# Šablony funkcí a tříd

## BD5B37PPC – Programování v jazyce C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

# Přehled témat

- Část 1 – Šablony funkcí a tříd

  Generické funkce

  Generické třídy

# Část I

# Šablony funkcí a tříd

# I. Šablony funkcí a tříd

Generické funkce

Generické třídy

# Generické funkce

- Generická funkce (šablona) je parametrizovaná deklarace a definice funkce.
- Generický parametr (parametr šablony) je datový typ, který je kompilátorem substituován, když vzniká nová instance generické funkce.
- Generické funkce mohou parametrizovat datové typy svých parametrů nebo návratové hodnoty.

**Příklad**

```
// generická funkce
T f ( T x, T y );
// instance generické funkce
int f ( int x, int y );
double f ( double x, double y );
char f ( char x, char y );
```

# Generické funkce

```
template < class T > // T je generický parametr
T max ( T x, T y ) {
return x > y ? x : y;
}
template < typename T > // alternativní syntaxe
T max ( T x, T y ) {
return x > y ? x : y;
}
```

- Funkce je generickou funkcí – reálná funkce (instance generické funkce) je odvozena, když je generická funkce použita (tj. volána).
- V cílovém programu může existovat více instancí generické funkce – jedna instance pro každý typ dat.

# Generické funkce – instance

- Instance jsou vytvořeny, když je generická funkce použita:

```
template < class T >
T max ( T x, T y ) {
  return x > y ? x : y;
}
// Protože jméno max je v konfliktu s std::max ,
// použijeme plně kvalifikované jméno ::max .
int i = 10, j = 20;
unsigned u = 40;
char c = 'a';
cout << ::max ( i, 10 ); // int max ( int, int )
cout << ::max ( i, j ); // int max ( int, int )
cout << ::max ( c, 'b' ); // char max ( char, char )
cout << ::max ( i, c ); // chyba - nejednoznačné
cout << ::max ( i, u ); // chyba - nejednoznačné
```

Při vytváření instance generické funkce nejsou používány standardní typové konverze.

# Generické funkce – instance

- Když skutečné parametry funkce nedávají jednoznačný výběr datového typu její šablony, musí parametr šablony napsat programátor explicitně:

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

int i = 10, j = 20;
unsigned u = 40;
char c = 'a';

cout << ::max<char> (i, c); // char max (char, char)
cout << ::max<int> (i, u);  // int max (int, int)
```

# Generické funkce – instance

- Jsou-li typy specifikovány explicitně, může šablona parametrizovat dokonce datový typ návratové hodnoty:

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

int i = 10;
char c = 'a';

cout << ::max<char> (c, 'b') << endl; // zobrazeno b
cout << ::max<int> (c, 'b') << endl; // zobrazeno 98
cout << ::max<int,int> (i, c) << endl; // zobrazeno
 97
```

# Generické funkce – přetížení

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

template <class T> T max ( T x, T y, T z ) {
  return x > y ? ( x > z ? x : z ) : ( y > z ? y : z )
    ;
}

int main ( ) {
  int a = 10, b = 20, c = 30;
  cout << ::max ( a, b ) << endl; // 20
  cout << ::max ( a, b, c ) << endl; // 30
  return 0;
}
```

# Generické funkce – přetížení

- Generická funkce a obyčejná funkce mohou být přetíženy.
- Obyčejná funkce má přednost.

```
template <class T> T max (T x, T y) {
  return x > y ? x : y;
}
const char * max (const char * x, const char *y) {
  return strcmp (x, y) > 0 ? x : y;
}

int main () {
  const char *a = "Hello", *b = "Hi";
  cout << ::max (a, b) << endl; // Hi
  cout << ::max ((void*)a, (void*)b) << endl;
  return 0;
}
```

# Generické funkce – explicitní instance

- Instance generické funkce může být vytvořena explicitně.
- To může pomoci při hledání chyb v generické funkci:
  - kompilátor má jen omezené šance najít chybu v generické funkci. Kompilátor nezná datové typy, když čte a rozebírá zdrojový text generické funkce, proto nemůže validovat parametry,
  - generická funkce musí být kompilována znovu a znovu pro každý generický parametr,
  - kompilátor může najít zbývající chyby až při vytvoení instancí.

```
template < class T > T max ( T x, T y ) {
  return x > y ? x : y;
}
// explicitní vytvoření instance generické funkce:
template int max (int x, int y);
// potlačení generické funkce pro určitý datový typ:
const char * max (const char * x, const char * y);
// Ve skutečnosti kompilátor hledá "obyčejnou" funkci.
// Není-li funkce implementována, nastane chyba.
```

# Generické funkce – explicitní instance

```
template < class T > T max ( T x, T y ) {
  return x > y ? x : y;
}
struct S {
  int a;
};

int main ( ) {
  S x = {1}, y = {2}, z;
  z = ::max (x, y); // zde je chyba
  cout << z.a << endl;
  return 0;
}
```

Jak opravit chybu?

Přidat přetížený operátor > pro datový typ S.

# Příklad

```
// Zobrazit pole, n - počet prvků
// rowLen - formátování (počet prvků na řádku)
template < class T >
void printArray ( T *arr, int n, int rowLen = 10 ) {
  int i;
  for ( i = 0; i < n; i++ ) {
    if ( i % rowLen != 0 )
      cout << ' ';
  cout << arr[i];
  if ( i % rowLen == rowLen - 1 )
    cout << endl;
  }
  if (i % rowLen != 0 )
    cout << endl;
}
```

# Příklad

```
// Seřadit pole, n - počet prvků
template < class T >
void sortArray ( T *arr, int n ) {
  for ( int i = 0; i < n - 1; i ++ ) {
    int min = i;
    for ( int j = i + 1; j < n; j ++ )
      if ( arr[j] < arr[min] )
        min = j;
    T tmp = arr[i];
    arr[i] = arr[min];
    arr[min] = tmp;
  }
}
```

# I. Šablony funkcí a tříd

Generické funkce

Generické třídy

# Generické třídy

- Generická třída je parametrizovaná implementace třídy.
- Kompilátor odvozuje instanci generické třídy nahrazením generických parametrů skutečnými hodnotami.
- Generické třídy jsou obvykle parametrizovány typovým(i) parametrem(y).

# Příklad

```
template <class T>
class CCounter {
  T m_Val;
  T m_Init;
public:
  CCounter ( T init ) { m_Init = init; reset (); }
  void increment ( ) { m_Val ++; }
  void decrement ( ) { m_Val --; }
  void reset ( ) { m_Val = m_Init; }
  T get ( ) { return m_Val; }
};
...
CCounter<int> intCnt (0);
CCounter<char> chCnt ('A');
cout << intCnt.get () << endl;
chCnt.increment ();
```

# Příklad

- Pokud jsou metody implementovány vně deklarace generické třídy, musí každá metoda začínat deklarací šablony template.

```
template <class T>
class CCounter {
  T m_Val;
  T m_InitVal;
public:
  CCounter ( T init );
  void increment ( ) { m_Val ++; }
...
};
template <class T>
CCounter<T>::CCounter ( T init ) { m_InitVal = init; reset
    (); }
template <class T>
void CCounter<T>::increment ( ) { m_Val++; }
```

# Příklad – dynamické pole

- Pole bude obsahovat prvky jakéhokoli typu (generický parametr).
- Implementace bude korektně implementovat kopírující konstruktor a operátor =.

```
template <class T>
class Array {
  T * array_data;
  int array_size;
public:
  Array (int size = 10);
  ~Array ();
  Array (const Array<T> & src);
  int size() const { return array_size; }
  Array<T> & operator = (const Array<T> & src);
  T & operator [] (int idx);
  const T & operator [] (int idx) const;
};
```

# Příklad – dynamické pole

```
template <class T>
Array<T>::Array (int size) {
  array_size = size;
  array_data = new T [array_size];
}

template <class T>
Array<T>::~Array ( ) {
  delete [] array_data;
}

template <class T>
Array<T>::Array (const Array<T> & src ) {
  array_size = src.array_size;
  array_data = new T[array_size];
  for (int i = 0; i < array_size; i++)
    array_data[i] = src.array_data[i];
}
```

# Příklad – dynamické pole

```
template <class T>
Array<T> & Array<T>::operator = (const Array<T> & src) {
  if (this != &src) {
    delete [] array_data;
    array_size = src.array_size;
    array_data = new T[array_size];
    for (int i = 0; i < array_size; i++)
      array_data[i] = src.array_data[i];
  }
  return *this;
}

template <class T>
T & Array<T>::operator [] (int idx) {
  if (idx < 0 || idx >= array_size)
    throw "Spatny index";
  return array_data[idx];
}
```

# Příklad – dynamické pole

```
template <class T>
const T & Array<T>::operator [] (int idx) const {
  if (idx < 0 || idx >= array_size)
    throw "Spatny index";
  return array_data[idx];
}

template <class T>
ostream & operator<< (ostream & o, const Array<T> & x) {
  for (int i = 0; i < x.size (); i++)
    o << x[i] << ' ';
  return o;
}
```

# Příklad – dynamické pole

```
Array<int> a (5);
for (int i = 0; i < a.size (); i++) a[i] = i;
cout << "array a: " << a << endl; // [0 1 2 3 4]

Array<int> b = a;
b[1] = 10;
cout << "array a: " << a << endl; // [0 1 2 3 4]
cout << "array b: " << b << endl; // [0 10 2 3 4]

Array<double> c (5);
c[1] = 20;
cout << "array c: " << c << endl; // [?  20 ?  ?  ?]

Array<double> d = c;
d[2] = 30;
cout << "array d: " << d << endl; // [?  20 30 ?  ?]
```

# Příklad – dynamické pole

- Mohou mít prvky pole generický datový typ?

```
int main() {
  Array< Array<int> > a ( 5 );
  for ( int i = 0; i < a.size ( ); i ++ )
    for ( int j = 0; j < a[i].size ( ); j ++ )
      a[i][j] = i + j;
  cout << "array a: " << a << endl;
  Array< Array<int> > b = a;
  b[1][2] = -10;
  cout << "array a: " << a << endl;
  cout << "array b: " << b << endl;
  return 0;
}
```

- Ano, pro inicializaci polí řádků je volán implicitní konstruktor.
- Výsledkem bude matice 5x10.