4. Šablony funkcí, tříd a další aspekty programování v C++

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Další aspekty programování v C++
  - Rozložení
  - Přetypování
  - Kopírující konstruktor
- Část 2 – Šablony funkcí a tříd
  - Generické funkce
  - Generické třídy

Část I

Další aspekty programování v C++

Kopírování objektů

- Často potřebuje kopírovat objekty podobně jako triviální typy

```
int b = a;
```

- I tato zdánlivě jednoduchá operace má svá úskalí

```
Trida* instance = new Trida;
Trida* jinaInstance = instance;
```

- Je zřejmé, že `jinaInstance` není kopie – oba odkazy ukazují na jedno paměťové místo
- Pokud chceme docílit opravdové kopie, musí být pro danou třídu implementován **kopírovací konstruktor**

I. Další aspekty programování v C++

Rozložení

- Termín **rozložení** odkazuje na to, jak jsou atributy a metody třídy uspořádány v paměti
- V některých případech, jako jsou např. virtuální funkce, může kompilátor rozložit instanci do paměti na více než jedno souvislé místo
- Z hlediska optimalizace je to výhodné, ovšem ztěžuje to přenos objektů (tj. bloků paměti) mezi aplikacemi, kopírování rychlými nízkoúrovňovými funkcemi (`memcopy`) serializaci, atd.
- Od `C++14` lze rozdělit datové typy na
  - triviální
  - standardní rozložení
  - POD – Pure Old Data
- Standardní knihovna má šablony funkcí pro určení kategorie typu
  - `is_trivial<T>`
  - `is_standard_layout<T>`
  - `is_pod<T>`

Triviální typy

- Instance triviálních typů zabírají v paměti souvislý blok, který je možné je kopírovat do pole typu `char` nebo `unsigned char` a z nich zpět do proměnné
- Triviální datový typ se vyznačuje tím, že má
  - triviální konstruktor
  - triviální kopírovací konstruktor
  - triviální operátor přiřazení
  - triviální destruktor

O kopírovacím konstruktoru se dozvíme v další části přednášky.

- Triviální v kontextu konstruktor/destruktor/operátor znamená, že
  (a) není definován uživatelem a
  (b) třída
  - nemá žádné virtuální funkce
  - není potomkem bázové třídy s netriviálním konstruktorem/operátorem/destruktorem
  - žádné atributy (metody) nejsou datového typu s netriviálním konstruktorem/operátorem/destruktorem

Triviální typy – příklad

```
struct A {
  int m;
};
struct B {
  B() {}
};
struct C {
  C(int a, int b) {}
  C() = default;
};
//..
std::cout << std::is_trivial<A>::value << '\n';
std::cout << std::is_trivial<B>::value << '\n';
std::cout << std::is_trivial<C>::value << '\n';
```

lec04/01-layouts.cpp

Klíčové slovo `default`

- Explicitní požadavek na vytvoření defaultního konstruktoru kompilátorem
- Vhodné pokud ve třídě existuje už jiný konstruktor → defaultní není automaticky vytvořen
- Zjevně redundantní, ale pokud je defaultní konstruktor definován uživatelem, není splněna podmínka triviálního datového typu

Klíčové slovo lze spojit i s destruktorem a kopírovacím konstruktorem, vhodné pro zpřehlednění kódu

```
struct A {
  // parametrizovany konstruktor
  A(int x) {}
  // instrukce pro vytvoreni def.
  // konstruktoru kompilatorem
  A() = default;
};
```

```
struct B {
  // chyba, neni konstruktor
  int func() = default;
  // chyba, neni def.  konst.
  B(int, int) = default;
  // chyba, konstruktor
  // s def.  argumentem
  B(int = 0) = default;
};
```

Standardní rozložení

- Datový typ se standardním rozložením splňuje následující vlastnosti
  - všechny nestatické datové členy (atributy a metody) mají totožnou viditelnost
  - všechny nestatické atributy a bázové třídy mají také standardní rozložení
  - nemá žádné virtuální funkce ani virtuálně nedědí bázovou třídu
  - není odvozena z bázové třídy s nestatickými datovými členy

Příklad

```
struct Base1 {
  int i;
};
// Derived1 nemá std.  rozložení
struct Derived1 : public Base1 {
  int x;
};
```

```
struct Base2 {
  void Foo() {}
};
// Derived2 má std.  rozložení
struct Derived2 : public Base2 {
  int x;
};
```

lec04/02-standard-derived.cpp

I. Další aspekty programování v C++

Explicitní konverze

- V C++ lze stejně jako v C použít explicitní a implicitní přetypování
- K dispozici je ještě alternativní formát zápisu – konverzní konstruktor
  - má právě jeden parametr
  - pokud se při vytváření instance objeví přiřazení jiného datového typu, překladač se pokusí najít odpovídající konstruktor

```
double a = 3.1415;
// implicitní konverze
int i = a;
// explicitní konverze známá z jazyka C
int j = (int)a;
// konverzní konstruktor - funkcionální přetypování
int k = int(a);
```

Konverzní konstruktor – příklad

```
struct Double {
  Double (int a) {
    std::cout << "int\n";
  }
  // zákaz implicitní konverze
  explicit Double (long a) {
    std::cout << "long\n";
  }
};
// --
Double a = 10L;
// 10L je long, ale implicitni konverze long -> Double je zakazana
// kompilator proto hleda jiny vhodny konstruktor
// Je treba ji provest explicitne
Double b = Double(10L);
```

lec04/04-converse-constructor.cpp

Implicitní konverze

```
struct A {};
struct B {
  // konverze z A (konstructor)
  B (const A & x) {}
  // konverze z A (přiřazení)
  B & operator= (const A & x) {return *this;}
  // konverze do A (type-cast operátor)
  operator A() {return A();}
};
// --
A foo;
B bar = foo;     // volá se konstruktor
bar = foo;       // operátor přiřazení
foo = bar;       // volá se type-cast operátor
```

lec04/05-implicit-cast.cpp

Přetypování – `dynamic_cast`

- Pracuje s ukazateli nebo referencemi na třídy (nebo `code *`)
  - **pointer upcast** – konverze z ukazatele na derivovanou třídu na ukazazatel na bázovou třídu
  - **pointer downcast** – konverze z ukazatele na bázovou třídu na ukazatel na derivovanou třídu, pouze pokud je odkazovaný objekt kompletním objektem cílového typu

Příklad

```
Base * pba = new Derived;
Base * pbb = new Base;
Derived * pd;

pd = dynamic_cast<Derived*>(pba);
if (pd==0) std::cout << "Nulovy ukazatel v prvnim pripade.\n";

pd = dynamic_cast<Derived*>(pbb);
if (pd==0) std::cout << "Nulovy ukazatel v druhem pripade.\n";
```

lec04/06-dynamic-cast.cpp

Přetypování – static_cast a reinterpret_cast

static_cast

- Pracuje s ukazateli na třídy v relaci
- Na rozdíl od dynamic_cast může odkazovat na nekompletní objekt

```
class Base {};
class Derived: public Base {};

Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

---

reinterpret_cast

- Může provádět konverze mezi ukazateli na libovolné datové typy

```
class A {};
class B {};

A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

I. Další aspekty programování v C++

Rozložení

Přetypování

Kopírující konstruktor

Motivační příklad 1/3

- Mějme třídu `Vektor`, která popisuje vektor celých čísel, data uložena v dynamické paměti
- Implementace obsahuje přetížený operátor pro přístup k datům
- Přístup k indexům mimo alokaci vyvolá výjimku

```
class Vektor {
  int * data, velikost;
public:
  explicit Vektor (int v): velikost(v) {
    data = new int[velikost];
    for (int i = 0; i < velikost; i++) data[i] = 0;
  }
  ~Vektor () {delete [] data;}
  int & operator[] (int idx);
};
```

Motivační příklad 2/3

```
int & Vektor::operator[] (int idx) {
  if ( idx < 0 || idx >= velikost )
    throw "Index mimo meze";
  return data[idx];
}
int main() {
  Vektor a(5);
  a[1] = 10;
  for (int i = 0; i < 5; i++)
    std::cout << a[i] << " ";
  std::cout << std::endl;
  return 0;
}
```

Motivační příklad


```
int main() {
  Vektor a(5), b(5);

  a[4] = 3;
  b = a;      // problém č. 1
  b[3] = 5;   // problém č. 2

  for (int i = 0; i < 5; i++) std::cout << a[i] << " ";
  std::cout << std::endl;
  for (int i = 0; i < 5; i++) std::cout << b[i] << " ";
  std::cout << std::endl;
  return 0;
}
```

lec04/07-vektor.cpp

Mělká kopie

- Objekt byl okopírován použitím operátoru `=`
- Třída `Vektor` nemá operátor `=` přetížený, takže si jej kompilátor vymyslí:
  - objektové členské proměnné kopíruje jejich operátorem `=`
  - primitivní datové typy (včetně ukazatelů a ukazatelů na objekty) kopíruje po bitech
- Binární kopie ukazatele na dynamicky alokovaná data referencuje stejnou paměť (tedy stejná data)
- To se nazývá mělká kopie (shallow copy).
- Mělká kopie není problémem, pokud jsme si toho vědomi.
- Destruktor ale problémem je (stejný dynamicky alokovaný blok je uvolněn dvakrát nebo i vícekrát).

Hluboká kopie

- Sémantika operátoru = (deep copy) je jasná:
  - chceme okopírovat zdrojová data (vpravo) na cílová (vlevo),
  - původní data na levé straně budou zřejmě zničena, objekty musí zůstat nezávislé.
- Kopírování adresy je nedostatečné, je třeba kopírovat obsah:
  - musí být připraven nový dynamicky alokovaný blok,
  - obsah pole musí být zkopírován, dříve alokované pole musí být zrušeno.

```
Vektor & Vektor::operator = (const Vektor & src) {
  if ( &src == this ) return *this;
  delete [] data;
  velikost = src.velikost;
  data = new int [velikost];
  for ( int i = 0; i < velikost; i ++ )
    data[i] = src.data[i];
  return *this;
}
```

Kopírující konstruktor 1/3

- Deklarace

```
Vektor b = a;
```

  deklaruje nový objekt `b` a inicializuje jej obsahem objektu `a`.
- Deklarace je identická jako

```
Vektor b ( a );
```

- V obou předchozích případech inicializace se volá kopírující konstruktor :

```
Vector( const CVector & src );
```

- Když kopírující konstruktor neexistuje (programátor ho nenapsal), kompilátor automaticky připraví kopírující konstruktor:
  - objektové členské proměnné jsou kopírovány jejich kopírujícími konstruktory,
  - primitivní datové typy (včetně ukazatelů a ukazatelů na objekty) jsou kopírovány po bitech.

Toto je v podstatě mělká kopie objektu.

Kopírující konstruktor 2/3

```
Vektor::Vektor ( const Vektor & src ) {
  velikost = src.velikost;
  data = new int [velikost];
  for ( int i = 0; i < velikost; i++ )
    data[i] = src.data[i];
}
```

- Konstruktor je podobný operátoru =, ale
  - chybí podmínka,
  - chybí delete.

Proč?

Kopírující konstruktor 3/3

- Kopírující konstruktor a operátor = jsou použity pro vytvoření hluboké kopie objektu.
- Rozdíl je v cílovém objektu:
  - operátor = má na levé straně plně funkční objekt → zdroje alokované pro cílový objekt musí být před kopírováním uvolněny,
  - kopírující konstruktor nemá žádný objekt k modifikaci → místo toho má cíl již rezervovanou (neinicializovanou) paměť, která musí být inicializována.
- Kopírující konstruktor, operátor = a destruktor jsou těsně svázány. Je-li důvod pro implementaci destruktoru, je také důvod pro implementaci kopírujícího konstruktoru a operátoru = (pokud kopírování není zakázáno).
- Pravidlo 3 (rule-of-three): buď třída implementuje všechny tři (destruktor, kopírující konstruktor a operátor =), nebo ani jeden z nich.

Část II

Šablony funkcí a tříd

II. 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;
}
// jméno max je v konfliktu s std::max, použijeme plně kvalifikované ::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 2/3

- 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 3/3

- 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 1/2

```
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;
  std::cout << ::max (a, b) << std::endl; // 20
  std::cout << ::max (a, b, c) << std::endl; // 30
  return 0;
}
```

Generické funkce – přetížení 1/2

- 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 () {
  std::const char *a = "Hello", *b = "Hi";
  std::cout << ::max (a, b) << std::endl; // Hi
  std::cout << ::max ((void*)a, (void*)b) << std::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 – zobrazení pole

```
// 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 – řazení pole

```
// 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;
  }
}
```

II. 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 – čítač 1/2

```
template <class T>
class Counter {
  T value, init;
public:
  Counter (T in_init): init(in_init) {reset ();}
  void increment () {value++;}
  void reset () {value = init;}
  T get () {return value;}
};
// --
Counter<int> a(0);
Counter<char> b('A');
std::cout << a.get() << std::endl;
b.increment();
```

Příklad – čítač 2/2

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

```
template <class T>
class Counter {
  T value, init;
public:
  CCounter (T in_init);
  void increment () {value++;}
// --
};
template <class T>
Counter<T>::Counter (T in_init) {value = in_init; reset();}
template <class T>
void Counter<T>::increment () {value++;}
```

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 * a_data;
  int a_size;
public:
  Array (int size = 10);
  ~Array ();
  Array (const Array<T> & src);
  int size() const {return a_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): a_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 ) {
  a_size = src.a_size;
  a_data = new T[a_size];
  for (int i = 0; i < a_size; i++) a_data[i] = src.a_data[i];
}
```

Příklad – dynamické pole


```
template <class T>
Array<T> & Array<T>::operator = (const Array<T> & src) {
  if (this != &src) {
    delete [] a_data;
    a_size = src.a_size;
    a_data = new T[a_size];
    for (int i = 0; i < a_size; i++) a_data[i] = src.a_data[i];
  }
  return *this;
}
template <class T>
T & Array<T>::operator [] (int idx) {
  if (idx < 0 || idx >= a_size) throw "Spatny index";
  return a_data[idx];
}
```

Příklad – dynamické pole


```
template <class T>
const T & Array<T>::operator [] (int idx) const {
  if (idx < 0 || idx >= a_size)
    throw "Spatny index";
  return a_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 je matice 5x10