Wzorce funkcji (szablony)

- Wzorce funkcji (ang. *function template*) dają możliwość wielokrotnego wykorzystywania tego samego kodu funkcji dla różnych typów danych.
- Załóżmy, że chcemy zdefiniować funkcję `min()`, która wybiera mniejszą z dwóch liczb całkowitych lub rzeczywistych.

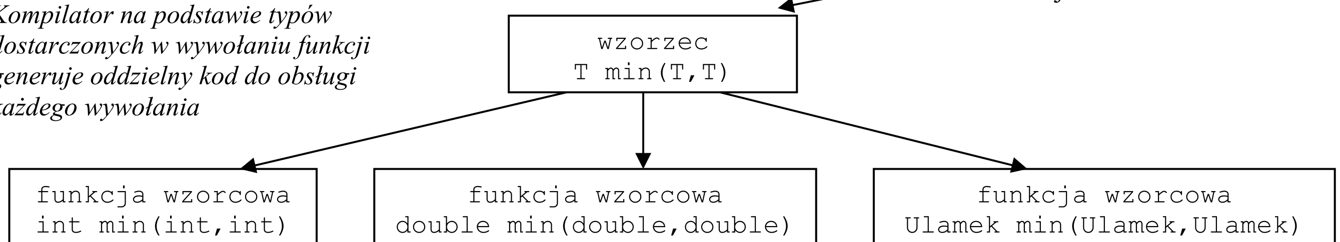
a) Podejście tradycyjne: tworzymy dwie funkcje

```
int min(int a, int b) {
  if (a<b) return a;
  else return b;
}
```

```
double min(double a, double b){
  if (a<b) return a;
  else return b;
}
```

b) Zastosowanie wzorca funkcji: tworzymy jeden kod, z którego kompilator generuje potrzebne funkcje:

```
template <class T> T min(T a, T b){
  if (a<b) return a;  // return a < b ? a : b;
  else return b;
}
```



- Analiza nagłówka wzorca min:

słowa kluczowe — *nazwa zastępcza typu danych używana w definicji wzorca, zawsze musi być poprzedzona słowem* `class`

```
template <class T> T min(T a, T b)
```

Nazwa zastępcza typu danych jest używana w definicji funkcji jako typ danych, kompilator zastąpi ją nazwą rzeczywistego typu danych podczas tworzenia określonej wersji funkcji.
Uwaga: standard C++ wprowadza nowe słowo kluczowe `typename`:

```
template <typename T> T min(T a, T b)
```

które lepiej oddaje znaczenie tej konstrukcji.

- Jak to działa: wywołanie funkcji min() z aktualną listą parametrów powoduje wygenerowanie funkcji, w której typy wzorca zostaną zastąpione typami aktualnymi, czyli zostanie wygenerowana odpowiednia wersja funkcji `int min(int, int)` lub `double min(double, double)`.
- Przykład użycia wzorca:

```
int main(){
  int i=2, j=5 ;
  double x=1.5, y=2.3 ;
  cout << "min(i,j) = "<< min(i,j) <<"\n" ;  // int min(int, int)
  cout << "min(x,y) = "<< min(x,y) <<"\n" ;  // double min (double, double)
}
```

- Pytanie: czy możemy użyć ten wzorzec dla napisów, na przykład:

```
char *wsk1="piotr", *wsk2="adam";
cout << "min(wsk1,wsk2) = " << min(wsk1,wsk2) <<endl;
```

Wzorce klas

- Wzorce klas (szablony, ang. *templates*) pozwalają realizować zależne od typu danych wersje klas ogólnych.
- Pojęcia:
  - konkretyzacja (ang. *instantiation*) - przysposobienie klasy do przechowywania konkretnych obiektów
  - parametry typów (ang. *type parameters*) - ogólne identyfikatory typów wykorzystane do definicji wzorca
- Przykład definicji klasy `Punkt`:

```
class Punkt {
  int x; int y;
public:
  Punkt(int xx=0, int yy=0)
  { x = xx ; y = yy ;}
  void DrukujWsp();
  ...
};

void Punkt::Drukuj()
{
  cout << "Wspolrzedne : " << x << " " << y << endl ;
}
```

- Chcemy tę definicję uogólnić, tak aby współrzędne mogły być różnych typów:

```
template <class T>
class Punkt {
  T x ; T y ;
public :
  Punkt (T xx=0, T yy=0)   // definicja funkcji inline
  {  x = xx ; y = yy ;}
  void Drukuj();         // funkcja zdefiniowana na zewnątrz klasy
} ;

template <class T>
void Punkt<T>::Drukuj()
{
  cout << "Wspolrzedne : " << x << " " << y << endl ;
}
```

- Przykład funkcji testującej:

```
int main ()
{
  // deklaracja obiektu parametryzowanej klasy
  Punkt<int> ai(3, 5);
  Punkt<double> ad(3.5, 2.3) ;
  ai.Drukuj();
  ad.Drukuj();
}
```

Przykład - klasa `vector`

- Klasa **`vector`** pozwala przechowywać obiekty różnych typów, nazywana jest kontenerem *(ang. container class* w języku polskim spotyka się tłumaczenia *kolekcja, kontener, zasobnik*) :

```
vector <int> a;       // wektor pusty elementów typu int
vector <double> b(2); // wektor o 2 elementach typu double
vector <string> c(5);  // wektor o 5 elementach typu string
```

- Stanowi alternatywę dla tablicy, typu wbudowanego w język C++:

```
int tab1[10];         // tablica wbudowana
vector<int> tab2(10); // tablica jako wektor o określonej długości
```

- Korzystanie z klasy **`vector`** wymaga dołączenia pliku nagłówkowego:

```
   #include <vector>
```

  i przestrzeni nazw `std`.

- Przykład: przygotowujemy książkę telefoniczną

```
// wersja a) - "tradycyjna" z tablicą wbudowaną:
struct Pozycja {
  string nazwisko;
  int numer;
};
Pozycja ksiazka_tel[1000];
void drukuj_pozycje(int i)
{
  cout << ksiazka_tel[i].nazwisko <<' '
       << ksiazka_tel[i].numer <<endl;
}
```

  Wada: tablica wbudowana ma stały rozmiar

```
// wersja b) - z wykorzystaniem klasy vector
#include <vector>
using namespace std;
struct Pozycja {
  string nazwisko;
  int numer;
};
<vector> Pozycja ksiazka_tel(1000); // wektor o 1000 elementów
void drukuj_pozycje(int i)
{
  // korzysta się tak samo jak z tablicy
 cout << ksiazka_tel[i].nazwisko <<' '
      << ksiazka_tel[i].numer <<endl;
}
// zwiększenie rozmiaru o n: funkcja resize()
 void dodaj_pozycje(int n)
 {
   ksiazka_tel.resize(ksiazka_tel.size()+ n);
 }
```

Wstawianie wartości do wektora

```
// metoda 1: za pomocą funkcji push_back()
  vector<int> v;           // pusty wektor o elementach typu int
  for (int i=0 ; i<10; i++)
    v.push_back(i);        // dodaj na końcu wektora wartość i

  vector<string> s;        // pusty wektor o elementach typu string
  string tekst;
  while ( cin >> tekst)
    s.push_back(tekst);    // dodaj na końcu wektora napis

  vector<int> v1;
  int t[4]={1,2,3,4};
  for (int i=0;i<4; ++i)
    v1.push_back(tab[i]);

// metoda 2:
  vector<int> v0 ;         // wektor pusty
  int t[4]={1,2,3,4};
  vector<int> v1(4, 99) ;  // wektor o 4 elementach typu int o wartości 99
  vector<int> v2(7, 0) ;   // wektor o 7 elementach typu int o wartości 0
  vector<int> v3(t, t+2) ; // wektor zbudowany z tablicy t:
                           // pierwsze 2 elementów
  vector<int> v4(v1);      // inicjalizacja innym wektorem
```

Działania na wektorach

```
 for (i=0 ; i<v1.size(); i++) // dostęp do elementów wektora
    cout << v[i] << " " ;
  cout << "\n";

for (i=0 ; i<v2.size(); i++) // dostęp do elementów wektora
    v2[i] = i*i ;

  v1.push_back(99) ;           // dodanie elementu na końcu
  v2.pop_back() ;              // usunięcie elementu z końca
  v3 = v2 ;                    // przypisywanie wektorów
  v1.assign (t+1, t+3) ;       // przypisywanie wartości wektorowi v1:
                               // od elementu t[1] do elementu t[3]
```

W jaki sposób wektor rośnie?

- Aby samemu zmieniać rozmiar wektora dynamicznie trzeba:
  - przydzielić nową, większą pamięć,
  - skopiować poprzednią zawartość pamięci w nowe miejsce,
  - zwolnić poprzednią, niepotrzebną już pamięć,
  - jeśli elementy są obiektami typu klasy, trzeba zdefiniować i użyć odpowiednie konstruktory i destruktory
- Jak jest to realizowane w klasie `vector` ?

  Rezygnuje się ze zwiększania wektora przy każdym wstawianiu. Jeżeli wektor musi się zwiększyć, jest on zwiększany nadmiarowo, tak, że będzie można wstawić wiele elementów bez potrzeby przydzielania pamięci. Ile wynosi nadmiar, zależy od implementacji. Dla różnych typów mogą to być różne wartości.
- Dla każdego wektora definiowany jest więc:
  - rozmiar (ang. *size*) - ilość elementów wektora,
  - pojemność (ang. *capacity*) - liczba elementów, które trzeba wstawić, zanim zwiększy się przydział pamięci.
- Przykład:

```
#include <vector>
#include <iostream>

using namespace std;
class Punkt {
int x,y;
};
int main()
{
  vector<int> x;
  cout << "x: rozmiar= " << x.size()
       << " pojemosc= " << x.capacity() << endl;
  for (int i=0; i<300; ++i) {
    x.push_back(i);
    cout << "x: rozmiar= " << x.size()
       << " pojemosc= " << x.capacity() << endl;
  }
}
```

Wyniki (g++ Linux):

```
x: rozmiar= 0   pojemnosc= 0
x: rozmiar= 1   pojemnosc= 1
x: rozmiar= 2   pojemnosc= 2
...
x: rozmiar= 8   pojemnosc= 8
x: rozmiar= 9   pojemnosc= 16   // podwojenie
...
x: rozmiar=16...pojemnosc= 16
x: rozmiar=17...pojemnosc= 32
...
x: rozmiar= 300 pojemnosc= 512
```

- Przykład: drukowanie składowych wektora rozdzielanych spacją

```
#include <iostream>
#include <vector>

using namespace std ;
void drukuj(vector<int> &);

int main() {
  int t[] = {1,2,3,4,5,6,7,8,9,10} ;
  vector<int> v(t,t+10);
  drukuj(v);
  return 0;
}
void drukuj(vector<int> &v)
{
  cout << '{' ;
  for (int i=0 ; i<v.size(); i++) // dostęp do elementów wektora
    cout << v[i] << ' ' ;
  cout << '}' << endl;
}
```

- To samo z przeciążaniem operatora wyjścia, składowe rozdzielane przecinkiem

```
#include <iostream>
#include <vector>

using namespace std ;
ostream &operator<<(ostream &,const vector<int> &);

int main()
{
  int t[] = {1,2,3,4,5,6,7,8,9,10} ;
  vector<int> v(t,t+10);
  cout << v << endl;
  return 0;
}

ostream &operator<<(ostream &S, const vector<int> &v){
  S<<'{';
  for(int i=0;i<v.size();i++)
  {
    if(i) S <<',';
      S <<v[i];
  }
  S <<'}';
  return S;
}
```

- Przykład: wczytanie pliku do wektora; każdy element wektora zawiera jeden wiersz tekstu

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream we("plik.txt");
  string wiersz;
  while(getline(we, wiersz, '\n'))
    v.push_back(wiersz);
  for (int i=0; i<v.size(); i++)
    cout << i << ": " << v[i] << endl;
}
```

- Przykład: wczytanie pliku do wektora, każdy element wektora zawiera pojedynczy wyraz

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream we("plik.txt");
  string wyraz;
  while(we >> wyraz)
    v.push_back(wyraz);
  for (int i=0; i<v.size(); i++)
    cout << i << ": " << v[i] << endl;
}
```

- Wyświetl plik tekstowy wierszami od końca, numeruj wiersze

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream we("plik.txt");
  string wiersz;
  while(getline(we, wiersz, '\n'))
    v.push_back(wiersz);
  int licznik = v.size();
  for(int i = 0; i < licznik; i++) {
    int nrWiersza = licznik-i-1;
    cout << nrWiersza << ": " << v[nrWiersza] << endl;
  }
}
```

- Wyświetl plik tekstowy wyraz po wyrazie (wyrazy są oddzielane spacjami)

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream we("plik.txt");
  string slowo;
  while(we>>slowo)
    v.push_back(slowo);
  int licznik = v.size();
  for(int i = 0; i < licznik; i++) {
    cout << v[i] << endl;
  }
}
```

- Przykład: wyszukiwanie binarne, gdy tablica jest zdefiniowana jako obiekt klasy.

```
const int nieZnaleziono=-1;

int binSzukaj(const vector<int> &t, int x) {
  int dolnaGranica=0;
  int gornaGranica=t.size()-1;
  while (dolnaGranica <= gornaGranica)
  {
    int aktInd = (dolnaGranica +
                  gornaGranica)/2;
    if (x==t[aktInd]) return aktInd;
    if (x<t[aktInd])
      gornaGranica=aktInd-1;
    else
      dolnaGranica=aktInd+1;
  }
  return nieZnaleziono;
}
```