Linguagens e Ambientes de Programação (2018/2019)

Teórica 14 (29/abr/2019)

Pré-processador.
Polimorfismo implementado usando mecanismos de baixo nível.
Módulos genéricos (polimórficos) em C.

Pré-processador

Programa que corre sempre antes do compilador de C e que faz substituição de macros, inclusão de ficheiros, e possibilita compilação condicional.

Em Linux, o pré-processador chama-se `cpp` e pode ser invocado diretamente pelo utilizador. Mas é mais prático deixar que seja o compilador de C a invocar o pré-processador automaticamente. Em todo o caso, vejamos o que diz o início do manual do comando `cpp`:

```
$ man cpp

CPP(1)                                      GNU                                      CPP(1)

NAME
       cpp - The C Preprocessor

SYNOPSIS
       cpp [-Dmacro[=defn]...] [-Umacro]
           [-Idir...] [-iquotedir...]
           [-Wwarn...]
           [-M|-MM] [-MG] [-MF filename]
           [-MP] [-MQ target...]
           [-MT target...]
           [-P] [-fno-working-directory]
           [-x language] [-std=standard]
           infile outfile

       Only the most useful options are listed here; see below for the remainder.

DESCRIPTION
       The C preprocessor, often known as cpp, is a macro processor that is used automatically by
       the C compiler to transform your program before compilation.  It is called a macro proces-
       sor because it allows you to define macros, which are brief abbreviations for longer con-
       structs.

       The C preprocessor is intended to be used only with C, C++, and Objective-C source code.
```

Exemplos de macros:

```
#define max(a,b)     ((a) > (b) ? (a) : (b))
#define new(type)    ((type *)malloc(sizeof(type)))
#define not          !
#define MAX_STACK    1000

#undef max
#undef new
```

Exemplos de inclusão de ficheiros

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include "MyQueue.h"
```

Exemplos de compilação condicional

```
#define DEBUGLEVEL   1

#ifdef DEBUG
    ...
#  if DEBUGLEVEL > 2
    ...
#  else
    ...
#  endif
    ...
#else
    ...
#endif


#ifndef DEBUG
    ...
#else
    ...
#endif
```

No caso das macros com argumentos, note que existe a necessidade de proteger todas as utilizações dos argumentos com parêntesis, para compensar o facto da expansão das macros ser textual. Por exemplo, considere a seguinte chamada `mult(1+1,5)` no caso da macro abaixo não seguir esta regra...

```
#define mult(a,b)     ((a) * (b))
```

Polimorfismo implementado usando mecanismos de baixo nível

Função monomórfica

A seguinte função permite trocar os valores de duas variáveis inteiras. É uma função monomórfica pois os seus argumentos são de tipo fixo.

```
void swap(int *a, int *b)
{
    int aux = *a ;
    *a = *b ;
    *b = aux ;
}
```

Esta função pode ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(&x, &y) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A função `swap` é segura pois o compilador valida todas as suas utilizações.

Polimorfismo implementado usando manipulação direta de memória

A seguinte função permite trocar os valores de duas variáveis de qualquer tipo. É uma função polimórfica pois os seus argumentos podem ser aplicados a argumentos de tipos diversos.

```
#include <string.h>

void swap(void *a, void *b, int n)
{
    char aux[n] ;     /* array criado com tamanho variável - novidade do C99 */
    memcpy(aux, a, n) ;
    memcpy(a, b, n) ;
    memcpy(b, aux, n) ;
}
```

Esta função ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(&x, &y, sizeof(int)) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A função `swap` não é segura pois o compilador não tem a possibilidade de validar as chamadas.

Polimorfismo implementado usando macros

A seguinte macro também permite trocar os valores de duas variáveis de qualquer tipo.

```
#define swap(a,b,T)         \
    do {                    \
        T __aux = (a)  ;    \
        (a) = (b) ;         \
        (b) = __aux ;       \
    } while( 0 )
```

Esta macro ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(x, y, int) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A macro `swap` é segura pois o compilador consegue detetar qualquer erro de tipo na sua utilização. Repare que o tipo dos valores a trocar é passado como parâmetro.

Repare nos cuidados que é preciso ter para escrever uma macro que não dê problemas. Os argumentos devem ser envolvidos em parêntesis, e qualquer nova variável que seja introduzido deve ter um nome que não entre em conflito com possíveis nomes existentes. Porque todos estes cuidados? E porque razão a macro foi definida usando um `do-while`? (Elimine o `do-while`, teste, e logo descobrirá o problema.)

Polimorfismo implementado usando manipulação direta de memória e macros

A seguinte implementação da operação `swap` é a mais agradável de usar do ponto de vista sintático. Até parece que está em causa uma operação primitiva da linguagem. Contido este implementação não é segura, pelo que o utilizador tem de ter algumas cautelas.

```
#include <string.h>

void __swap(void *a, void *b, int n)
{
    char aux[n] ;
    memcpy(aux, a, n) ;
    memcpy(a, b, n) ;
    memcpy(b, aux, n) ;
}

#define swap(a,b)    __swap(&(a), &(b), sizeof(a))
```

Esta função ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(x, y) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A macro `swap` não é segura pois o compilador não tem a possibilidade de validar as chamadas.

Em GCC

O pré-processador do GCC possui uma construção `typeof` que permite escrever macros ainda mais sofisticadas do que as anteriores e seguras. Por exemplo, a expressão `typeof(a)` representa o tipo da variável `a` e pode ser usada como um tipo normal, inclusivamente para definir outras variáveis com o mesmo tipo de `a`.

É pena o C padrão não suportar a construção `typeof`.

Exemplo de polimorfismo na biblioteca padrão do C - função qsort

A função **qsort** permite ordenar vetores de qualquer tipo.

```
$ man qsort

QSORT(3)                                Linux Programmer's Manual                                QSORT(3)

NAME
       qsort - sorts an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));

DESCRIPTION
       The qsort() function sorts an array with nmemb elements of size size.  The base argument points to the
       start of the array.

       The contents of the array are sorted in ascending order according to a comparison function pointed  to
       by compar, which is called with two arguments that point to the objects being compared.

       The  comparison function must return an integer less than, equal to, or greater than zero if the first
       argument is considered to be respectively less than, equal to, or greater than  the  second.   If  two
       members compare as equal, their order in the sorted array is undefined.

RETURN VALUE
       The qsort() function returns no value.

CONFORMING TO
       SVr4, 4.3BSD, C89, C99.
```

Exercício: Como faria para ordenar um vetor com componentes do seguinte tipo:

```
typedef struct
{
    int day, month, year ;
} Date;
```

Módulos genéricos (polimórficos) em C

É possível escrever módulos genéricos em C usando macros com várias linhas e a operação de concatenação de tokens que se escreve ##.

Os módulos genéricos são muito difíceis de escrever e de ler, mas são fáceis de usar.

No seguinte módulo, o tipo `Data` passa a ser argumento de duas macros.

Ficheiro Generic_LinkedList.h

```
#include <stdbool.h>

#define Generic_LinkedListHeader(Data)                          \
                                                                \
typedef struct Data##Node {                                     \
    Data data ;                                                 \
    struct Data##Node *next;                                    \
} Data##Node, *Data##List ;                                     \
                                                                \
int Data##ListSize(Data##List l) ;                              \
bool Data##ListGetByIndex(Data##List l, int idx, Data *res) ;   \
Data##List Data##ListPutAtHead(Data##List l, Data val) ;        \
```

```
Data##List Data##ListPutAtEnd(Data##List l, Data val) ;         \
void Data##ListPrint(Datasexto##List l) ;
```

Ficheiro Generic_LinkedList.c

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

#include "Generic_LinkedList.h"

#define Generic_LinkedListImplementation(Data, Print)           \
                                                                \
Generic_LinkedListHeader(Data)                                  \
                                                                \
static Data##List Data##NewNode(Data val, Data##List next) {    \
    Data##List n = malloc(sizeof(Data##Node)) ;                 \
    if( n == NULL )                                             \
        return NULL ;                                           \
    n->data = val ;                                             \
    n->next = next ;                                            \
    return n ;                                                  \
}                                                               \
                                                                \
int Data##ListSize(Data##List l) {                              \
    int count ;                                                 \
    for( count = 0 ; l != NULL ; l = l->next, count++ ) ;       \
    return count ;                                              \
}                                                               \
                                                                \
bool Data##ListGetByIndex(Data##List l, int idx, Data *res) {   \
    int i ;                                                     \
    for( i = 0 ; i < idx && l != NULL ; i++, l = l->next ) ;    \
    if( l == NULL )                                             \
        return false ;                                          \
    else {                                                      \
        *res = l->data ;                                        \
        return true ;                                           \
    }                                                           \
}                                                               \
                                                                \
Data##List Data##ListPutAtHead(Data##List l, Data val) {        \
    return Data##NewNode(val, l) ;                              \
}                                                               \
                                                                \
Data##List Data##ListPutAtEnd(Data##List l, Data val) {         \
    Data##List n = Data##NewNode(val, NULL) ;                   \
    if( n == NULL )                                             \
        return NULL ;                                           \
    if( l == NULL )                                             \
        return n ;                                              \
    else {                                                      \
        Data##List p ;                                          \
        for( p = l ; p->next != NULL ; p = p->next ) ;          \
        p->next = n ;                                           \
        return l ;                                              \
    }                                                           \
}                                                               \
                                                                \
void Data##ListPrint(Data##List l) {                            \
    for(  ; l != NULL ; l = l->next )                           \
        Print(l->data) ;                                        \
}
```

Exemplo de instanciação do módulo genérico

Ficheiro LinkedList_double.h

```
#include "Generic_LinkedList.h"

Generic_LinkedListHeader(double)
```

Ficheiro LinkedList_double.c

```
```

```
#include "Generic_LinkedList.c"

static doublePrint(double d) {
    printf("%lf\n", d) ;
}

Generic_LinkedListImplementation(double, doublePrint)
```

#80