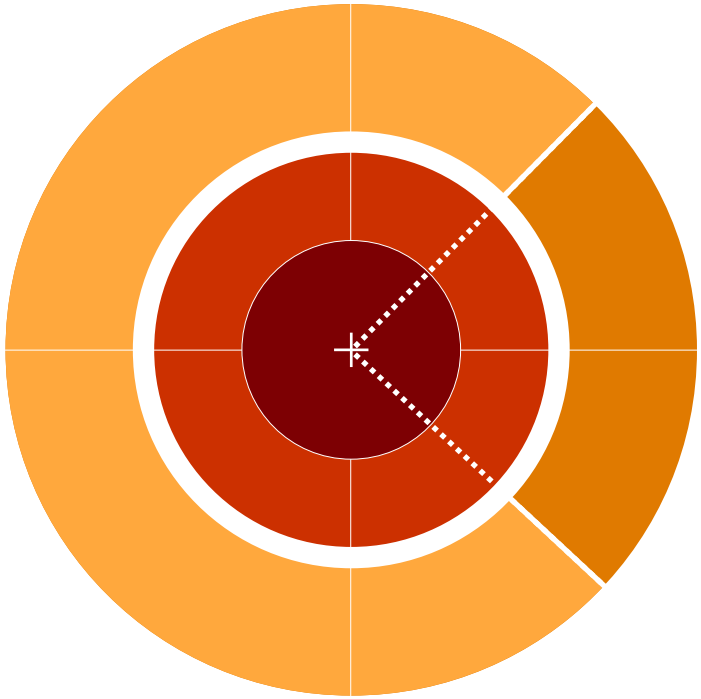
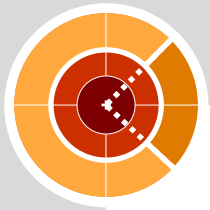
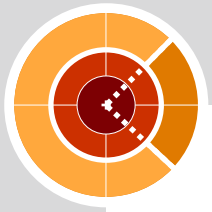




Šablony

- šablona je syntaktická konstrukce pro popis nekonečné množiny objektů (tříd, funkcí) jediným zdrojovým kódem
- šablony do C++ zavádí pojem **genericity**, v praxi se neužívají příliš často, protože jsou syntakticky značně komplikované a nepřehledné – používají se hlavně tzv. **kontejnery**, jsou definovány v knihovně ***Standard Template Library***.
- účelem šablon je **snížení zátěže programátora**, který popíše algoritmus (resp. třídu) pouze jednou, překladač pak podle šablony automaticky generuje kód pro různé datové typy (podle toho, jak je daná funkce/metoda či třída užita v programu)
- **v C++ je možné vytvářet šablony tříd a funkcí/metod**



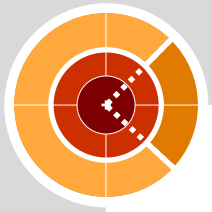
Šablony funkcí

- Šablony funkcí fungují podobně jako přetížení funkce, tj. existuje funkce s jediným jménem pro různé kombinace typů argumentů a návratových hodnot, **ale**:
  a) při přetížení je třeba tělo funkce nadefinovat pro všechny možné kombinace argumentů a návratových hodnot,
  b) šablona má jedinou definici těla, která je generická, popisuje algoritmus pro obecný případ datového typu `T`.

- Definice šablony je podobná definici funkce, před prototypem je uvedeno `template <class T>`, symbol `T` je volitelný a představuje jméno typu:

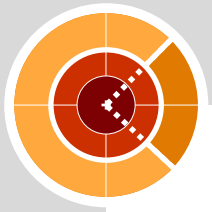
```
template <class T> T max(T a1, T a2) {
  return a1 > a2 ? a1 : a2;
}
```

operátor musí být pro daný typ přetížen



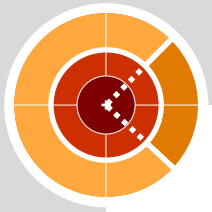
Šablony funkcí - příklad č. 1

```
#include <iostream>
using namespace std;

template <class T> T get_max(T val1, T val2) {
  return val1 > val2 ? val1 : val2;
}

void main() {
  int i1 = 5, i2 = 10, imax = 0;
  double d1 = 2.71, d2 = 3.14, dmax = 0.0;

  imax = get_max(i1, i2);
  dmax = get_max(d1, d2);

  cout << "imax: " << imax << endl;
  cout << "dmax: " << dmax << endl;
}
```



Šablony funkcí - příklad č. 2

```
#include <string>
using namespace std;

template <class T> void argswap(T &a, T &b) {
  T temp = a;
  a = b;
  b = temp;
}

void main() {
  int i1 = 3, i2 = 5;
  string s1 = "Retezec1", s2 = "Retezec2";

  argswap(i1, i2);
  argswap(s1, s2);
}
```



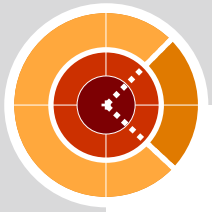
Šablony funkcí pro více typů

- V jedné šabloně lze specifikovat i více typů:

```
template <class T1, class T2, class T3> …
```

```
#include <iostream>
using namespace std;

template <class T1, class T2, class T3>
  double disp(T1 x, T2 y, T3 z) {
  cout << "x: " << x << " y: " << y << " z: " <<
    z << endl;
  return x * y * z;
}

void main() {
  double r = disp(1, 2.1F, 3.1);
  cout << "r: " << r << endl;
}
```

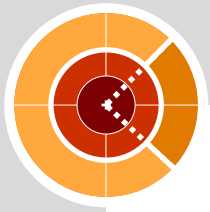
šablona funguje pouze pro číselné datové typy (**int**, **float**, **double**)...



Šablony tříd

- Syntakticky podobné šablonám funkcí, slouží ke generová-
  množiny tříd. V praxi se užívají zřídka, ale standardní kni-
  hovna je využívá intenzivně.
- Deklarace objektu: **`nazev_sablony<typ> promenna;`**

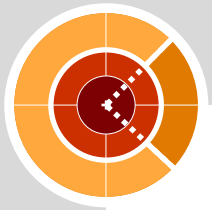
```
template <class T> class Vector3d {
private:
  T x, y, z;

public:
  Vector3d() { x = 0; y = 0; z = 0; }
};

void main() {
  Vector3d<int> vect_int;
  Vector3d<double> vect_double;
  ...
}
```



Šablony tříd - příklad

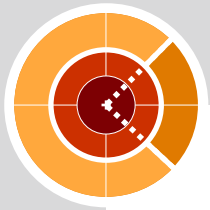
```
#include <iostream>
using namespace std;

template <class T> class Vector3d {
private:
  T x, y, z;
public:
  Vector3d() { x = 0; y = 0; z = 0; }
  void print_coords();
};

template <class T> void Vector3d<T>::print_coords() {
  cout << "Coords: " << x << ", " << y << ", " << z << endl;
}

void main() {
  Vector3d<int> vect_int;
  Vector3d<double> vect_double;

  vect_int.print_coords();
  vect_double.print_coords();
}
```

symbol typu **<T>** **musí být uveden**, jinak překladač hlásí chybějící definici deklarované metody



STL - Standard Template Library

- Součástí standardní knihovny jazyka C++ je **standardní knihovna šablon** (STL).
- V STL jsou definovány např. šablony funkcí implementujících algoritmy pracující s poli (řazení, vyhledávání).
- Z šablonových tříd definovaných v STL se nejčastěji používají tzv. **kontejnery**, sloužící k ukládání objektů libovolného typu.
- V praxi patrně nejužívanější je kontejner **`vector`**, který představuje zobecněný koncept pole (narozdíl od staticky deklarovaného pole není počet prvků v kontejneru předem dán, kontejner se adaptivně zvětšuje při vložení prvku, počet vložených prvků je omezen pouze dostupnou pamětí).
- Detaily: např. **http://www.cplusplus.com/reference/stl/**



Šablona kontejneru `vector`

- Připojit knihovnu příkazem **`#include <vector>`**

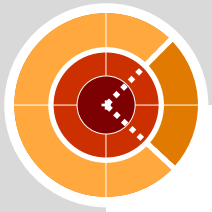
- Kontejner **`vector`** má mj. např. tyto metody:
  `front()` – vrací první prvek
  `back()` – vrací poslední prvek
  `push_back()` – vloží kopii prvku na konec
  `pop_back()` – vyjme poslední prvek
  `clear()` – smaže všechny prvky
  `insert()` – vloží prvek na zadanou pozici
  `size()` – vrací počet vložených prvků
  `empty()` – indikuje, zda počet vložených prvků == 0

- Přetížený operátor **`[]`** umožňuje přistupovat k prvkům kontejneru jako u běžného pole, tj. **`vect[i] = 0`**.
- Přetížený operátor **=** dovoluje přiřazovat celé kontejnery, tj. včetně úplného obsahu jediným příkazem **`v1 = v2;`**

Šablona kontejneru `vector` – příklad č. 1

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

class Circle {
  ...
};

void main() {
  Circle c;
  vector<Circle> circles;

  while (...) {
    c.load_from_file(inpstream);
    circles.push_back(c);
  }

  for (int i = 0; i < circles.size(); i++)
    circles[i].draw(hdevice);
}
```

staticky deklarovaný kontejner pro objekty typu `Circle`.



Šablona kontejneru `vector` – příklad č. 2

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

void main () {
  vector<int> vect(10);
  unsigned int i;

  vector<int>::size_type vectsize = vect.size();

  for (i = 0; i < vectsize; i++) vect[i] = i;

  for (i = 0; i < vectsize / 2; i++) {
    int temp = vect[vectsize - 1 - i];
    vect[vectsize - 1 - i] = vect[i];
    vect[i] = temp;
  }

  cout << "vect: ";
  for (i = 0; i < vectsize; i++) cout << " " << vect[i];
  cout << endl;
}
```

statická definice typu uvnitř šablony kontejneru.