Sugestões para escrever bom código

Laboratório de Algoritmia 1

Laboratórios de Informática 2

Ano Letivo 2023/24

Sumário

1 Introdução

Escrever código é uma tarefa desafiante. No início não sabemos muito bem o que estamos a fazer e é fácil de cometer erros. Este documento pretende dar algumas sugestões.

2 Submissão do projeto

A única maneira de submeter o projeto para o MOOshak é usando um zip. A maneira mais simples de gerar um zip em linux é ir para a consola, para onde está o código e escrever:

```
$ zip -9r *.[hc]
```

Este comando tem a vantagem de só arquivar os ficheiros relevantes. Se vir a `makefile` mais à frente, também está lá um target para criar o zip.

3 Restrições para a submissão

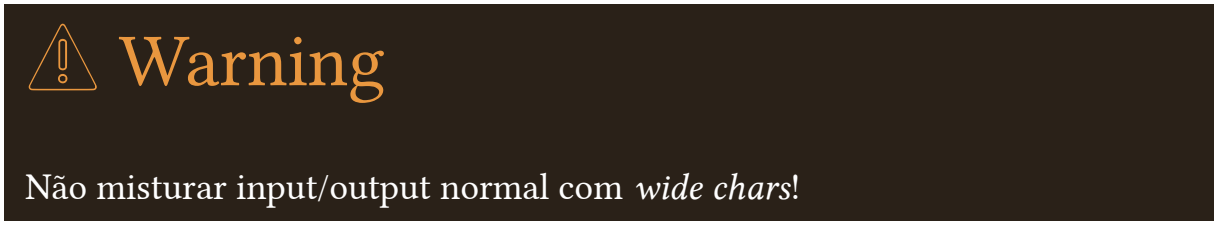
Para forçar a escrita de bom código e *tentar* ensinar alguns hábitos de programação, as submissões do projeto verificam vários requisitos:

test_compilacao Compilar o programa;
test_complexidade Testar se o programa contém funções demasiado grandes (mais do que 30 instruções) e/ou complexas (complexidade ciclomática superior a 20;
test_ficheiros Testar se o programa só contém ficheiros C num local;
test_goto Testar se o programa contém instruções goto;
test_includes Testar se o código inclui ficheiros que também contém código;
test_labels Testar se o código contém labels (para continue, break, gotos, etc);
test_variaveis_globais Testar se o programa contém variáveis globais

O primeiro teste é o normal do MOOshak. Os outros são mais ou menos óbvios pela pequena descrição, excepto o da complexidade.

O teste da complexidade verifica se as funções são pequenas e pouco complexas correndo a ferramenta `pmccabe` e verificando o número de instruções (*statements*) e a complexidade ciclomática (**CC**).

4 Sugestões de código para o projeto



- A maneira como tratam os buffers é **diferente**!
- Não reinventar a roda!
- Ler o manual do wchar.h para ver que funções existem
- Ler todas as secções deste documento com **atenção**!

5 O mais importante de tudo é ...

1. **Pensar** antes de codificar!
2. Código **legível**!

3. Funções **pequenas**!
4. Testar, testar, testar! **Aprende a usar o gdb**!
5. Complexidade baixa.
6. Usar **assert** para garantir os invariantes.

Planeia antes de começares a escrever código.

Vais passar mais tempo a depurar o código do que a escrever. Logo aprende a usar o gdb! Vai ver a secção correspondente e vê os vídeos!

Quando tens bugs ficas mais stressado, logo pensas pior. Se inventares ao escrever o código não vais perceber o que se passa! O teu código deve ser pequeno, simples, legível e falhar o mais rapidamente possível!

Quando estás stressado fazes asneiras. Pára! Vai dar uma volta. Bebe um café!

6 Makefile

Eis um exemplo de uma makefile. A makefile tem alguns targets interessantes:

cards é o target por omissão que é usado quando escrevemos simplesmente make;

complexidade imprime uma tabela com a complexidade das funções do código, desde a mais complexa até à menos complexa;

check Usa o comando **cppcheck** para procurar problemas ou sugerir alterações no código;

codigo.zip Cria o zip para enviar para o MOOshak (talvez faça sentido fazer isto doutra maneira se o código de todas as etapas do projeto estiver na mesma pasta).

```
CFLAGS=-Wall -Wextra -pedantic -O2
OBJS=$(patsubst %c, %o, $(wildcard *.c))

cards: cards.o main.o
  $(CC) $(CFLAGS) -o $@ $^

# Procura funções demasiado complexas
complexidade:
  @echo Only printing when Modified McCabe Cyclomatic Complexity is above 5
  @echo | pmccabe -v
  @pmccabe *.c| sort -nr | awk '{if($$1>5)print}'

# Procura problemas no código
check:
  cppcheck --enable=all --suppress=missingIncludeSystem .

# Esta parte foi gerada usando o comando gcc -MM *.c
cards.o: cards.c cards.h
main.o: main.c cards.h

codigo.zip: $(wildcard *.h) $(wildcard *.c)
  zip -9r $@ $^

clean:
  rm $(OBJS) a.out cards
```

7 Ferramentas úteis

valgrind útil para descobrir erros com a memória
cproto extrai todos os protótipos de ficheiros C
clang-format formata automaticamente código C
cppcheck procura erros comuns no código e dá sugestões de melhoria
pmccabe Calcula algumas medidas de complexidade como:

- o número de linhas de código de cada função
- o número de instruções e
- a complexidade ciclomática

8 Asserts

É importante criar asserts para verificar invariantes no nosso código. As vantagens dos asserts são as seguintes:

- O programa falha o mais cedo possível
- Testam o código através de condições
- Caso as condições não sejam verdade, o código rebenta e é gerado um traceback para a linha onde houve o problema
- Deve-se testar problemas frequentes, como por exemplo:

- Aceder a memória inválida, e.g.;
  - índices negativos de arrays, ou
  - além do maior índice possível
- Argumentos inválidos, e.g.;
  - tentámos eliminar um valor de uma lista vazia
  - tentámos inserir um valor que já existia
  - tentámos apagarr um valor que não existia
- Testar consistência nos dados (e.g., o nº de bits de uma representação de vírgula flutuante tem que ser igual a 1 + E + M)

9 Código legível

É muito importante escrever código legível:

- Funções **pequenas** (que cabem num ecrã)
- Nomes de funções **sugestivos** (que indicam o que fazem)
- Funções auxiliares com nomes **relevantes** (e não aux1, aux2)
- Variáveis com nomes que *indiquem* a sua função (e não a, b, c)
- Escrever funções auxiliares para aceder a campos das estruturas
- Escrever funções auxiliares para testar coisas comuns a várias funções
- Reescreva o seu código para o tornar mais simples utilizando **refactoring** (eis como o pode fazer no CLion)

10 Complexidade

Consulte o artigo da wikipédia sobre a complexidade ciclomática. No fundo, o importante a reter é que, por ordem de importância:

- devemos escrever código legível (veja o início do vídeo **Back to Basics: Debugging in CPP**)
- devemos escrever funções com **poucas linhas**
- devemos minimizar o **nesting** (níveis de instruções condicionais ou ciclos)
- devemos tentar manter o valor da complexidade ciclomática abaixo de 10;

```
$ make complexidade
Only printing when Modified McCabe Cyclomatic Complexity is above 5
Modified McCabe Cyclomatic Complexity
|   Traditional McCabe Cyclomatic Complexity
|       |    # Statements in function
|       |        |   First line of function
|       |        |       |   # lines in function
|       |        |       |       |  filename(definition line number):function
|       |        |       |       |           |
9       9       24     159      29      cards.c(159): suggest_play
8       8        6     267      10      cards.c(267): is_exception
8       8       20     132      26      cards.c(132): get_combination
7       7       16     230      16      cards.c(230): get_all_seqs
6       6        7     278      11      cards.c(278): can_play
6       6        4      17       9      cards.c(17): aux_cmp_hands
```

11 Modularidade

Divida o seu código em vários módulos. Cada módulo deve possuir todas as funções de um dado tipo. Cada módulo deve possuir:

Ficheiro .h que só deve possuir protótipos de funções, definições de tipos que tem que ser conhecidos por outros módulos, macros e constantes que precisam de ser conhecidas por outros módulos. Este tipo de ficheiro não deve possuir código;

Ficheiro .c que deve incluir o ficheiro .h correspondente e definir as funções; este ficheiro deve incluir o ficheiro .h correspondente para garantir que a definição das funções está de acordo com os protótipos.

11.1 Include fences

Quando criamos um ficheiro .h, devemos colocar um *include fence*. Esta construção permite que se possa incluir várias vezes o mesmo ficheiro sem provocar erros por múltipla definição da mesma coisa.

```
#ifndef __FICHEIRO_H__
#define __FICHEIRO_H__
// Aqui coloca o conteúdo (e.g., tipos de dados, protótipos)
#endif
```

11.2 Protótipos

O comando `cproto` permite extrair todos os protótipos de um ficheiro e assim facilita a criação do ficheiro .h correspondente. Os protótipos possuem várias vantagens:

- Evitam erros de conversão implítica;
- Evitam problemas de definir uma função depois dela ser utilizada;
- Permite utilizar recursividade dupla;

- Permitem que outros módulos possam utilizar funções sem problemas de declaração implícita.

11.3 Estrutura de dados

- Pense com cuidado antes de criar a estrutura de dados;
- Leia o enunciado e pense no que vai ser necessário;
- Crie uma estrutura que lhe permita aceder rapidamente a todo o tipo de informação que precisa;
- Use estruturas para lhe permitir agrupar a informação;
- Use *enum* em vez de criar várias constantes, também tem a vantagem de ajudar quando inspecionar essas variáveis no `gdb`;
- O *enum* permite, se quisermos, atribuir inteiros a alguns valores;
- Use apontadores para passar a informação entre funções;
- Use apontadores *const* quando a função não deve poder modificar a informação mas só aceder;
- Use funções para aceder e modificar a estrutura de dados em vez de aceder diretamente à informação;
- Assim se modificar a informação só terá que alterar as funções que a alteram diretamente ao invés de alterar todas as funções do seu código.

12 Debugging

Eis alguns links de tutoriais do GDB, podes clicar no QR code ou usar o teu telemóvel:

Tutorial do gdb	Back to Basics: Debugging in CPP	Cool new stuff in GDB

13 Erros com a memória

Os erros típicos de memória são normalmente os mais difíceis de encontrar em C. Eles estão sempre relacionados com más práticas de programação:

- Usar endereços da *stack* que deixaram de estar reservados em vez de usar memória da *heap*;
- Não alocar memória suficiente;
- Não libertar a memória (*memory bloat*);
- Não inicializar a memória;
- Libertar a memória mais do que uma vez;
- Corromper a *stack* ou a *heap* ao escrever fora da área de memória permitida:
  - Índices negativos de arrays;
  - Índices para além do tamanho alocado.

13.1 Boas práticas

- Inicializar sempre as áreas de memória!
- Usar o **calloc** em vez do **malloc** visto que este inicializa toda a memória a zero;
- Contar bem o espaço e alocar espaço para os terminadores de string;
- Usar o **sizeof** para calcular quanto espaço se precisa.
- Não esquecer de libertar **toda** a memória!
- Aceder aos dados através de poucas funções;
- Colocar *asserts* para verificar invariantes;
- Usar o valgrind.

13.2 Exemplo

Eis um exemplo da declaração e criação de uma lista ligada e da função de inserção de um elemento à cabeça da lista:

```
typedef struct cel {
  void *elem;
  struct cell *prox;
} CEL;

typedef struct lista {
  CEL *cabeca;
  int tamanho;
} LISTA;

LISTA *criar_lista() {
  // Já inicializou a cabeça e o tamanho a zero!
  LISTA *L = (LISTA *) calloc(1, sizeof(LISTA));
  // Safety check, temos memória?
  assert(L != NULL);
  return L;
}
```

```c
void inserir_elem(LISTA *L, void *elem) {
  // O prox já é NULL!
  CEL *C = (CEL *) calloc(1, sizeof(CEL));
  assert(C != NULL); // Safety check
  /*
   * Paranoia!
   * Não aceitamos a inserção de um elemento
   * sem nada (pode não fazer sentido mas é um sanity check extra!)
  */
  assert(elem != NULL);
  C->elem = elem;
  if(L->cabeca == NULL) {
    L->cabeca = C; // Não foi preciso dizer que C->prox = NULL
  } else {
    C->prox = L->cabeca;
    L->cabeca = C;
  }
  L->tamanho++;
}
```

Se quisermos alocar um array dinâmico e inicializar a área a zero, também podemos usar o *calloc*

```c
  TIPO *array = (TIPO *) calloc(tamanho, sizeof(TIPO));
```

Se quisermos voltar a inicializar o array mais tarde:

```c
  bzero(array, sizeof(array));
```

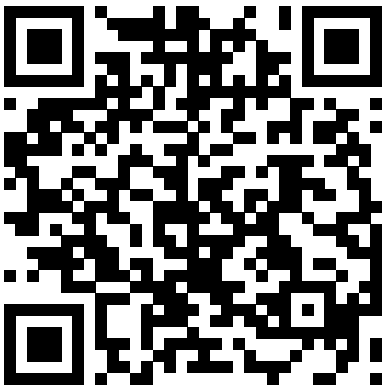
Se soubermos o tamanho de um array e lhe quisermos aceder, não seria má ideia algo do género:

```c
  void invariante(int indice, unsigned tamanho_maximo) {
    assert(indice >= 0);
    assert(indice < tamanho_maximo);
  }
  TIPO aceder(TIPO *array, int indice) {
    invariante(indice, tam);
    return array[indice];
  }
  TIPO modificar(TIPO *array, int indice, TIPO valor) {
    invariante(indice, tam);
    return array[indice] = valor;
  }
```

Poderíamos fazer uma única função usando varargs. Deixo o exercício a quem o quiser tentar.

13.3 Valgrind

O **valgrind** é uma ferramenta extremamente útil para descobrir problemas de memória. Sugere-se vivamente que veja este vídeo sobre como usar o **valgrind** e ao mesmo tempo ver erros típicos de má utilização de memória:



Eis um tutorial em texto que ajuda a perceber como funciona, para quem não tiver paciência para o vídeo: