tworzenia obiektu tej klasy. Każda tablica przechowuje również swój zadany rozmiar. Zdefiniować również konstruktor, destruktor i metodę wyświetlającą zawartość tablicy.

```
template <class typ>
class tablica {
     typ* dane;
     int rozmiar;
 public:
     tablica(int p_rozmiar) : rozmiar(p_rozmiar)
     { dane = new typ [p_rozmiar]; };
     ~tablica() { delete [] dane; };
     void wyswietl() {
          for (int i=0; i<rozmiar; i++)
          cout<<"dane["<<i<<"]="<<*(dane+i)<<endl;
     };
};
```

Wzorce (szablony) klas

```
void main() {
     //wskaźnik do obiektów klasy szablonowej
     tablica<int>* tI = new tablica<int>(4);
     tablica<char>* tC = new tablica<char>(6);
     tablica<int> tI1(5); //obiekt statyczny
     tablica<char> tC1(7); //obiekt statyczny

     tI->wyswietl();
     tC->wyswietl();
     tI1.wyswietl();
     tC1.wyswietl();

     delete tI;
     delete tC;
     return 0;
}
```

Wzorce (szablony) klas

- **Kiedy kompilator sięga po szablon klas (generuje klasę szablonową)**
  - Definicja obiektu klasy szablonowej

```
koszyk<int> zm1, zm2, zm3;
```

  - Definicja wskaźnika mogącego pokazywać na obiekty klasy szablonowej

```
//wsk jest wskaźnikiem mogącym pokazywać na obiekty klasy
//koszyk<float>
koszyk<float> *wsk;
```

  - Deklaracja funkcji zawierającej nazwę klasy szablonowej

```
void funkcja (char t, koszyk<float> zm);
```

  - Nazwa klasy szablonowej użyta jako nazwa klasy podstawowej w dziedziczeniu.

```
class pojemnik : koszyk<int>{
    //ciało funkcji
};
```

Wzorce (szablony) klas

- **Obiekt klasy szablonowej może być składnikiem innego szablonu klas.**

```
template <class typ> //pierwszy szablon
class stol {
    //…
};
//drugi szablon, korzysta z pierwszego
template <class typ>
class pokoj {
    stol<float> drewniany;
    /*typ stolu będzie zależał od tego jakiej konkretnie klasy
    szablonowej zażądamy*/
    stol<typ> metalowy;
    //…
};
int main() {
    //w realizacji klasy szablonowej stol<typ> metalowy -> stol<char>
    pokoj<char> lewy;
    pokoj<int> prawy;
    //…
}
```

Wzorce (szablony) klas

- **Zagnieżdżenie definicji klasy w szablonie**
  - Szablon może być definiowany tylko w zakresie globalnym, więc nie można umieścić definicji szablonu wewnątrz innej klasy lub szablonu klas.
  - **W szablonie można natomiast zagnieździć klasę.**

```
template <class typ1, class typ2>
class duza {
typ1 zm;
//klasa pomocnicza korzysta z parametrów szablonu klasy
class pomocnicza {
    typ1 lewy;
    typ2 prawy;
    void pisz();
};
pomocnicza zm2;
//…
};
```

Efekt takiego zagnieżdżenia klasy w szablonie jest taki jakbyśmy w szablonie klas zagnieździli inny szablon klas.

Wzorce (szablony) klas

- **Parametry szablonu klas.**
  - Szablon klasy może być parametrem innego szablonu klas. Możliwa jest też rekursja np.:

```
template <class typ, int rozmiar>
class tablica {
     typ tab[rozmiar];
 public:
     tablica (){ };
     //…
};
tablica< tablica< double, 5>, 8> tab2;
```

    dla powyższego szablonu klas do obsługi tablic jest równoważne:

```
double tab2[8][5] ;
```

  - Szablon klas może zawierać kilka parametrów.
  - Możliwe jest określenie domyślnych wartości parametrów szablonu klas (jeśli parametry te zostaną pominięte w wywołaniu to zostaną podstawione domyślne wartości). Tak jak dla funkcji parametry domyślne powinny być jako ostatnie na liście.

Wzorce (szablony) klas

- **Parametry szablonu klas cd.**
  - Przykład z domyślnymi parametrami w szablonie:

```
template <class T1, class T2=double, int n = 2>
class Pojemnik {
 T1 t_1;
 T2 t_2;
public:
 Pojemnik(T1 a, T2 b): t_1(a), t_2(b){}
 T2 wartosc() {return (T2) pow(t_2, n);}
 T1 nazwa() {return t_1;}
};
int main() {
Pojemnik<string> p1("Ala", 12.3);
Pojemnik<string, float, 3> p2("Antek", 12.3);
Pojemnik<string, long double, 4> p3("Ola", 12.3);

cout << p1.nazwa() << ", wartosc: " << p1.wartosc() << endl;
cout << p2.nazwa() << ", wartosc: " << p2.wartosc() << endl;
cout << p3.nazwa() << ", wartosc: " << p3.wartosc() << endl;
}
```

Wzorce (szablony) klas

- **Specjalizacja, a szablon klasy**
  - Przykład:

```
template <class typ>
class akumulator{
 typ zloze;
public:
 akumulator(): zloze(0){}
 void dodaj(typ co){ zloze += co;}
 typ wydaj();
};

template <class typ>
typ akumulator<typ>::wydaj(){
 typ tmp = zloze;
 zloze = 0;
 return tmp;
};
```

Wzorce (szablony) klas

- **Specjalizacja, a szablon klasy cd.**

```
//definicja specjalizowanej klasy szablonowej, parametrem
//aktualnym jest char*
template <> class akumulator<char*>{
 char* zloze;
public:
 akumulator(): zloze(NULL){}
 void dodaj(char* co);
 char* wydaj(){
     char* tmp = zloze;
     zloze = 0;
     return tmp;
 }
 //konieczne było zdefiniowanie destruktora – żeby zwolnił pamięć
 ~akumulator(){
     if (zloze) delete zloze;
 }
};
```

Wzorce (szablony) klas

- **Specjalizacja, a szablon klasy cd.**

```
/* definicja funkcji na zewnątrz klasy. Nie ma tutaj słowa
template, dlatego, że jest to funkcja składowa konkretnej klasy,
a nie szablonu */
void akumulator<char*>::dodaj(char* co) {
 if (!zloze) {
      zloze = new char[strlen(co)+1];
      zloze[0] = NULL;
 } else {
      char *old = zloze;
      //rezerwacja pamięci na tekst stary i nowy
      zloze = new char[strlen(old)+strlen(co)+1];
      strcpy(zloze, old);
      delete old;
 }
 //doklejenie do starego tekstu nowego
 strcat(zloze, co);
}
```

Wzorce (szablony) klas

- **Specjalizacja, a szablon klasy cd.**

```
int main() {
    akumulator<float> kasa;
    kasa.dodaj(120.01);
    cout << "Wyplac: " << kasa.wydaj() << endl;

    //kompilator nie generuje klasy z szablonu, tylko
    //wykorzystuje klasę specjalizowaną
    akumulator<char*> opis;
    opis.dodaj("Ala ma kota");
    opis.dodaj(" i psa");
    cout << "Wypisz: " << opis.wydaj() << endl;
    return 0;
}
```

  - **Definicja specjalizowanej klasy szablonowej może wystąpić w programie dopiero po definicji szablonu, który ma „ulepszać”.**
    - Inaczej błąd kompilacji.
    - Kompilator musi sprawdzić czy klasa szablonowa o takiej nazwie mogłaby rzeczywiście powstać z danego szablonu.

Wzorce (szablony) klas

- **Specjalizacja, a szablon klasy cd.**
  - Jeśli zdecydujemy się sami definiować specjalizowaną klasę szablonową, to musimy zdefiniować ją w całości.
    - Nie można oczekiwać od kompilatora, że część klasy zrobi z szablonu.
  - Specjalizowana klasa szablonowa nie musi mieć takich samych składników, jakie występują w ciele szablonu.
    - Można zdefiniować dodatkowe składowe (np. dodatkowa funkcja składowa, brak definicji jakiejś funkcji z szablonu)
  - **Można też zdefiniować jako specjalizowaną tylko funkcję składową szablonu klas**, np.:

```
void akumulator<long>::wydaj(long co) {
    zloze += co;
    cout << "Zloze: " << zloze << endl;
}
```

Jeśli dla pewnego specjalnego parametru klasa szablonowa zachowuje się dobrze, a tylko jedna/kilka funkcji powinna działać w tym specjalnym przypadku inaczej, to można te funkcje zdefiniować jako specjalizowane.

Wzorce (szablony) klas

- **Specjalizacja, a szablon klasy cd.**
  - Kompilator generuje z szablonu klasę szablonową. Wygeneruje też wszystkie potrzebne jej funkcje składowe (wszystkie oprócz tych, które zostały zdefiniowane jako specjalizowane).
- **Specjalizacja szablonów klas – podsumowanie:**
  - Szablon klasy jest jedynie przepisem na zdefiniowanie klasy, nie jest jeszcze definicją.
  - Kompilator generuje kod definicji gdy zajdzie taka potrzeba (np. wtedy gdy potrzebujemy obiektu kasy szablonowej) – **konkretyzacja niejawna**.
  - W procesie konkretyzacji niejawnej kompilator wykonuje tyle pracy, ile to konieczne i tak późno jak to jest możliwe (**podejście leniwe**). W przypadku klas konkretyzacji podlegają tylko te metody, które są faktycznie używane.
  - Powoduje to, że generowany jest możliwie najkrótszy kod. Zwykle oznacza to, że programy są mniejsze i działają szybciej.
  - Oprócz konkretyzacji niejawnej możliwe jest wymuszenie konkretyzacji we wskazanym miejscu i czasie (bez konieczności tworzenia obiektów) – **konkretyzacja jawna**.

Wzorce (szablony) klas

- **Specjalizacja szablonów klas – podsumowanie, cd. :**
  - Jawna specjalizacja stosowana jest gdy:
    - mamy działający szablon klas
    - chcemy, by dla pewnego zestawu parametrów szablonu klasa była zdefiniowana inaczej (”nieszablonowo”) (lub też – klasa nie da się zdefiniować poprzez szablon)
    - definicja takiej klasy poprzedzona deklaracją

      `template <> class NazwaKlasy<konkretne parametry>`
  - Możliwa jest **specjalizacja częściowa**, która polega na zastąpieniu części parametrów szablonu typami konkretnymi.
  - Typy, które jeszcze nie są ustalone – wypisane w nawiasach ostrych po słowie kluczowym *template*
    - Przykłady:

```
//wersja ogólna
template <class T1, class T2, class T3> class Moja { … }
//wersja częściowo specjalizowana: T2=int
template <class T1, class T3> class Moja<int> { … }
//wersja częściowo specjalizowana: T2=T3;
template <class T1, class T2> class Moja<T1, T2, T2>
```

Wzorce (szablony) klas

- **Specjalizacja szablonów klas – podsumowanie, cd. :**
  - Jeżeli jest taka możliwość, to kompilator wybierze zawsze najbardziej skonkretyzowaną wersję szablonu. Np., gdy mamy następujące wersje:

```
template <class T1, class T2> class Para { … } //def. ogólna
template <class T1> class Para <T1, int> { …} //T2=int
template <> class Para<int, int> { ... } //T1=T2=int
```

to poniższe deklaracje oznaczają:

```
Para<double, double> p1; //użycie szablonu ogólnego Para
Para<double, int> p2; //użycie szablonu Para<T1, int>
Para<int, int> p3 ; //użycie jawnej konkretyzacji Para<int, int>
```

Wzorce (szablony) klas

- **Przyjaźń, a szablony klas.**
  - Klasy szablonowe mogą mieć przyjaciół (funkcje i klasy zaprzyjaźnione).
  - **Możliwe są dwie sytuacje:**

1. **Jeden wspólny przyjaciel.**

   Wszystkie klasy wygenerowane z szablonu będą miały tego samego, wspólnego przyjaciela (funkcję lub klasę).

   - **Przykład:**

```
/*wszystkie składowe są prywatne – tylko przyjaciel może utworzyć obiekt klasy szablonowej pole*/
template <int zm> class pole {
 int rola[zm];  //wielkość tablicy zależy od parametru szablonu
 pole(){cout << "Prywatny konstruktor Pola" << endl;}
 void fun(){cout << "Prywatna funkcja pola: " << zm << endl;}
 /*Klasa rolnik będzie przyjacielem wszystkich klas szablonowych tworzonych z szablonu pole – będzie miała dostęp do wszystkich ich składowych*/
 friend class rolnik;
 /*Funkcja uprawa będzie przyjacielem wszystkich klas szablonowych tworzonych z szablonu pole – jak powyżej*/
 friend void uprawa();
};
```

Wzorce (szablony) klas

```
class rolnik {
public:
 void rok() {
      cout << "Dziala zaprzyjazniony rolnik: " << endl;
      /*w tej klasie można tworzyć obiekty klasy pole jej
      prywatnym konstruktorem */
      pole<6> duze;
      duze.rola[0] = 3; //dostęp do prywatnej składowej
      duze.fun(); //można wywołać prywatną funkcję
 }
};

//funkcja przyjaciel klasy pole
void uprawa(){
 cout << "Dziala zaprzyjazniona funkcja: " << endl;
 /*w tej funkcji można tworzyć obiekty klasy pole jej prywatnym
 konstruktorem – czyli pole nie może istnieć bez rolnika lub uprawy*/
 pole<2> male;
 male.fun(); //można wywołać prywatną funkcję
}
```

Wzorce (szablony) klas

```
int main() {
 rolnik stefan;
 stefan.rok();

 uprawa();
}
```

2. **Każdy ma swojego przyjaciela, właściwego tylko jemu.**
   - Deklaracja funkcji (klasy) zaprzyjaźnionej zależy od parametrów szablonu klasy.
   - Aby dla każdej możliwej klasy szablonowej znalazł się odpowiadający jej przyjaciel, **należy zdefiniować inny szablon generujący przyjaciół**.

Wzorce (szablony) klas

```
template <class typ> class rolnik; // deklaracja zapowiadająca


/* szablony przyjaciół mają sens jeśli zależą oni od typu dla danej
klasy */
template <class typ>                // szablon klasy pole
class pole { // dla konkretnej klasy powstanie konkretny przyjaciel
    typ rola;
public:
    pole(typ a):rola(a){} // konstruktor
    void fun(){cout << "Funkcja pola: " << rola << endl;}
    /*Klasa szablonowa rolnik będzie przyjacielem wszystkich klas
    szablonowych tworzonych z szablonu pole. To jaka konkretnie
    powstanie klasa szablonowa rolnik zależy od tego jaka powstanie
    klasa szablonowa pole, np. pole<int> uzna za przyjaciela tylko
    rolnik<int>*/
    friend class rolnik<typ>; //definiowane inaczej niż dla funkcji!


    /*Funkcja szablonowa uprawa będzie przyjacielem wszystkich klas
    szablonowych tworzonych z szablonu pole. Typ argumentu funkcji
    jest parametryzowany, czyli argument jest obiektem jakiejś klasy
    szablonowej. Zaprzyjaźniona funkcja szablonowa dla np. pole<int>
    to void uprawa(pole<int> obj)*/
    template <class typ> friend void uprawa(pole<typ> obj);
};
```

Wzorce (szablony) klas

```
/*Funkcja przyjaciel klasy pole, będąca szablonem. Jej argumentem
jest obiekt jakiejś klasy szablonowej, raz to będzie np. pole<int>,
innym razem pole<char> etc.*/
template <class typ>
 void uprawa(pole<typ> obj){
      cout << "Dziala zaprzyjazniona funkcja: " << obj.rola << endl;
 }

/*Definicja szablonu klasy rolnik. Jej składnikiem jest wskaźnik do
obiektu klasy szablonowej powstałej z szablonu pole.*/
template <class typ>
class rolnik {
 pole<typ> *duze;
public:
 rolnik(pole<typ> *p):duze(p){} // konstruktor, tworzy składową duze
/*może odwołać się do prywatnej składowej klasy szablonowej pole,
bo są zaprzyjaźnione (definicja w klasie pole).*/
 void rok() {
      cout << "Dziala zaprzyjazniony rolnik: " << duze->rola << endl;
 }
};
```

Wzorce (szablony) klas

```
int main() {
 pole<int> male(10);  // tworzy obiekt typu pole
 male.fun();

/*Jeżeli w programie będzie wiele różnych klas powstałych z szablonu pole to powstanie wiele funkcji o przeładowanej nazwie uprawa*/
 uprawa(male);

/*Taką klasę rolnik<int>, klasa pole<int> uzna za swojego przyjaciela i da jej dostęp do wszystkich swoich składowych*/
 rolnik<int> stefan(&male);
 stefan.rok();
 return 0;
}
```

Wzorce (szablony) klas

- **Dziedziczenie, a szablony klas.**
  - Klasy szablonowe (powstałe z jednego szablonu) nie mają ze sobą nic wspólnego, poza tym, że pochodzą od tego samego szablonu.
  - Klasy szablonowe tak jak i inne klasy mogą być użyte w dziedziczeniu (być klasami bazowymi lub pochodnymi).
  - **Możliwe są następujące przypadki:**
    1. **Zwykła klasa może odziedziczyć klasę szablonową.**
       - Klasa szablonowa to już konkretna klasa, więc na liście pochodzenia nowej klasy umieszcza się po prostu nazwę tej klasy szablonowej, która ma być rodzicem,
       - np. chcemy odziedziczyć klasę szablonową *schowek<float>* (w pełni określona konkretna klasa, w dziedziczeniu zachowuje się jak zwykła klasa):

```
//schematycznie wygląda to następująco
class bagaznik : public schowek<float> {
    //…
};
```

Wzorce (szablony) klas

```
template <class typ> class schowek {
 typ zawartosc;
public:
 schowek(typ a) : zawartosc(a){}
 void wloz(typ a) {zawartosc=a;}
 typ wyjmij() {return zawartosc;}
};

//klasa, która dziedziczy klasę szablonową. To nie jest
//dziedziczenie szablonu!
class portfel : public schowek<float> {
 const int rozmiar;
public:
 /*wywołanie konstruktora klasy bazowej schowek<float> - portfel
 nie będzie pusty*/
 portfel(int w) : rozmiar(w), schowek<float>(0.10) {}
 void raport() {
 cout << "W portfelu jest " << wyjmij() << " zl." << endl;
 }
};
```

Wzorce (szablony) klas

```
int main() {
     portfel czarny(15);
     portfel bialy(12);

     czarny.wloz(czarny.wyjmij()+30);
     czarny.wloz(czarny.wyjmij()-2);
     bialy.wloz(bialy.wyjmij()+100);
     bialy.wloz(bialy.wyjmij()-25);

     czarny.raport();
     bialy.raport();
     return 0;
}
```

2. **Specjalizowana klasa szablonowa może odziedziczyć zwykłą klasę lub klasę szablonową.**

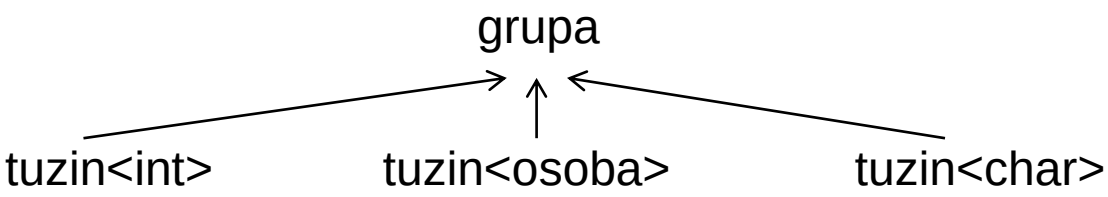
   Sytuacja, w której definiujemy sami klasę szablonową.

Wzorce (szablony) klas

3. **Szablon może dziedziczyć zwykłą klasę, inny szablon lub klasę szablonową.**

```
//szablon dziedziczy zwykłą klasę; schematycznie:
template <class typ>
class tuzin : public grupa { //grupa to nazwa klasy
    //…
};
```

Klasa *grupa* jest wspólnym rodzicem dla wszystkich klas szablonowych, które kiedykolwiek zostaną wygenerowane z tego szablonu. Może powstać więc taka przykładowa hierarchia:



Przydaje się np. w przypadku, gdy definiując szablon klasy chcemy nawiązać do klasy już istniejącej lub wykorzystać funkcje (rozbudowane) niezależne od parametrów szablonu.

Wzorce (szablony) klas

Inne przypadki:

Definiujemy	Mamy i chcemy z niej dziedziczyć		
	Klasę zwykłą	Klasę szablonową	Szablon klas
Zwykłą klasę	+	+	-
Szablon klas	+	+	+
Klasę szablonową (specjalizowaną)	+	+	-

- Do konkretnej klasy (zwykłej czy specjalizowanej) nie można odziedziczyć szablonu.
- **Dziedziczenie szablonu jest możliwe tylko od innego szablonu.**
- Klasa (zwykła lub szablonowa) może odziedziczyć tylko klasę (zwykłą czy szablonową).