INF1018 - Software Básico (2019.2)
Segundo Trabalho

Gerador Dinâmico de Funções

O objetivo deste trabalho é implementar em C uma função **`cria_func`**, que recebe o endereço de uma função **`f`** e a descrição de um conjunto de parâmetros. A função `cria_func` deverá gerar, dinamicamente, o código de uma "nova versão" de `f`, e retornar o endereço do código gerado.

Também deve ser implementada uma função **`libera_func`**, que é responsável por liberar a memória alocada para o código criado dinamicamente.

Instruções Gerais

Leia com atenção o enunciado do trabalho e as instruções para a entrega. Em caso de dúvidas, não invente. Pergunte!

- O trabalho deve ser entregue **até 23:59h do dia 25 de novembro**.
- Trabalhos entregues com atraso perderão **um ponto por dia de atraso**.
- Trabalhos que não compilem no ambiente Linux **não serão considerados** (ou seja, receberão grau zero).
- Os trabalhos podem ser feitos em grupos de **no máximo** dois alunos.
- Alguns grupos poderão ser chamados para apresentações orais / demonstrações dos trabalhos entregues.

Amarrando Parâmetros

O propósito de gerarmos dinamicamente uma "nova versão" de uma função **`f`** é podermos "amarrar" valores pré-determinados a um ou mais dos parâmetros de **`f`**. Dessa forma, não precisaremos passar esses valores como argumentos quando chamarmos a nova versão gerada.

Considere, por exemplo, um exemplo trivial: uma função que retorna o produto de seus dois parâmetros:

```
int mult (int x, int y);
```

A função **`cria_func`** nos permite criar dinamicamente uma nova função, baseada em `mult`, que sempre retorna o valor de seu parâmetro multiplicado por 10. Para criar essa nova função, `cria_func` *amarra* o segundo parâmetro de `mult` a um valor fixo (10). Ou seja, `cria_func` constrói, em tempo de execução, o código de uma nova função que *chama* `mult`, passando dois argumentos: o primeiro é argumento recebido por essa nova função, e o segundo é o valor constante 10.

Especificação das funções

A função **`cria_func`** deve ter o protótipo

```
void* cria_func (void* f, DescParam params[], int n);
```

onde **f** tem o endereço da função original a ser chamada pelo código gerado, o array **params** contém a descrição dos parâmetros para chamar essa função e **n** é o número de parâmetros descritos por **params**.

O número mínimo de parâmetros é 1, e o máximo é 3!

O tipo **DescParam** é definido da seguinte forma:

```
typedef enum {INT_PAR, PTR_PAR} TipoValor;
typedef enum {PARAM, FIX, IND} OrigemValor;

typedef struct {
   TipoValor    tipo_val;  /* indica o tipo do parametro (inteiro ou ponteiro) */
   OrigemValor  orig_val;  /* indica a origem do valor do parametro */
   union {
     int v_int;
     void* v_ptr;
   } valor;         /* define o valor ou endereço do valor do parametro (quando amarrado/indireto) */
} DescParam;
```

O campo **orig_val** indica se o parâmetro deve ser "amarrado" ou não; ele pode conter os seguintes valores:

- **PARAM**: o parâmetro não é amarrado, ou seja, deve ser recebido pela nova função e repassado à função original.
- **FIX**: o parâmetro deve ser amarrado a um valor constante, fornecido no campo **valor**.
- **IND**: o parâmetro deve ser amarrado a uma variável, cujo endereço é fornecido no campo **valor**. Isto é, deve ser passado à função original o valor corrente dessa variável.

A função **libera_func** deve ter o protótipo a seguir:

```
void libera_func (void* func);
```

O arquivo **cria_func.h** contém as definições acima, e pode ser obtido AQUI. O trabalho deve seguir **estritamente** as definições constantes nesse arquivo.

Um exemplo de uso

O programa abaixo usa **cria_func** para criar dinamicamente uma nova versão de `mult` que multiplica seu parâmetro por 10, e depois chama essa função para obter as dezenas de 1 a 100:

```
#include <stdio.h>
#include "cria_func.h"

typedef int (*func_ptr) (int x);

int mult(int x, int y) {
  return x * y;
}

int main (void) {
  DescParam params[2];
  func_ptr f_mult;
  int i;

  params[0].tipo_val = INT_PAR; /* o primeiro parâmetro de mult é int */
  params[0].orig_val = PARAM;   /* a nova função repassa seu parâmetro */

  params[1].tipo_val = INT_PAR; /* o segundo parâmetro de mult é int */
  params[1].orig_val = FIX;     /* a nova função passa para mult a constante 10 */
  params[1].valor.v_int = 10;

  f_mult = (func_ptr) cria_func (mult, params, 2);

  for (i = 1; i <=10; i++) {
    printf("%d\n", f_mult(i)); /* a nova função só recebe um argumento */
  }
```

```
  libera_func(f_mult);
  return 0;
}
```

Na variação do programa mostrada abaixo, obtemos as dezenas de 1 a 100 *amarrando* o primeiro parâmetro a uma variável e o segundo parâmetro ao valor constante 10. Neste caso, não passamos nenhum argumento para a função gerada dinamicamente.

Note que devemos passar, na descrição do primeiro parâmetro, o **endereço** da variável à qual o parâmetro está *amarrado*:

```
#include <stdio.h>
#include "cria_func.h"

typedef int (*func_ptr) ();

int mult(int x, int y) {
  return x * y;
}

int main (void) {
  DescParam params[2];
  func_ptr f_mult;
  int i;

  params[0].tipo_val = INT_PAR; /* a nova função passa para mult um valor inteiro */
  params[0].orig_val = IND;     /* que é o valor corrente da variavel i */
  params[0].valor.v_ptr = &i;

  params[1].tipo_val = INT_PAR; /* o segundo argumento passado para mult é a constante 10 */
  params[1].orig_val = FIX;
  params[1].valor.v_int = 10;

  f_mult = (func_ptr) cria_func (mult, params, 2);

  for (i = 1; i <=10; i++) {
    printf("%d\n", f_mult()); /* a nova função não recebe argumentos */
  }

  libera_func(f_mult);
  return 0;
}
```

Outro exemplo de uso

No exemplo a seguir, criamos uma nova versão da função de comparação de bytes **`memcmp`**, da biblioteca padrão de C.

A função `memcmp` recebe duas strings e um número *n*, e compara os *n* primeiros bytes das duas strings, retornando 0 se são iguais.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Em outras palavras, podemos usar `memcmp` para verificar se as duas strings fornecidas possuem um mesmo *prefixo*, de tamanho `n`.

Podemos usar **`cria_func`** para criar uma versão de `memcmp` que testa se uma dada string possui um mesmo prefixo que uma string pré-determinada (ou seja, *amarrada*), Veja este exemplo de uso abaixo:

```
#include <stdio.h>
#include <string.h>
#include "cria_func.h"
```

```
typedef int (*func_ptr) (void* candidata, size_t n);

char fixa[] = "quero saber se a outra string é um prefixo dessa";

int main (void) {
  DescParam params[3];
  func_ptr mesmo_prefixo;
  char s[] = "quero saber tudo";
  int tam;

  params[0].tipo_val = PTR_PAR; /* o primeiro parâmetro de memcmp é um ponteiro para char */
  params[0].orig_val = FIX;     /* a nova função passa para memcmp o endereço da string "fixa" */
  params[0].valor.v_ptr = fixa;

  params[1].tipo_val = PTR_PAR; /* o segundo parâmetro de memcmp é também um ponteiro para char */
  params[1].orig_val = PARAM;   /* a nova função recebe esse ponteiro e repassa para memcmp */

  params[2].tipo_val = INT_PAR; /* o terceiro parâmetro de memcmp é um inteiro */
  params[2].orig_val = PARAM;   /* a nova função recebe esse inteiro e repassa para memcmp */

  mesmo_prefixo = (func_ptr) cria_func (memcmp, params, 3);

  tam = 12;
  printf ("'%s' tem mesmo prefixo-%d de '%s'? %s\n", s, tam, fixa, mesmo_prefixo (s, tam)?"NAO":"SIM");
  tam = strlen(uma);
  printf ("'%s' tem mesmo prefixo-%d de '%s'? %s\n", s, tam, fixa, mesmo_prefixo (s, tam)?"NAO":"SIM");

  libera_func(mesmo_prefixo);
  return 0;
}
```

Implementação

A função **`cria_func`** deve ser implementada em C, mas ela deve gerar, em um bloco de memória **alocado dinamicamente**, um trecho de código **em linguagem de máquina** que corresponde à nova função. O valor de retorno de **`cria_func`** será um ponteiro para esse bloco de memória.

Para facilitar a criação do código da nova função, você pode utilizar uma variação da instrução call (a instrução *call indireto*), onde o endereço da função a ser chamada está em um registrador. Por exemplo, a instrução abaixo faz uma chamada para a função cujo endereço foi armazenado no registrador %rax:

```
  call *%rax
```

De forma geral, `cria_func` irá percorrer o array com a descrição dos parâmetros e gerar um código em linguagem de máquina que:

- alinhe a pilha (usando o prólogo); isto é necessário porque poderemos chamar funções de biblioteca!
- coloque os valores a serem passados para a função original nos registradores correspondentes, respeitando os tipos e eventuais valores amarrados especificados no array `params`;

  Atenção: lembre-se que a localização (registradores) dos parâmetros **não amarrados**, recebidos pela função gerada dinamicamente, pode não ser a mesma para a chamada à função original. Cuidado para não perder o valor desses parâmetros!

- chame a função original usando a instrução `call` (indireto);
- desfaça o registro de ativação (leave)
- retorne o controle para seu chamador, mantendo inalterado qualquer valor de retorno da função original.

Você deve criar um arquivo fonte chamado **cria_func.c** contendo as funções `cria_func` e `libera_func` e funções auxiliares, se for o caso. Esse arquivo não deve conter uma função `main` nem depender de arquivos de cabeçalho além de `cria_func.h` e dos cabeçalhos das bibliotecas padrão!

Para testar o seu programa, crie um outro arquivo, por exemplo `teste.c`, contendo a função `main`. Crie seu programa executável, por exemplo `teste`, com a linha

```
gcc -Wall -Wa,--execstack -o teste cria_func.c teste.c
```

(sem a opção *execstack* , o sistema operacional abortará o seu programa, por tentar executar um código armazenado na área de dados).

Dicas

Recomendamos fortemente uma implementação gradual, desenvolvendo e **testando** passo-a-passo cada nova funcionalidade implementada.

Comece, por exemplo, com um esqueleto que aloca espaço para o código a ser gerado, coloca um código bem conhecido nessa região e retorna o endereço da região alocada. Teste a chamada a essa função gerada dinamicamente. Para obter um código "bem conhecido" você pode compilar um arquivo *assembly* contendo uma função bem simples (que, por exemplo, retorna o valor do seu parâmetro) usando:

```
minhamaquina> gcc -c code.s
```

A opção -c é usada para compilar e não gerar o executável. Depois de compilar, veja o código de máquina gerado usando:

```
minhamaquina> objdump -d code.o
```

A seguir, implemente a geração dinamica do código, e comece a testá-la. Por exemplo, comece usando `cria_func` para gerar um código que chama uma função que retorna o valor de seu único parâmetro inteiro. Teste primeiro sem amarrar o parâmetro, e depois o amarrando a um valor fixo e ao valor de uma variável.

Vá então aumentando gradualmente o número e os tipos de parâmetros tratados, testando combinações diferentes. Você pode usar os exemplos dados no enunciado do trabalho, mas faça também outros testes!

Entrega

Leia com atenção as instruções para entrega a seguir e siga-as estritamente. **Atenção para os nomes e formatos dos arquivos!**

Devem ser entregues **via Moodle** dois arquivos:

1. o arquivo fonte **cria_func.c**

   Coloque no início do arquivo fonte, como comentário, os nomes dos integrantes do grupo da seguinte forma:

   ```
   /* Nome_do_Aluno1 Matricula Turma */
   /* Nome_do_Aluno2 Matricula Turma */
   ```

   Lembre-se que esse arquivo não deve conter a função `main`!

2. um arquivo texto (não envie um .doc ou .docx), chamado **relatorio.txt**, contendo um pequeno relatório que descreva os testes realizados.

   Esse relatório deve explicar o que está funcionando e o que não está funcionando. Mostre exemplos de testes executados com sucesso e os que resultaram em erros (se for o caso).

   Coloque também no relatório os nomes dos integrantes do grupo.

Coloque na área de texto da tarefa do Moodle os nomes e turmas dos integrantes do grupo.

- Para grupos de alunos de uma mesma turma, **apenas uma entrega é necessária** (com o *login* de um dos integrantes do grupo).
- Caso os integrantes do grupo sejam de turmas diferentes, o trabalho deve ser entregue pelos dois alunos.