Laboratório 1

Uso das ferramentas de desenvolvimento de programas em C++.

Objetivo

O objetivo deste exercício é realizar a implementação de um programa na linguagem de programação C++ utilizando ferramentas de suporte ao programador, tais como o compilador e um sistema de controle de versão, e aplicando modularização e outras boas práticas de programação. O programa a ser implementado é uma espécie de calculadora que fornece as medidas de diversas figuras geométricas planas e espaciais.

Questão 01

A *Geometria* (do grego γεωμετρια; *geo-* = "terra", *-metron* = "medida") é um ramo da Matemática que estuda questões relacionadas à forma, tamanho e posição relativa de figuras e a propriedades do espaço. Essa ciência surgiu em diversas culturas da Antiguidade como um conjunto de conhecimentos práticos sobre comprimento, área e volume, partindo das necessidades e observações do ser humano para resolver problemas em agricultura, astronomia, arquitetura e engenharia. Com efeito, conhecimentos em Geometria são aplicados ainda hoje nos mais variados campos do conhecimento humano, tais como física, química, geologia, astronomia, engenharia, biologia, cartografia e computação.

Dentre as divisões da Geometria, encontram-se as chamadas *Geometria Plana* e *Geometria Espacial*. A Geometria Plana refere-se ao estudo das figuras geométricas definidas em um plano de duas dimensões, enquanto que a Geometria Espacial se encarrega do estudo das figuras geométricas (também chamadas de sólidos geométricos) definidas no espaço, ou seja, aquelas que possuem mais de duas dimensões e ocupam um lugar no espaço. As principais figuras geométricas planas são o *triângulo*, o *quadrado*, o *retângulo* e o *círculo*. Já as principais figuras geométricas espaciais são o *cubo*, a *esfera*, o *cone*, a *pirâmide*, o *paralelepípedo* e o *cilindro*.

Três conceitos são de suma importância para o entendimento das Geometrias Plana e Espacial, a saber, a *área*, o *perímetro* e o *volume*. A área de uma figura geométrica, seja ela plana ou espacial, expressa o tamanho de tal figura sobre uma superfície, de modo que quanto maior a superfície da figura, maior a sua área. O perímetro de uma figura geométrica é definido como a medida do contorno que delimita a figura, sendo resultante da soma das medidas de todos os seus lados. Por fim, o volume corresponde à medida do espaço ocupado por uma figura geométrica. Para encontrar os valores dessas medidas, é importante analisar o tipo da figura (se plana ou espacial) e a forma da figura, isto é, quantos e quais são os lados.

Área, perímetro e volume de figuras geométricas planas e espaciais

As Tabelas 1 e 2 apresentam a definição das principais figuras geométricas planas e espaciais, bem como as fórmulas utilizadas para calcular as medidas de área, perímetro e volume. É importante notar que, pelo fato de as figuras geométricas planas serem definidas em um plano de duas dimensões, elas não possuem volume.

Tabela 1. Área e perímetro das figuras geométricas planas

Figura	Definição	Área	Perímetro
Triângulo*	Figura fechada formada por três lados	$A = \frac{base \times altura}{2}$	$P = lado1 + lado2 + lado3$
Retângulo	Figura fechada formada por quatro lados que formam ângulos retos (90º)	$A = base \times altura$	$P = 2 \times (base + altura)$
Quadrado	Figura fechada formada por quatro lados congruentes (isto é, de medidas iguais) que formam ângulos retos	$A = lado^2$	$P = 4 \times lado$
Círculo	Figura fechada por uma linha curva chamada circunferência	$A = \pi \times r^2$	$P^{**} = 2 \times \pi \times r$

* Para este exercício, você deverá considerar um triângulo equilátero, no qual os três lados são congruentes.

** O perímetro de um círculo é chamado de *comprimento da circunferência*. π é uma constante definida com o valor aproximado de 3,1415 e *r* é a medida do *raio* do círculo, isto é, a distância entre o centro e a extremidade do círculo.

Tabela 2. Área e volume das figuras geométricas espaciais

Figura	Definição	Área	Volume
Pirâmide	Figura composta por uma base poligonal* (triangular, quadrangular, etc.) e um vértice que une as faces laterais da pirâmide	$A = area\_base + area\_lateral^{**}$	$V = \frac{1}{3} \times area\_base \times altura$
Cubo	Figura composta por seis faces quadrangulares	$A = 6 \times aresta^2$	$V = aresta^3$
Paralelepípedo	Figura composta por seis faces, tendo três pares de faces idênticas e paralelas entre si	$A = (2 \times aresta1 \times aresta2) + (2 \times aresta1 \times aresta3) + (2 \times aresta2 \times aresta3)$	$V = aresta1 \times aresta2 \times aresta3$
Esfera	Figura resultante do conjunto de pontos do espaço cuja distância ao centro é igual ou menor que o raio	$A = 4 \times \pi \times r^2$	$V = \frac{4}{3} \times \pi \times r^3$

* Para este exercício, você deverá considerar uma pirâmide com base quadrangular, ou seja, contendo uma base formando um quadrado e quatro faces laterais triangulares.

** A área lateral de uma pirâmide é dada pela soma das áreas de todas as faces laterais triangulares.

Tarefas

A tarefa principal a ser realizada neste exercício é a implementação de um programa que calcula as medidas de diversas figuras geométricas planas e espaciais. O programa em execução deve apresentar ao usuário uma lista de opções referentes às figuras e, após a escolha de uma dessas opções, deve solicitar ao usuário que forneça os dados necessários aos cálculos das medidas. Como resultado, o programa deve exibir as respectivas medidas (área, perímetro, volume) da figura escolhida. Note que, para figuras geométricas planas, o programa deve exibir apenas a área e o perímetro; para figuras geométricas espaciais, o programa deve exibir apenas a área e o volume da figura. Para calcular tais medidas, você deve fazer uso das equações matemáticas apresentadas nas Tabelas 1 e 2, além dos seus próprios conhecimentos matemáticos na área de Geometria.

Seguindo boas práticas de modularização, o seu programa deve ser composto de uma série de arquivos cabeçalho (`.h`) e corpo (`.cpp`), listados na Tabela 3. Você deve declarar os protótipos das funções que calculam as medidas de área, perímetro e volume das figuras geométricas nos arquivos cabeçalho e a respectiva implementação dessas funções deve ser feita nos arquivos corpo. Além disso, você também deverá implementar outros arquivos cabeçalho e corpo contendo funções que solicitem do usuário os dados necessários ao cálculo das medidas e que chamem as respectivas funções que realizarão esse cálculo.

Tabela 3. Arquivos que deverão ser implementados no programa

Arquivo	Descrição
`area.h`	Definição dos protótipos das funções que calculam a área de figuras geométricas planas e espaciais
`area.cpp`	Implementação das funções que calculam a área de figuras geométricas planas e espaciais
`calcula.h`	Definição dos protótipos das funções que solicitam ao usuário os dados necessários ao cálculo da área de acordo com a figura geométrica e faz chamada às funções de cálculo
`calcula.cpp`	Implementação das funções que solicitam ao usuário os dados necessários ao cálculo da área de acordo com a figura geométrica e chamam as funções que realizam essa operação
`perimetro.h`	Definição dos protótipos das funções que calculam o perímetro de figuras geométricas planas
`perimetro.cpp`	Definição dos protótipos das funções que solicitam ao usuário os dados necessários ao cálculo do perímetro de acordo com a figura geométrica
`volume.h`	Definição dos protótipos das funções que calculam o volume de figuras geométricas espaciais
`volume.cpp`	Implementação das funções que calculam o volume de figuras geométricas espaciais
`main.cpp`	Implementação da função principal do programa

Na implementação das funções que calculam as medidas de área, perímetro e volume das figuras, você pode fazer uso da biblioteca `<cmath>`. Dentre as funções disponibilizadas por essa biblioteca, existe (i) a função `pow`, que realiza a operação de potenciação de uma base e um expoente, recebidos

como parâmetro, e (ii) a função `sqrt`, que calcula a raiz quadrada de um número recebido como parâmetro.

Por fim, seu programa deverá conter ainda um arquivo com nome `main.cpp` com a implementação da função principal do programa e fazendo a inclusão dos arquivos cabeçalho necessários.

Uma vez que será necessário compilar e ligar vários arquivos de uma vez, você deverá escrever um arquivo `Makefile` para facilitar esse processo. O exemplo de `Makefile` apresentado a seguir leva em consideração a estrutura de diretórios apresentada em sala, que deverá ser seguida. Com isso, os arquivos usados na compilação e geração do(s) binário(s)/executável(is) já são buscados nos respectivos diretórios.

Estude o `Makefile` dado como modelo a seguir e adapte-o à sua implementação.

```
# Exemplo mais completo de um Makefile
#
# Algumas variaveis sao usadas com significado especial:
#
# $@ nome do alvo (target)
# $^ lista com os nomes de todos os pre-requisitos sem duplicatas
# $< nome do primeiro pre-requisito
#

# Comandos do sistema operacional
# Linux: rm -rf
# Windows: cmd //C del
RM = rm -rf

# Compilador
CC=g++

# Variaveis para os subdiretorios
LIB_DIR=./lib
INC_DIR=./include
SRC_DIR=./src
OBJ_DIR=./build
BIN_DIR=./bin
DOC_DIR=./doc
TEST_DIR=./test

# Outras variaveis


# Opcoes de compilacao
CFLAGS  = -Wall -pedantic -ansi -std=c++11 -I. -I$(INC_DIR)

# Garante que os alvos desta lista nao sejam confundidos com arquivos de mesmo nome
.PHONY: all clean distclean doxy

# Define o alvo (target) para a compilacao completa.
# Define os alvos questao01, questao02 e questao03 como dependencias.
```

```
# Ao final da compilacao, remove os arquivos objeto.
all: questao01 questao02 questao03

debug: CFLAGS += -g -O0
debug: questao01 questao02 questao03

# Alvo (target) para a construcao do executavel questao01
# Define os arquivos classe11.o, classe12.o e main1.o como dependencias
questao01: $(OBJ_DIR)/classe11.o $(OBJ_DIR)/classe12.o $(OBJ_DIR)/main1.o
	@echo "============="
	@echo "Ligando o alvo $@"
	@echo "============="
	$(CC) $(CFLAGS) -o $(BIN_DIR)/$@ $^
	@echo "+++ [Executavel questao01 criado em $(BIN_DIR)] +++"
	@echo "============="

# Alvo (target) para a construcao do objeto classe11.o
# Define os arquivos classe11.cpp e classe11.h como dependencias.
$(OBJ_DIR)/classe11.o: $(SRC_DIR)/questao01/classe11.cpp
$(INC_DIR)/questao01/classe11.h
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do objeto classe12.o
# Define os arquivos classe12.cpp, classe12.h e classe11.o como dependencias.
$(OBJ_DIR)/classe12.o: $(SRC_DIR)/questao01/classe12.cpp
$(INC_DIR)/questao01/classe12.h $(OBJ_DIR)/classe11.o
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do objeto main1.o
# Define o arquivo main1.cpp como dependencias.
$(OBJ_DIR)/main1.o: $(SRC_DIR)/questao01/main1.cpp
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do executavel questao02
# Define os arquivos classe21.o e main2.o como dependencias.
questao02: $(OBJ_DIR)/classe21.o $(OBJ_DIR)/main2.o
	@echo "============="
	@echo "Ligando o alvo $@"
	@echo "============="
	$(CC) $(CFLAGS) -o $(BIN_DIR)/$@ $^
	@echo "+++ [Executavel questao02 criado em $(BIN_DIR)] +++"
	@echo "============="

# Alvo (target) para a construcao do objeto classe21.o
# Define os arquivos classe21.cpp e classe21.h como dependencias.
$(OBJ_DIR)/classe21.o: $(SRC_DIR)/questao02/classe21.cpp
$(INC_DIR)/questao02/classe21.h
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do objeto main2.o
# Define o arquivo main2.cpp como dependências.
$(OBJ_DIR)/main2.o: $(SRC_DIR)/questao02/main2.cpp
	$(CC) -c $(CFLAGS) -o $@ $<
```

```
# Alvo (target) para a construcao do executavel questao03
# Define os arquivos classe31.o e main3.o como dependências.
questao03: $(OBJ_DIR)/classe31.o $(OBJ_DIR)/classe31.o $(OBJ_DIR)/main3.o
	@echo "============="
	@echo "Ligando o alvo $@"
	@echo "============="
	$(CC) $(CFLAGS) -o $(BIN_DIR)/$@ $^
	@echo "+++ [Executavel questao03 criado em $(BIN_DIR)] +++"
	@echo "============="

# Alvo (target) para a construcao do objeto classe31.o


# Define os arquivos classe31.cpp e classe31.h como dependências.
$(OBJ_DIR)/classe31.o: $(SRC_DIR)/questao03/classe31.cpp
$(INC_DIR)/questao03/classe31.h
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do objeto main3.o
# Define o arquivo main3.cpp como dependências.
$(OBJ_DIR)/main3.o: $(SRC_DIR)/questao03/main3.cpp
	$(CC) -c $(CFLAGS) -o $@ $<


# Alvo (target) para a construcao do objeto main2.o
# Define o arquivo main2.cpp como dependências.
$(OBJ_DIR)/main2.o: $(SRC_DIR)/questao02/main2.cpp
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do executavel questao03
# Define os arquivos classe31.o e main3.o como dependências.
questao03: $(OBJ_DIR)/classe31.o $(OBJ_DIR)/classe31.o $(OBJ_DIR)/main3.o
	@echo "============="
	@echo "Ligando o alvo $@"
	@echo "============="
	$(CC) $(CFLAGS) -o $(BIN_DIR)/$@ $^
	@echo "+++ [Executavel questao03 criado em $(BIN_DIR)] +++"
	@echo "============="

# Alvo (target) para a construcao do objeto classe31.o
# Define os arquivos classe31.cpp e classe31.h como dependências.
$(OBJ_DIR)/classe31.o: $(SRC_DIR)/questao03/classe31.cpp
$(INC_DIR)/questao03/classe31.h
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a construcao do objeto main3.o
# Define o arquivo main3.cpp como dependências.
$(OBJ_DIR)/main3.o: $(SRC_DIR)/questao03/main3.cpp
	$(CC) -c $(CFLAGS) -o $@ $<



# Alvo (target) para a construcao do executavel de teste
test: $(TEST_DIR)/main.o questao01 questao02 questao03
```

```
	@echo "============="
	@echo "Ligando o alvo $@"
	@echo "============="
	$(CC) $(CFLAGS) -o $(BIN_DIR)/testador $<
	@echo "+++ [Executavel testador criado em $(BIN_DIR)] +++"
	@echo "============="

# Alvo (target) para a construcao do objeto main.o
# Define o arquivo main.cpp como dependencias.
$(TEST_DIR)/main3.o: $(TEST_DIR)/main.cpp
	$(CC) -c $(CFLAGS) -o $@ $<

# Alvo (target) para a geração automatica de documentacao usando o Doxygen.
# Sempre remove a documentacao anterior (caso exista) e gera uma nova.
doxy:
	$(RM) $(DOC_DIR)/*
	doxygen Doxyfile

# Alvo (target) usado para limpar os arquivos temporarios (objeto)
# gerados durante a compilacao, assim como os arquivos binarios/executaveis.
clean:
	$(RM) $(BIN_DIR)/*
	$(RM) $(OBJ_DIR)/*


# FIM do Makefile
```

Exemplos de entrada e saída

Conforme mencionado anteriormente, o programa em execução deve primeiramente apresentar ao usuário uma lista com nove opções, numeradas de 0 a 8, referentes às figuras geométricas planas e espaciais das quais deseja-se obter as medidas. Quando o usuário digitar 0 como opção, o programa deverá ter sua execução finalizada. Caso seja digitada uma opção diferente das disponíveis, o programa deverá emitir uma mensagem informando da entrada inválida e solicitando que o usuário digite uma nova opção.

```
$ ./geometrica
Calculadora de Geometria Plana e Espacial
(1) Triangulo equilatero
(2) Retangulo
(3) Quadrado
(4) Circulo
(5) Piramide com base quadrangular
(6) Cubo
(7) Paralelepipedo
(8) Esfera
(0) Sair

Digite a sua opcao:
```

Depois de o usuário digitar a opção escolhida, o programa deverá solicitar os dados necessários para o cálculo das medidas de acordo com a figura em questão. Por exemplo, suponha que o usuário

tenha escolhido a opção 2, para cálculo das medidas referentes a um retângulo. Pelo fato de um retângulo ser uma figura geométrica plana, serão calculados apenas a sua área e o seu perímetro, de modo que é necessário informar o tamanho da base e da altura do retângulo.

```
Digite a sua opcao: 2
Digite o tamanho da base do retangulo: 5
Digite o tamanho da altura do retangulo: 3
```

Uma vez que o usuário informou os dados necessários aos cálculos, o programa deverá chamar as respectivas funções de cálculos das medidas de acordo com a figura geométrica em questão e exibir os resultados correspondentes. Ainda considerando o exemplo anterior das medidas referentes a um retângulo, o programa exibirá as medidas da área e do perímetro do retângulo.

```
Area do retangulo: 15
Perimetro do retangulo: 16
```

Após a exibição dos resultados das medidas de uma figura geométrica, o programa deverá exibir novamente a lista com as nove opções do menu inicial.

Questão 02

Escreva um programa chamado *anterior* que lê um valor inteiro e retorna o maior número primo inteiro anterior ao valor do fatorial desse número. Você deverá obedecer aos seguintes detalhes de implementação:

a. O valor do número inteiro deve ser lido através da linha de comando.
b. Implemente o seu programa de forma modular. Crie o conjunto de arquivos *fatorial.h/.cpp* (com a implementação da função de fatorial), *primalidade.h/.cpp* (com a implementação das funções que testam a primalidade de um valor inteiro e que retorna o maior primo inteiro anterior a X) e um arquivo *main.cpp* (contendo o programa principal). Crie um *Makefile* para a compilação e geração do binário/executável.
c. Utilize conceitos de recursividade para o cálculo do fatorial e para a obtenção do maior número primo anterior ao valor do fatorial do número em questão.

Você pode utilizar os seguintes casos de teste:

Entrada	Fatorial	Saída
5	120	113
3	6	5
9	362880	362867

Um exemplo de execução do programa seria:

```
$ ./anterior 5
$ Maior numero primo anterior ao fatorial de 5 (120) eh 113.
```

Com a ajuda do profiller Gprof, analise a execução de seu código e indique qual a função que é mais chamada e qual consome mais tempo.

Questão 03

Com a ajuda do depurador GDB, apresente os valores das variáveis *arg1* e *arg2* após a chamada de cada função no código abaixo, justificando os valores apresentados.

```
#include <iostream>

int funcX (int a, int b)
{
    ++a;
    b++;
    int result = a + b;
    return result;
}

int funcY (int* a, int b)
{
    int* y;
    (*y) = (*a);
    (*y) *= 5;
    int result = (*y) + b;
    return result;
}

void funcZ (int a, int b, int* result)
{
    a++;
    (*result) += a + 2*b;
}

int main(int argc, char* argv[])
{
    int arg1 = 11;
    int arg2 = 23;
    funcX (arg1,arg2);
    funcY (arg1,arg2);
    int resultado = 0;
    funcZ (arg1,arg2,&resultado);

    return 0;
}
```

Autoria e política de colaboração

Esta atividade é individual. Todos os arquivos de código fonte deverão ser mantidos em um repositório remoto Git para controle de versões, hospedado em algum serviço da Internet, a exemplo do GitHub, Bitbucket, Gitlab ou outro de sua preferência. Registre **todas as atividades** de implementação por meio de *commits*, cujo histórico será analisado durante a avaliação.

O trabalho em cooperação entre estudantes da turma é estimulado, sendo aceitável a discussão de ideias e estratégias. Contudo, tal interação não deve ser entendida como permissão para utilização

de (parte de) código fonte de outras equipes, o que pode caracterizar situação de plágio. Trabalhos copiados em todo ou em parte de outras equipes ou da Internet serão sumariamente rejeitados.

Entrega

Você deverá submeter um único arquivo compactado no formato `.zip` contendo todos os códigos fonte resultantes da implementação deste exercício, sem erros de compilação e devidamente testados e documentados, **até às 23h59 do dia 15 de agosto de 2017** através da opção *Tarefas* na Turma Virtual do SIGAA. Você deverá ainda informar, no campo *Comentários* do formulário de submissão da tarefa, o endereço do repositório Git no qual foram disponibilizados todos os arquivos referentes ao programa implementado. O repositório deve incluir um arquivo README contendo a identificação completa do aluno e do laboratório, a descrição de como compilar e rodar o programa - descrevendo inclusive as dificuldades encontradas.

Orientações gerais

Você deverá observar as seguintes observações gerais na implementação deste exercício:

1) Apesar da completa compatibilidade entre as linguagens de programação C e C++, seu código fonte não deverá conter recursos da linguagem C nem ser resultante de mescla entre as duas linguagens, o que é uma má prática de programação. Dessa forma, deverão ser utilizados estritamente recursos da linguagem C++.

2) Você deverá utilizar apenas um editor de texto simples (tais como o Gedit ou o Sublime) e o compilador em linha de comando, por meio do terminal do sistema operacional Linux.

3) Durante a compilação do seu código fonte, você deverá habilitar a exibição de mensagens de aviso (*warnings*), pois eles podem dar indícios de que o programa potencialmente possui problemas em sua implementação que podem se manifestar durante a sua execução.

4) Aplique boas práticas de programação. Codifique o programa de maneira legível (com indentação de código fonte, nomes consistentes, etc.) e documente-o adequadamente na forma de comentários. Seu código fonte deverá inclusive ser anotado para dar suporte à geração automática de documentação utilizando a ferramenta Doxygen (http://www.doxygen.org/). Consulte o documento disponibilizado na Turma Virtual do SIGAA com algumas instruções acerca do padrão de documentação e uso do Doxygen.

5) Busque desenvolver o seu programa com qualidade, garantindo que ele funcione de forma correta e eficiente. Pense também nas possíveis entradas que poderão ser utilizadas para testar apropriadamente o seu programa e trate adequadamente possíveis entradas consideradas inválidas.

6) A fim de garantir a boa manutenção de seu repositório, configure corretamente o arquivo `.gitignore` em seu repositório Git.

Avaliação

O trabalho será avaliado sob os seguintes critérios: (i) utilização correta dos conteúdos vistos anteriormente e nas aulas presenciais da disciplina; (ii) a corretude da execução do programa implementado, que deve apresentar saída em conformidade com a especificação e as entradas de dados fornecidas; (iii) a aplicação correta de boas práticas de programação, incluindo legibilidade, organização e documentação de código fonte, e; (iv) a utilização correta do repositório Git, no qual deverá estar registrado todo o histórico da implementação por meio de *commits*. A presença de mensagens de aviso (*warnings*) ou de erros de compilação e/ou de execução, a modularização inapropriada e a ausência de documentação são faltas que serão penalizadas. Este trabalho contabilizará nota de até 2,0 pontos na 1ª Unidade da disciplina.