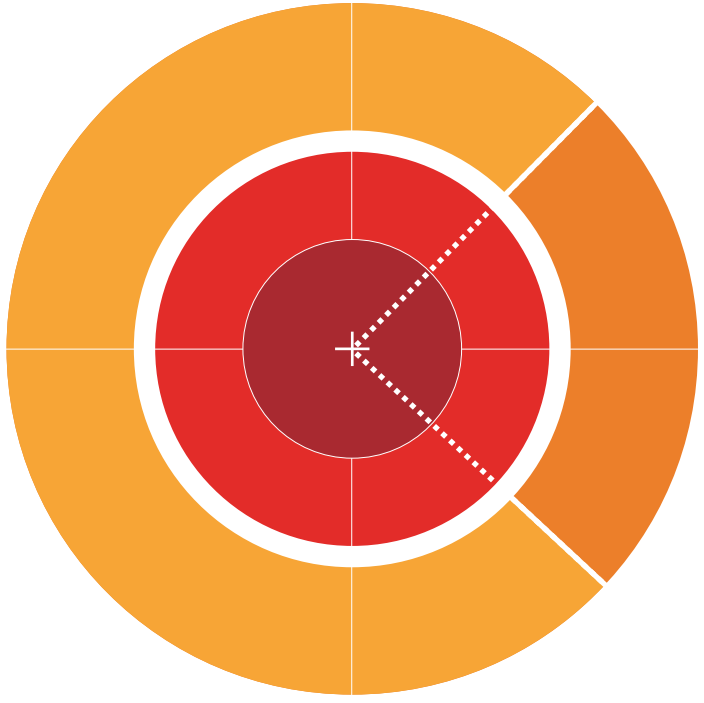
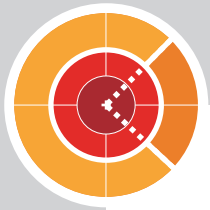


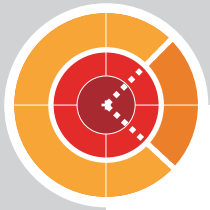


Preprocesor jazyka C -

- makro procesor (lexikálně-syntaktický analyzátor), který zpracovává zdrojový text programu ještě před překladačem
- příkazy preprocesoru jsou součástí ZK, vždy na samostatné řádce, začínají znakem **#**
- při preprocessingu se řádky s příkazy preprocesoru odstraní; výsledkem předzpracování musí být platný program v C
- po rozvinutí makra preprocesoru se ve výsledném programu už nesmí vyskytovat žádné příkazy preprocesoru:

```
#define INCLUDEMATH #include <math.h>
INCLUDEMATH
```

rozvinutím makra INCLUDEMATH vznikne další příkaz preprocesoru, který se ale předá překladači, který ho nezná



Lexikální konvence preprocesoru

- nepřekládá ZK, ale rozděluje ho na atomy
- nevejde-li se makro na jeden řádek, je možné pokračovat na dalším po ukončení předchozího řádku zpět. lomítkem

```
#define err(flag, msg) if (flag) \
    printf(msg)
```

- **POZOR:** řádek následující po řádku, který končí \, se nikdy nepovažuje za řádek preprocesoru, i když začíná #

```
#define BACKSLASH \
#define ASTERISK *
```

tuto definici preprocesor **ignoruje**

Definice a nahrazení - #define

- ve zdrojovém textu C **nahradí** výskyt identifikátoru makra tělem makra

```
#define sum(x,y) x+y
...

i = sum(5, a * b);
```

preprocesor zajistí rozvinutí makra na

```
i = 5 + a * b;
```

- preprocesor **neidentifikuje klíčová slova** jazyka C, takže je možné makro pojmenovat stejně, jako klíčové slovo C, ale to je **nebezpečné a hloupé** (špatný prog styl)

```
#define ITEM_SIZE 0x100
#define BUFFER_SIZE (256*ITEM_SIZE)
#define QUITMSG "Konec...\n"
```

Obvyklé chyby a problémy s #define

```
#define WIDTH = 640
#define HEIGHT 480;
…

size_x = WIDTH;
msize = HEIGHT * 8;
```

preprocesor zajistí rozvinutí makra na
`size_x = = 640;`

preprocesor zajistí rozvinutí makra na
`msize = 480; * 8;`

- **pozor na středníky** za definicí těla makra
- nevkládat **žádné znaky navíc**, za jménem následuje za mezerou okamžitě definice těla makra
- **mezera** slouží jako oddělovač - nespoléhejte na syntaktickou analýzu preprocesoru a nevkládejte mezery do definice těla, příp. identifikátoru makra

Definice makra s parametry

- obecně **#define ident(*ident1*, *ident2*, ... , \ *identN*)** ***posloupnost-atomů***

```
#define add(x,y) ((x)+(y))
...

return add(a + 3, b);
```

závorky nejsou nutné, ale jsou **vhodné** - zajišťují dodržení priority operátorů

```
#define getchar() getc(stdin)
...

while ((c = getchar() != EOF) {...}
```

- makro může mít **nulový počet** p-metrů, v tom případě musí i při volání být seznam p-metrů prázdný

"Rekurze" makra

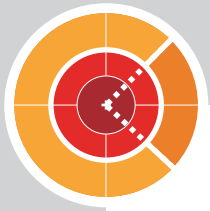
- makra, která se objeví ve svém vlastním rozvoji, se v ANSI C **znovu nerozvíjejí**
- lze definovat novou funkci pomocí původní definice

```
#define sqrt(x) ((x) < 0 ? \
    sqrt(-x) : sqrt(x))
```

- starší preprocesory tuto "rekurzi" nedokážou detekovat a zacyklí se

```
#define plus(x,y) add(y,x)
#define add(x,y) ((x)+(y))
```

v pořádku - makra se nerozvíjejí v definici jiného makra



Příklad makra s parametry

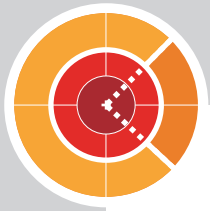
```
#define step(v,l,h) \
   for ((v) = (l); (v) <= (h); (v)++)

int main() {
   int i;

   step(i, 1, 20)
      printf("%2d %6d\n", i, i * i);

   return 0;
}
```

- zdánlivé nadužití závorek zajišťuje správnou interpretaci komlexních parametrů makra (např. položek struktur)

Předdefinovaná makra v ANSI C

__LINE__	číslo právě zpracovávaného řádku programu (desítková celočíselná konstanta)
__FILE__	jméno právě zpracovávaného souboru (řetězcová konstanta)
__DATE__	aktuální kalendářní datum (řetězcová konstanta tvaru *mm dd yyyy*)
__TIME__	čas překladu (řetězcová konstanta tvaru *hh:mm:ss*)
__STDC__	má hodnotu 1 <=> překladač ANSI C

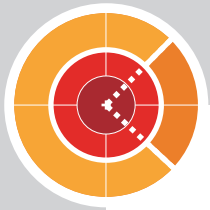
```
if (n != m)
   printf("Chyba na radku %d v souboru %s\n",
      __LINE__, __FILE__);
```



Zrušení makra - příkaz #undef

- definice makra se ruší příkazem **#undef** *<jméno>*
- lze zrušit i makro, které nebylo nadefinováno

```
#define madd(a,b) ((a)+(b))

int main() {
    int i = 3, j = 5, k, l;

    k = madd(i, j);
#undef madd
    l = madd(i, j);  /* chyba */

    return 0;
}
```



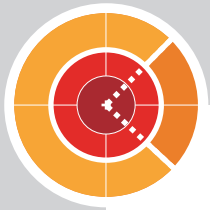
Převod atomů na řetězce - operátor

- **pouze ANSI**, starší preprocesory operátor **#** neznají

```
#define test(a,b) \
    printf(#a "<" #b ": %d\n", (a) < (b))

int main() {
    test(-5, 5);
    return 0;
}
```

```
printf("-5<5: %d\n", (-5)<(5));
```

- každá posloupnost prázdných znaků v rozvoji formálního parametru se nahradí jednou mezerou
- každému znaku \ a " se **předřadí zpětné lomítko**, aby se zachoval jejich význam v řetězci



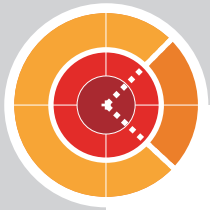
Spojování atomů v rozvoji maker - operátor

- **pouze ANSI**, starší preprocesory operátor **##** neznají

```
#define var(i) var ## i
...
var(1) = var(2);
```

```
var1 = var2;
```

- užitečný nástroj, ale **používat s rozumem**
- častá nutnost použití tohoto operátoru naznačuje nevhodné řešení problému

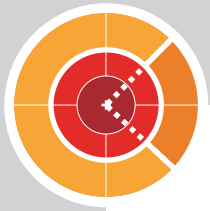


Vkládání souborů - příkaz #include

```
#include <stdio.h>
#include "mylib.h"
#include "ext/mylib.h"
```

Win: lomítko může být \ i /
UNIX: jen /

- je-li jméno vkládaného souboru uzavřeno v **<...>**, hledá se v instalaci překladače
- je-li v **"..."**, pak se prohledává aktuální adresář (kde je ZK)
- vkládaný soubor může sám obsahovat **`#include`** - min. garantovaná povolená hloubka vnoření (ANSI C) je **6**



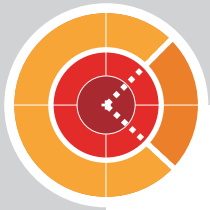
Podmíněný překlad -

preproc2.c

```
int main() {
   printf("Environment: ");
#ifdef WIN32
   printf("Win32\n");
#elif UNIX
   printf("UNIX\n");
#elif VAX
   printf("VAX\n");
#else
   printf("Unknown...\n");
#endif
   return 0;
}
```

použití `#elif` v kombinaci s `#ifdef` funguje, ale není to úplně čisté

```
C:\>cl -DWIN32 preproc2.c
```

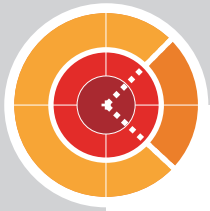


Příkazy #if a #endif

```
#if 1<<16
...
#endif
```

konstantní výraz preprocesoru
(vzhledem k tomu, že preprocesor neprovádí komplexní syntaktickou analýzu, **téměř nic nevyhovuje**)

- tento kód **nesprávně** testuje, zda je **`int`** větší než 16 bitů, ve skutečnosti ale preprocesor pracuje vždy pouze s typem **`long`** nebo **`unsigned long`** podle znaménka operandu
- KVP jen celočíselné konstanty, znakové konstanty a spec. operátor **`defined`** (ANSI C)

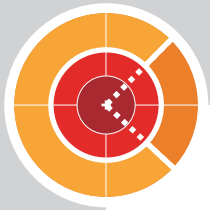


Příkazy #elif a #else a operátor defined

```
...
#if defined(WIN32)
    printf("Win32\n");
#elif defined(UNIX)
    printf("UNIX\n");
#else
    printf("Unknown\n");
#endif
...
```

- silnější nástroj, než prostý **#ifdef** nebo **#ifndef**
- umožňuje budovat složité výrazy

```
#if defined(WIN32) && !defined(UNIX)
...
#endif
```



Příkaz #pragma

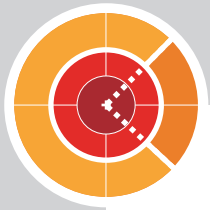
- novinka v ANSI C, slouží k přidávání nových funkcí preprocesoru nebo k **předávání informací překladači** (závisí na implementaci)
- příkaz **`#pragma`** je **silně závislý** na p-formě a implementaci
  => používat podmíněně

```
#if defined(VAX) && defined(__STDC__)
#pragma builtin(abs), inline(myfunc)
#endif
```

- neexistuje dohoda o standardních tvarech, lze nalézt v dokumentaci k překladači

```
align, pack                extend
rom                        small, medium, large
optimize                   debug
check stack                inline
```

ukázka některých tvarů
v současných překladačích



Příkaz #error

- novinka v ANSI C, umožňuje **zachytit nesrovnalosti** např. při překladu řízeném skripty

```
#if defined(WIN32) && defined(UNIX)
#error "Chyba v Makefile!"
#endif
```

```
#include "sizes.h"
...
#if (SIZE % 256) != 0
#error "SIZE musí být násobek 256"
#endif
```