Programação em Sistemas Computacionais

Turma LI31N – Aula de 2009-10-30

Na aula anterior (2009-10-27) desenhámos uma função genérica para encontrar o menor elemento em qualquer *array* cujos elementos tenham exactamente 4 *bytes*.

[*ver ficheiros em anexo:* findmin4.s *e* findmin4_test.c]

O elemento-chave no desenho desta solução genérica é a utilização de ponteiros para função, neste caso para indicar a função de comparação a utilizar com cada *array*. Quando se invoca *findmin4* sobre um *array* de inteiros indica-se a função *icmp* como função de comparação; para um *array* de *floats* usa-se *fcmp*; e para *strings* pode-se utilizar directamente *strcmp* (da biblioteca *standard*). Dentro da função *findmin4*, basta utilizar como argumento da instrução *call* o ponteiro para função passado como terceiro argumento de *findmin4*, em vez de invocar uma função de comparação fixa.

Tendo em conta a solução desenhada, salientam-se os seguintes defeitos[1]:

- A função só é aplicável a *arrays* cujos elementos tenham 4 *bytes*.
- Existe dificuldade em lidar com o tipo de retorno de *findmin4*, uma vez que não existe forma genérica de designar um tipo cujo tamanho seja de 4 *bytes*.

Para resolver o primeiro problema seria necessário passar um quarto argumento (*dim*) à função *findmin*, para que nesta se avançasse de *dim* em *dim bytes* sobre o *array*, em vez dos saltos de 4 que estamos a usar. Para além disso, passaria a ser possível passar argumentos de dimensão *dim* à função de comparação, bem como tratar adequadamente o valor de retorno de *findmin*. Infelizmente, basta tentar escrever o código de *findmin* de acordo com esta ideia para se perceber como este último ponto leva a uma solução demasiado complexa. E continuamos a não conseguir resolver o segundo problema indicado atrás.

Ambos os problemas podem ser facilmente resolvidos se, em vez de lidarmos directamente com os valores presentes no *array*, utilizarmos apenas ponteiros para esses elementos. Por um lado, todos os ponteiros têm a mesma representação física (32 *bits*, na arquitectura que estamos a considerar), como existe mesmo um tipo próprio para representar ponteiros genéricos em C, o tipo *void **.

Com base nesta ideia, redesenha-se a solução para que:

- A função de comparação receba, como argumentos, ponteiros para os elementos a comparar e não directamente os elementos.
- A função *findmin* retorne um ponteiro para o menor valor encontrado, em vez de retornar directamente o próprio valor.

[1] Para além dos defeitos indicados, também não existe justificação para se utilizar um algoritmo recursivo, uma vez que consome mais recursos em tempo de execução, não apresentando ganhos reais de legibilidade do código fonte que o compensem. Sugere-se, por isso, que redesenhem o algoritmo para a forma iterativa.

Naturalmente, a função *findmin* passa a receber um quarto argumento indicando a dimensão em *bytes* de cada elemento presente no *array*.

Nova assinatura de *findmin*:

```
void * findmin(void * data, unsigned int dim, unsigned int num,
               int (*cmp)(void * pval1, void * pval2));
```

ou, recorrendo a *typedef* para abreviar nomes de tipos:

```
typedef unsigned int uint;

typedef int (*cmp_f)(void * pval1, void * pval2);

void * findmin(void * data, uint dim, uint num, cmp_f cmp);
```

Nos ficheiros anexos *findmin.s* e *findmin_test.s* pode ver-se a nova versão do código, agora utilizável sobre *arrays* de qualquer tipo. Consegue ver as duas únicas alterações realizadas para transformar *findmin4* em *findmin*?

Para utilizar esta nova versão de *findmin*, é agora necessário converter os ponteiros genéricos para ponteiros concretos sempre que é preciso aceder ao valor apontado. Veja-se, em particular, as implementações das funções de comparação e o tratamento dado aos retornos de *findmin* em *main*.

Vejamos um segundo exemplo de função genérica, desta vez implementado exclusivamente em C. Pretende-se uma função genérica para percorrer *array*, realizando uma determinada acção sobre cada elemento. A assinatura desta função será a seguinte:

```
void foreach(void * data, uint dim, uint num, void (*action)(void * elem));
```

Note-se como o triplo *(data, dim, num)* caracteriza completamente o *array* a percorrer, já que nos dá o endereço do primeiro elemento, a dimensão de cada elemento em *bytes* e o número de elementos. A função indicada por *action* realiza uma acção sobre um elemento e será invocada para cada um dos elementos do *array*.

Para implementar esta função em C é necessário ter em conta que a única operação realizável sobre um ponteiro genérico é copiá-lo (para outra variável ou como argumento ou retorno de uma função). Em particular, não tem significado a adição de um valor numérico *n* a um ponteiro genérico em C, com o intuito de o fazer referir outro endereço a *n bytes* de distância, uma vez que a adição de um valor inteiro *n* a um ponteiro, em C, produz um ponteiro para *n* elementos adiante e não para *n bytes* depois. Isto é, dado um ponteiro *p* para uma instância de T (em que T é um tipo qualquer, excepto *void*), a expressão, em C, p + n produz um ponteiro para o endereço final p + n * sizeof(T). Assim, para avançar sobre o *array*, vamos converter o ponteiro *data* para *char **, conseguindo dar significado a *p + dim*, com a vantagem de, devido a *sizeof (char) == 1*, a expressão *p + dim* produzir directamente o endereço desejado.

Esta versão está implementada em *foreach.v1.c*.

Numa outra versão, talvez mais útil, de *foreach*, utiliza-se um argumento adicional que é recebido por *foreach* e passado (sem qualquer alteração) à primeira invocação da acção *action*. Por seu turno, a função *action* retornará um novo valor que será passado à invocação seguinte de *action*, e por aí adiante, até que foreach retornará o valor retornado pela última invocação de *action*. Estes valores não são usados pelo algoritmo genérico *foreach*, mas podem ser utilizados por *action* para manter estado entre as várias invocações que ocorrerão.

```
void * foreach(void * data, uint dim, uint num,
          void * (*action)(void * elem, void * ctx),
          void * ctx) {
   char * p = (char *)data;
   uint i;
   for (i = 0; i != num; ++i) {
      ctx = (*action)(p, ctx);
      p += dim;
   }
   return ctx;
}
```

Esta forma de *foreach* é suficientemente genérica para ser possível realizar uma nova versão de *findmin*. Veja o código de *foreach.v2.c*.

Exercícios:

1. Utilizando a última versão de *foreach*, escreva as funções *action* que permitem acumular todos os elementos do *array*. No caso dos inteiros e dos *float* pretende-se calcular o somatório de todos os valores e no caso das *strings*, concatená-las (use *strcat*, da biblioteca *standard*, se necessário).

   Sugestão: invoque *foreach* com *ctx = 0* no caso dos inteiros, *ctx = 0.0* no caso dos *floats* e *ctx = all_strs* no caso das *strings*, com *char all_strs[64] = “”;*

2. Utilizando a última versão de *foreach*, escreva as funções *action* que permitem descobrir, simultaneamente, o maior e o menor valor presentes num *array*.
   Sugestão: defina o tipo auxiliar struct MaxMin { void * pMax; void * pMin }, crie uma instância de *struct MaxMin* em *main* e passe o seu endereço como argumento ctx de *foreach*.