Universidade Tecnológica Federal do Paraná
Departamento Acadêmico de Informática
CSF13 – Fundamentos de Programação 1
Prof. Bogdan Tomoyuki Nassu
Profa. Leyza Baldo Dorini

Trabalho 1

Esta é a especificação do primeiro projeto da disciplina CSF13 - Fundamentos de Programação 1, profs. Bogdan T. Nassu e Leyza B. Dorini, para o período 2022/2.

I) Equipes

O trabalho é individual. A discussão entre colegas para compreender a estrutura do trabalho é recomendada e estimulada, mas cada um deve apresentar suas próprias soluções para os problemas propostos. Indícios de fraude (cópia) podem levar à avaliação especial (ver item IV). Não compartilhe códigos (fonte nem pseudocódigos)!!!

II) Entrega

O prazo de entrega é 07/10/2022, 23:59. Trabalhos entregues após esta data terão sua nota final reduzida em 0,00025% para cada segundo de atraso. Cada estudante deve entregar, através da página da disciplina no Classroom, dois arquivos (separados, sem compressão):

- Um arquivo chamado *t1-x.c*, onde *x* é o seu número de matrícula. O arquivo deve conter as implementações das funções pedidas (com o cabeçalho idêntico ao especificado). Exceto afirmação em contrário, as funções pedidas não devem fazer uso de funções da biblioteca-padrão. Você também não pode usar vetores, matrizes ou outros recursos da linguagem C que não tenham sido explorados em aula antes da especificação deste trabalho. O autor do arquivo deve estar identificado no início, através de comentários.

IMPORTANTE: as funções pedidas não envolvem interação com usuários. Elas não devem imprimir mensagens (por exemplo, através da função `printf`), nem bloquear a execução enquanto esperam entradas (por exemplo, através da função `scanf`). O arquivo também não deve ter uma função `main`.

- Um arquivo no formato PDF chamado *t1-x.pdf*, onde *x* é o seu número de matrícula. Este arquivo deve conter um relatório breve (em torno de 1 a 2 páginas), descrevendo os desafios encontrados e a forma como eles foram superados. Não é preciso seguir uma formatação específica. O autor do arquivo deve estar identificado no início.

III) Avaliação (normal)

Todos os testes serão feitos usando a IDE Code::Blocks. Certifique-se de que o seu trabalho pode ser compilado e executado a partir dela.

Os trabalhos serão avaliados por meio de baterias de testes automatizados. A referência será uma implementação criada pelos professores, sem “truques” ou otimizações sofisticadas. Os resultados dos trabalhos serão comparados àqueles obtidos pela referência. Os seguintes pontos serão avaliados:

III.a) Compilação. Cada erro que impeça a compilação do arquivo implica em uma penalidade de até 20 pontos. Certifique-se que seu código compila!!!

III.b) Corretude. Sua solução deve produzir o resultado correto para todas as entradas testadas. Uma função que produza resultados incorretos terá sua nota reduzida. Quando possível, o professor corrigirá o código até que ele produza o resultado correto. A cada erro corrigido, a nota será multiplicada por um valor: até 0.7 se o erro for facilmente detectável em testes simples, ou 0.85 do contrário.

III.c) Atendimento da especificação. Serão descontados até 10 pontos para cada item que não esteja de acordo com esta especificação, como nomes de arquivos e funções fora do padrão.

III.d) Documentação. Comente a sua solução. Não é preciso detalhar tudo linha por linha, mas forneça uma descrição geral da sua abordagem, assim como comentários sobre estratégias que não fiquem claras na leitura das linhas individuais do programa. Uma função sem comentários terá sua nota reduzida em até 25%.

III.e) Organização e legibilidade. Use a indentação para tornar seu código legível, e mantenha a estrutura do programa clara. Evite construções como *loops* infinitos terminados somente por `break`, ou *loops* `for` com varías inicializações e incrementos, mas sem bloco de comandos. Evite também replicar trechos de programa que poderiam ser melhor descritos por repetições. Um trabalho desorganizado ou cuja legibilidade esteja comprometida terá sua nota reduzida em até 30%.

III.f) Variáveis com nomes significativos. Use nomes significativos para as variáveis – lembre-se que `n` ou `x` podem ser nomes aceitáveis para um parâmetro de entrada que é um número, mas uma variável representando uma "soma total" será muito melhor identificada como `soma_total`, `soma` ou `total`; e não como `t`, `aux2` ou `foo`. Uma função cujas variáveis internas não tenham nomes significativos terá sua nota reduzida em até 20%.

III.g) Desempenho. Se a estrutura lógica de uma função for pouco eficiente, por exemplo, realizando muitas operações desnecessárias ou redundantes, a sua nota pode ser reduzida em até 15%.

IV) Avaliação (especial)

Indícios de fraude podem levar a uma avaliação especial, com o aluno sendo convocado e questionado sobre aspectos referentes aos algoritmos usados e à implementação. Além disso, alguns alunos podem ser selecionados para a avaliação especial, mesmo sem indícios de fraude, caso exista uma grande diferença entre a nota do trabalho e a qualidade das soluções apresentadas em atividades anteriores, ou se a explicação sobre o funcionamento de uma função for pouco clara.

V) Nota

A nota final do trabalho será igual à soma das notas de cada função. A nota máxima de cada função é listada junto à sua descrição. A nota máxima para o relatório é 5 pontos.

VI) Apoio

Caso tenha dúvidas, procure a monitoria. Os professores também estarão disponíveis para tirar dúvidas a respeito do projeto, nos horários de atendimento previstos (e em casos excepcionais, fora deles). A comunicação por e-mail ou via Discord pode ser usada para pequenas dúvidas sobre aspectos pontuais.

Será disponibilizado um "pacote" contendo os seguintes arquivos:

1) Um arquivo main.c, contendo o código-fonte de um programa para testar as funções pedidas. Note que você não precisa compreender os testes.

2) Um arquivo trabalho1.h, contendo as declarações das funções pedidas e de funções auxiliares. Este arquivo deve ser incluído no seu arquivo .c através da diretiva `#include`. As funções se dividem em 3 tipos:

- Funções do trabalho: estas são as funções pedidas, que cada um deve implementar.

- Funções auxiliares que devem ser chamadas pelos alunos: estas são as funções mencionadas na descrição das funções pedidas. Você deve fazer uso destas funções.

- Funções auxiliares que não devem ser chamadas pelos alunos: estas funções são usadas nos testes, mas não devem ser usadas no seu trabalho.

3) Um arquivo trabalho1.c, contendo as implementações das funções auxiliares, assim como dados internos usados para os testes. Você não precisa compreender o conteúdo deste arquivo, nem mesmo das funções auxiliares usadas pelo seu trabalho. Lembre-se: é preciso abstrair!

4) Arquivos .txt contendo dados de teste. A correção final será baseada em casos diferentes daqueles disponibilizados para testes, portanto as suas funções não devem cobrir apenas os casos de teste, mas qualquer outro caso que se encaixe na especificação.

Para usar os arquivos disponibilizados:

1) Crie no Code::Blocks um projeto vazio (Empty project).

2) Coloque todos os arquivos disponibilizados no mesmo diretório criado para o projeto.

3) Adicione ao projeto os arquivos main.c, trabalho1.c e trabalho1.h.

4) Crie o arquivo para as suas funções, e adicione-o ao projeto.

5) Crie corpos “vazios” para as funções pedidas. Os protótipos podem ser copiados/colados diretamente do arquivo trabalho1.h. Use valores de retorno arbitrários, por enquanto.

6) Verifique se o projeto pode ser compilado e o programa executado.

Funções 1 e 2:

Você está participando da criação de um jogo no qual bombas de tinta são disparadas de canhões. O objetivo é cobrir os alvos do adversário com a cor da sua equipe. Neste jogo, a região de cada alvo é modelada como um círculo, definido por 3 parâmetros: o raio e as coordenadas (x, y) do centro em um plano cartesiano. Alvos complexos são construídos como agrupamentos de alvos mais simples. A região afetada por um disparo é modelada da mesma forma que um alvo, como um conjunto de círculos. Você deve escrever duas funções que testam os efeitos de um disparo sobre os alvos dos adversários.

Dado um componente de um disparo e um alvo, existem 4 possíveis resultados (Fig. 1 a 4):

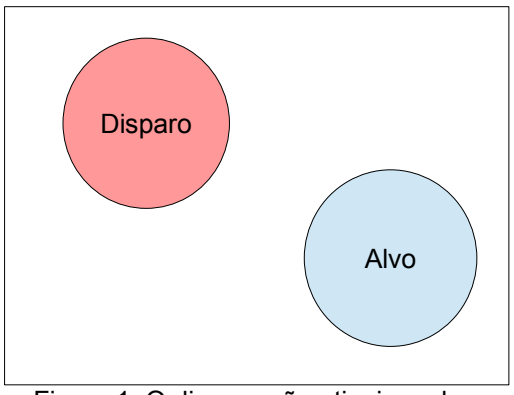

Figura 1: O disparo não atingiu o alvo.

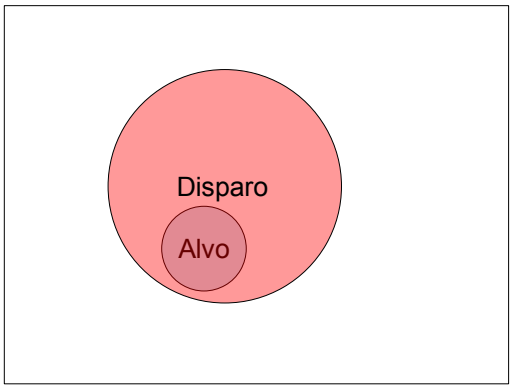

Figura 2: O disparo cobre completamente o alvo.

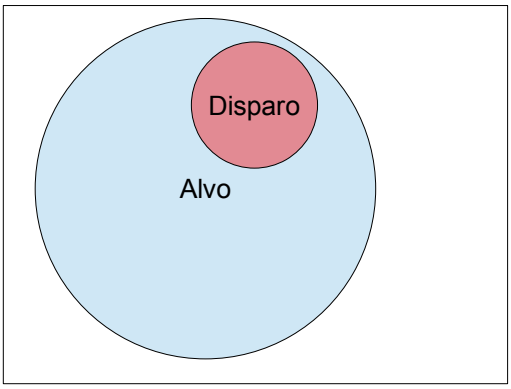

Figura 3: O disparo é completamente coberto pelo alvo.

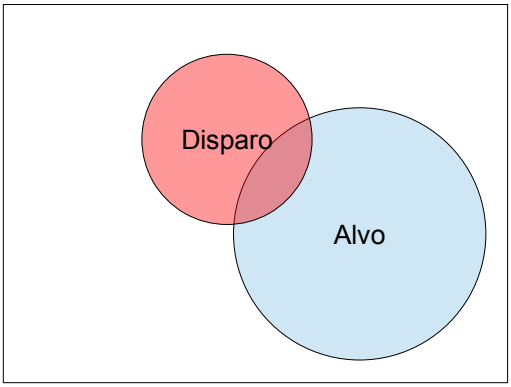

Figura 4: O disparo atingiu o alvo.

Note que um único disparo pode atingir múltiplos alvos, e que um único alvo pode ser atingido por múltiplos disparos. Estes casos são apenas combinações dos 4 casos acima.

Função 1 (50 pontos):

```
unsigned int testaDisparo (int disparo, float x, float y, float raio);
```

A primeira função que você deve escrever testa os efeitos de um único disparo sobre os alvos do adversário. Ela recebe como parâmetros o identificador do disparo e os valores que definem a região afetada por ele. O identificador do disparo é um inteiro entre 0 e *N*-1, onde *N* é o número de disparos realizados (lembrando que uma área complexa é modelada como um conjunto de disparos). A função deve verificar, para cada alvo, qual dos 4 casos ilustrados anteriormente ocorreu. Os alvos são identificados por números de 0 a *M*-1, onde *M* é o número de alvos do adversário. Para obter informações sobre os alvos, a sua função deve invocar 4 outras funções, declaradas no arquivo trabalho1.h:

```
unsigned int pegaNAlvos ();
```

Retorna o número de alvos (*M*).

```
float pegaXAlvo (unsigned int alvo);
float pegaYAlvo (unsigned int alvo);
```

Retornam respectivamente as coordenadas no eixo *x* e *y* do centro do alvo com o identificador dado.

```
float pegaRaioAlvo (unsigned int alvo);
```

Retorna o raio do alvo com o identificador dado.

A função `testaDisparo` deve avaliar os efeitos do disparo em cada alvo, analisando os alvos em ordem, do menor para o maior identificador. Se um alvo tiver sido atingido (casos ilustrados nas Figuras 2, 3 e 4), deve-se invocar uma função, declarada no arquivo trabalho1.h, com o seguinte protótipo:

```
void registraAcerto (int tipo, int disparo, int alvo);
```

O nome da função `registraAcerto` é autoexplicativo: ela registra a ocorrência de um acerto. Ela recebe como parâmetros os identificadores do disparo e do alvo, assim como um número que indica o tipo de acerto ocorrido. Este número é dado por 3 macros, declaradas no arquivo trabalho1.h: `ALVO_COBERTO`, `DISPARO_COBERTO` e `ACERTO_COMUM`, que correspondem respectivamente aos casos ilustrados nas Figuras 2, 3 e 4.

Para identificar os acertos, a sua função deve realizar cálculos relacionando as distâncias e áreas dos círculos que representam o disparo e os alvos. Estes cálculos podem fazer uso das funções `sqrt` ou `sqrtf` da biblioteca-padrão. Você também pode criar funções para realizar sub-tarefas, caso julgue necessário.

Ao final, a função `testaDisparo` deve retornar o número total de alvos atingidos pelo disparo.

Função 2 (25 pontos):

```
unsigned int testaDisparos ();
```

A segunda função que você deve escrever testa os efeitos de todos os disparos sobre os alvos do adversário. Ela deve chamar a função `testaDisparo` para cada disparo realizado, em ordem, do menor para o maior identificador. Para obter os dados sobre os disparos, a sua função deve invocar 4 outras funções, declaradas no arquivo trabalho1.h:

```
unsigned int pegaNDisparos ();
```

Retorna o número de disparos realizados.

```
float pegaXDisparo (unsigned int disparo);
float pegaYDisparo (unsigned int disparo);
```

Retornam respectivamente as coordenadas nos eixos *x* e *y* do centro da região afetada pelo disparo com o identificador dado..

```
float pegaRaioDisparo (unsigned int disparo);
```

Retorna o raio da região afetada pelo disparo com o identificador dado.

Ao final, a função `testaDisparos` deve retornar o número total de acertos em alvos – note que este número não é necessariamente igual ao número de alvos atingidos, já que um mesmo alvo pode ser atingido por mais de um disparo.

Função 3 (25 pontos):

```
unsigned short pegaCampo (unsigned short seq, int pos, int nbits);
```

Em implementações de protocolos de rede, drivers de dispositivos, *firmwares*, montadores de linguagem de máquina (*assemblers*), controladores de dispositivos, e emuladores de processadores, é muito comum encontrarmos uma série de números “empacotados” em um mesmo conjunto de bits. Por exemplo, um inteiro de 16 bits poderia ser dividido em campos, da seguinte forma:

Neste exemplo hipotético, os primeiros 3 bits do número seriam o código de um comando, os 12 bits seguintes representariam os valores de 2 parâmetros para o comando, e o último bit seria um código para detecção de erros. Desta forma, a seguinte sequência de 16 bits:

0011000100011001

Seria repartida da seguinte forma:

001 100010 001100 1

Comando:	001	(1 decimal)
Dado1:	100010	(34 decimal)
Dado2:	001100	(12 decimal)
Checksum:	1	(1 decimal)

A função `pegaCampo` recebe como parâmetros uma sequência de 16 bits, a posição de um campo na sequência e o número de bits do campo, e retorna o valor para aquele campo. No exemplo anterior, se a sequência estivesse em uma variável `foo`, `pegaCampo(foo,0,3)` retornaria 1, `pegaCampo(foo,3,6)` retornaria 34, `pegaCampo(foo,9,6)` retornaria 12, e `pegaCampo(foo,15,1)` retornaria 1.