ISEL | DEETC

Programação de Sistemas Computacionais

4ª Série de Exercícios

Semestre de Inverno de 2009/2010

Autores:
30896 – Ricardo Canto
31401 – Nuno Cancelo
33595 – Nuno Sousa

Enunciado

ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Licenciatura em Engenharia Informática e de Computadores

Programação em Sistemas Computacionais

Quarta série de exercícios – Inverno de 2009/10

Entregue o código desenvolvido em linguagem C, devidamente indentado e comentado, e um relatório com a descrição das soluções. Inclua na entrega o *makefile* que permita gerar os ficheiros executáveis a partir do código fonte. O relatório deverá ser um guia para a compreensão do código desenvolvido e não uma mera tradução deste para língua natural. Contacte o docente se tiver dúvidas. Encoraja-se a discussão de problemas e soluções com colegas de outros grupos, mas recorda-se que a partilha directa de soluções leva, no mínimo, à anulação das entregas de todos os envolvidos.

O calendário de exames das unidades curriculares (UC) de um curso deve cumprir um conjunto de regras. Por exemplo: deve ser garantido um intervalo mínimo de dias entre as provas de primeira e segunda época de uma UC; as provas de duas UC do mesmo semestre devem ter entre si um intervalo mínimo de dias; se, numa das épocas, as provas de duas UC de semestres próximos coincidem, então na outra época essa coincidência não se deve repetir.

Para facilitar a validação destes mapas foi desenvolvido um programa em Java, cujas fontes se encontram em anexo, que valida um calendário de exames segundo um conjunto configurável de regras. Algumas destas regras aplicam-se às datas de cada UC individualmente, enquanto que as outras se aplicam a cada par possível de UC. O programa recebe na linha de comando o nome de dois ficheiros de texto, um com informação sobre as UC do curso (*ProgramCourses*) e outro com o calendário a avaliar (*Exams*). Exemplos destes ficheiros também se encontram em anexo.

Os ficheiros descritos têm uma linha para cada UC com elementos separados por uma barra vertical. Por exemplo: a linha "`PI|B|16|AVE|RCp;LS`" no primeiro ficheiro indica que PI é do tipo B (oBrigatória), funciona apenas no 5º semestre (*bitmap* com bit a 1 nos semestres em que funciona), tem dependência forte de AVE e fraca de RCp e LS; a linha "`PI|8|22`" no segundo ficheiro indica que PI tem a primeira época no dia 8 do mapa e a segunda época no dia 22 do mapa (valores entre 0 e N).

1. Inspirando-se nas fontes do programa em Java, escreva um programa em C com os mesmos objectivos. A estrutura geral do programa em C deve seguir a do programa em Java, garantindo-se a mesma facilidade de alteração:
   a) Deve existir, pelo menos, um módulo em C correspondente a cada módulo em Java.
   b) O módulo para carregamento de informação (*DataLoader*) é usado para ler todos os ficheiros de texto.
   c) As regras a aplicar são indicadas nos ficheiros "*validators1.txt*" e "*validators2.txt*"
   d) A alteração, remoção ou adição de regras não deve obrigar a gerar uma nova versão do ficheiro executável que contém o programa de validação.
2. Acrescente um validador para garantir uma distância mínima *mi* entre UC de semestres consecutivos, excepto se forem dependências fortes, caso em que a distância mínima será de *md*, com *md* <= *mi*.
3. Acrescente outro validador que faça sentido e que use a informação sobre as disciplinas serem ou não obrigatórias

Data limite de entrega: 11 de Janeiro de 2010

Bom trabalho!

NOTA: O calendário de exames do DEETC contém, neste semestre, marcações de provas para cerca de oito dezenas de UC em que aproximadamente metade delas funcionam em mais do que um dos cinco cursos do departamento (LEIC, LEETC, LERCM, MEIC e MEET). Se pretender validar o calendário de exames real da LEIC no contexto do DEETC, contacte o docente.

Exercício 1

Pela análise do enunciado e como é pedido na alínea a) ficou claro que o trabalho fosse realizado por módulos e estruturado como se tratasse de classes em Java.

Na nossa implementação optámos por tentar ser fiéis ao código Java apresentado como exemplo e "traduzir" tanto quanto possível para c, tendo em conta as limitações do mesmo.

Encontrámos algumas dificuldades iniciais como código Java uma vez que algumas soluções apresentadas no mesmo nos eram completamente desconhecidas. Após a aula prática ficámos a perceber melhor como todo este conjunto de ficheiros funciona para estruturarmos o programa utilizando o c.

O ficheiro DataLoader irá ser utilizado por todos os outros módulos para ler os ficheiros de texto de entrada e criar as estruturas necessárias ao processamento dos dados. Sendo assim tivemos de proceder a algumas alterações após a criação da primeira versão de modo a que os métodos passasem a receber e a retornar ponteiros void. Assim manteve-se a possibilidade deste módulo ser usado para construir várias estruturas de dados bem diferenciadas.

Osficheiro ProgramCourse e Exam procedem a construção das estruturas de Unidades Curriculares (UC) e Exames das mesmas.

Em qualquer um dos objectos das "classes" mencionadas foi nossa preocupação que pudessem ser usados por qualquer outro módulo por isso a todos os objectos foi adicionada uma tabela de métodos virtuais que incluem pelo menos o destrutor do objecto. A sua criação será sempre efectuada com a função <nome_do_objecto>_ctor().

As funções *ctor* efectuam a alocação dinâmica de memória, e inicializam a tabela de métodos virtuais com os métodos respectivos. Optámos por não separar o construtor do método para inicializar os dados uma vez que esses métodos não existiam nas classes Java e não se tornava necessário alterar os dados iniciais destes objectos.

No caso do DataLoader, embora o destrutor, o *build* e o *loadfrom* sejam assignados na construção os módulos que os pretendem usar são obrigados a atribuir a estes métodos virtuais métodos desenvolvidos para as especificidades de cada um. Uma vez que implementam a interface DataLoader são obrigados a implementar os métodos abstractos *newInstance* e *newArray*.

Durante o desenvolvimentos dos módulos ProgramCourse e Exam chegámos à conclusão que deveríamos arranjar uma solução para a criação de arrays destes objectos uma vez que percebemos que iria ser necessário. Assim a solução encontrada passou por criar uma estrutura que contem uma tabela de métodos virtuais com o seu destrutor, um ponteiro para um array de ponteiros (para os

objectos do array) e um inteiro que contém o nº de elementos presentes nesse array. A outra solução possível seria ter um ponteiro a NULL no final mas nesse, mas uma vez que o custo em termos de memória é idêntico (4 bytes para o ponteiro ou 4 bytes para o inteiro) assim é mais fácil saber rapidamente o nº total de elementos no array sem ter de o percorrer todo.

Para facilitar a utilização desta estrutura em todos os módulos e para facilidade de leitura definimos a estrutura no DataLoader e depois criámos alias mais indicados a cada um dos módulos.

Para utilização da função *parseInt* (que faz, como o nome indica a passagem de String para inteiros) e da função xstrtrim (que efectua o trim de uma String retirando os espaços no início e fim da mesma e os caracteres de fim de linha) e outros alias para facilitar a leitura do código, criámos o ficheiro myLib onde se encontram as definições e/ou implementações destas funções e objectos.

ficheiro: DataLoader.h

```
typedef struct DataLoader_vtable{
        void                (*dtor)             (dldr *this);
        ItemType*           (*newArray)         (int numEntries);
        ItemType            (*newInstance)      (String* elems, int nbr);
        arrayItem*          (*loadFrom)         (dldr *this, arrayItem* arr, String filename);
        arrayItem*          (*build)            (dldr *this, arrayItem* arr, int n);
} dldrMethods;

typedef struct ItemTypeArray_vtable{
        void                (*dtor)             (arrayItem* this);
} arrayMethods;

struct DataLoader{
        dldrMethods *vptr;
        FILE* input;
};

struct ItemTypeArray{
        arrayMethods *vptr;
        ItemType* array;
        int size;
};

#define length(this)                    (((const arrayItem * const)(this))->size)
#define getArray(this)                  (this->array)
#define getArrayPos(var,i)              (var->array[i])

dldr*           DataLoader_ctor             ();
void            DataLoader_dtor             (dldr * this);
arrayItem*      loadFrom                    (dldr *this, arrayItem* arr, String filename);
arrayItem*      build                       (dldr *this, arrayItem* arr, int n);
```

Podemos verificar no código acima o tipo de estrutura que implementámos em todo o trabalho:

vtable com os métodos virtuais do módulo

estruturas com as "variáveis de instância" e um ponteiro (vptr) para a tablea de métodos virtuais

definição de macros para facilitar o acesso aos vários campos das estruturas

Para se poder usar o array com qualquer tipo de dados optámos por criar uma estrutura que na realidade tem um ponteiro para um array de dados. O tipo escolhido foi void para permitir depois facilmente o cast para o tipo "real dos dados".

A construção do ficheiro ProgramCourse e Exam foram as que mais trabalho nos

deram por serem as primeiras e para garantir que o DataLoader irá funcionar com qualquer outro módulo que se pretenda. Aqui foi necessário algum tempo para perceber bem como passar os argumentos, quais eram necessários e como processar correctamente a informação recebida do ficheiro.

O código é bastante fácil de perceber uma vez que se tentou que fosse fiel ao original em Java, acrescentando apenas os construtores, destrutores virtuais de objectos e de arrays de objectos.

Nos métodos *loadfrom* tentámos utilizar o qsort da biblioteca do c mas não estava a funcionar e não conseguimos resolver o problema. Do mesmo modo, em vez do bsearch no *indexOf* efectuamos uma pesquisa linear.

O módulo *ExamValidator* foi mais fácil de desnvolver depois do trabalho preparatório mas por outro lado obrigou-nos a estudar bem o funcionamento das bibliotecas dinâmicas para efectuar o load dos validadores passados nos ficheiros txt. Uma vez que a estrutura de arrays criada funciona com void pointers criámos apenas uma estrutura de arrays para acomodar os tipos OneExamValidator e TwoExamValidator. Depois criámos macros no ficheiro .h para retirar deste objecto o array ou o objectos que está na posição x do array fazendo logo o cast para um dos 2 tipos de objecto.
Na função *newInstance* quando chamamos a função *setArgs* optámos por efectuar o cast para o tipo base BaseExamValidator uma vez que tanto podemos utilizar um dos outros dois tipos derivados.

Uma vez bem definidos e estruturados todos estes módulos a adaptação dos módulos com os validadores foi bastante fácil. Estes tinham sido os primeiros a ser "traduzidos" do Java para o C mas acabaram por não ser necessárias muitas alterações ao trabalho efectuado inicialmente. Uma vez que foi tudo preparado para que as estruturas do C fossem utilizadas como se objectos Java se tratassem ficou muito mais fácil a escrita do código dos validadores.

Numa tentativa de implementarmos algo parecido com as excepções utilizamos um módulo Exception.h retirado do documento "Implementing Exceptions in C" de Eric. S. Roberts com definições para o TRY, CATCH e THROW das excepções de Java mas neste caso utilizando jumps (jmp). Infelizmente não tivemos tempo de estudar o funcionamento deste módulo para podermos desenvolver este conceito e concretizar as excepções para todos os módulos do trabalho. Assim, foi apenas utilizado nos locais onde usamos a função *parseInt.*

Exercício 2 e 3

Para podermos executar os validadores pedidos nestes dois exercícios foi necessário "voltar atrás" e alterar a estrutura inicial de ProgramCourse para

acrescentar as dependências fracas e fortes das UCs. Para tal foi necessário criar uma função auxiliar chamada pelo construtor do objecto para processar as UCs dependentes devido a algumas UCs terem mais do que uma dependência. Assim o campo das UCs dependentes (fortes e fracas) são arrays de Strings (ou char*) como preferirmos. Neste caso optámos por manter uma posição a mais do que o máximo permitido para garantir que existe sempre um ponteiro com 0 no final permitindo correr estes arrays sem sair dos mesmos e sem termos um contador.
A análise das UCs que ocorrem em semestres consecutivos foi efectuada recorrendo a shifts de um bit para a direita e esquerda. Ao efectuar o AND bit a bit com o inteiro a comparar apenas iremos receber um valor diferente de 0 se as UCs funcionarem em semestres consecutivos. No exercício 3 adaptámos esta função para verificar também UCs do mesmo semestre e validar se existem UCs do mesmo semestre ou semestres consecutivos que sejam obrigatórias com exames no mesmo dia.
No filtro do Exercício 2 foi criado um método auxiliar semelhante ao utilizado para processar as UCs dependentes no módulo ProgramCourse para processar os argumentos com as duas distancias mínimas passadas.

A produção de novos filtros é bastante simples devido à estrutura utilizada para processar os mesmos. Pouco mais é necessário desenvolver à parte do método *isValid* onde são aplicadas as regras desejadas.

Este trabalho foi um grande desafio para nós devido à pouca prática na utilização de métodos virtuais, criação de "objectos" e utilização de características que a linguagem c não tem como a herança. Se acrescentarmos a isto o facto da linguagem c ter algumas especificidades e ser muito "pobre" a reportar os erros de execução tornou-se um grande desafio perceber alguns dos erros que fomos obtendo durante o desenvolvimento.

Penso que o objectivo do trabalho foi cumprido tornando-se mais claro agora como funcionam realmente os objectos nas linguagens de mais alto nível e pensamos que esta abordagem é muito importante para melhorar o desenvolvimento de código noutras linguagens tornando-o mais eficaz e "poupado" na utilização dos recursos disponíveis.

Juntamos em anexo todos os ficheiros fonte do trabalho, assim como o Makefile que permite a compilação dos mesmos.