Preprocesorul C

Funcţii cu număr variabil de argumente

6 decembrie 2005

Preprocesorul C

– extensii (macro-uri) pentru scrierea mai concisă a programelor
– preprocesorul efectuează transformarea intr-un program C propriu-zis
– directivele de preprocesare au caracterul # la început de linie

`#include <numefisier>` sau `#include "numefisier"`
– include textual fişierul numit (în mod tipic definiţii)
(a doua variantă: caută întâi în directorul curent apoi în cele standard)

```
#define LEN 20 /* substituţie textuală simplă */
int tab[LEN]; /* pentru definirea de constante simbolice */
for (i = 0; i < LEN; i++) { ... } /* mai robust la modificări */
```

forma generală: `#define nume(arg1,...,argn) substituţie`

```
#define max(A, B) ((A) > (B) ? (A) : (B))
#define swapint(a, b) { int tmp; tmp = a; a = b; b = tmp; }
```

Obs: substituţia se face textual ⇒ pot apărea probleme subtile
– folosiţi paranteze în jurul argumentelor (evită erori de precedenţă)
– argumentele: evaluate la fiecare apariţie textuală (ex. de 2x în max)
⇒ rezultat incorect la evaluarea repetată a expresiilor cu efect lateral

Operatorii # și

Operatorul `#` aplicat unui argument de macro produce un șir de caractere care are ca și conținut argumentul macro-ului.

```
#define mkstring(x)     # x
```

În acest caz, `mkstring(nume)` devine `"nume"`

Operatorul `##` are ca efect concatenarea elementeror lexicale de dinainte și de după. Exemplu:

```
#define concat(x,y)   x ## y
```

În acest caz, `concat(unu, doi))` devine `unudoi`

Obs.: în standardul C două *constante șir* consecutive sunt echivalente cu concatenarea lor: `"unu" "doi"` e echivalent cu `"unudoi"`).

Observație: de regulă, *definiția* unui macro nu include un `;` final; acesta e scris în program după *folosirea* macro-ului, obținând un aspect uniform la scriere (ca și cum ar fi un apel de functie)

Compilarea condiţională

pt. a compila selectiv porţiuni de cod din program în funcţie de opţiuni (caracteristici arhitecturale; pt. depanare; funcţionalitate în plus; etc)

Sintaxa: *grup-cod* ::= *test-if grup-cod grupuri-elif*$_{opt}$ *grup-else*$_{opt}$ `#endif`
test-if ::= `#if` *expr-const* | `#ifdef` *identificator* | `#ifndef` *identificator*
grup-elif ::= `#elif` *expr-const grup-cod* (toate # apar
grup-else ::= `#else` *grup-cod* pe linie nouă)
expr-const ::= cele obişnuite | `defined` *identificator*

```
#define DEBUG /* dacă depanăm */
/* cod obişnuit */
#ifdef DEBUG
printf("am ajuns aici, x = ...");
#endif
/* alt cod obişnuit */
```

```
#if defined __GNUC__ /* compilator GNU */
#if __GNUC__ == 2    /* versiunea 2 */
  /* cod specific pt. versiune */
#else      /* altă versiune */
  /* cod specific pt. altă versiune */
#endif
#endif
```

Identificatori predefiniţi: `__FILE__` `__LINE__` `__DATE__` `__TIME__`

Ex: `fprintf(stderr, "eroare în %s linia %d\n", __FILE__, __LINE__);`

Funcţii cu număr variabil de argumente

Funcţiile de tipul `printf/scanf` au număr variabil de argumente (...)
Pentru a implementa o astfel de funcţie, trebuie un mod de acces la argumentele cu număr variabil, pornind de la ultimul arg. numit.
$\Rightarrow$ limbajul C defineşte o serie de macro-uri în `stdarg.h`
– tipul `va_list` pentru a reţine informaţii despre lista de argumente
`void va_start(va_list ap,`*ultimarg*`);`
– iniţializează `ap` pornind de la adresa ultimului argument
tip `va_arg(va_list ap,` *tip*`);`
– returnează următorul argument din listă, presupus a fi de tipul *tip*
apelată repetat pentru fiecare argument; tipul argumentelor şi numărul lor trebuie deduse din argumentele fixe (ex. formatul la `print/scanf`)
`void va_copy(va_list dest, va_list src);`
– copiază un `va_list`, inclusiv punctul curent de prelucrare atins
`void va_end(va_list ap);`
– apelat pentru încheierea corectă a prelucrării argumentelor

Funcții de intrare-ieșire formatată cu nr. variabil de argumente

– declarații similare, dar in loc de ... apare `va_list ap`:

```
int vprintf(const char *format, va_list ap);
int vscanf(const char *format, va_list ap);
```

similar: `vfprintf`, `vfscanf`, `vsprintf`, `vsscanf`, `vsnprintf`
– se folosesc când dorim să facem întâi o prelucrare preliminară,
și apoi să apelăm `printf/scanf` etc. cu argumentele rămase

```
void errprintf(const char *format, ...)
{
  va_list ap;
  va_start(ap, format);
  fprintf(stderr, "Error: %s; ", strerror(errno));
  vfprintf(stderr, format, ap);
  va_end(ap);
}
```