ORGANIZAÇÃO GERAL DOS PROJETOS

Serão desenvolvidos alguns projetos durante o semestre. Todos serão organizados como descrito a seguir.

Basicamente, os programas lerão 2 arquivos: um arquivo que descreve um conjunto de dados a serem armazenados em alguma estrutura de dados e um arquivo descrevendo operações sobre o primeiro conjunto. As operações podem causar a delição, modificação, inserção de dados ao primeiro conjunto. O resultado final do processamento de cada conjunto é gravado em um ou mais arquivos de saída.

ENTRADA DE DADOS

A entrada de dados, via de regra, ocorrerá por meio de um ou mais arquivos-texto. Estes arquivos estarão sob um diretório, referenciado por **$DIR_ENTRADA** neste texto.[1]

SAIDA DE DADOS

O dados produzidos serão mostrados na saída padrão e/ou em diversos arquivos-texto. Alguns resultados serão gráficos no formato SVG. Os arquivos de saída serão colocados sob um diretório, referenciado por **$DIR_SAIDA** neste texto.[2]

PROCESSO DE COMPILAÇÃO E TESTES DO TRABALHO

O trabalho deve ser armazenado em um repositório GIT (público), que será clonado pelo professor. O repositório deve estar organizado da seguinte forma:

`LEIA-ME.txt`	*colocar matrícula e o nome do aluno.*
`*`	*Outros arquivos podem ser solicitados a cada fase.*
`/src`	*(arquivos-fonte)*
`makefile`	*deve ter target para a geração do arquivo objeto de cada módulo e o target* **`ted`** *que produzirá o executável de mesmo nome dentro do mesmo diretório* **`src`**. *Os fontes devem ser compilados com a opção -fstack-protector-all.* ** adotamos o padrão C99. Usar a opção -std=c99.*
`*.h e *.c`	***Atenção:*** *não devem existir outros arquivos além dos arquivos fontes e do makefile. Outros arquivos serão apagados antes da compilação.*

O que entregar

Submeter ao Classroom um arquivo-texto (.txt) com uma linha, com o formato abaixo. A URL fornecida será usada para clonar o repositório (git clone) e o apelido é usado para identificar o aluno. Pede-se que o apelido seja composto pelo primeiro nome do aluno seguido por algumas iniciais do sobrenome.

---

1 Indicado pela opção -e.
2 Indicado pela opção -o.

`apelido url-do-repositorio`
Exemplo: `joseMane https://github.com/zemane/t2.git`

Organização do diretório para a compilação e correção dos trabalhos (no computador do professor):

[HOME DIR]

`*.py`	*scripts para compilar e executar*
\t	*diretório contendo os arquivos de testes*
`*.geo *.qry`	*arquivos de consultas, talvez, distribuídos em alguns outros sub-diretórios*
\alunos	*(contém um diretório para cada aluno)*
\apelido	*diretório criado pela operação* `git clone`*. O nome do diretório é o apelido informado no arquivo-texto entregue.*
	outros subdiretórios para os arquivos de saída informados na opção `-o`

Os passos para correção serão os seguintes:

1. O repositório git informado pelo aluno é clonado dentro do diretório alunos, conforme mostrado acima
2. O makefile provido pelo aluno será usado para compilar os módulos e produzir o executável. Os fontes serão compilados com o compilador **gcc** em um máquina **Linux**. Os executáveis devem ser produzidos no mesmo diretório dos arquivos fontes O professor usará o **GNU Make**. Será executado (a partir dos scripts) o seguinte comando:[3]
   - **`make ted`**
3. O programa será executado automaticamente várias vezes: uma vez para cada teste e o resultado produzido será inspecionado visualmente pelo professor. Cada execução produzirá (pelo menos) um arquivo **.svg** diferente dentro do diretório **$DIR_SAIDA**, informado na opção **-o**. Possivelmente serão produzidos outros arquivos .**svg** e .**txt**.

3 O nome do executável poderá ser modificado em cada trabalho.

Os Arquivos de Entrada e Saída

A entrada do programa é composta por, em geral, dois arquivos-texto:

- arquivo .**geo**: Um arquivo com a extensão .geo (por exemplo, teste1.geo, arquiv.geo, etc) contém os comandos que especificam os dados que serão manipulados pelo programa;
- arquivo .**qry**: contém consultas, alterações e delições de dados provenientes do arquivo .geo. Alguns comandos também podem provocar a inserção de novos comandos.

Os arquivos de entrada são compostos, basicamente, por conjunto de comandos (um por linha). Cada comando tem um certo número de parâmetros separados por um espaço (branco)

A figura abaixo mostra um exemplo de um arquivo de entrada (hipotético). Note que a extensão do arquivo é **`.geo`**. Note que cada linha é iniciada por um comando (`c` e `r`, no exemplo), seguido por alguns parâmetros, separados por um espaço em branco. Estes parâmetros podem ser número inteiros, número reais, cadeias de caracteres.

```
c 1 50.00 50.0 30.00 grey magenta
r 6 121.0 46.0 100.0 30.0 cyan yellow
c 2 120.0 45.0 15.0 grey magenta
r 4 10.0 150.0 90.0 40.0 cyan yellow
c 5 230.0 180.0 13.0 grey magenta
```

***a01*.geo**

O arquivo .**`qry`** é semelhante ao anterior. Note que, via de regra, um comando deste arquivo pode fazer referência a alguma "entidade" criada no arquivo .**`geo`** por meio de um identificador (número inteiro ou cadeia de caracteres). No exemplo abaixo, o comando `o?` faz referência às entidades `2` e `6`, criadas, respectivamente, pelos comando `c` e `r`. Também, o comando `i?` referencia a entidade de identificador `5` criada pelo comando `c`.

```
o? 2 6
i? 5 210.0 160.0
```

q.qry

A Saída

Normalmente, os dados ("entidades") criados pelo arquivo .geo e manipulados pelo arquivo .qry são representadas por formas geométricas de duas dimensões. Assim, via de regra, o programa produzirá um arquivo **`.svg`** e um arquivo **`.txt`** ambos com o mesmo nome base do arquivo **`.geo`**.

O arquivo .**`svg`** produzido deve mostrar as formas geométricas resultantes do processamento dos arquivos de entrada. Existem várias ferramentas que renderizam arquivos .svg. As figuras abaixo mostram um exemplo de arquivo **`.svg`** e sua respectiva renderização.

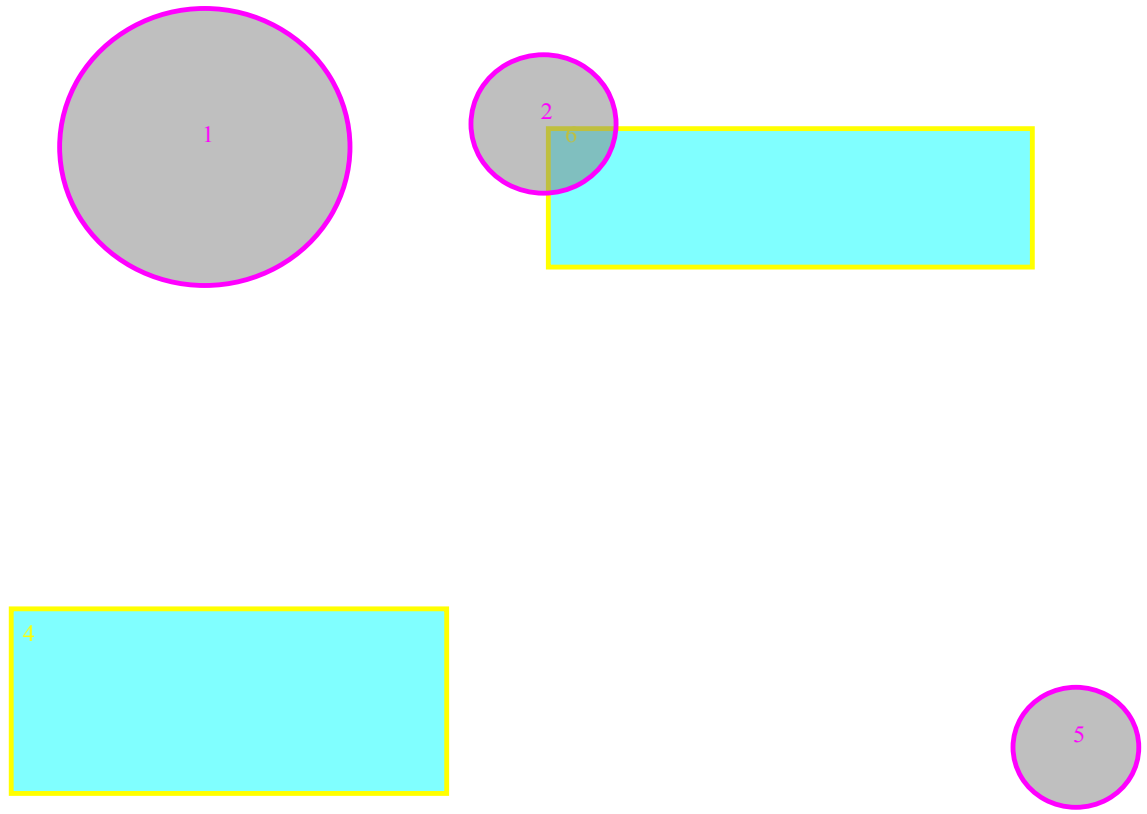

Illustration 1: Arquivo a01.svg (renderizado)

```
<svg xmlns:svg="http://www.w3.org/2000/svg"
   xmlns="http://www.w3.org/2000/svg"
   version="1.1">
  <circle style="fill:#808080;fill-opacity:0.5;stroke:#ff00ff"
     r="30"    cy="40.693146"    cx="41.200001" />
  <text style="font-size:5px;line-height:0%;fill:fuchsia"
     font-size="5"  y="40.69"  x="41.2">  1  </text>
  <rect style="fill:#00ffff;fill-opacity:0.5;stroke:#ffff00"
     height="30"  width="100"   y="36.693146"  x="112.2" />
  <text  style="font-size:5px;line-height:0%;fill:yellow"
     font-size="5"  y="40.69"  x="116.2">  6  </text>
  <circle  style="fill:#808080;fill-opacity:0.5;stroke:#ff00ff"
     r="15" cy="35.693146"  cx="111.2" />
  <text  style="font-size:5px;line-height:0%;fill:fuchsia"
     font-size="5"   y="35.69"  x="111.2">  2  </text>
  <rect   style="fill:#00ffff;fill-opacity:0.5;stroke:#ffff00"
     height="40"   width="90"  y="140.69315"   x="1.2" />
  <text   style="font-size:5px;line-height:0%;fill:yellow"
     font-size="5"   y="148.7"   x="4.2">  4  </text>
  <circle  style="fill:#808080;fill-opacity:0.5;stroke:#ff00ff"
     r="13"   cy="170.69315"   cx="221.2" />
  <text    style="font-size:5px;line-height:0%;fill:fuchsia"
     font-size="5"   y="170.7"   x="221.2">  5  </text>
</svg>
```

Arquivo a01.svg

Se o arquivo .geo (a01.geo, por exemplo) for processado em conjunto com um arquivo .qry

(q.qry, por exemplo), além de a01.svg, **deverá** ser produzido um arquivo svg (cujo nome é a **combinação** do nome dos 2 arquivos, por exemplo: a01-q.svg) e um arquivo txt (também com nome combinado, por exemplo: a01-q.txt). O arquivo .svg deve representar o estado final do "banco de dados", após o arquivo .qry ter sido completamente processado. Por sua vez, o arquivo .txt deve conter o resultado textual de todas as consultas, conforme as especificações dos comandos. Neste arquivo deve ser copiado em uma linha o texto da consulta, precedido por "`[*]` " e, na linha seguinte, o seu resultado.

```
[*] inp 10
círculo
âncora em (123.500, 18.350)
raio: 15.00
preenchimento: blue
borda: yellow

[*] sel 50.0 21.0 60.0 35.00
18: circulo
29: linha
5: texto
9: texto
21: retâgulo
```

Arquivo a01-q.txt

O PROGRAMA

O nome do programa deve ser **ted** e aceitar alguns parâmetros:[4]

ted [-e *path*] -f *arq*.geo [-q consulta.qry] -o *dir*

O primeiro parâmetro (**-e**) indica o diretório base de entrada (**$DIR_ENTRADA**). É opcional. Caso não seja informado, o diretório de entrada é o diretório corrente da aplicação. O segundo parâmetro (**-f**) especifica o nome do arquivo de entrada que deve ser encontrado sob o diretório informado pelo primeiro parâmetro. O terceiro parâmetro (**-q**) é um arquivo de consultas (também sob o diretório de entrada). O último parâmetro (**-o**) indica o diretório onde os arquivos de saída (***.svg e *.txt**) deve ser colocados. Note que os nomes de arquivos podem ser precedidos por um caminho relativo; ***dir e path*** podem ser caminhos absolutos ou relativos . Note, também que a ordem dos parâmetros pode variar.

A seguir, alguns exemplos de possíveis invocações de **ted**:

- **ted** -e /home/ed/testes/ -f t001.geo -o /home/ed/alunos/aluno1/o/
- **ted** -e /home/ed -f ts/t001.geo -o /home/ed/alunos/al1/o
- **ted** -f ./tsts/t001.geo -e /home/ed -o /home/ed/alunos/aluno1/o/
- **ted** -o ./alunos/aluno1/o -f ./testes/t001.geo
- **ted** -o ./alunos/aluno1/o -f ./testes/t001.geo -q ./t001/q1.qry
- **ted** -e ./testes -f t001.geo -o ./alunos/aluno1/o/ -q ./q1.qry

4 Novos parâmetros poderão ser acrescentados.

RESUMO DOS PARÂMETROS DO PROGRAMA SIGUEL

Parâmetro / argumento	Opcional	Descrição
`-e` ***`path`***	S	Diretório-base de entrada (**$DIR_ENTRADA**)
`-f` ***`arq`*`.geo`**	N	Arquivo com a descrição da cidade. Este arquivo deve estar sob o diretório **$DIR_ENTRADA**.
`-o` ***`path`***	N	Diretório-base de saída (**$DIR_SAIDA**)
`-q` ***`arqcons`*`.qry`**	S	Arquivo com consultas. Este arquivo deve estar sob o diretório **$DIR_ENTRADA**.

RESUMO DOS ARQUIVOS PRODUZIDOS

-f	-q	comando com sufixo	arquivos
arq.geo			arq.svg
arq.geo	***arqcons***.qry		arq.svg arq-arqcons.svg arq-arqcons.txt
arq.geo	***arqcons***.qry	***sufx***	arq.svg arq-arqcons.svg arq-arqcons.txt arq-arqcons-sufx.[svg|txt][5]

ATENÇÃO:
* os fontes devem ser compilados com a opção **`-fstack-protector-all`**.
* adotamos o padrão C99. Usar a opção **`-std=c99.`**

5 Podem ser produzidos os respectivos arquivos .svg e/ou .txt, dependendo da especificação do comando.

APÊNDICES

Exemplo de Tratamento dos Parâmetros

A seguir, exemplo de um programa que faz tratamento do parâmetros recebidos em sua invocação via linha de comando. Note os parâmetros **argc** e **argv** do procedimento main.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#define PATH_LEN 250
#define FILE_NAME_LEN 100
#define MSG_LEN 1000

void trataPath(char *path, int tamMax, char* arg){
  int argLen = strlen(arg);

  assert(argLen<tamMax);
  if(arg[argLen-1]='/'){
    arg[argLen-1]='\0';
  }
  strcpy(path,arg);
}


void trataNomeArquivo(char *path, int tamMax, char* arg){
  int argLen = strlen(arg);

  assert((argLen+4)<tamMax);
  sprintf(path,"%s.txt",arg);
}


void main(int argc, char *argv[]){
  char dir[PATH_LEN],  arq[FILE_NAME_LEN], msg[MSG_LEN];
  char *fullNameArq;
  FILE *f;

  /* MOSTRA OS PARAMETROS */

  for(int i= 0; i< argc; i++){
    printf("argv[%d] = %s\n",i,argv[i]);
  }

  /* TRATA PARAMETROS */
```

```
  int i = 1;
  strcpy(msg,"");

  while (i < argc){
    if (strcmp(argv[i],"-d")==0){
      i++;
      /* se i >= argc: ERRO-falta parametro */
      trataPath(dir,PATH_LEN, argv[i]);
    }
    else if (strcmp(argv[i],"-f") == 0){
      i++;
      /* se i >= argc: ERRO-falta parametro */
      trataNomeArquivo(arq,FILE_NAME_LEN,argv[i]);
    }
    else{
      strcat(msg,argv[i]);
      strcat(msg," ");
    }
    i++;
  } //while

  /* GRAVA MENSAGEM NO ARQUIVO (DE ACORDO COM OS PARAMETROS) */

  int pLen = strlen(dir);
  int fLen = strlen(arq);

  fullNameArq = (char *)malloc((pLen+fLen+2)*sizeof(char));
  sprintf(fullNameArq, "%s/%s",dir,arq);
  f = (FILE *)fopen(fullNameArq,"w");
  fprintf(f,"%s\n",msg);
  fclose(f);
  free(fullNameArq);
}
```

Makefile

Como já dito, utilizaremos GNU Make e makefiiles. Para saber mais sobre eles:

- https://www.gnu.org/software/make/
- https://www.gnu.org/software/make/manual/html_node/Introduction.html
- `http://opensourceforu.com/2012/06/gnu-make-in-detail-for-beginners/`

Abaixo é mostrado um modelo de makefile que, fortemente, sugere-se ser utilizado. Note que este modelo deve ser preenchido com detalhes específicos do trabalho e do aluno.

```
PROJ_NAME=ted

ALUNO=
LIBS=
OBJETOS=

# compilador
CC=gcc

# Flags

CFLAGS= -ggdb -O0 -std=c99 -fstack-protector-all -Werror=implicit-function-
declaration

LDFLAGS=-O0

$(PROJ_NAME): $(OBJETOS)
	$(CC) -o $(PROJ_NAME)$(LDFLAGS)  $(OBJETOS) $(LIBS)

%.o : %.c
	$(CC) -c $(CFLAGS) $< -o $@

#
# COLOCAR DEPENDENCIAS DE CADA MODULO
#

# Exemplo: suponha que o arquivo a.c possua os seguintes includes:
#
#   #include "a.h"
#   #include "b.h"
#   #include "c.h"
#
# a.o: a.h b.h c.h a.c
```

Valgrind

O valgrind é uma ferramenta para analisar a execução de programas e reportar possíveis bugs escondidos. É fortemente recomendado que testem o programa com o valgrind antes de entregá-lo. Consulte: https://valgrind.org/

CLI (Comand Line Interface)

A compilação do programa e execução dos testes não usará nenhum GUI nem nenhuma IDE. Usaremos uma interface de linha de comando (Bash). Por isso, é recomendável alguma experiência com este tipo de interface. Consultar, por exemplo:

- https://www.gnu.org/software/bash/
- https://youtu.be/oxuRxtrO2Ag