Universidade Federal do Espírito Santo

Departamento de Computação e Eletrônica - CEUNES
Estrutura de Dados I
Prof. Oberlan Romão

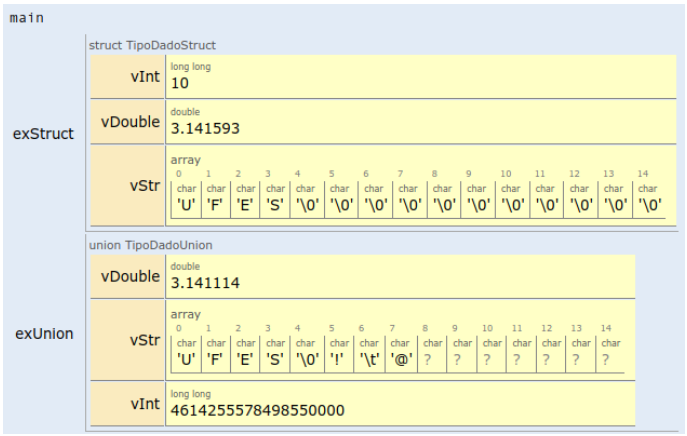
Listas Genéricas

Atenção: Ao terminar, não se esqueça de enviar as soluções no AVA.

Como vimos nas aulas teóricas, uma lista genérica pode ser vista como uma lista que armazena diferentes tipos de valores em cada nó. Por exemplo, em um nó podemo ter um `int` em outro um `double` e em outro um `string`. Basicamente, existem duas utilizadas para construir uma lista genérica: usando `union` ou usando ponteiro genérico (mais comum). Se não estiver familiarizado com esse tópico você pode ser o texto sobre Listas Genéricas.

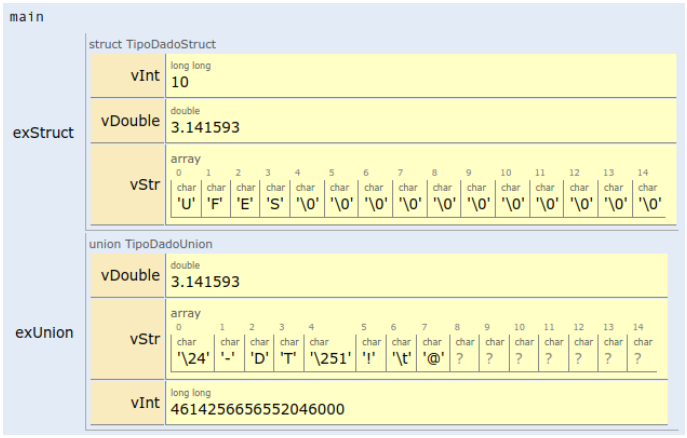
Antes de praticarmos listas genéricas vamos entender um pouco melhor como funciona um `union` e um ponteiro genérico (`void *`). Além disso, também vamos aprender/revisar alguns detalhes de como `string` (array de `char`) funciona em C. Baixe o arquivo `Codigos.zip` no AVA e extraia em alguma pasta.
Exercício 1: Abra e analise o arquivo `ex1.c`. Copie o seu conteúdo para o site C Tutor. Clique no botão "`Visualize Execution`" e, em seguida, clique algumas vezes em "`Next >`" (se precisar, volte os passos da execução clicando em "`< Prev`").

Ao executar a linha 25 (`strcpy(exUnion.vStr, "UFES");`) você obterá uma imagem parecida com a seguinte:

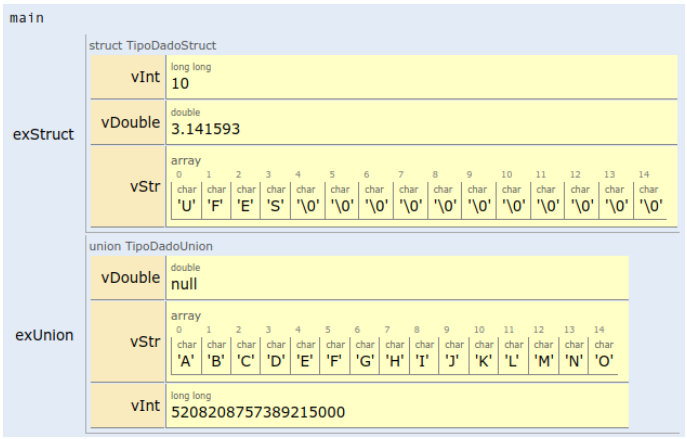


Note que os campos do `struct` permanecem com todos os dados coerentes. Por outro lado, no `union`, apenas o campo `vStr` contém uma informação coerente com as atribuições feitas a variável `exUnion`.
Pergunta: Observe os `'\0'` no campo `vStr` da variável `exStruct` e `exUnion`. Você consegue dizer qual o significado/utilização deles (em especial do primeiro `'\0'`)? Para saber mais sobre `string` em C, leia esse documento: String in C.

Volte a execução (clicando em "`< Prev`") para que setinha verde (➡) fique na linha 24 (`exUnion.vDouble =PI;`). Você obterá uma imagem parecida com a seguinte:



Note que agora, apenas o campo `exUnion.vDouble` possui o valor correto. Ao executar a linha 25 (`strcpy (exUnion.vStr, "UFES");`) note o `'\0'` após o caractere `S` na variável `exUnion.vStr`. Mais uma pista do significado do `'\0'` em um string. Continue a execução do código (clicando no botão "`Next >`"). Ao fim, você, provavelmente, obterá algo assim:



Pergunta: Após tudo que você aprendeu, o código `strcpy(exUnion.vStr, "ABCDEFGHIJKLMNO");` (linha 28) possui algum problema? Qual?
Quando estamos trabalhando com `string` em C, esse é um dos erros mais comuns e mais difíceis de se encontrar. Tente usar o Valgrind para ver se ele acusa algum erro. Provavelmente ele terminará com 0 erros. Um site muito útil para análise de códigos, inclusive com a geração do código Assembly de diferentes compiladores, é o Compiler Explorer. Nesse link `https://gcc.godbolt.org/z/4zKPx5Ena` você terá acesso ao código do `ex1.c` juntamente com a análise o PVS-Studio (uma ferramenta para detectar bugs em códigos). Note que na aba associada a análise do PVS-Studio, é apresentado o erro: `<source>:28:1: error: V512 A call of the 'strcpy' function will lead to overflow of the buffer 'exUnion.vStr'`. O que esse erro está indicando?

Clique em "`Edit this code`" e troque `exUnion.vInt = 10;` (linha 22) por `exUnion.vInt = 65;`. Execute o código até a linha 22 ou 23. O que aconteceu com o campo `vStr` de `exUnion`? Teste outros valores (por exemplo, 66, 67, 68...). O que esses valores têm a ver com um `char`? Dê uma olhada na Tabela ASCII. Qual caractere é representado pelo decimal 65?

Exercício 2: Abra e analise o arquivo `ex2.c`. Compile-o sem as flags que usamos na disciplina, ou seja, compile apenas com: `gcc ex2.c -o ex2`. Provavelmente, serão apresentados alguns avisos (*warnings*). Em seguida, execute o programa: `./ex2`. Note que resultado não saiu como "esperado" para os dois primeiros `printf`. Isso acontece porque um ponteiro de um determinado tipo só deve receber endereços de variáveis desse mesmo tipo. Assim, ao fazermos, por exemplo `int *pInt = &y;` estamos violando essa restrição, já que y é do tipo `double`, e isso certamente irá causar problemas. Existe uma exceção para essa regra: usar um ponteiro genérico, ou seja, `void *`. Entretanto, para obtermos o valor ao qual o ponteiro está apontando, devemos fazer um *cast* (conversão de tipo) para o ponteiro adequado, ou seja, se ele recebeu um endereço de uma variável do tipo `int`, devemos fazer a conversão para `int *` sempre que precisarmos usar o valor apontado pelo ponteiro genérico. Por exemplo:

```
1 int x = 10;
2 double pi = 3.1415;
3 void *ptr = &x;
4 printf("%d\n", *(int *)ptr); //Irá imprimir o valor apontado por ptr
5 ptr = &pi;
6 printf("%d\n", *(double *)ptr);
```

Por que devemos usar essa sintaxe na conversão, ou seja, `*(int *)` ou `*(double *)`? R. Quando trabalhamos com ponteiro `void` devemos sempre convertê-lo para o ponteiro adequado. Como queremos pegar o valor que esse ponteiro aponta, usamos o operador `*`. Assim, chegamos ao `*(int *)ptr` da linha 4 e `*(double *)ptr` da linha 6. O mesmo deve ser feito se precisarmos mudar o valor apontado por um ponteiro `void *`:

```
1 int x = 10;
2 void *ptr = &x;
3 printf("%d\n", *(int *)ptr); //10
4 *(int *)ptr = 15;
5 printf("%d\n", *(int *)ptr); //15
```

Exercício 3: Como você pode imaginar, um ponteiro genérico também pode receber um endereço de um `struct`. Veja um exemplo:

```
1  typedef struct {
2      double x, y, z;
3  }Ponto;
4
5  int main() {
6     Ponto p = {1.5, 2.7, 0.5};
7     void *ptr1 = &p;
8     void *ptr2 = malloc(sizeof(Ponto));
9     //...
10 }
```

Abra, analise, compile e execute o arquivo `ex3.c`. Note que para acessarmos um campo de um `struct` que é apontado por um ponteiro `void` não podemos usar o operador `->`. Para resolvermos esse problema, devemos fazer um *cast* antes (atente-se a sintaxe):

```c
void *ptr = malloc(sizeof(Ponto));
//ptr->x = 10; //Erro
((Ponto *)ptr)->x = 10;
//ou
(*(Ponto *)ptr).x = 10;
```

Uma estratégia mais simples e legível é criar uma variável auxiliar do tipo ponteiro para `Ponto`:

```c
void *ptr = malloc(sizeof(Ponto));
Ponto *ptrPonto = (Ponto *)ptr; //Uma boa prática é fazer o cast para Ponto * do
                                    ponteiro void
ptrPonto->x = 5.5; //ou (*ptrPonto).x = 5.5
ptrPonto->y = 8.7;
ptrPonto->z = 3.4;
```

Exercício 4: Agora que já entendemos melhor sobre como trabalhar com `union` e `void *`, vamos implementar uma lista genérica que pode armazenar em cada nó um `int`, ou um `double` ou um `string`. Analise os códigos da pasta `ListaGenerica/Union`.

- Note que nessa lista, um nó é chamado de `Objeto` (`Objetos.h` e `Objetos.c`). Além disso, diferente do código visto em aula, a função de inserção de objetos na lista (`insereLista`) deve receber um ponteiro de um objeto já alocado. Por isso, é conveniente e prático termos funções responsáveis pela alocação desses objetos (uma para cada tipo);
- Analise a implementação da função de alta ordem `imprimeLista` (`Lista.h` e `Lista.c`). Em seguida, observe como podemos chamá-la (arquivo `main.c`). Analise também a implementação das funções `imprimeInformacaoObjeto` e `imprimeItemObjeto` (`Objetos.h` e `Objetos.c`);
- Baseado nas funções `criaObjetoInteiro` e `criaObjetoReal`, complete a `criaObjetoString`. Lembre-se dos detalhes que você aprendeu/revisou sobre `string` no Exercício 1;
- Complete as funções `liberaObjeto` (`Objetos.c`) e `liberaLista` (`Lista.c`);
- Execute o código com "`make run`" e teste possíveis vazamentos de memória (`make memcheck`). Se tiver algum erro, corrija-os :)

Exercício 5: Podemos usar essa lista genérica para armazenar expressões matemáticas simples, por exemplo, $10 + 15$ ou -10 ou $-3 + 5$ ou $-3.5 - 1.3$. Considerando que os objetos do tipo `STR` sempre conterão `"+"` ou `"-"`, implemente a função `analisaExpressao` que retorna `true` se a expressão matemática está correta e `false` caso contrário. Exemplos de expressões incorretas:

- $10 + 15-$ (tem um $-$ sobrando);
- $+$ (operador sem operandos);
- $10 + +10$ (considere essa expressão como errada, mesmo ela fazendo sentido matematicamente);
- $10\ 15$ (expressão sem operador)

Pergunta: Você consegue pensar em quais seriam os passos de uma função que recebe uma expressão correta e retorna o seu resultado? Por exemplo, $10 + 15$ deve retornar 25.
Desafio: Implemente essa função :)
Exercício 6: Por fim, vamos analisar essa mesma lista genérica, mas agora usando ponteiro `void`. Analise os códigos da pasta `ListaGenerica/PonteiroGenerico`.

- Observe que agora não precisamos mais do `union Item` (`Objetos.h`). Agora no `struct objeto` o campo `item` é do tipo `void *` (ponteiro genérico). Com essa mudança, precisamos fazer alguns ajustes nas funções que acessam o campo `item`;
- Implemente a função `criaObjetoInteiro` (`Objetos.c`). Para isso:
  - faça a alocação de memória para um `int` (usando `malloc`); e
  - atribua ao endereço alocado o valor passado como parâmetro para a função.
- Em `criaObjetoInteiro`, ao invés de fazermos a alocação do espaço, poderíamos fazer:

```
1 obj->item = &valor;
```

  Por quê?
- Implemente as funções `criaObjetoReal` e `criaObjetoString`;
- Complete a função `liberaObjeto`. Lembre-se de fazer a desalocação dos `itens` dos objetos;
- Descomente o código da função `imprimeInformacaoObjeto`. Note que é apresentado um erro ao tentarmos imprimir o conteúdo do `item` do objeto. Faça as correções necessárias (o Exercício 2 pode te ajudar);
- Faça o mesmo na função `imprimeItemObjeto`;
- Execute o código com "`make run`" e teste possíveis vazamentos de memória (`make memcheck`). Se tiver algum erro, corrija-os :)

Bom trabalho e divirta-se!