Matheus Laidler Programação C - UFRJ

Daniel Bastos

# UNIONS: “A união faz a força!”

Muito se utiliza das famosas estruturas para declarações de várias variáveis que serão importantes para o nosso código. Porém, quando se pensa no ponto de vista prático da máquina e sua otimização, esta opção deixa de ser tão valiosa e passamos a preferir uma alternativa mais viável para o desenvolvimento do código. E se eu te falasse que podemos unir essas variáveis e alocarmos todas elas no mesmo espaço da memória? Isso é possível, embora já devemos imaginar que deverão ser utilizadas uma por vez, por se tratar da mesma região da memória.

As ‘unions’ funcionam praticamente como uma ‘struct’, entretanto ela é mais otimizada no sentido de ocupação da memória, que é algo importante a se levar em consideração na vida de um programador. Se você não vai precisar utilizar as variáveis declaradas ao mesmo tempo, utilizar as ‘unions’ passa a ser um “dever moral” do programador em muitos casos.

Digo isso pois, imagina que você faz um programa grande e não utiliza das ‘unions’ para sua produção. Você vai fazer seu programa alocar tudo na memória, muitas vezes deixando ela presa a ‘lixos’, isto é, informações que não serão mais utilizadas, e acaba, ainda sim, utilizando mais espaço para suas novas/outras funções. Se você tem aquele espaço do ‘lixo’ que não será mais utilizado, porque alocar mais espaço na memória? Pois é.

. *Escrevi um código C para entender melhor na prática como tudo funciona.*

. *Veremos então agora um exemplo de UNIONS:*

## Entendendo Union na prática [lab1]

### Código:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
union labB{
        short int x;
        unsigned char z;
};
int main(){
        union labB m;
        m.x = 5;
        m.z = 'a';
        printf("\tx = %d\n",m.x);
        printf("\tz = %c\n",m.z);
        printf("\t----------\n\n");
        m.x = 8;
        printf("\tx = %d\n",m.x);
        m.z = 'z';
        printf("\tz = %c\n",m.z);
        printf("\t----------\n\n");
        sleep(5);
        return 0;
};
```

O código quase inteiro foi colocado acima. Nele vemos os testes do que é printado pela ordem que escrevemos na memória.

**Conclusões dos testes práticos:**

```
$>  gcc firstunion.c -o union1
$> ./union1
      x = 97
      z = a
      ----------
      x = 8
      z = z
      ----------
```

Vemos pela saída dos primeiros resultados que o resultado foi o mesmo. Ou seja, entregaram a mesma resposta só que com os tipos diferentes. O valor de ‘x’ precisaria ser um ‘int’, portanto o resultado foi o valor de ‘a’ em decimal, que podemos confirmar pela tabela ASCII.
Já na segunda parte a saída foi correta, já que printamos antes de utilizar a memória para atribuir o valor de ‘z’. Nenhuma chamada do compilador ocorreu nesta execução do primeiro exemplo.

Concluímos, assim, que precisamos de apenas um espaço na memória para a utilização dos dois, porém necessariamente precisam ser usados de forma separada. Pois, se estamos escrevendo na memória o valor de ‘x’ e logo depois escrevemos o valor de ‘z’, tal qual no primeiro exemplo do código, então sobrescrevemos o que foi passado para ‘x’ e colocamos o que foi passado para ‘z’, assim eles acabam sendo mostrados como o mesmo valor. Ao mesmo tempo que a união faz a força e pode deixar nosso programa mais otimizado, se não foi bem utilizado por falta de entendimento, o resultado pode ser um programa cheio de ‘bugs’ e problemas que, se não for reconhecido logo, poderá ser de difícil identificação do problema no futuro, pois o compilador não te dará nenhuma mensagem de aviso ao compilar (na maior parte dos casos).

Podemos, ainda, fazermos uma adição ao código para que ele nos mostre o valor total de ‘union’ e ‘struct’ para observarmos a diferença entre eles.

Podemos também colocar para mostrar o tamanho de cada declaração ‘union’ para vermos que o valor total será o mesmo daquele que tiver maior espaço a ser alocado, já que será neste tamanho de espaço em que ambos conseguirão trabalhar. Vamos então utilizar o sizeof() para isso.

*union labB m;struct labA n;int a = **sizeof(m)**; int b = **sizeof(n)**;*

printf("\n\tUnion: **%d** bytes \n\tStruct: **%d** bytes\n\n",***a,b***);

Para usarmos de modo que mostre também de cada declaração dentro de ‘union’:

*int a = **sizeof(m);**int a1 = **sizeof(m.x)**;int a2 = **sizeof(m.z)**;*

printf("\n\n\tBytes de Union:**%d**\t \n.Sendo eles: **%d** de (x) e **%d** de (z) \n\t\n\n",***a,a1,a2***);

**Saida1:**

*Union>2bytes*

*Struct>4bytes*

**Saida2:**

*Bytes de Union:2*

*.Sendo eles: 2 de (x) e 1 de (z)*

*(Podemos tbm fazer para ver os bytes de x e z da estruct, o que eu fiz, que ficou> Bytes de Struct:4 .Sendo 2 de (x) e 1 de (z);)*

*Assim concluímos sobre a importância do uso das uniões em relação a otimização do uso da memória em contraste com uma utilização desenfreada da struct, que muitas vezes é usada em momentos os quais a ‘unions’ seria mais adequada para o programa e ao usuário final.*

### *Definição e/da Declaração*

Vamos falar da **declaração** e **definição** na prática vendo pelo código do programa abaixo:

---

```
#include <stdio.h>
union value {
  int ival;
  char *sval;
};
struct record {
  char *name; int type;
  union value v;
};
#define INT 0
#define STR 1
int main() {
  struct record r; r.name = "Love Story"; r.type = INT; r.v.ival = 1970;
  printf("name = %s, year = %d, size = %ld\n", r.name, r.v.ival, sizeof r.v);
  r.name = "Dr. Jhivago"; r.type = STR; r.v.sval = "Omar Sharif";
  printf("name = %s, star = %s, size = %ld\n", r.name, r.v.sval, sizeof r.v);
  return 0;
}
```

---

A ***/union value/*** é uma declaração, definição ou ambos? Pois bem, union value no início do programa é apenas uma declaração, isto é, o programa apenas avisa o que será usado, mas não tem nenhum espaço reservado para armazenamento na memória neste momento. Já quando definimos a coisa muda de figura, por exemplo:

union value { .. } w;  Obs: '*union value { ... } w*;' é a mesma coisa se fizermos '*union value w;'*

Ao declararmos '*union value'* e **definirmos** como *'w'*, agora um **espaço** na memória é alocado. Se fizermos '*union value m*', como fiz no meu programa anterior, só depois disso que é **definido** um espaço na memória para alocar a variável. Só foi possível pois **declaramos** anteriormente. Podemos, então, concluir que a declaração apenas nos diz quem é o *type*, já quando nos dá uma variável que podemos usar, isso significa que foi *definido*. Em outras palavras, quando você coloca *'int a'* no seu *'main', por exemplo,* 4 bytes já foram alocados, portanto houve uma *declaração com definição*. Porém, se fizéssemos '*extern int a;'* seria diferente, pois apenas estou declarando um tipo sem precisar alocar memória para a variável, tendo em vista que a memória já foi alocada externamente. Quando apenas declaramos uma estrutura ou união, nenhum espaço ainda foi armazenado na memória. Entretanto, temos que prestar atenção se a **definição** não está **dentro de uma declaração**, pois aí a coisa muda de figura e tudo **continua sendo apenas uma declaração**, como se só apenas declarasse que a definição irá acontecer. Podemos observar este fenômeno na ***struct record*** do código acima. Ela é inteira apenas uma **declaração**, portanto a ***union value v*** dentro dela **também** é apenas uma **declaração**, porque ainda não teve nenhum espaço alocado na memória ainda, logo para ela também não terá.

(É interessante comentar que em meus testes observei que se existir um *static int* em *union* ou em um *struct* o código não é compilado).

Com tudo isso em mente, elas aparentemente se comportam como as próprias estruturas quando passadas a procedimentos, tendo em vista que ela já tem o espaço definido para tudo dela em um mesmo ambiente tendo o tamanho de seu maior elemento, diferente de um array que é armazenado em um local específico da memória dos quais os membros, então, estão alocados em locais diferentes da memória. Com as unions não podemos usar todos os seus elementos, diferentemente de um array.Porém também faz sentido se elas apenas decaem para ponteiro.

*-Para ser bem sincero, não sei se essa minha interpretação está correta.-*

Pois então, será que é possível criar *array* com *union*? É melhor usar um *struct*, mas.. veremos… Fiz um teste com o código abaixo:

```
#include <stdio.h>
#include <string.h>
union value {
   int credito; char lista[5];
 } u;
void main () {
 u.credito = 5; // inicializando credito
 printf ("O valor de credito é: %d\n", u.credito);
 strcpy (u.lista, "abelha");
 printf ("O valor do elemento 0 da lista é: '%c' -> %d\n", u.lista[0],u.lista[0]);
 printf("\t Elementos da lista:
[%c,%c,%c,%c,%c,%c]\n",u.lista[0],u.lista[1],u.lista[2],u.lista[3],u.lista[4],u.lista[5]);
 printf("\ncomo ta o credito agr?  %c | %d\n",u.credito,u.credito);
}
```

saída na próxima página:Claro, tive que respeitar a regra de union e deu tudo certo. Veremos;

*$ ./union3*

*O valor de credito é: 5*

*O valor do elemento 0 da lista é: 'a' -> 97*

*Elementos da lista: [a,b,e,l,h,a]*

*como ta o credito agr? a | 1818583649*

**Union com Array;**

Para finalizar meus testes com o estudo a respeito de ‘union’ fui tentar testar a criação de arrays. A ideia por trás do meu código era testar unions e depois implementar array nele. Com isso declarei um inteiro de teste e a array que queremos testar. Coloquei um espaço adequado para escrever algo aleatório de teste (abelha). Adicionei um valor ao inteiro para testar no início e no final do programa para vermos sua mudança. Após printar ele na tela usei strcpy() para adicionarmos a nossa string ‘abelha’ na nossa lista. Com isso o valor do inteiro antigo deve ser sobrescrito. Daí vem os prints do resultado que queremos ver. Inclusive em um deles coloquei para printar cada elemento da lista e em outro coloquei o novo valor do inteiro inicial.
Foi possível criar e usar o array com Union, apesar de não ser a melhor opção na maioria dos casos, ele pode ser útil em determinadas situações.

**(Sobre o que escrevi da penúltima tarefa, a respeito de ser mais parecido com struct ou com arrays, isso me pega um pouco pq me parece uma opção viável ela decair para ponteiros como arrays, já que ela reserva um espaço na memória pra aquilo tudo e pode simplesmente apontar ao invés de copiar pro parâmetro. Eu realmente fiquei com essa pulga atrás da orelha… poderia me mandar em um comentário na atividade algo para clarear a mente? não encontrei algo q me ajude nisso. abração professor)**