

C++



- Slajdovi su modifikacija slajdova Nenada Čaklovića (VSITE, Zagreb).

C++ template



template (predložak)

- predlošci omogućuju metaprogramiranje - kôd koji generira kôd u *compile-time*u
- predlošci omogućuju **generičko programiranje** - tipovi mogu biti parametri
- template koristi špicaste zagrade: <Content >, gdje Content može biti:
 - class T / typename T
 - tip podataka koji odgovara T-u
 - cjelobrojne konstante (int, char, long, unsigned)
 - cjelobrojne konstante/pointeri/reference koji odgovaraju specifikaciji iz prethodne točke
- dva su tipa templatea
 - *function template*
 - *class template*

template funkcije (*function template*)

- funkcija parametrizirana tipom (ili tipovima) - *template type parameter*

```
void swap(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}
void swap(double& a, double& b)
{
    double t = a;
    a = b;
    b = t;
}
void swap(Number& a, Number& b)
{
    Number t = a;
    a = b;
    b = t;
}
```

```
template<typename TYPE>
void swap(TYPE& a, TYPE& b)
{
    TYPE t = a;
    a = b;
    b = t;
}
int main()
{
    int n1(2), n2(3);
    swap(n1, n2);
    swap<int>(n1, n2);
}
```

- template vrijedi do kraja vitičaste zagrade funkcije
- swap se ne kompajlira sve dok se ne pozove za određeni tip (swap je predložak, *template code*, a ne *source code*)
- ovisno o proslijeđenim parametrima, kompajler će odlučiti kojeg je tipa TYPE

template funkcije (*function template*)

```
template<typename TYPE>
void swap(TYPE& a, TYPE& b)
{
    TYPE t = a;
    a = b;
    b = t;
}

int main()
{
    int n1(2), n2(3);
    swap(n1, n2);
    swap<int>(n1,n2);

    double a(2.3), b(3.14);
    swap(a, b);
}
```

- prilikom poziva funkcije
swap(n1, n2) nastane *source code*
funkcije

```
void swap(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}
```

- prilikom poziva funkcije swap(a, b)
nastane *source code* funkcije

```
void swap(double& a, double& b)
{
    double t = a;
    a = b;
    b = t;
}
```

template funkcije (*function template*)

```
template<typename T>
T sum(T x, T y)
{
    T res;
    res = x + y;
    return res;
}
```

- kompajler kompajlira ovaj kôd u dvije faze
- kompajler u prvom prolazu provjerava osnovnu sintaksu
- neće provjeriti da li:
 - T ima *default* konstruktor
 - T ima podržan operator +
 - T ima copy konstruktor
- u drugoj fazi kompajlira svaku instancu funkcijskog templatea
 - prilikom poziva funkcije stvara se objektni kôd funkcije za određeni tip

template funkcije (*function template*)

03.cpp

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

template<typename T>
double get_average(T arr[], int n)
{
    T sum = T(); // explicit call to
                 // default constructor
    for (int i = 0; i < n; ++i)
        sum += arr[i];
    return (double)sum/n;
}

int main()
{
    int iarr[] = {1, 2, 3, 4, 5, 6};
    float farr[] = {1.1f, 2.2f, 3.3f};

    cout << get_average(iarr, sizeof iarr/
                        sizeof iarr[0]) << endl;
    cout << get_average(farr, sizeof farr/
                        sizeof farr[0]) << endl;
}
```

- tip T mora imati *default* konstruktor
- tip T mora imati operator+=
- tip T mora se moći konvertirati u double

- prilikom prvog poziva funkcije `get_average` nastaje funkcija

```
double get_average(int arr[], int n);
```

- prilikom drugog poziva funkcije `get_average` nastaje funkcija

```
double get_average(float arr[], int n);
```


template funkcije (*function template*)

- template parametara može biti više

05.cpp

```
#include <iostream>

template <typename T1, typename T2>
void change(T1& a, T2& b)
{
    a /= 2;
    b *= 2;
}

int main()
{
    int a = 3;
    double b = 3.14;
    change(a, b);
    std::cout << a << " " << b;
    change(b, a);
    std::cout << b << " " << a;
}
```

- kreirati će se dvije funkcije

```
void change(int, double);
void change(double, int);
```

eksplicitna specifikacija template argumenata

- kompajler po pozivu funkcije zaključuje u koje tipove će se pretvoriti template parametri
- mi možemo tražiti eksplicitno da generira funkciju sa određenim tipom koristeći u pozivu špicaste zgrade
- template parametri funkcije ne moraju se eksplicitno navesti, osim za return type

04.cpp

```
#include <iostream>
template <typename T1, typename T2>
T1 cast(T2 x)
{
    return (T1)x;
}

int main()
{
    std::cout << cast<int>(10.5);
    std::cout << cast<double>(10);
    std::cout << cast<char>(65);
}
```

- kreirati će se tri funkcije

04.cpp

```
int cast(double);
double cast(int);
char cast(int);
```

function template vs. template function

- *function template* je funkcija označena ključnom riječi `template` i špicastim zagradama
 - ona nije prava funkcija, neće se do kraja iskompajlirati i linker je ne povezuje
- *template function* nastaje prilikom poziva *function templatea*
 - ona je prava funkcija, nju prevodi kompajler do kraja i povezuje je linker

function template vs. template function

- *function template*

```
template<typename T>  
void show(T a)  
{}
```

- *pri pozivu function template*

```
show<double>(12);
```

- *nastaje template function*

```
void show<double>(double a){};
```

template klase (*class template*)

06.h

```
class A
{
    int n;
public:
    A() : n(0){}
    void set_data(int n){ this->n = n;}
    int get_data() const {return n;}
    void print_data() {std::cout << n <<
                        std::endl;}
};
```

06.h

```
class B
{
    double n;
public:
    B() : n(0){}
    void set_data(double n){ this->n = n;}
    double get_data() const {return n;}
    void print_data() {std::cout << n <<
                        std::endl;}
};
```

```
int main()
{
    A a;
    a.set_data(10);
    a.print_data();

    B b;
    b.set_data(10.25);
    b.print_data();
}
```

template klase

- *class template* definira apstraktni tip
- klasa parametrizirana tipom (ili tipovima)
- za razliku od funkcijskih predložaka, gdje iz argumenata funkcije kompajler zaključuje o kojem se tipu radi, kod predložaka klase **eksplicitno** treba proslijediti tip u špicastim zagradama

07.h

```
template<typename T>
class A
{
    T n;
public:
    A() : n( T() ){}
    void set_data(T n){ this->n = n;}
    T get_data() const {return n;}
    void print_data() {std::cout << n <<
                        std::endl;}
};
```

07.cpp

```
int main()
{
    A<int> a;
    a.set_data(10);
    a.print_data();

    A<double> b;
    double res = b.get_data();
    std::cout << res;
}
```

- A nije tip. On postaje tip kada kompajler sazna točno s kojim tipom ga treba napraviti.
- A je predložak za nastanak tipa. A<int> je instanciranje templatea (nastaje novi tip). Objekt a je objekt toga tipa.

template klase - definicija članova

- non-inline definiciju funkcija članova prethodi template <typename T>

08.h

```
template<typename T>
class A
{
    T n;
public:
    A();
    void set_data(T n);
    T get_data() const;
    void print_data();
};
```

08.h

```
template<typename T>    A<T>::A() : n( T() ){}
template<typename T>    void A<T>::set_data(T n){ this->n = n;}
template<typename T>    T A<T>::get_data() const {return n;}
template<typename T>    void A<T>::print_data() {std::cout << n;}
```

- s obzirom da ne znamo za koji tip će se instancirati klasa, u konstruktoru se poziva konstruktor tipa
- sav template kôd mora biti u headerima

09.cpp

```
int main()
{
    A<int> a;
    a.set_data(10);
    a.print_data();

    A<double> b;
    double res = b.get_data();
    std::cout << res;
}
```

- prvo instanciranje sa tipom `int` generira sljedeće funkcije:
 - `A<int>::A()` konstruktor
 - `void A<int>::set_data(int)` funkciju
 - `void A<int>::print_data()const` funkciju
- instanciranje s tipom `double` generira sljedeće funkcije:
 - `A<double>::A()` konstruktor
 - `double A<double>::get_data()const` funkciju
- sljedeće funkcije neće doći do druge faze kompajliranja
 - `int A<int>::get_data()const`
 - `void A<double>::set_data(double)`
 - `void A<float>::print_data()const`

dvije faze kompajliranja

- u prvoj fazi kompajliranja provodi se bazična provjera sintakse za template klasu
- druga faza kompajliranja provodi se nakon instanciranja kada je određen template tip

10.cpp

```
template<typename T> A<T>::A() : n( T() ){}  
template<typename T> void A<T>::set_data(T n){ this->n = n%10;}  
template<typename T> T A<T>::get_data() const {return n;}  
template<typename T> void A<T>::print_data() {std::cout << n;}  
  
int main()  
{  
    A<int> a;  
    a.set_data(15);  
    a.print_data();  
  
    A<double> b;  
    b.get_data();  
    b.print_data();  
}
```

- iako za double ne postoji operator %, nema greške jer se funkcija

`void A<double>::set_data(double)` nije ni kompajlirala

class template sa više parametara

11.h

```
struct point
{
    int x, y;
};

struct money
{
    int kn, lp;
};

struct root
{
    int n;
    double r;
};
```

- u svakoj od ovih struktura je par
- ovisno o tome kakav nam par treba, napraviti ćemo instancu s određenim tipovima

11.h

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

11.cpp

```
int main()
{
    pair<int, int> point1;
    point1.first = 10;
    point1.second = 2;

    pair<int, double> sroot;
    sroot.first = 100;
    sroot.second = sqrt(sroot.first);
}
```

class template sa više parametara

11.h

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()){}
    pair(const T1& t1, const T2& t2) :
        first(t1), second(t2){}
    pair(const pair<T1, T2>& other) :
        first(other.first), second(
            other.second){}
    bool operator== (const pair<T1, T2>&
        other) const
    {
        return first == other.first &&
            second == other.second;
    }
};
```

11.cpp

```
int main()
{
    pair<int, double> p;
    pair<int, double> p1(0, 0.13);
    pair<int, double> p2(p1); // OK
    pair<int, int> point4(point3); // ERROR

    if (p1 == p2)
        std::cout << p2.second;
}
```

- uočimo da je copy konstruktoru potrebno specificirati tip parametra

template funkcije članovi (template member functions)

- funkcija član može biti template u template klasi ili običnoj klasi
- ne može biti virtualna

template funkcija u klasi

```
class A {  
public:  
    template<typename T>  
    T fun(T a) { return a*a; }  
};  
  
int main(){  
    A a;  
    int n = a.fun(5);  
}
```

- kôd za funkciju fun će se kompajlirati kad zatreba

template funkcija u template klasi

```
template<typename T> class A  
{  
    T m;  
public:  
    A(T t) : m(t) {}  
  
    template<typename ARG>  
    void f(ARG a) { cout << m*a; }  
};  
  
int main(){  
    A<unsigned char> a(2);  
    a.f(5);  
}
```

predložci s konstantnim parametrima (*non-type*)

- parametri predložka mogu biti integralne konstante (`char`, `int`, `long`, `unsigned`), nisu dozvoljeni `float` i `double`
- prilikom instanciranja, samo konstante vrijednosti mogu biti argumenti (`10`, `10+10`, `10*2`)

12.h

```
template <class T, int N>
class array
{
    T arr[N];
    int sz;
public:
    array() : sz(N)
    {
        for (int i = 0; i < sz; ++i)
            arr[i] = T();
    }
    int size() {return sz;}
    T& operator[] (int i) { return arr[i];}
};
```

12.cpp

```
int main ()
{
    array <int, 4> a1;
    array <float, 4> a2;
    a1[3] = 10;
    std::cout << a1[3] << '\n';
    std::cout << a2[3] << '\n';
    return 0;
}
```

default template argumenti

- template može imati više parametara, zadnji mogu imati default vrijednosti i ne moraju se navesti (kao kod default vrijednosti argumenata funkcija)

```
template<typename T, int SIZE=64>
class array { ... };

int main(){
    array<int> ar1;
    array<unsigned long, 32> ar2;
}
```

```
template<typename T=int, int SIZE=64>
class array { ... };

int main(){
    array<> ar1;
    array<unsigned long, 32> ar2;
}
```

specijalizacija (specialization)

- alternativna implementacija za pojedini tip
- koristi se kada se jednim predloškom ne mogu obuhvatiti svi slučajevi realizacije s različitim tipovima
- za tipove kojima realizacija odstupa od predloška pišemo specijalizaciju

Za klasu koju definiramo predloškom

```
template<typename T>class ime{...}
```

specijalizacija za konkretni tip se zapisuje u obliku

```
template<>class ime<tip>{...}
```

specijalizacija (specialization)

```
template<>
class Array<bool>
{
    char* p;
public:
    Array(int size) : p(new char[(size-1)/8+1]){ }
    ~Array() { delete[] p; }
    bool operator[](int i) { return p[i/8] & (1<<(i%8)); }
};

int main()
{
    Array<int> ar1(30); // koristi originalni template
    Array<bool> ar2(64); // koristi specijalizirani template
}
```

- npr. u standardnoj biblioteci vector ima specijalizaciju za bool

parcijalna specijalizacija (partial specialization)

- specijalizacija, ali ne po svim parametrima

```
template <typename T, typename U, typename V> class A {};  
  
template <typename T, typename V> class A<T, int, V> {};  
template <typename V> class A<double, long, V> {};  
template <typename U> class A<int, U, int*> {};
```

eksplicitna konverzija (cast) u C++

u c++ ne radimo sa klasičnim cast operatorom

- `const_cast`
- `reinterpret_cast`
- `static_cast`
- `dynamic_cast`

eksplicitna konverzija (cast) u C++

- `static_cast` - za uobičajene (C-ovske) pretvorbe i upcasting

```
int n=5; double d = static_cast<double>(n);  
Derived pd; Base* pb = static_cast<Base*>(&pd);
```

- `dynamic_cast` - za downcast, vraća `NULL` ako ne uspije
- klasa mora imati virtualne funkcije, potrebno uključiti u projekt RTTI (run-time type information)

```
Base b;  
Derived d;  
  
Base* pb = static_cast<Base*>(&d);  
Derived* pd = dynamic_cast<Derived*>(pb);  
Derived* pd2 = dynamic_cast<Derived*>(&b); // pd2 je NULL!
```

eksplicitna konverzija (cast) u C++

- `const_cast` - za skidanje i stavljanje konstantnosti

```
void f(const A& a)
{
    A& ra = const_cast<A&>(a);
    ra.change();
    A* pa = const_cast<A*>(&a);
    pa->change();
}
```

- `reinterpret_cast` - za kompletnu promjenu tipa, opasno

```
void f(long lp)
{
    A* pa = reinterpret_cast<A*>(lp);
}

int main()
{
    A a;
    long xxx = reinterpret_cast<long>(&a);
    f(xxx);
    f(1); // argh!!
}
```