Y36PJC Programování v jazyce C/C++

Ukazatele, dynamická alokace paměti, řetězce

Ladislav Vagner

Dnešní přednáška

- Dynamická alokace paměti v C a v C++.
- Ukazatele a vícerozměrná pole.
- Ukazatele na funkce.
- Řetězce v C a v C++.

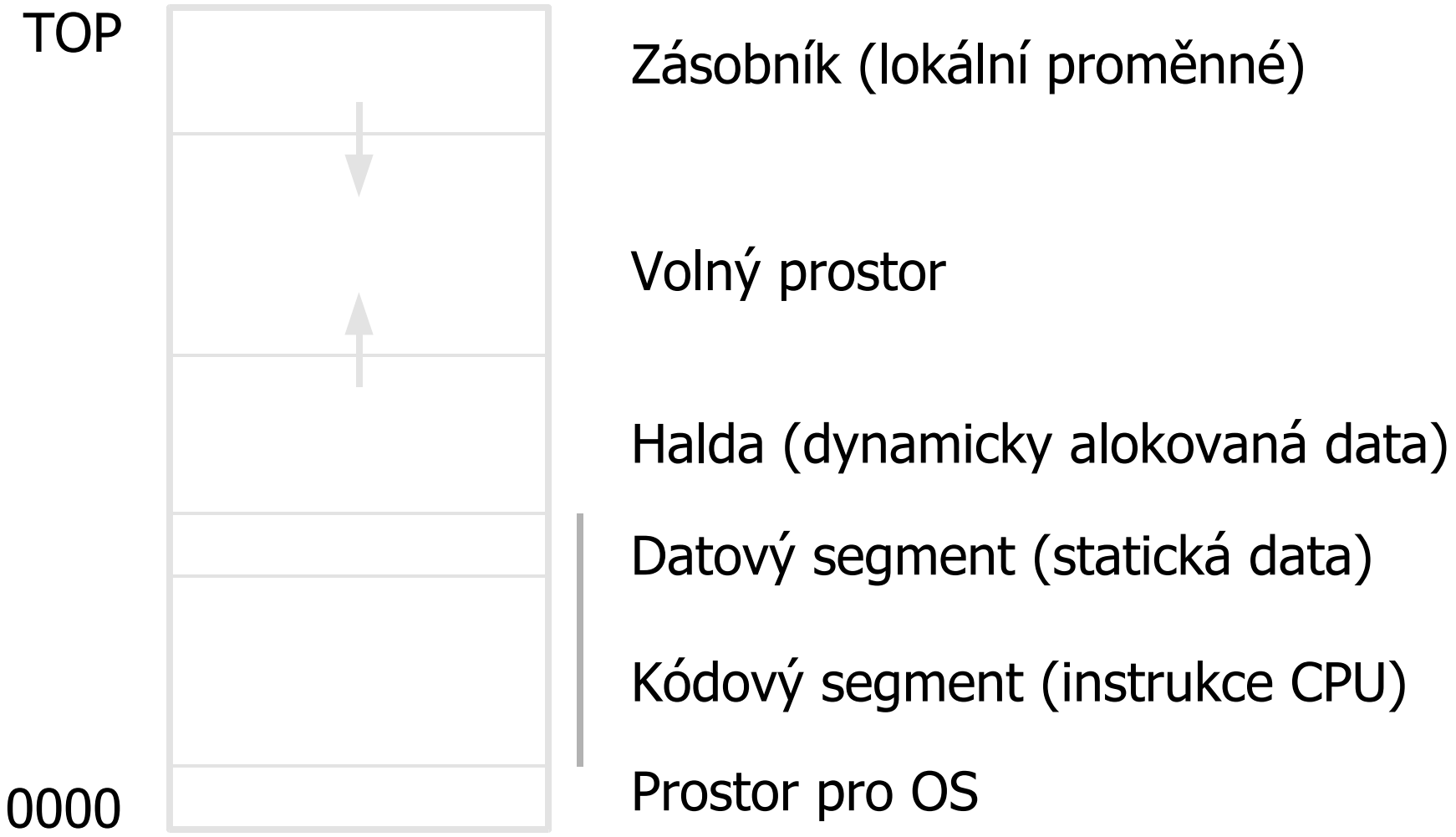
Minulá přednáška

- Staticky alokovaná pole.
- Vícerozměrná staticky alokovaná pole.
- Ukazatele.
- Ukazatelová aritmetika.
- Reference, vztah k ukazatelům.

Dynamická alokace paměti

- Statická alokace – v době překladu se rozhodne o umístění proměnné:
  - globální a lokální s paměťovou třídou **`static`** – datový segment,
  - lokální – zásobník.
- Dynamická alokace – program řídí za chodu přidělování paměti požadovaným proměnným.
- Nevýhoda – program je zodpovědný za:
  - alokaci,
  - uvolnění (v C/C++ není garbage collector).
- Výhoda - velikost alokované paměti je možné přizpůsobit velikosti řešeného problému.

Přidělení paměti procesu



Dynamická alokace paměti v C++

- Alokace paměti – 2 varianty operátoru **`new`**:

```
new <typ> [ <velikost> ]
```

- Operátor alokuje paměť pro pole prvků požadovaného typu a počtu, vrátí ukazatel na tento alokovaný prostor.

```
new <typ>
```

- Operátor alokuje paměť pro jeden prvek požadovaného typu, vrací ukazatel na něj.
- Používá se zejména s objekty.

Dynamická alokace paměti v C++

- Uvolnění paměti – 2 varianty operátoru **`delete`**:

`delete [] <adresa>`

- Operátor uvolní paměť alokovanou pomocí **`new T []`**.

`delete <adresa>`

- Operátor uvolní paměť alokovanou pomocí **`new T`**.
- Adresa je vždy ukazatel vrácený odpovídajícím operátorem **`new`**.

Dynamická alokace paměti v C++

```
int main ( int argc, char * argv [] )
 {
   int i, cnt, *data;
   do {
     cout << "Zadej pocet prvku" << endl;
     cin >> cnt;
    } while ( cnt <= 0 );
   data = new int [cnt];
   for ( i = 0; i < cnt; i ++ )
    cin >> data[i];
   cout << "Reverzovane:" << endl;
   for ( i = cnt - 1; i >= 0; i-- )
    cout << data[i];
   delete [] data;
   return ( 0 );
 }
```

Dynamická alokace paměti v C++

- Vadí, když zapomeneme **`delete`**?
  - Alokovaný blok paměti zůstane označený jako obsazený.
  - Dříve či později paměť dojde.
  - Vadí zejména u programů v režimu 24x7x365 (databáze, WWW server).
- Uvolní se paměť po ukončení programu?
  - Ano, OS odstraní z paměti celý proces.
  - Alokovaná zůstane pouze paměť sdílená mezi procesy.
  - I tak je potřeba po sobě uklízet (využití kódu jinde, předání kódu někomu jinému, ...).

Dynamická alokace paměti v C

- Řešena knihovními funkcemi.
- Paměť se alokuje po bajtech (potřeba přepočítat).
- Vyžaduje přetypování.
- Alokace paměti:
  `malloc ( <počet bajtů> )`

- Uvolnění paměti:
  `free ( <adresa> )`

- Změna velikosti bloku:
  `realloc ( <adresa>, <počet bajtů> )`

Dynamická alokace paměti v C

```
int main ( int argc, char * argv [] )
 {
   int i, cnt, *data;
   do {
     cout << "Zadej pocet prvku" << endl;
     cin >> cnt;
    } while ( cnt <= 0 );
   data = (int*) malloc ( cnt * sizeof ( *data ) );
   for ( i = 0; i < cnt; i ++ )
    cin >> data[i];
   cout << "Reverzovane:" << endl;
   for ( i = cnt - 1; i >= 0; i-- )
    cout << data[i];
   free ( data );
   return ( 0 );
 }
```

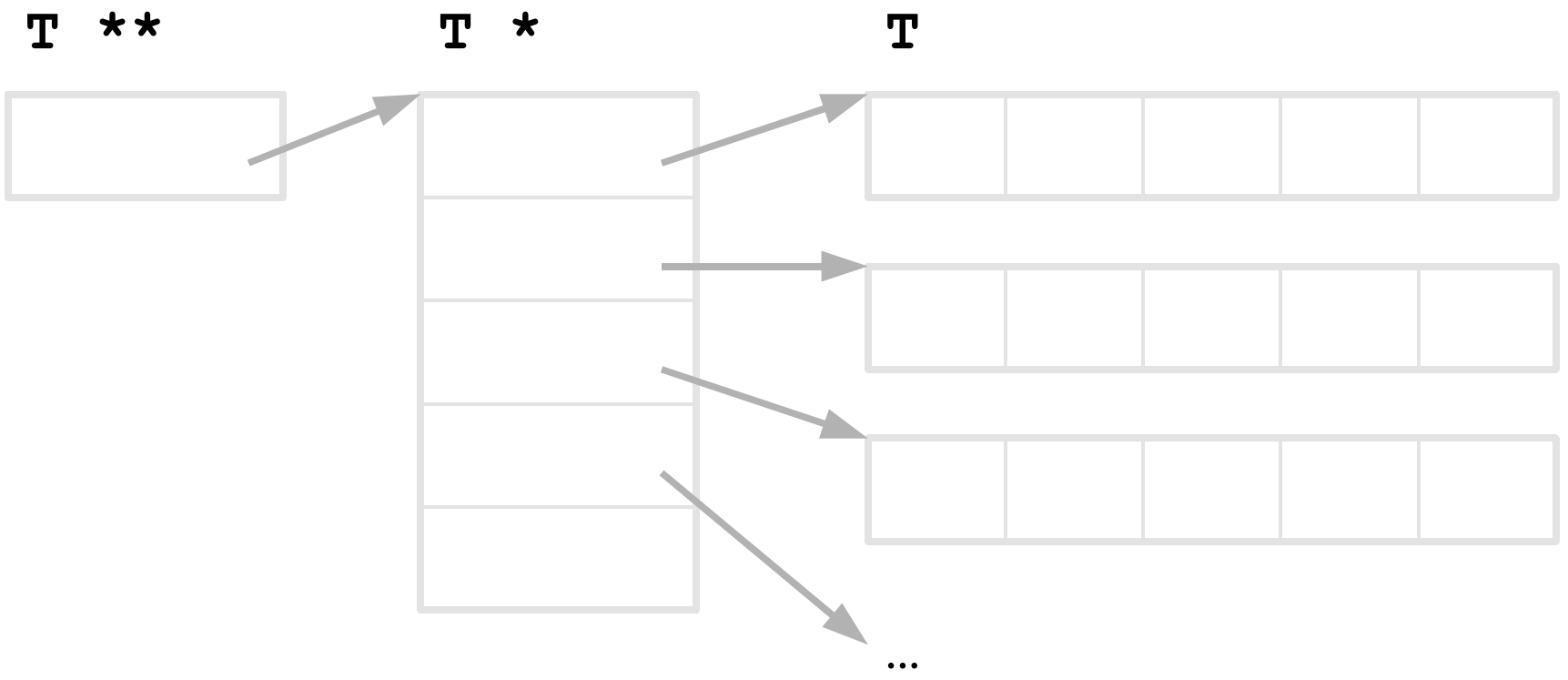
Dynamická alokace paměti v C/C++

- Výhody **`new/delete`**:
  - není potřeba přetypování,
  - lze je použít s objekty,
  - není potřeba přepočítávat velikost na bajty.
- Výhody **`malloc/realloc/free`**:
  - lze efektivněji realokovat blok,
  - ale pouze pro neobjektové datové typy.
- Nemíchejte používání **`new/delete`** s **`malloc/realloc/free`**.

Dynamická alokace vícerozměrných polí

- Obdoba alokace neobdélníkových polí v Javě:
  - alokované jednotlivé řádky,
  - alokován sloupec ukazatelů na řádky.
- Typ ukazatele na řádek – `T *`.
- Typ ukazatele na sloupec – `T **`.
- Uvolnění opakem alokace:
  - uvolněné jednotlivé řádky,
  - uvolněn sloupec odkazů.

Dynamická alokace vícerozměrných polí



Dynamická alokace vícerozměrných polí

```
double ** allocMatrix ( int rows, int cols )
 {
   int i;
   double ** mat;

   mat = new double * [ rows ];
   for ( i = 0; i < rows; i ++ )
    mat[i] = new double [cols];

   return ( mat );
 }
```

Dynamická alokace vícerozměrných polí

```
void freeMatrix ( double ** mat, int rows )
 {
   int i;

   for ( i = 0; i < rows; i ++ )
    delete [] mat[i];

   delete [] mat;
 }
```

Dynamická alokace vícerozměrných polí

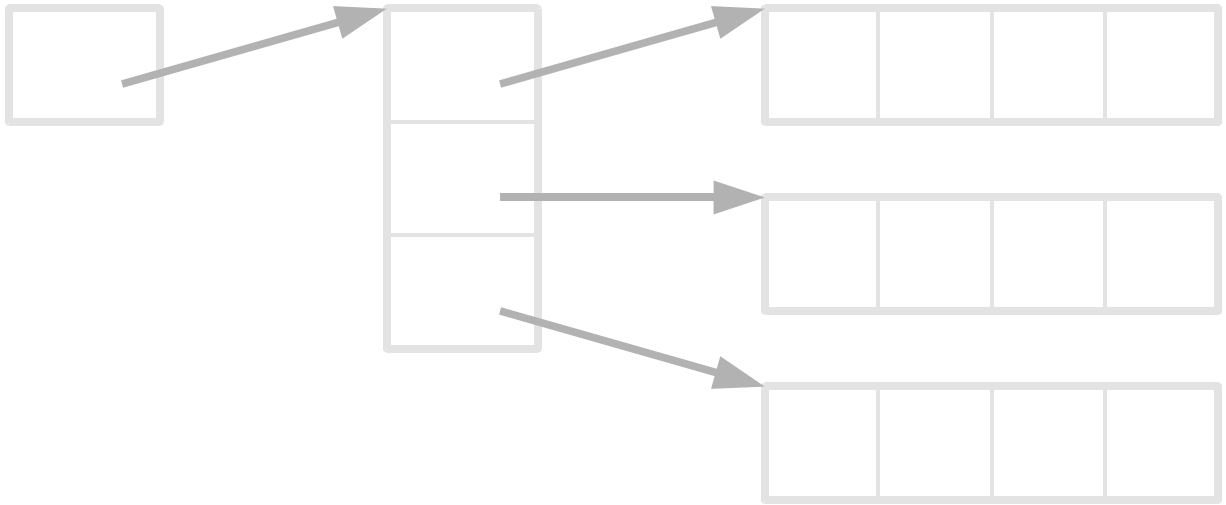
```
void printMatrix ( double ** mat,
                   int rows, int cols )
 {
   int i, j;

   for ( i = 0; i < rows; i ++ )
    {
      for ( j = 0; j < cols; j ++ )
       cout << mat[i][j] << " ";
      cout << endl;
    }
 }
```

Vícerozměrná pole – statická,dynamická

```
int pole[3][4];
```



```
int ** pole = allocMatrix ( 3, 4 );
```



Vícerozměrná pole – statická,dynamická

```
void foo1 ( int ** a, int r, int c );
void foo2 ( int (*a)[20], int r );

int ** mat1 = allocMatrix ( 10, 20 );
int    mat2[10][20];

foo1 ( mat1, 10, 20 );
foo2 ( mat2, 10 );
```

Ukazatele na funkce

- Lze deklarovat ukazatel na funkci.
- Ukazatel lze použít k volání funkce.
- Využití:
  - zpětná volání (callbacks),
  - práce s více vlákny (thready),
  - reakce na události (event handlers, signals),
  - spolupráce s OS.

Ukazatele na funkce

Příklad – realizace inject-into operace nad polem:

```
int max ( int param, int item )
 { return ( item > param ? item : param ); }
int sum ( int param, int item )
 { return param + item; }
int prod ( int param, int item )
 { return param * item; }

int injectInto ( const int * array, int arrayLen,
                 int (*func)( int, int ), int start )
 {
   int res = start, i;
   for ( i =0; i < arrayLen; i ++ )
    res = func ( res, *array ++ );
   return ( res );
 }
```

Ukazatele na funkce

```
int arr [5] = { 1, 2, 3, 4, 5 };

cout << "Maximum: " <<
        injectInto ( arr, 5, max, arr[0] ) << endl;

cout << "Soucet: " <<
        injectInto ( arr, 5, sum, 0 ) << endl;

cout << "Soucin: " <<
        injectInto ( arr, 5, prod, 1 ) << endl;
```

Řetězce v C/C++

- Posloupnost znaků v paměti.
- Ukončené znakem s bin. hodnotou 0.
- Znak:
  - **`char`** - ASCII,
  - **`wchar_t`** - UNICODE.
- Řetězec určen adresou svého počátku v paměti:
  - **`char *`** resp. **`const char *`**,
  - **`wchar_t *`** resp. **`const wchar_t *`**.
- Řetězcové konstanty:
  - znaky řetězce fyzicky uloženy v datovém segmentu,
  - pracujeme s ukazatelem na počátek konstanty,
  - **`"abcd"`** - **`const char *`**
  - **`L"efgh"`** - **`const wchar_t *`**

Řetězce v C/C++

```
char k1 [] = "test";
```

t	e	s	t	\0

```
char k2 [10] = "test";
```

t	e	s	t	\0	\0	\0	\0	\0	\0

```
const char * k3 = "test";
```


```
char * k4 = "test";
```

Práce s řetězci v C/C++

- Práce s řetězci = práce s poli.
- Délka řetězce – počet znaků do ukončující `\0`.
- Nejčastější chyba – špatná alokace paměti.
- Časté operace:
  - **`strlen`** – délka řetězce,
  - **`strncpy,strcpy`** – kopie řetězce,
  - **`strncat,strcat`** – zřetězení,
  - **`strcmp`** – porovnání řetězců.
- Součást standardní knihovny C/C++.
- Hlavičkový soubor **`#include <cstring>`**
- (v C **`#include <string.h>`**).

Délka řetězce v C/C++

```
int strlen ( const char * str );
```

- **strlen** není totéž co **sizeof**.
- Pozor, pro řetězec potřebujete o 1 znak více, než vrátí **strlen** (pro ukončující \0).

```
char t1 [] = "Priklad";
const char * t2 = "a";

cout << strlen ( t1 ) << ","<< sizeof ( t1 ); // 7, 8
cout << strlen ( t2 ) << ","<< sizeof ( t2 ); // 1, 4
t2 = t1;
cout << strlen ( t2 ) << ","<< sizeof ( t2 ); // 7, 4
```

Kopie řetězců v C/C++

```
char * strncpy (char * dst, const char * src, int n);
char * strcpy  (char * dst, const char * src);
```

- **`strcpy`** kopíruje znaky zdrojového řetězce do cílového. **`strncpy`** také, ale kopíruje maximálně **`n`** znaků.

```
char         t1 [8] = "Priklad";
const char * t2;
char         t3 [10];
t3        = t1;                          // !! chyba
strncpy   ( t2, t1, sizeof ( t2 ) ); // !! chyba
strncpy   ( t3, t1, sizeof ( t3 ) );
//strcpy  ( t3, t1 ); zde ok, ale radeji nepouzivat
t2        = t1;
t1[0]     = "p";
cout << t1 << t2 << t3; // priklad priklad Priklad
```

Zřetězení řetězců v C/C++

```
char * strncat (char * dst, const char * src, int n);
char * strcat  (char * dst, const char * src);
```

- **`strcat`** kopíruje znaky zdrojového řetězce za konec cílového. **`strncat`** také, ale kopíruje maximálně **`n`** znaků.

```
char         t1 [8] = "Priklad";
const char * t2;
char         t3 [20];
char         t4 [10];
t3      = t1 + " zretezeni";        // !! chyba
t2      = t1 + " zretezeni";        // !! chyba
strncpy ( t3, t1, sizeof ( t3 ) );
strncat ( t3, " zretezeni", sizeof(t3)-strlen(t3)-1);
strcpy  ( t4, t1 );
strcat  ( t4, " zretezeni" );       // !! chyba
cout << t3 << t4;
```

Porovnání řetězců v C/C++

```
int strcmp(const char * s1, const char * s2 );
```

- `== 0` - shodné řetězce,
- `< 0` - první řetězec je lexikograficky menší,
- `> 0` - první řetězec je lexikograficky větší.

```
char          t1 [8] = "Priklad";
char          t2 [8] = "Priklad";

if ( t1 == t2 )
  cout << "Jsou shodne";       // nejsou

if ( !strcmp ( t1, t2 ) )
  cout << "Jsou shodne";       // jsou
```

Řetězce v C/C++ - časté chyby

```
char str1[5] = "Pokus"; // a co ukoncujici 0

char * str2;
char * str3[20];
char   str4[10];

cout << "Zadej jmeno" << endl;
cin >> str2;
      // !! neinicializovany ukazatel

cin >> str3[0];
      // str3 je pole 20 neinicializovanych ukazatelu
      // str3[0] je neinicializovany ukazatel

cin >> str4;
      // co kyz zadame vice nez 10 znaku
```

Řetězce v C/C++ - časté chyby

Jak správně načíst řetězec ze vstupu:

```
char   str[10];

cout << "Zadej jmeno" << endl;
cin >> setw ( sizeof ( str )) >> str;
      // proste omezime delku
```

Řetězce v C/C++ a buffer overflow

- Načítání řetězců bez omezení délky – chyba typu buffer overflow.
- Paměť procesu je přepsána vstupními daty.
- Přepsaná paměť – typicky část zásobníku (aktivační záznam).
- Využitím buffer overflow lze na vzdáleném počítači provést vlastní kód:
  - získání přístupu k počítači,
  - zcizení přístupových kódů,
  - zcizení dat / úprava dat (např. navýšení zůstatku na vlastním kontě...).

Řetězce v C/C++ a buffer overflow

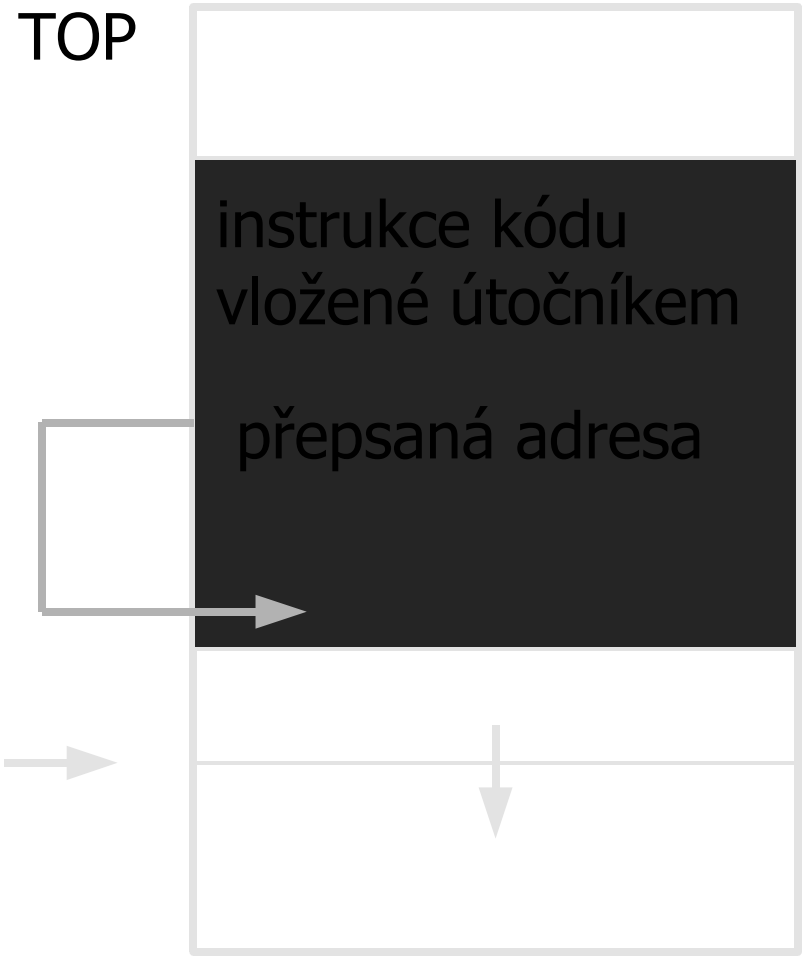
- Zásobník typicky roste směrem k nižším adresám.
- Na zásobníku jsou typicky lokální proměnné a servisní data (návratová adresa).
- Pokud zapisujeme do lokální proměnné za její hranici, můžeme přepsat návratovou adresu.
- Na tuto adresu je předáno řízení po skončení kódu volané funkce
- Útočník typicky:
  - kód (instrukce CPU) vloží na zásobník do proměnné typu řetězec (např. URL požadavku),
  - přepíše návratovou adresu tak, aby směřovala na jeho kód.

Řetězce v C/C++ a buffer overflow

```
void foo (void)
 {
   char req[1024];
   …
   cin >> req;
   …
 }
```

Řetězce v C/C++ a buffer overflow

```
void foo (void)
 {
   char req[1024];
   …
   cin >> req;
   …
 }
```



Ochrana proti buffer overflow

- Alokovat velké buffery:
  - NE - vždy lze poslat ještě delší požadavek.
- Požádat, aby požadavek nebyl delší než XYZ bajtů:
  - NE – útočník asi neposlechne.
- Využívat HW ochranu kódových segmentů (execute-only, HW šifrování/dešifrování instrukcí při zavádění do paměti/načítání instrukcí):
  - NE – není 100% ochrana, pouze záplata na stávající špatně napsaný SW.
- Omezit velikost načítaného vstupu při každém načítání/kopírování:
  - ANO.

Řetězce v C++

- Realizované třídou string ze standardní knihovny.
- Hlavičkový soubor: **`#include <string>`**
- Přetížené operátory pro kopírování a zřetězování.
- Metody pro porovnávání, hledání, vkládání, …

```
string a = "ab", b = "cd";

a += b;                  // zretezeni
cout << a << " " << a . length () << endl; // abcd 4

b = a;                   // kopie
a . at ( 3 ) = 'z';  // pristup ke znakum
cout << a << " " << a . length () << endl; // abcz 4
cout << b << " " << b . length () << endl; // abcd 4
```

Dotazy...

Děkuji za pozornost.