

# PB071 – Principy nízkoúrovňového programování

Preprocesor, assert, varargs, zbývající klíčová slova C99

**Slidy pro komentáře (děkuji!):**

<https://drive.google.com/file/d/1n4E1Rq8A1yLQ8OisQxk5k4TejJaeX-BX/view?usp=sharing>

# Preprocesor

# Překlad po částech

## 1. Preprocessing "gcc -E hello.c > hello.i"

- rozvinutí maker, expanze include...

## 2. Kompilace "gcc -S hello.i -o hello.s"

- syntaktická kontrola kódu, typicky chybová hlášení

## 3. Sestavení "as hello.s -o hello.o"

- assembly do strojového kódu

## 4. Linkování "gcc hello.o"

- nahrazení relativních adres absolutními



*Při běžném překladu proběhnou všechny kroky automaticky, nemusíme pouštět každý zvlášť*

# Preprocessor

- Většina příkazů preprocesoru začíná znakem #
- Znak # nemusí být na začátku řádku, ale nesmí před ním být žádný další token
  - NE `int a = 1; #define DEBUG`
  - může být odsazeno
- Dělá textové náhrady nad zdrojovým kódem
  - jazykově nezávislé, "neví" nic o syntaxi jazyka C
- Příkazy preprocesoru jsou z předzpracovaného kódu odstraněny
  - (přidá některé značky pro kompilátor `#line`)

# Preprocessor – makra bez parametrů

- `#define JMÉNO_MAKRA hodnota_makra`
  - jméno\_makra se v kódu nahradí za hodnota\_makra
- Označení jmen maker velkými písmeny je jen konvence
  - je zřejmé, co jsou makra a budou tedy nahrazena
- Často používáno např. pro konstanty (legacy kód)

- `#define ARRAYSIZE 10000`

hodnota makra

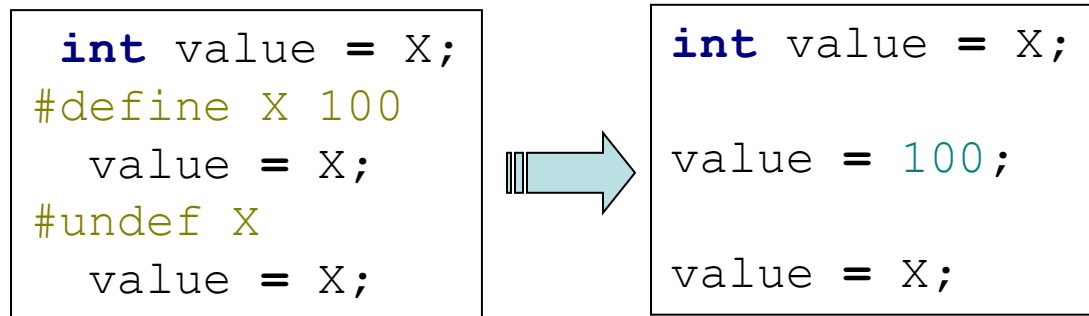
jméno makra

Pozn.: pro konstanty ale raději `const int ARRAYSIZE = 10000;`

- Nahradí se jen samostatné tokeny, nikoli podčásti tokenu
  - `int arraySize = ARRAYSIZE;`
  - `int arraySize = ARRAYSIZEBAD;` => bez změny
  - `int array[ARRAYSIZE];` => `int array [10000];`

# Rozsah platnosti makra

- Makro je v kódu platné od řádku jeho uvedení
  - nahrazení proběhne až pro následující řádky
  - pozor, makro může být platné i v dalších souborech (#include)
- Platnost makra lze zrušit pomocí **#undef** jméno\_makra



- Makro lze definovat v kódu nebo jako přepínač překladač
  - #define DEBUG
  - gcc **-D**jméno\_makra => gcc **-D**DEBUG
  - gcc **-D**jméno\_makra=hodnota\_makra => gcc **-D**ARRAYSIZE=100

# Makro - redefinice

- Makro může být předefinováno

- často spíše nezáměrná chyba, proto varování preprocesoru

- `warning: "VALUE" redefined`

```
#define VALUE 100  
#define VALUE 1000
```

- Pokud potřebujete předefinovat, oddefinujte nejprve předchozí

```
#define VALUE 100  
#undef VALUE  
#define VALUE 1000
```

- Hodnota makra může být prázdná

- `#define DEBUG`
- často použito pro podmíněný překlad (viz. následující)

# Preprocessor – podmíněný překlad

- Chceme vždy přeložit celý zdrojový kód?
  - ne nutně, část kódu může vynechat
  - chceme mít zároveň jediný zdrojový kód, ne násobné (nekonzistentní) kopie
- Např. v ladícím režimu chceme dodatečné výpisy
  - `#ifdef DEBUG printf("Just testing")`

```
#if (DEBUG > 0) && (DEBUG < 2)
printf("Debug level 1");
#endif
```
- Např. máme části programů závislé na platformě
  - little vs. big endian, Unix vs. Windows
  - `#ifdef _WIN32`
- Příkazy preprocesoru pro podmíněný překlad
  - `#if, #ifdef, #ifndef, #else, #elif, #endif` (`#elifndef` od C23)
- Podmínky se vyhodnotí v době překladu!



# Ukázka #ifdef

```
#include <stdlib.h>
#include <stdio.h>

#define VERBOSE

int main(void) {
    int a = 0;
    int b = 0;
    scanf("%d %d", &a, &b);
#ifdef VERBOSE
    printf("a=%d b=%d\n", a, b);
#endif
    printf("a+b=%d\n", a + b);

    return 0;
}
```

Zakomentováním odstraníme  
dílčí výpis

Namísto definice VERBOSE v kódu  
můžeme použít:  
Přepínač překladače  
gcc -DVERBOSE  
(Pokud používáte IDE, hledejte  
příslušné nastavení pro překladač)

Pokud není VERBOSE  
definován, tento řádek  
nebude vůbec přítomný ve  
výsledné binárce

# Zamezení opakovanému vkládání souboru

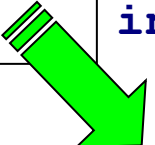
- Opakované vložení hlavičkového souboru je nepříjemné
  - překladač hlásí násobnou deklaraci funkcí apod.
  - obtížné pohlídat vkládání souboru jen jednou "manuálně"
- S pomocí podmíněného překladače lze řešit
  - vložení souboru podmíníme (ne-)existující definicí makra
    - `#ifndef JMENOSOUBORU_H`
  - ve vkládaném souboru makro definujeme
    - `#define JMENOSOUBORU_H`
- Při prvním vkládání se obsah souboru vloží a zadefinuje makro JMENOSOUBORU\_H, které zamezí dalšímu vložení
- U souborů \*.c se typicky nepoužívá
  - nepoužíváme `#include "soubor.c"` ale `#include "soubor.h"`

```
#ifndef MYLIB_H
#define MYLIB_H
// obsah souboru mylib.h
#endif
```

# Preprocesor – makra s parametry

- Makra můžeme použít pro parametrizaci nahrazení opakujícího se kódu (tzv. function-style macro)
- `#define JMÉNO_MAKRA (argumenty) tělo_makra`
- Preprocesor nahradí (rozvine) výskyty makra včetně jejich zadaných argumentů
  - argumenty makra v definici se nahradí za reálné argumenty při použití makra

```
#define ADD(a, b) (a + b)
int main(void) {
    int z = ADD(5, 7) * 3;
    return 0;
}
```



```
int main(void) {
    int z = (5 + 7) * 3;
    return 0;
}
```

```
#define ADD(a, b) (a + b)
int main(void) {
    int x = 0;
    int y = 0;
    int z = ADD(x, y);
    z = ADD(x, 3.4);

    return 0;
}
```

# Preprocesor – makra s parametry

```
#define ADD(a, b) (a + b)
int add(int a, int b) {
    return a + b;
}
int main(void) {
    int x = 0;
    int y = 0;
    int z = ADD(x, y);
    z = add(x, y);

    return 0;
}
```

```
23      int z = ADD(x, y);
0x00401399 <+85>:      mov     0x48(%esp),%eax
0x0040139d <+89>:      mov     0x4c(%esp),%edx
0x004013a1 <+93>:      add     %edx,%eax
0x004013a4 <+96>:      mov     %eax,0x44(%esp)

24      z = add(x, y);
0x004013a8 <+100>:     mov     0x48(%esp),%eax
0x004013ac <+104>:     mov     %eax,0x4(%esp)
0x004013b0 <+108>:     mov     0x4c(%esp),%eax
0x004013b4 <+112>:     mov     %eax,(%esp)
0x004013b7 <+115>:     call   0x401d5c <_Z3addii>
0x004013bc <+120>:     mov     %eax,0x44(%esp)
```

- Pozor na Debug vs. Release (optimalizace)

# Makro vs. Funkce inline

```
inline int add(int a, int b) {  
    return a + b;  
}  
// ...  
int z = add(x, y);
```

- Makra s parametry typicky zavedeny z důvodu rychlosti
  - pokud je standardní funkce, musí se připravit zásobník...
  - např. jednoduché sečtení dvou čísel může znatelně zpomalovat
- Při použití makra vložen přímo kód funkce namísto volání
- Optimalizující překladač ale může sám vložit kód funkce namísto volání (tzv. inlining)
  - rychlostní optimalizace na úkor paměti (delší kód)
  - překladače jsou obecně v optimalizaci velice dobré!
- Pomocí klíčového slova **inline** signalizujeme funkci vhodnou pro vložení
  - překladač ale může ignorovat (jen doporučení)
- Ladění **inline** funkcí je výrazně snazší než maker!

# Makra – problémy s typem

- Makro nemá žádný typ

- jde o textovou náhradu během preprocessingu
- velmi náchylné na textové překlepy

- Např. pozor na

```
#define ARRAYSIZE 10000;  
int array[ARRAYSIZE];
```

- hodnotou makra je zde `10000; // int array[10000];`
- => chyba, ale tuto odhalí už překladač

- Překladač nemůže kontrolovat typovou správnost

- zvyšuje se riziko nesprávné interpretace dat

- Často problém díky automatickým implicitním konverzím

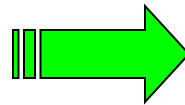
- díky automatické konverzi překladač neohlásí chybu
- paměť s konstantou 100 je interpretována jako řetězec
- => chyba se projeví až za běhu ☹

```
#define VALUE 100  
printf("%s", VALUE);
```

# Makra – problémy s rozvojem

- Velký pozor na přesný výsledek rozvoje

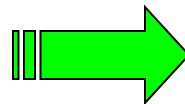
```
#define ADD(a, b) a + b
int main(void) {
    int z = ADD(5, 7) * 3;
    return 0;
}
```



```
int main(void) {
    int z = 5 + 7 * 3;
    return 0;
}
```

**používejte preventivní  
uzávorkování**

```
#define ADD(a, b) (a + b)
int main(void) {
    int z = ADD(5, 7) * 3;
    return 0;
}
```



```
int main(void) {
    int z = (5 + 7) * 3;
    return 0;
}
```

# Makra - shrnutí

- Makra se vyhodnocují při překladu, nikoli při běhu
- Snažte se minimalizovat jejich používání
  - `#include "library.h"` ✓
  - `#ifndef SOUBOR_H` ✓
  - `#define MAX 10 ...` raději `const int MAX = 10;` ✓
  - `#define VALUE_T int ...` lépe `typedef int VALUE_T;` ✓
- Používejte inline funkce namísto funkčních maker
  - **inline int** `add(int, int);` namísto `#define ADD(a,b) (a+b)`
- Podmíněný překlad častý při využití platformově závislých operací ✓



# Zbývající klíčová slova jazyka C

# Klíčová slova C99

<https://en.cppreference.com/w/c/keyword>

`_Bool`

`_Complex`

`_Imaginary`

`auto`

`break`

`case`

`char`

`const`

`continue`

`default`

`do`

`double`

`else`

`enum`

`extern`

`float`

`for`

`goto`

`if`

`int`

`inline`

`long`

`register`

`restrict`

`return`

`short`

`signed`

`sizeof`

`static`

`struct`

`switch`

`typedef`

`union`

`unsigned`

`void`

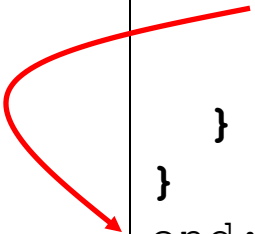
`volatile`

`while`

# Problém s goto

- goto skočí na místo označené návěštím (nepodmíněný skok)
- Syntaxe: `goto jméno_návěští;`
- Návěští znáte z příkazu switch
  - jméno\_návěští:

```
for (/*anything*/) {  
    if (error_occured) {  
        error = -1;  
        goto end;  
        // step outside all nested blocks  
    }  
}  
end:  
// do some error handling
```



# Použití goto (zkráceno)

```
#include <stdio.h>
int foo() {
    int error = 0;
    for (int i = 0; i < 10; i++) {
        // Regular code, possible allocation of resources (memory, files)
        if (i < 0) {
            error = -1; // Shall not happen
            goto out;
        }
        if (i > 10) {
            error = -2; // Shall not happen
            goto out;
        }
    }

    out:
        // Release unneeded resources here (fclose(), free()...)

        // Check if error happened based on variable error (error != 0)
        if (error != 0) { printf("Some error: %d", error); }
        else { printf("No error"); }
        return error;
}

int main() {
    foo();
}
```

```

#include <stdio.h>
int foo() {
    int error;
    for (int i = 0; i < 10; i++) {
        // Regular code, possible allocation of resources (memory, files)

        if (i < 0) { // Shall not happen
            error = -1;
            goto out;
        }
    }
    // Code was executed and there are no errors
    error = 0;

out:
    // Release unneeded resources here (fclose(), free()...)

    // Check if error happened based on variable error (error != 0)
    if (error != 0) { printf("Some error: %d", error); }
    else { printf("No error"); }
    return error;
}

int main() {
    foo();
}

```

```

for (int i = 0; i < 10; i++) {
00007FF63DF517D2 mov     dword ptr [rbp+24h],0
00007FF63DF517D9 jmp     foo+33h (07FF63DF517E3h)
00007FF63DF517DB mov     eax,dword ptr [rbp+24h]
00007FF63DF517DE inc     eax
00007FF63DF517E0 mov     dword ptr [rbp+24h],eax
00007FF63DF517E3 cmp     dword ptr [rbp+24h],0Ah
00007FF63DF517E7 jge     foo+4Ch (07FF63DF517FCh)
        // Regular code, possible allocation of resources (memory, files)

        if (i < 0) { // Shall not happen
00007FF63DF517E9 cmp     dword ptr [rbp+24h],0
00007FF63DF517ED jge     foo+4Ah (07FF63DF517FAh)
            error = -1;
00007FF63DF517EF mov     dword ptr [error],0FFFFFFFh
            goto out;
00007FF63DF517F6 jmp     $out (07FF63DF51803h)
        }
    }
00007FF63DF517FA jmp     foo+2Bh (07FF63DF517DBh)

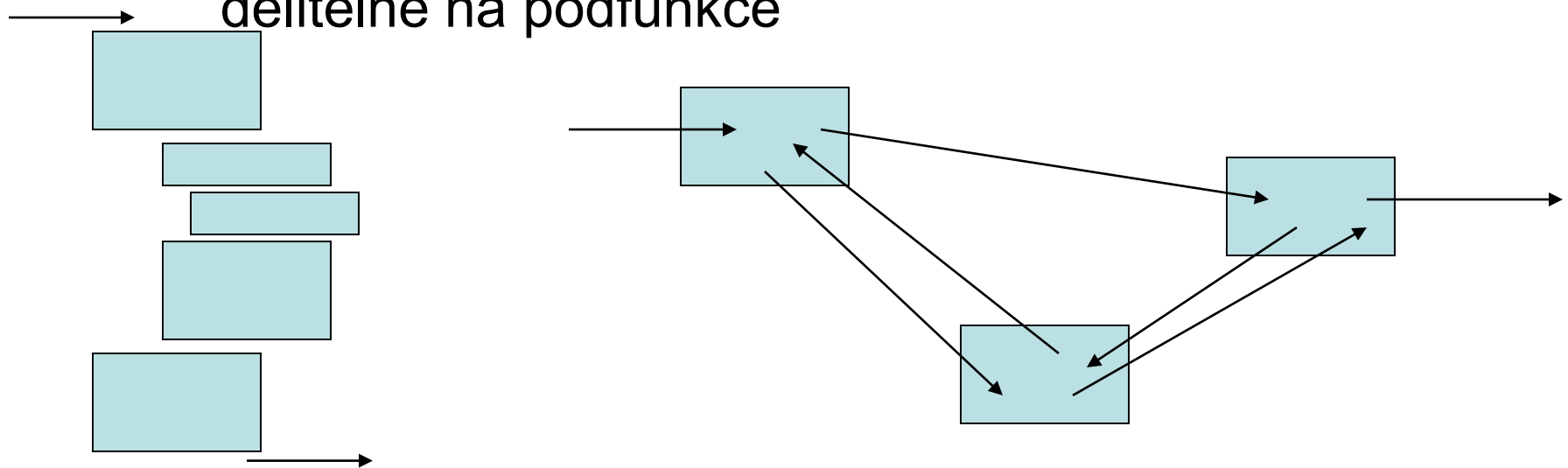
out:
    // Release unneeded resources here (fclose(), free()...)

    // Check if error happened based on variable error (error != 0)

```

# goto – styl programování

- Strukturované programování
  - přemýšlíme o funkcích, které mají vstup a výstup
  - problém řešíme jejich vhodným poskládáním
- Využití goto typicky vede ke špagetovému kódu
  - dlouhé kusy kódu, silně provázané, jen obtížně dělitelné na podfunkce



# goto – další informace

- Kód s goto typicky snižuje čitelnost a ztěžuje ladění
  - z pohledu na kód lze těžko říct, které části se provedou
- Kód s goto lze vždy přepsat na kód bez něj
- Dijkstra. *Go To Statement Considered Harmful*.  
Communications of the ACM 11(3),1968
  - [http://www.u.arizona.edu/~rubinson/copyright\\_violations/Go\\_To\\_Considered\\_Harmful.html](http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html)
- Problém není v samotném slovu, ale ve způsobu použití
  - a stylu programování, ke kterému svádí
  - <https://web.archive.org/web/20120604082341/http://blog.julipedia.org/2005/08/using-gotos-in-c.html>

# goto – korektní použití

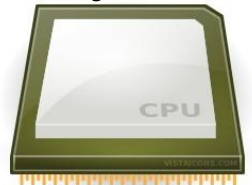
- C nepodporuje výjimky (na rozdíl od C++, Java...)
- goto může poskytnout rozumný způsob ošetření chy a uvolňování zdrojů při násobném vnoření bloků
- Celkově se ale nedoporučuje používat
  - protože lze vždy přepsat bez něj
  - speciálně začátečníky svádí k nevhodnému stylu programování
- Příklad správného použití viz seminář 8, příklad 3
  - <https://gitlab.fi.muni.cz/pb071/seminars/-/blob/master/seminar-08/solution/cipher.c#L184-289>



# Modifikátory u proměnných

# Koncept umístění hodnot v paměti

- Procesor (registry CPU)
  - vykonání instrukce procesoru potřebuje argumenty v registrech
- RAM (zásobník)
  - register ESP ukazuje na aktuální pozici v zásobníku
  - lokální proměnné
- RAM (halda)
  - dynamicky alokovaná paměť
- Ostatní paměť (HDD, ...)
  - umístění dat mimo paměť aktuálního programu
  - např. soubory na disku



# Motivace – sečtení dvou čísel z/do souboru

## 1. Načtení hodnot z HDD do RAM paměti

- `fscanf(file, "%d", &value);`

## 2. Přesun hodnot z RAM paměti do registru CPU

- `MOV 0x48(%esp), %eax`
- `MOV 0x44(%esp), %edx`

## 3. Provedení instrukce procesoru (např. ADD)

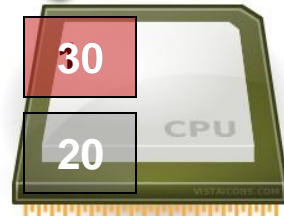
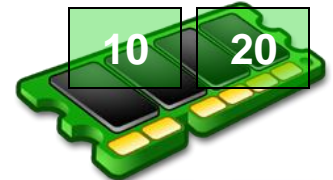
- `ADD %edx, %eax`

## 4. Uložení výsledku registru CPU do RAM

- `MOV %eax, 0x48(%esp)`

## 5. Uložení výsledku z RAM do souboru

- `fprintf(file, "%d", value);`



# Klíčové slovo `auto`

```
auto int a = 10;
```

- Defaultní paměťová třída pro lokální proměnné
  - automatický vznik na zásobníku
  - automatické odstranění při konci bloku
- V kódu se tedy explicitně neuvádí
  - (pozor, v C++11 má jiný význam)
    - (automatické doplnění typu)

```
void foo() {  
    auto int a = 10;  
    printf("%d\n", a);  
    a += 10;  
}  
  
int main(void) {  
    foo(); // => 10  
    foo(); // => 10  
    foo(); // => 10  
    return 0;  
}
```

# Klíčové slovo `static`

```
static int a = 10;
```

- Proměnná deklarovaná se `static` zachová svou hodnotu i po konci bloku s deklarací
- Statické proměnné jsou inicializovány v době překladač
  - trvalé místo pro proměnnou stejně jako pro globální proměnné
  - při novém "vzniku" proměnné obsahuje poslední předešlou hodnotu
- Zachování hodnoty proměnné je jen v rámci jednoho spuštění programu
- Proměnná se `static` je lokální v rámci souboru
  - Vyplývá z faktu, že je viditelná nejvýše v bloku s její deklarací

# Klíčové slovo static - ukázka

```
void foo() {
    static int a = 10;
    printf("%d\n", a);
    a += 10;
}

int main(void) {
    foo(); // => 10
    foo(); // => 20
    foo(); // => 30
    return 0;
}
```

- Využíváno je zřídka (např. `strtok()` – každé zavolání vrátí další část vstupu – musí si pamatovat mezistav)
- Zavádí globální stav podobně jako globální proměnné
  - Typicky nežádoucí (funkce nejsou samostatné)
  - **static** proměnná má viditelnost omezenou pouze na blok deklarace => nižší pravděpodobnost nechtěného použití

# Klíčové slovo `volatile`

```
volatile int a = 10;
```

- Proměnná může být měněna i mimo náš kód
  - rutinou přerušení, sdílená paměť...
  - Pouze z analýzy zdrojového kódu nelze určit místa změny proměnné
- Vynutí nahrání a zápis hodnoty proměnné ze zásobníku do registru CPU před každou operací
  - nelze provést optimalizaci předpokládající přítomnost hodnoty proměnné v registru z předchozí operace
  - pokud by došlo ke změně mimo náš kód, hodnota by nebyla aktuální
    - Pozor, neznamená atomičnost práce s proměnnou (nutné použít C11 `_Atomic`)

# Klíčové slovo `register`

```
register int a = 10;
```

- Doporučení pro překladač, aby byla proměnná uložena přímo v registru CPU
  - před vykonáním instrukce musí být hodnoty do registru přeneseny (instrukce `mov` atp.)
  - pokud je ale již v registru přítomná => zrychlení
- CPU má ale jen omezený počet registrů
  - `register` je jen doporučení, překladač může ignorovat
- Některé proměnné mohou být umístěny v registru i bez specifikace `register`
  - překladač sám analyzuje a vybere často používané proměnné
  - Obecně se použití `register` nedoporučuje (překladač je optimálnější v jejím použití)



# Klíčové slovo register - ukázka

```
void foo4() {  
    register int value = 10;  
    printf("%d\n", value);  
    value += 12;  
}
```

Pro proměnnou value  
byl vyhrazen registr ebx  
Číslo 10 (0xa) je přímo  
uloženo do registru ebx

```
34      register int value = 10;  
0x004013c6 <+7>:      mov     $0xa,%ebx  
  
35      printf("%d\n", value);  
0x004013cb <+12>:     mov     %ebx,0x4(%esp)  
0x004013cf <+16>:     movl   $0x402034,(%esp)  
0x004013d6 <+23>:     call   0x401ca0 <printf>  
  
36      value += 12;  
0x004013db <+28>:     add    $0xc,%ebx
```

Číslo 12 (0xc) je přímo  
přičteno k registru ebx

## Debug mód – proměnná **i** je na adrese [ebp-20h]

```
for (int i = 0; i < 10; i++) {
00FA1807 jmp      main+72h (0FA1812h)
00FA1809 mov     eax,dword ptr [ebp-20h]
00FA180C add     eax,1
00FA180F mov     dword ptr [ebp-20h],eax
00FA1812 cmp     dword ptr [ebp-20h],0Ah
00FA1816 jge     main+8Bh (0FA182Bh)
        printf("%d", i);
00FA1818 mov     eax,dword ptr [ebp-20h]
00FA181B push    eax
00FA181C push    offset string "%d" (0FA6B3Ch)
00FA1821 call   _printf (0FA1320h)
00FA1826 add     esp,8
    }
00FA1829 jmp     main+69h (0FA1809h)
```

## Release mód – proměnná **i** pouze v registru esi

```
for (int i = 0; i < 10; i++) {
00D1104E xor     esi,esi
        printf("%d", i);
00D11050 push   esi
00D11051 push   offset string "%d" (0D12
00D11056 call   printf (0D11010h)
00D1105B inc     esi
00D1105C add     esp,8
00D1105F cmp     esi,0Ah
00D11062 jl     main+10h (0D11050h)
    }
```

- Některé proměnné mohou být umístěny v registru **i** bez specifikace register

# Klíčové slovo `restrict` (od C11)

```
void foo(int* restrict pA, int* restrict pB, int* restrict pVal);
```

- Paměťová třída pouze pro ukazatel
  - slibujeme překladači, že na danou paměť ukazuje jen tento a žádný jiný používaný ukazatel
  - pokud bude paměť měněna, tak pouze přes tento ukazatel
- Překladač může generovat optimalizovanější kód
  - např. nemusí nahrávat opakovaně hodnotu do registru, pokud je zřejmé, že nebyla změněna
  - příklad viz. <http://en.wikipedia.org/wiki/Restrict>
- Pokud porušíme, dojde k nedefinovanému chování
  - zvažte, zda rychlostní optimalizace vyváží riziko zanesení chyby

# Klíčové slovo `extern`

```
extern int globalValue;
```

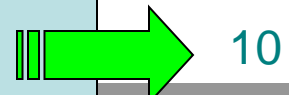
- Proměnná nebo funkce je definovaná jinde, typicky v jiném zdrojovém souboru
- Defaultní volba pro funkční prototyp
  - po vložení hlavičky s prototypem může překladač pokračovat, aniž by znal implementaci funkce
- Až během linkování se hledá implementace funkce resp. umístění proměnné
- Pro proměnné se používá v případě globální proměnné dostupné z několika zdrojových kódů
  - Nepoužívejte, pokud nevíte přesně proč potřebujete
  - Vhodné např. pro *mutex* (proměnná sloužící k zamykání sdíleného zdroje - např. bloku paměti - při vícevláknovém programování)

`file1.c`

```
int globalValue = 10;
```

`main.c`

```
extern int globalValue;
int main(void) {
    printf("%d\n", globalValue);
    return 0;
}
```



10

# Modifikátory u proměnných - shrnutí

- Dodatečné modifikátory mohou pomoci:
  - optimalizovat rychlost (**register**, **restrict**, **const**)
  - zamezit chybám z optimalizace (**volatile**)
  - zamezit chybám programátora (**const**)
  - ovlivnit životní cyklus proměnné (**auto**, **static**)
  - změnit umístění proměnné (**register**, **extern**)
- Některé modifikátory jsou jen doporučení pro překladač (**register**, **auto**, **inline**)
- Nejprve piště korektní kód, optimalizujte až poté!

**<assert.h>**

# <assert.h> – pomocník při ladění

- Při psaní kódu předpokládáme platnost některých podmínek
  - invarianty / konzistence stavu
  - např. energie Avatara nikdy neklesne pod 0
  - např. pokud je zavolána funkce `attack()`, tak by Avatar měl být živý
- Pro potřeby snazšího ladění lze tyto invarianty hlídat
  - a snadno získat lokalizaci místa, pokud je invariant porušen
- **void** `assert` (**int** `expression`)
  - pokud se `expression` vyhodnotí na `false` (0), tak vypíše identifikaci řádku s `assert()` na standardní chybový výstup a program skončí
  - některé vývojové nástroje umožní připojení debuggeru
    - např. MS Visual Studio
- Pozor, využití jen pro ladící režim!

# Assert - ukázka

```
#include <assert.h>
#include <string.h>
#include <stdbool.h>
int main() {

    assert(1 + 1 == 2);
    assert(strlen("Hello world") == 11);

    int value = 0;
    assert(value == 0);
    assert(value != 0);
    assert(false);

    return 0;
}
```



# Assert – vhodnost použití

*Pozn.: Assert použijeme pro ošetření tzv. neočekávatelných chyb (test co by mělo vždy platit).*

*Očekávatelné chyby (chyby od uživatelského vstupu, selhání knihovných volání) ošetříme standardní podmínkou*

- NEjedná se o ošetřování uživatelského vstupu a syst. chyb
  - tam je nutné použít běžnou podmínku v kódu
  - jde o stav (proměnné...) uvnitř našeho kódu, která není typicky přímo nastavována uživatelem
  - pokud je ale podmínka porušena, značí to nekorektní chování našeho kódu → chceme rychle najít
- NEpoužívá se pro detekci chyby v produkčním kódu
  - zde je makro odstraněno → podmínky se nevyhodnocují
  - `#define assert(ignore)((void) 0)`
- **Pozor na `assert(foo())` !!!**
  - v Debug režimu funguje dobře
  - v Release režimu se `foo()` vůbec nevolá – je odstraněno!

# Výňatek z <assert.h>

```
#ifndef NDEBUG
/*
 * If not debugging, assert does nothing.
 */
#define assert(x)          ((void)0)

#else /* debugging enabled */
/*
 * CRTDLL nicely supplies a function which does the actual output and
 * call to abort.
 */
_CRTIMP void __cdecl __MINGW_NOTHROW _assert(const char*,const char*,int)
    __MINGW_ATTRIB_NORETURN;

/*
 * Definition of the assert macro.
 */
#define assert(e)          ((e) ? (void)0 : _assert(#e, __FILE__, __LINE__ ))

#endif /* NDEBUG */
```

**Pokud je definováno makro NDEBUG (Release), tak nedělej nic**

**Pokud se e vyhodnotí na true, tak nedělej nic**

**e vyhodnoceno na false => reaguj**

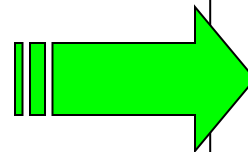
# Assert – překlad pro Release

```
#include <assert.h>
#include <string.h>
int main() {

    assert(1 + 1 == 2);
    assert(strlen("Hello world") == 11);

    int value = 0;
    assert(value == 0);
    assert(value != 0);
    assert(false);

    return 0;
}
```



```
#define NDEBUG
#include <assert.h>
#include <string.h>
int main() {

    ((void) 0);
    ((void) 0);

    int value = 0;
    ((void) 0);
    ((void) 0);
    ((void) 0);

    return 0;
}
```

# Funkce s proměnným počtem argumentů

# Funkce s proměnným počtem argumentů

- Některé funkce má smysl používat s různými počty a typy argumentů
  - `printf("Hello 4 world");`
  - `printf("%s%c %d %s", "Hell", 'o', 4, "world");`
  - nemá smysl definovat funkce pro všechny možné kombinace
- Argumenty na konci seznamu lze nahradit výpustkou `...`
  - `int printf ( const char * format, ... );`
  - první argument je formátovací řetězec, dále 0 až N argumentů
- Výslovně uvedené argumenty jsou použity normálně
- Argumenty předané na pozici výpustky jsou přístupné pomocí dodatečných maker
  - hlavičkový soubor `stdarg.h`
  - `va_start`, `va_arg` a `va_end`

# Přístup k argumentům

1. Definujeme ve funkci proměnnou typu `va_list`
  - `va_list arguments;`
2. Naplníme proměnnou argumenty v proměnné části
  - `va_start(arguments, name_of_previous_argument);`
3. Opakovaně získáváme jednotlivé argumenty
  - `va_arg(arguments, type_of_argument);`
4. Ukončíme práci s argumenty
  - `va_end(arguments);`

# Poznámky k argumentům

- Seznam argumentů lze zpracovat jen částečně a předat další funkci (která může zpracovat zbytek)
- Jazyk C neumožňuje zjistit počet argumentů při volání funkce
  - lze přímo předat prvním parametrem:
    - `void foo(int num, ...);`
  - lze odvodit z prvního argumentu:
    - `int printf(const char* format, ...);`
    - `format = "%s%c %d %s" -> 4 args, char*,char,int,char*`

# Proměnný počet argumentů - příklad

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
void varFoo(int number,...) {
    va_list arg;
    va_start(arg,number);
    int valueInt = va_arg(arg,int);
    char* valueChar = va_arg(arg,char*);
    printf("%d dynamic params are: %d, %s\n",
           number,valueInt,valueChar);
    va_end(arg);
    return;
}
int main(void) {
    varFoo(2,123,"end");
    return 0;
}
```



# Proměnný počet argumentů – na zásobník

```
printf("%s%d%d", "pb071", 42, 1);
```

```
c7 44 24 0c 01 00 00 00 movl $0x1,0xc(%esp)
c7 44 24 08 2a 00 00 00 movl $0x2a,0x8(%esp)
c7 44 24 04 64 40 40 00 movl $0x404064,0x4(%esp)
c7 04 24 6a 40 40 00    movl $0x40406a,(%esp)
e8 26 11 00 00        call 0x402778 <printf>
```

# Shrnutí

- Preprocessor
  - Od začátku důležitá součást tvorby C programů
  - Defacto textové nahrazování bez typů => nevýhody
    - Postupně částečné nahrazování (const, inline...)
- goto (skok na zadané návěští)
  - Neinformované použití vede k nepřehlednému kódu
  - Nepoužívejte (pokud ano, tak přesně chápejte)
- Paměťové třídy proměnné
  - Ovlivňují, jak s nimi překladač nakládá (optimalizace)
- Funkce s proměnným počtem argumentů
  - Často používané ve standardní knihovně (printf(“%d%s”, ...))
  - Možná chyba typů proměnné (neignorujte varování překladače)

# Co dál po PB071?

# Možné návaznosti PB071

- Programování v jazyce C++ (PB161)
  - principy objektově orientovaného programování
  - základy jazyka C++ (STL, šablony...), moderní C++ (11, 17...)
- Programování v jazyce Java (PB162)
- Úvod do vývoje v C#/.NET (PV178)
- Tématický vývoj aplikací (PB173)
  - zaměření na řešení praktických programátorských problémů v oblasti vašeho zájmu
  - tématické skupiny: Zpracování obrazu, Systémové programování Linux a Windows, Ovladače jádra Linux, Aplikovaná kryptografie a bezpečnost...
  - lze zapisovat opakovaně (různé seminární skupiny)
- (2025) „Plnotučné C“
  - Nový předmět po reorganizaci nízkoúrovňového programování
  - Zaměření na pokročilejší použití C, standard C23

# Možné návaznosti PB071 (pokračování)

- Programování se nenaučíte na cvičeních ve škole
- Najděte si zajímavý open source projekt
  - zkuste v něm odstranit reportovanou chybu + PULL request
  - zkuste implementovat nějakou funkčnost z Issues
  - Někdy označeno jako `good first issue`
- Vyberte si zajímavou laboratoř na škole
  - stačí chuť se učit novým věcem
- Nebojte se jiných jazyků
  - schopnost programovat je platformově nezávislá 😊

**DÍKY ZA VÁŠ ČAS A V  
PONDĚLÍ ZASE NA VIDĚNOU**



# Bonus 😊

```
btcbonus*
```

```
btcbonus**
```

```
~xsvenda/btcbonus.zip
```

