INF1018 - Software Básico (2015.2)
Segundo Trabalho

Gerador Dinâmico de Funções

O objetivo deste trabalho é implementar em C uma função **`cria_func`** que recebe uma função **`f`** e a descrição de um conjunto de parâmetros e retorna o endereço de uma "nova versão" de **`f`**, gerada dinamicamente (isto é, o código dessa nova versão é gerado em tempo de execução). Também deve ser implementada uma função **`libera_func`**, que é responsável por liberar a memória alocada para a função criada dinamicamente.

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

Considere, por exemplo, a função **`pow`**, da biblioteca matemática, que retorna o valor de seu primeiro parâmetro (x) elevado ao segundo (y):

```
double pow (double x, double y);
```

A função **`cria_func`** nos permite criar dinamicamente uma nova função que sempre eleva o seu parâmetro ao quadrado. Para criar essa função, `cria_func` *amarra* o segundo parâmetro de `pow` a um valor fixo, construindo, em tempo de execução, o código de uma função que chama `pow` com dois parâmetros: o primeiro é seu próprio parâmetro e o segundo é o valor 2.0. Quando `pow` retornar, essa função retorna o valor retornado por `pow`.

Podemos criar também uma versão de **`pow`** *amarrando* o segundo parâmetro da função original a uma variável. Neste caso, sempre que charmarmos essa nova função, o segundo parâmetro que ela passará para **`pow`** será o conteúdo atual dessa variável.

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

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 a ser chamada, o array **`params`** contém a descrição dos parâmetros de **`f`** e **`n`** é o número de parâmetros descritos por **`params`**.

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

```
typedef enum {INT_PAR, FLOAT_PAR, DOUBLE_PAR, PTR_PAR} TipoValor;
typedef enum {PARAM, FIX_DIR, FIX_IND} OrigemValor;

typedef struct {
   TipoValor    tipo_val;  /* indica o tipo do parametro (inteiro, float, double ou ponteiro) */
```

```
   OrigemValor  orig_val;  /* indica a origem do valor do parametro */
   union {
     int v_int;
     float v_float;
     double v_double;
     void* v_ptr;
   } valor;         /* define o valor/endereço do valor do parametro */
} DescParam;
```

O campo **`orig_val`** indica se o parâmetro deve ser "amarrado" ou não, e comoo desejamos amarrar o parâmetro. Ele pode conter os seguintes valores:

- **`PARAM`**: o parâmetro não é amarrado, e deve ser passado para a nova função.
- **`FIX_DIR`**: o parâmetro deve ser amarrado a um valor constante, fornecido no campo **`valor`**.
- **`FIX_IND`**: o parâmetro deve ser amarrado a uma variável, cujo endereço é fornecido no campo **`valor`**.

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.

Exemplos de uso

O programa abaixo usa **`cria_func`** para criar dinamicamente uma função que eleva números ao quadrado e depois invoca essa função para obter os quadrados de 1 a 10:

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

typedef double (*func_ptr) (double x);

int main (void) {
  DescParam params[2];
  func_ptr f_quadrado = NULL;
  int i;

  params[0].tipo_val = DOUBLE_PAR;
  params[0].orig_val = PARAM;
  params[1].tipo_val = DOUBLE_PAR;
  params[1].orig_val = FIX_DIR;
  params[1].valor.v_double = 2.0;

  f_quadrado = (func_ptr) cria_func (pow, params, 2);
  for (i = 1; i <= 10; i++) {
     printf("%d ^ 2 = %4.1f\n", i, f_quadrado(i));
  }
  libera_func(f_quadrado);
  return 0;
}
```

Neste outro programa, amarramos os dois parâmetros de **`pow`**, obtendo potências de 2:

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

```
typedef double (*func_ptr) ();

int main (void) {
  DescParam params[2];
  func_ptr f_potencia = NULL;
  double pot;

  params[0].tipo_val = DOUBLE_PAR;
  params[0].orig_val = FIX_DIR;
  params[0].valor.v_double = 2.0;
  params[1].tipo_val = DOUBLE_PAR;
  params[1].orig_val = FIX_IND;
  params[1].valor.v_ptr = &pot;

  f_potencia = (func_ptr) cria_func (pow, params, 2);
  for (pot = 0.0; pot <=  10.0; pot++) {
     printf("2.0 ^ %2.1f = %4.1f\n", pot, f_potencia());
  }
  libera_func(f_potencia);
  return 0;
}
```

Ainda um outro exemplo é um uso da função de comparação de bytes **`memcmp`**, da biblioteca padrão de C, para determinar se diferentes strings tem um mesmo prefixo que uma determinada string fixa.

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);
```

Podemos usar **`cria_func`** para fixar uma das strings, passando para a nova função apenas a string que queremos testar e o tamanho do prefixo:

```
#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 e_prefixo = NULL;
  char uma[] = "quero saber tudo";
  int tam;

  params[0].tipo_val = PTR_PAR;
  params[0].orig_val = FIX_DIR;
  params[0].valor.v_ptr = fixa;
  params[1].tipo_val = PTR_PAR;
  params[1].orig_val = PARAM;
  params[2].tipo_val = INT_PAR;
  params[2].orig_val = PARAM;

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

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

  libera_func(e_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.

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

- empilhe os parâmetros para a função original, respeitando os tipos e eventuais valores amarrados especificados no array `params`;
- chame a função original via a instrução `call`;
- jogue fora os parâmetros empilhados;
- 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 -m32 -Wa,--execstack -o teste cria_func.c teste.c -lm
```

(a opção **-lm** é necessária se a biblioteca matemática precisar ser incluída no passo de ligação)

Arquivos que não compilem no ambiente Linux não serão considerados!

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 simples (o retorno do seu parâmetro, por exemplo) usando:

```
minhamaquina> gcc -m32 -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, por exemplo, implemente e teste a geração de código para chamar uma função que retorna o valor de seu único parâmetro inteiro, primeiro sem amarrar e depois amarrando esse parâmetro. Vá depois aumentando gradualmente o número e os tipos de parâmetros tratados.

Você pode usar os exemplos dados no enunciado do trabalho mas faça também testes mais completos!

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.

Prazo

O trabalho deve ser entregue **até meia-noite do dia 20 de novembro**.

Será descontado **um ponto** por cada dia de atraso (incluindo sábado, domingo e feriados).

Observações

- O programa `objdump` será provavelmente a maior fonte de referência. Os opcodes das instruções em linguagem de máquina que devem ser geradas dinamicamente também podem ser obtidos no Intel Pentium Instruction Set Reference Manual (Volume 2).

- Os trabalhos devem preferencialmente ser feitos em grupos de dois alunos.

  Alguns grupos poderão ser chamados para apresentação oral de seus trabalhos. Neste caso, **ambos os membros do grupo deverão estar presentes na apresentação**.