

# Preprocessor

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

© Karel Richta, Martin Hořeňovský, Aleš Hrabalík, 2018

Programování v C++, B6B36PJC

11/2018, Lekce 9b

<https://cw.fel.cvut.cz/wiki/courses/b6b36pic/start>



# Preprocesor

**Příklad:** *Definice symbolické konstanty*

```
#define MAX 100  
printf("MAX=%d\n", MAX);
```



**MAX=100**

**Příklad:** *Definice makra*

```
#define SUM(X,Y) ((X)+(Y))  
printf("SUM(3,7)=%d\n", SUM(3, 7));
```



**SUM(3, 7) = 10**

# Ukázka programu v C++ se symb. konst.

```
/* převodní tabulka Fahrenheit - Celsius: C = (5/9)(F-32) */
#include <iostream>
#define POCATEK 0
#define KONEC 300
#define KROK 20
int main() {
    int F; float C;
    F = POCATEK;
    while (F <= KONEC) {
        C = (5.0 / 9.0) * (F - 32);           // spočte stupně v C
        std::cout << F << " - " << C << "\n"; // tiskne řádek tabulky
        F = F + KROK;                         // posune na další F
    }
    return 0;
}
```

definice  
symbolických  
konstant (makra  
bez parametrů)

# Ukázka programu v C++ s konstantami

```
/* převodní tabulka Fahrenheit - Celsius: C = (5/9)(F-32) */  
#include <iostream>  
const int pocatek = 0;  
const int konec = 300;  
const int krok = 20;  
int main()  
{  
    int F; float C;  
    F = pocatek;  
    while (F <= konec) {  
        C = (5.0 / 9.0) * (F - 32); // spočte stupně v C  
        std::cout << F << " - " << C << "\n"; // tiskne řádek tabulky  
        F = F + krok; // posune na další F  
    }  
    return 0;  
}
```

definice  
konstantních  
buněk

# Preprocesor (doplňky)

Konstanty zavedené pomocí **const** a **#define**:

- Poskytují podobnou funkcionalitu.
- **#define** je v C i v C++, **const** není v C89
- Debugger zná konstanty zavedené **const**, ale nezná konstanty nahrazené preprocesorem.
- Konstanty zavedené **#define** nerespektují jmenné prostory a pravidla zastiňování.
- **#define** je lepší použít, pokud píšeme rozhraní dynamicky linkované knihovny:
  - nešikovné řešení s **const** zavádí zbytečné relokace,
  - u dynamicky linkovaných knihoven není předem jasné, kde všude budou použity (zda vždy pouze v C++).

# Preprocesor (doplňky)

## Podmíněný překlad:

```
#if konstantní výraz
#ifdef jméno
#ifndef jméno
#elif konstantní výraz
#else
#endif
```

## Příklad:

```
#if MAX > 10
    Zde je kód pro MAX > 10
#else
    Zde je kód pro MAX <= 10
#endif
```

# Preprocesor (doplňky)

**Příklad:** vícenásobné větvení

```
#if MAX > 10
```

```
    Zde je kód pro MAX > 10
```

```
#elif MAX > 5
```

```
    Zde je kód pro MAX <= 10 a MAX > 5
```

```
#else
```

```
    Zde je kód pro MAX <= 5
```

```
#endif
```

# Preprocesor (doplňky)

**Příklad:** *korektní předefinování makra*

```
#ifdef MAX  
#undef MAX  
#endif  
#define MAX 100
```



# Preprocesor (doplňky)

**Příklad:** stráže hlaviček knihoven (umožňují libovolný počet vložení hlavičky)

```
#ifndef __STDIO_H
#define __STDIO_H
...
tělo knihovny
...
#endif
```

**Alternativa:**

```
#pragma once
```

# Preprocesor (doplňky)

- Preprocesor nabízí některá předdefinovaná makra:
  - **\_\_FILE\_\_** jméno zdrojového souboru,
  - **\_\_LINE\_\_** číslo řádky v aktuálním souboru,
  - **\_\_func\_\_** jméno aktuálně překládané funkce,
  - **\_\_cplusplus** příznak, zda je zdrojový kód kompilován překladačem C nebo C++.
- Kompilátory pak nabízejí předdefinovaná makra specifická pro výrobce/platformu:
  - **WIN32** příznak kompilace pro Windows,
  - **BORLANDC** Borland kompilátory,
  - **DEBUG** příznak kompilace pro ladění,
  - **RELEASE** příznak kompilace pro ostré nasazení.

# Preprocesor (doplňky)

Předdefinovaná makra mohou usnadnit ladění:

```
int divide(int num, int denom) {
    if (denom == 0)
    {
        cout << "Deleni nulou" << endl <<
            " funkce: " << __func__ <<
            " soubor: " << __FILE__ <<
            " radek: " << __LINE__ << endl;
        return (0);
    }
    return (num / denom);
}
```

# Preprocesor – operátory # a ##

- Definice maker připouští dva speciální operátory (# a ##) v těle makra:
- Operátor #, následovaný jménem parametru se nahradí řetězcem, který je parametrem (jako by byl uveden v uvozovkách):

```
#define str(x) #x  
cout << str(test);
```

- Bude nahrazeno textem:

```
cout << "test";
```

- Operátor ## spojí dva argumenty (aniž je oddělí mezerou):

```
#define glue(a,b) a ## b  
glue(c,out) << "test";
```

- Bude rovněž nahrazeno textem:

```
cout << "test";
```

# Preprocesor – direktiva `#line`

- Direktiva `#line` umožňuje změnu hlášení pozice chyby nalezené překladačem.

- Direktiva má tvar:

```
#line number "filename"
```

- kde jméno souboru není povinné.

- Např.:

```
10. #line 20 "cosi.txt"
```

```
11. int a7;
```

- Způsobí, že překladač hlásí chybu v souboru "`cosi.txt`" na řádku 20, místo v aktuálním souboru na řádku 11. Lze to použít pro odkazy na zdrojový kód z něž bylo C++ generováno.

# Preprocesor – direktiva `#error`

- Direktiva `#error` způsobí, že překladač hlásí chybu a ukončí překlad.

- Direktiva má tvar:

```
#error text chybové zprávy
```

- Např.:

```
#ifndef __cplusplus
```

```
#error A C++ compiler is required!
```

```
#endif
```

- Způsobí, že pokud není makro `__cplusplus` definováno, překladač ohlásí chybu `A C++ compiler is required!` a ukončí překlad.

# Preprocesor – direktiva #pragma

- Direktiva **#pragma** je určena pro speciální parametry různých prostředí. Pokud překladač direktivě neporozumí (neumí parametry interpretovat), ignoruje ji.

- Direktiva má tvar:

**#pragma parametry**

- Např.:

**#pragma once**

- Způsobí u některých překladačů, že soubor, který ji obsahuje bude překládán jen jednou.
- Pokud překladač této direktivě nerozumí, nebude jedinečný překlad zajištěn. Proto je lepší kombinovat tuto direktivu s běžnými strážemi hlavičkových souborů zajištěných podmíněným překladem.

# Preprocesor – assert

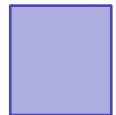
- Pomoc při ladění – makro **assert** v hlavičkovém souboru **<assert.h>** (**<cassert>**)
- Makro má jako parametr podmínku:
  - je-li podmínka platná, nic se nedělá,
  - není-li podmínka platná, zobrazí pozici a podmínku, která vedla k jeho vyvolání. Zároveň ukončí program.
- Volání **assert** se typicky nechají v kódu:
  - pro ladění se použijí,
  - ve finální verzi se hromadně odpojí definicí:

```
#define NDEBUG
```



# Preprocesor – assert (příklad)

```
void print_number(int* myInt) {  
    assert (myInt!=NULL);  
    printf ("%d\n",*myInt);  
}  
  
int main() {  
    int a = 10;  
    int *b = NULL;  
    int *c = NULL;  
  
    b = &a;  
  
    print_number(b);  
    print_number(c);  
}
```



Assertion failed: myInt != NULL, file  
d:\výuka\fel\a7b36pjc\prednasky\2015\priklady\assert\assert\assert.cpp, line 12  
abnormal program termination

# Deklarace funkce s proměnným počtem parametrů

- Funkce s **proměnným počtem parametrů** (nutno použít standardní knihovnu `<stdarg.h>`)

```
#include <stdarg.h>
```

```
int K(int, ...); /* proměnný počet parametrů */  
int printf(char *format, ...);
```

# Definice funkce s proměnným počtem parametrů

- Hlavička: paměťová třída, typ návratové hodnoty, jméno funkce a seznam deklarácí pevných parametrů zakončený ... (ellipsis):

```
#include <stdarg.h>
long Sum(int n, ... )
{
    long result = 0; va_list pt;
    va_start(pt, n);
    while (n-- >= 0) result += va_arg(pt, int);
    va_end(pt);
    return result;
}
long l = Sum(4, 2, 45, 3, 17);
```

- Při volání funkcí s proměnným počtem parametrů se kontrolují pevné parametry, proměnné parametry se samozřejmě kontrolovat nedají.

**The End**