

Y36PJC Programování v jazyce C/C++

Třídy a objekty II

Ladislav Vagner

Dnešní přednáška

- Více o konstruktorech:
 - implicitní konstruktor,
 - inicializace členských proměnných,
 - kopírující konstruktor,
 - konstruktory uživatelské konverze.
- Třídy jako datové typy:
 - zásobník,
 - fronta.

Minulá přednáška

- Deklarace třídy.
- Konstruktor, destruktory, metody.
- Zapouzdření.
- Třídní a instanční proměnné a metody.
- Třídy v C++ a struktury v C.

Konstruktory

- Volán automaticky při vytváření nové instance.
- Rozlišení konstrukturu – pravidla přetěžování funkcí.
- Speciální typy konstruktů:
 - implicitní,
 - kopírující,
 - uživatelské konverze.
- Volány systémem, pokud instance vzniká za specifických okolností.

Implicitní konstruktor

- Konstruktor volatelný bez parametrů:
 - nemá žádné formální parametry,
 - nebo má pro všechny parametry implicitní hodnoty.
- Použit, pokud instance vzniká a parametry nejsou k dispozici (pole objektů, členská proměnná jiného objektu).
- Vygenerován systémem automaticky, pokud:
 - má vzniknout instance a
 - ve třídě neexistuje žádný jiný konstruktor.
- Vygenerovaný implicitní konstruktor:
 - volá implicitní konstruktory staticky alokovaných členských proměnných – objektů,
 - ostatní členské proměnné nechá neinicializované.

Implicitní konstruktor

```
class CFoo { ... };

CFoo * array = new CFoo [100];
// celkem se vola 100 x implicitni konstruktor,
// pro každý prvek pole 1x
...
delete [] array;
// vola 100x destruktorku

// -----

CFoo * array2 = new CFoo [100];

delete array2;
// bez [] by se destruktorku pouzelo pouze 1x pro první objekt
```

Implicitní konstruktor

```
class Cfoo
{ ...
  Cfoo ( int x = 20 ) { ... } //impl. konstruktor Cfoo
};

class CBar
{
  Cfoo  Stat;
  Cfoo * Dyn;
  CBar :: CBar ( int x );
};

CBar::CBar ( int x )
{ // zde se vola impl konstruktor pro Stat
  Dyn = new Cfoo ( 20 );
}
```

Implicitní konstruktor

```
class Cfoo
{ ...
  Cfoo ( int x ) { ... } // konstruktor Cfoo, ne impl.
};
```

```
class CBar
{
  Cfoo  Stat;
  Cfoo * Dyn;
  CBar :: CBar ( int x );
};
```

```
CBar::CBar ( int x )
{ // zde se ma volat impl. konstruktor pro Stat,
  // neexistuje, nedoplni se automaticky -> chyba
  Dyn = new Cfoo ( 20 );
}
```


Implicitní konstruktor

```
class Cfoo
{ ...
  Cfoo ( int x ) { ... } // konstruktor Cfoo, ne impl.
};

class CBar
{
  Cfoo  Stat;
  Cfoo * Dyn;
  CBar :: CBar ( int x );
};

CBar::CBar ( int x ) : Stat ( 10 )
{
  Dyn = new Cfoo ( 20 );
}
```

Implicitní konstruktor a Java

- Existuje v Javě implicitní konstruktor?
- Java vytváří instance pouze dynamicky,
- Java neumí alokovat pole objektů.
- V Javě není implicitní konstruktor třeba.
- Toto je pole objektů:

```
string * array = new string [100]; // C++
```

- Toto není v Javě pole objektů, ale pole referencí:

```
String array [] = new String [100]; // Java
```

- Poli Java referencí odpovídá C++ deklarace:

```
string ** array = new string [100]; // C++
```

Kopírující konstruktor

- Konstruktor, který má parametrem konstantní referenci na instanci své třídy.
- Volán pokud:
 - vzniká nová instance,
 - instance vzniká jako kopie existující instance.
- Vygenerován automaticky, pokud ve třídě neexistuje.
- Vygenerovaný kopírující konstruktor:
 - vytvoří kopie staticky alokovaných členských proměnných – objektů voláním jejich kopírujících konstruktorů,
 - ostatní členské proměnné zkopíruje binárně.

Kopírující konstruktor

```
class CFoo { ... };

void foo1 ( CFoo a ) { ... }
void foo2 ( CFoo & a ) { ... }
void foo3 ( CFoo * a ) { ... }

CFoo x;
CFoo y (x); // volan copy konstruktor
CFoo z = x; // volan copy konstruktor

foo1 ( x ); // volan copy konstruktor
foo2 ( x ); // neni volan zadny konstruktor
foo3 ( &x ); // neni volan zadny konstruktor

CFoo a = x; // volan copy konstruktor
z = x; // neni volan copy konstruktor, ale
// operator = (pozdeji)
```

Kopírující konstruktor

- Kdy automaticky generovaný kopírující konstruktor nestačí?
 - Obsahuje-li instance dynamicky alokovaná data,
 - prostředky OS (soubory, sockety, semaforey, thready, mutexy, ...),
 - využíváme-li techniku počítaných referencí (bude vysvětleno později).
- "Pravidlo" pro návrh tříd – následující rozhraní mívají třídy buď celé nebo vůbec:
 - kopírující konstruktor,
 - přetížený operátor =,
 - destruktory.

Kopírující konstruktor

```
class Cfoo
{
    char * str;
    int   len;
public:
    Cfoo   ( const char * str = "" )
    {
        len       = strlen ( str );
        this -> str = new char [len + 1];
        strncpy   ( this -> str, str, len + 1 );
    }

    ~Cfoo   ( void )
    { delete [] str; }

    const char * GetStr ( void ) const
    { return str; }
}
```

Kopírující konstruktor

```
void SetChar ( int idx, char c )
    { if ( idx >= 0 && idx < len ) str[idx] = c; }
};

void f ( void )
{
    CFoo a( "test" ), b( "test" );

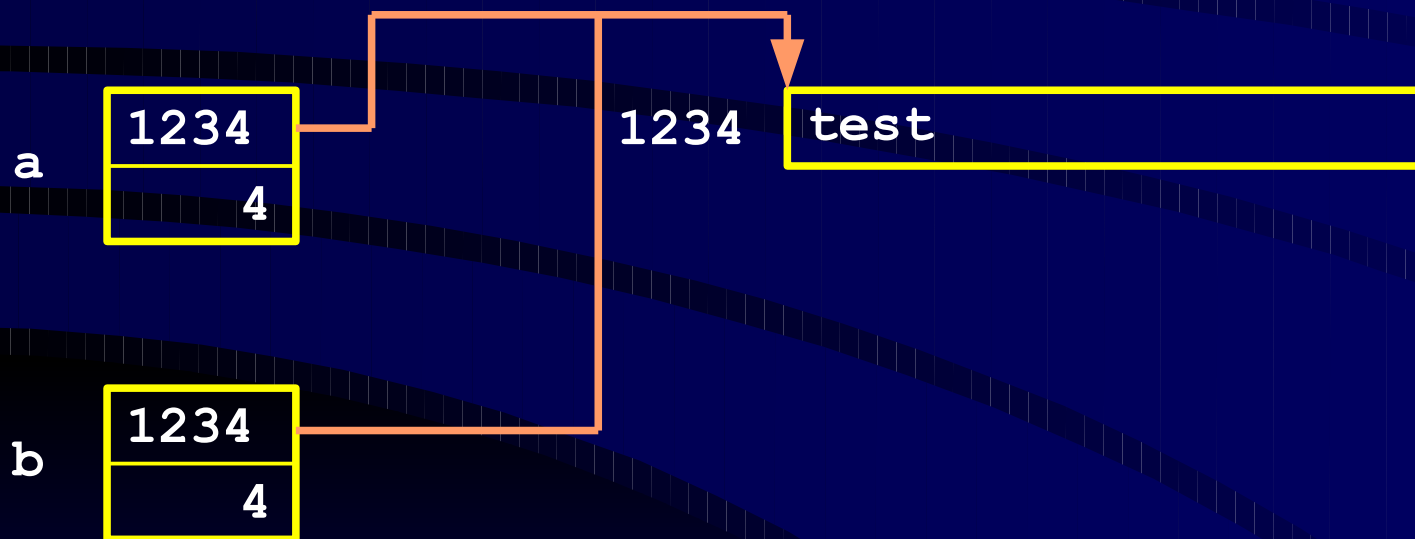
    a . SetChar ( 2, 'z' );
    b . SetChar ( 3, 'u' );
    cout << a . GetStr () << " " << b . GetStr ();
}
```

Kopírující konstruktor

```
void g ( void )
{
    CFoo a ( "test" );
    CFoo b = a;

    a . SetChar ( 2, 'z' );
    b . SetChar ( 3, 'u' );
    cout << a . GetStr () << " " << b . GetStr ();
}
```

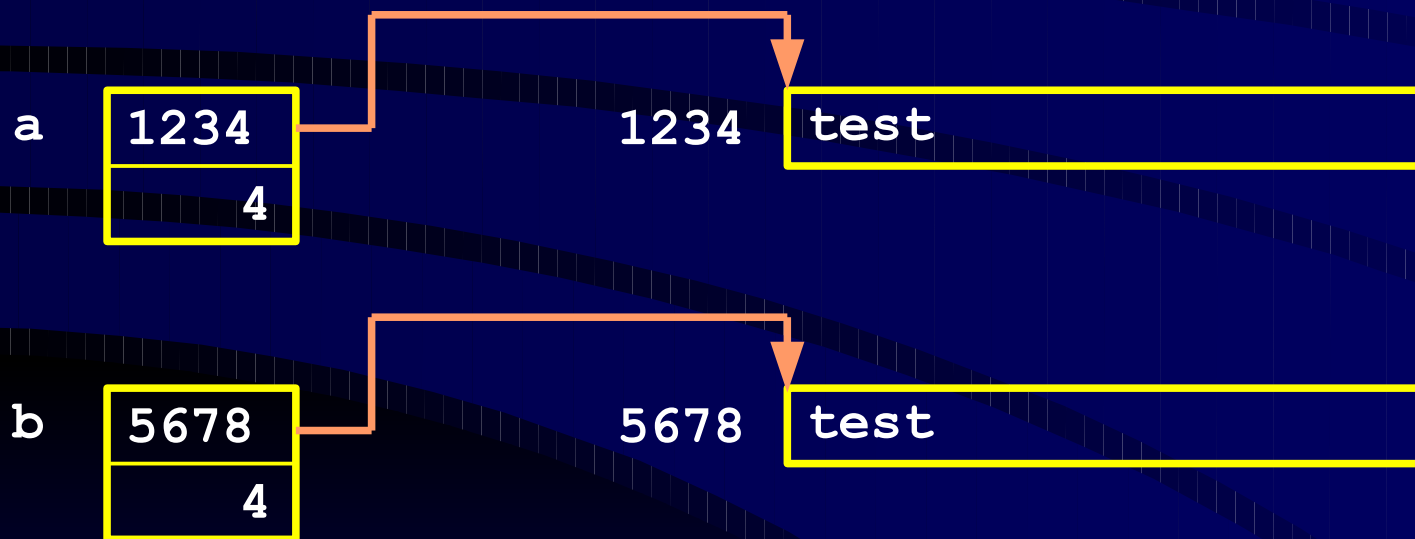

Kopírující konstruktor mělká kopie



Kopírující konstruktor

```
class Cfoo
{
    char * str;
    int   len;
public:
    ...
        Cfoo ( const Cfoo & src )
        {
            len      = src . len;
            str      = new char [len + 1];
            strncpy ( this -> str, src . str, len + 1 );
        }
    ...
};
```

Kopírující konstruktor hluboká kopie



Kopírující konstruktor a rychlost

- Kopírující konstruktor stojí mnoho času:
 - pokud je instance velká,
 - pokud je instance složitá (např. kopie stromu).
- Místo předávání objektů hodnotou:

```
class CFoo { ... };  
void Bar ( CFoo x ) { ... }
```

- Je rychlejší předávat pouze referenci či ukazatel (pokud objekt ve funkci `Bar` neměníme):

```
void Bar ( CFoo & x ) { ... }
```

- Ještě lepší je předávat konstantní referenci či ukazatel (kompilátor kontroluje, že objekt skutečně nezměníme):

```
void Bar ( const CFoo & x ) { ... }
```

Kopírující konstruktor a Java

- Má Java kopírující konstruktor?
- Problém #1 – uvolnění paměti:
 - garbage collector počítá reference,
 - paměť nikdy není uvolňovaná 2x (vícekrát).
- Problém #2 – sdílení dat:
 - nepracuje se přímo s objekty,
 - objekty jsou přístupné přes reference,
 - je samozřejmé, že reference na týž objekt sdílí data.
- Je-li třeba vytvořit kopii objektu v Javě:
 - interface `java.lang.Cloneable`,
 - metoda `java.lang.Object.clone`.

Přímé volání konstruktoru

- Konstruktore je volaný při vzniku instance automaticky:
 - součást volání operátoru `new`,
 - volán systémem při vytváření statické instance.
- Konstruktore lze zavolat explicitně:
 - vytvoří se nová instance,
 - instance se použije (zkopíruje, předá při volání, ...),
 - nakonec instance automaticky zaniká.
- Využití – pro konverze mezi objektovými typy.

```
class CFoo { ... };  
void foo ( CFoo x ) { ... }
```

```
foo ( CFoo ( 10 ) );  
foo ( * new CFoo ( 10 ) ); // !! chyba - jak delete
```

Konstruktor uživatelské konverze

- Konstruktor volatelný s jedním parametrem:
 - má právě jeden parametr,
 - má více parametrů, ale druhý (a všechny další) parametry mají implicitní hodnoty.
- Má-li konstruktor tvar:
`T (T1 x);`
- Pak se konstruktor uživatelské konverze bude automaticky volat tam, kde je očekávána instance typu `T` a skutečnou hodnotou je parametr typu `T1`.

Konstruktor uživatelské konverze

```
class CFoo
{
    public:
        ...
        CFoo ( int x ) { ... }
        ...
};

void foo1 ( CFoo a ) { ... }
void foo2 ( CFoo & a ) { ... }
void foo3 ( const CFoo & a ) { ... }

CFoo a(1);
CFoo b = 10; // CFoo b = CFoo ( 10 )

foo1 ( 5 ); // foo1 ( CFoo ( 5 ) );
foo2 ( 10 ); // chyba
foo3 ( 20 ); // foo3 ( CFoo ( 20 ) );
```


Konstruktor uživatelské konverze

- Uživatelská konverze – potenciální zdroj chyb:
 - méně přehledný kód,
 - problémy při přetěžování.
- Automaticky vkládanou uživatelskou konverzi lze u konstruktoru potlačit:
 - klíčové slovo `explicit`,
 - konverzi lze stále použít, je potřeba volání konstruktoru celé vypsát.

Konstruktor uživatelské konverze

```
class CFoo
{
    public:
        ...
        explicit CFoo ( int x ) { ... }
        ...
};

void foo1 ( CFoo a ) { ... }
void foo2 ( const CFoo & a ) { ... }

CFoo a(1);
CFoo b = 10;           // chyba
CFoo c = CFoo ( 10 ); // ok, ale neefektivni

foo1 ( 20 );          // chyba
foo1 ( CFoo ( 20 ) ); // ok
foo2 ( CFoo ( 20 ) ); // ok
```

Třídy a datové struktury

- Datové struktury často dynamicky alokují paměť:
 - vybavení destruktorem,
 - kopírující konstruktor,
 - přetížený operátor = (příští přednášku).
- Příklad z minulé hodiny (zásobník):
 - doplnění kopírujícího konstruktora pro realizaci polem,
 - doplnění kopírujícího konstruktora pro realizaci spojovým seznamem.

Třídy a datové struktury

```
class CStack
{ ...
    int     dataNr;           // first free index
    int     dataMax;        // max. size
    int *   data;           // dyn. alloc data
};

CStack::CStack ( const CStack & src )
{
    int i;
    dataNr = src . dataNr;
    dataMax = src . dataMax;
    data = new int [dataMax]; // deep copy
    for ( i = 0; i < dataNr; i ++ )
        data[i] = src . data[i];
}
```

Třídy a datové struktury

```
CStack x;  
int y;  
  
x . Push ( 10 ); x . Push ( 20 ); x . Push ( 30 );  
CStack z = x;  
  
x . Pop ( y ); x . Push ( 50 );  
  
cout << "Stack x" << endl;  
while ( ! x . IsEmpty ( ) )  
    { x . Pop ( y ); cout << y << endl; }  
  
cout << "Stack z" << endl;  
while ( ! z . IsEmpty ( ) )  
    { z . Pop ( y ); cout << y << endl; }
```

Třídy a datové struktury

```
class CStack
{ ...
    struct TItem
    {
        TItem * Next;
        int    Data;
    };
    TItem * top; // linked list
};
```

Třídy a datové struktury

```
CStack::CStack ( const CStack & src )
{
    TItem * n, *prev = NULL, *tmp;
    // tmp prochazi zdrojovy spoj. seznam.
    // prev ukazuje na posledni vlozeny prvek
    // vytvareneho seznamu.
    for ( tmp = src . top; tmp; tmp = tmp -> Next )
    {
        n = new TItem;
        n -> Data = tmp -> Data;
        if ( prev ) prev -> Next = n; else top = n;
        prev = n;
    }
    if ( prev ) prev -> Next = NULL; else top = NULL;
    // ukoncit vytvareny seznam.
}
```

Třídy a datové struktury

```
CStack::CStack ( const CStack & src ) // alternativa
{
    TItem * n, **wr = &top, *tmp;
    // tmp prochazi zdrojovy spoj. seznam.
    // wr ukazuje na misto, kam ma byt zapsana adresa
    // nove pridavaneho prvku vytvareneho spojoveho
    // seznamu
    for ( tmp = src . top; tmp; tmp = tmp -> Next )
    {
        n = new TItem;
        n -> Data = tmp -> Data;
        *wr = n;
        wr = &n -> Next;
    }
    *wr = NULL;
    // ukoncit vytvareny seznam.
}
```


Dotazy...

Děkuji za pozornost.