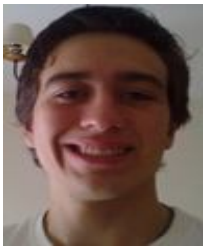
Universidade do Minho

LEI 2º Ano – 2º Semestre

LI3 – Projeto em C

Gesthiper

Grupo 21



Filipe de Oliveira - a67686 Pedro Cunha – a67677 Stéphane Fernandes – a67681

Conteúdo

1. Introdução

1.1. Apresentação do caso de estudo

Foi-nos proposto a elaboração de uma aplicação na linguagem C que permita a Gestão de Produtos, Clientes e Compras de um Hipermercado (seguindo princípios de modularidade e encapsulamento), aplicando os conhecimentos adquiridos nas Unidade Curriculares de Programação Imperativa, Algoritmos e Complexidade e Arquitetura de Computadores.

O programa deverá ser capaz de ler três ficheiros.txt e armazenar eficazmente os dados em memória, por forma a realizar as operações pretendidas com os mesmos, após uma breve análise dos três ficheiros recebidos (FichProdutos.txt, FichClientes.txt e Compras.txt) chegou-se aos seguintes pontos:

- **FichProdutos.txt** cada linha representa o código de um produto vendável no hipermercado, cada código é formado por duas letras maiúsculas e quatro dígitos.
- **FichClientes.txt** cada linha representa o código de um cliente identificado do hipermercado, cada código é formado por duas letras maiúsculas e três dígitos;
- **FichCompras.txt** cada linha representa o registo de uma compra efetuada no hipermercado, nestas linhas está representado um código de produto, um preço unitário, o número de unidades compradas, o tipo de compra (N - Normal ou P - Promoção), o código de cliente e o mês da compra (1..12);

Ao utilizador é disponibilizado um sistema de menus para escolher as queries que pretende consultar.

Objetivos:

→Com a realização deste trabalho é pretendida a implementação de código modular e encapsulado, utilizando Tipos Abstratos de Dados de forma a desenvolver uma aplicação segura.

→Desenvolvimento de estruturas de auxílio a processamento de grandes quantidades de dados, seguindo as normas delineadas pela equipa docente.

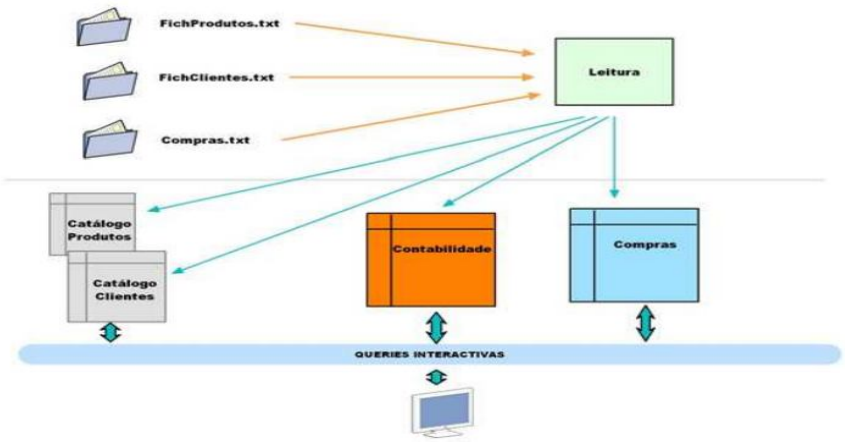


Figura 1 – Esquema da Aplicação

2. Módulos de Dados

2.1. Módulo de Catálogo

2.1.1. Estrutura de dados do Módulo

No ficheiro Catalog.h definiu-se o tipo abstrato de dados:

```
typedefstructCatalog_ *Catalog;
typedefstructTreeCatNode *TreeCat;
```

No ficheiro Catalog.c encontra-se definidaa estrutura de dados

```
struct Catalog_ {
    /*array de árvores (ponteiros) para cada letra*/
TreeCat indice[ALFABETO];
int num codigos[ALFABETO];
};
structTreeCatNode{
     Codigo codigo;
     struct TreeCatNode *left, *right;
};
```

Funções privadas ao módulo

`static int hashFunc(Codigo codigo)`– função que determina a posição onde vai inserir o código;

2.1.2. Desenho da estrutura de dados

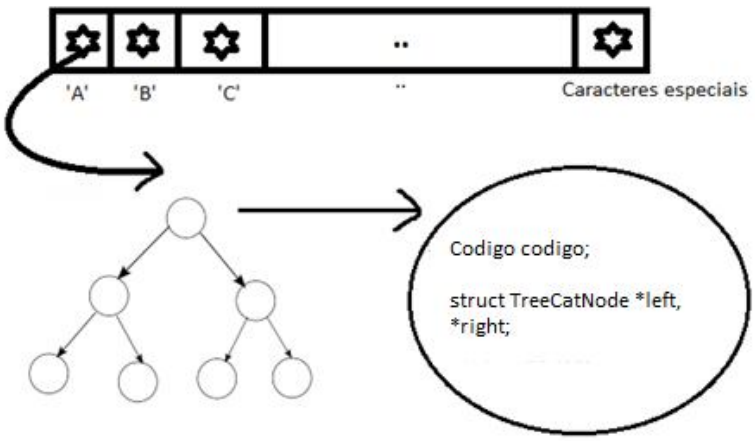


Figura 2 – Estrutura de dados do Módulo Catálogo

2.1.3. Funções de inicialização

Catalog newCat() - função que inicializa um novo catálogo.

TreeCat newNode(Codigo c) –função que recebe um código e cria uma nova árvore com esse código.

2.1.4. Funções de Inserção

`Catalog Cat_insert(Catalogindex, char *codigo)-` Função que recebe um código e um catálogo e usa a função de hash para calcular a posição onde vai inserir esse código.

`TreeCat TreeCat_insert(TreeCatt,Codigo c)–` função que insere o Cliente/Produto recebido como argumento numa árvore também recebida como argumento, caso este ainda não exista.

2.1.5. Funções que libertam memória

`void Cat_dispose(Catalogindex) –` função que recebe um catalogo e para cada uma das posições do array chama a função `disposeTreeCat`responsável por libertar a memória ocupada por a árvore presente nessa posição do array, por fim a função liberta a memória ocupada por cada índice do array.

2.1.6. Outras funções

`int Cat_getNumCodigos(Catalogindex,Codigo c)–` função que recebe um catálogo e retorna o número de Códigos existentes no Catálogo.

`Boolean TreeCat_search(TreeCat t, Codigo c)` **–**função que recebe uma árvore e um código e vai procurar esse código na árvore recebido retornando um Bool.

`Codigo ArrayCat_getTreeToArray(Catalog c, Codigocodigo)–` Função que recebe um catálogo e um código e retorna um CodigoArray com os códigos começados por essa letra ordenados alfabeticamente.

2.2. Módulo de Contabilidade

2.2.1. Estrutura de dados do Módulo

No ficheiro Contabilidade.h definiu-se o tipo abstrato de dados:

```
typedef struct contnode_* Contab;
```

No ficheiro Contabilidade.c encontra-se definidaa estrutura de dados

```
struct contnode_ {

Codigo codigo; // char* que representa o código do
cliente/produto

int vendasN[12], vendasP[12];

int NVendN[12], NVendP[12];

floa tfaturaN[12], faturaP[12];

struct contnode_ *left, *right;

};
```

`Vendas(N/P)` -Quantidade de produtos vendidos em modo (N/P) para cada um dos meses;

`NVend(N/P)`– array com a informação do número de vendas em modo(N/P) para cada um dos meses;

`Fatura(N/P)` – valor total faturado em modo (N/P) para cada um dos meses;

2.2.2. Desenho da estrutura de dados

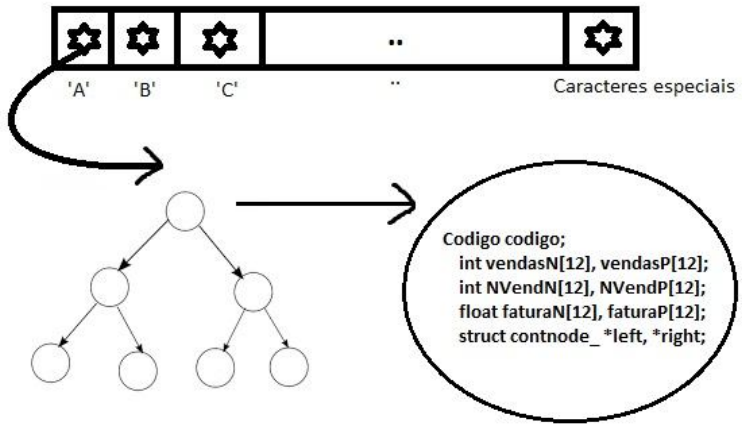


Figura 3- Estrutura de dados do módulo de Contabilidade

2.2.3. Funções de inicialização

`CTree newCT()` – Esta função inicializa um array de 27 posições, em que em cada posição existe um apontador para uma árvore a NULL.

`Contab newCont(Codigo codigo)` – Esta função inicializa cada parâmetro de cada nodo da árvore.

2.2.4. Funções de Inserção

`CTree CT_insert(CTree ct, Codigo codigo)`– Função que recebe uma árvore e um código e utilizando uma função de hash calcula a posição onde vai inserir esse código.

`Contab Cont_insert(Contab cont, Codigo codigo)`–Função que recebe um array de 27 posições e um código e vai inserir esse código na árvore.

`CTree CT_insereCompra(CTree ct, Codigo codigo, char modo, int qtd, float valor, int mes)`– Função que calcula a posição onde vai inserir a compra que recebe como parâmetro.

`Contab Cont_insereCompra(Contab c, Codigo codigo, char modo,int qtd, float valor, int mes)` – Função que recebe um array de 27 posições e uma compra, inserindo essa compra na posição do array previamente calculado, se nessa posição a árvore estiver vazia esse passa a ser a única compra da árvore, caso contrário vai verificar onde vai inserir (esquerda ou direita).

2.2.5. Funções que libertam memória

`void CT_dispose(CTree nodo):` Função que para cada uma das 27 posições do array vai fazer o free do apontador para a árvore.

`void Cont_dispose(Contab nodo):` Função que para cada nodo das árvores vai libertar a memória ocupada para canto.

2.2.6. Outras funções

`float Cont_getFaturacao`*`(Normal/Promo)`* `(Contab c, Codigo codigo, int mes)` – Função que recebe o array da contabilidade, um código e um mês e vai procurar esse código na árvore, se encontrar o código devolve a faturação(**Normal/Promoção**) no mês dado, caso contrário devolve -1.

`int Cont_getVendas`*`(Normal/Promo)`* `_(Contab c, Codigo codigo, int mes)` - Função que recebe o array da contabilidade, um código e um mês e vai procurar esse código na árvore, se encontrar o código devolve q quantidade de vendas (**Normal/Promoção**) no mês dado, caso contrário devolve -1.

`int Cont_getNVendas`*`(Normal/Promo)`* `_( (Contab c, Codigo codigo, int mes)` -Função que recebe o array da contabilidade, um código e um mês e vai procurar esse código na árvore, se encontrar o código devolve o numero de vendas (**Normal/Promoção**) no mês dado, caso contrário devolve -1.

`CodigoArray CT_produtosNaoComprados(CTree ct):` Função que cria um novo array de códigos e para cada posição do array da contabilidade chama a função abaixo descrita.

`CodigoArray Cont_insereProdutosNaoComprados(Contab ct, CodigoArray ca):` Função que em cada uma das árvores do array de contabilidade verifica os códigos dos produtos não comprados e insere estes num código array.

2.3. Módulo de Compras

2.3.1. Estrutura de dados do Módulo

No ficheiro "Compras.h" definiu-se o tipo abstrato de dados,

`typedefstruct produto* Produto; //` estrutura de cada nodo da árvore de produtos

`typedefstruct cliente* Cliente;` // estrutura de cada nodo da árvore de clientes

`typedefstructcomprasDB* ComprasDB; //`estrutura que contém as 3 árvores

No ficheiro Compras.c encontra-se definidas estrutura de dados:

```
struct cliente {
Codigo codigo;
int compraMes[12];
struct simpleProd* prodComprados;
int nCompras;
struct cliente* left,*right;
};

struct produto {
Codigocodigo;
struct simpleCli* cliCompradores;
int nVezesComprado;
int qtdComprada;
int compradoMes[12];
struct produto* left,*right;
};

structcomprasDB {
Cliente clientes;
Produto produtos;
};
struct simpleProd {
    Codigo codigo;
    int mes;
    int qtdCompradaTotal;
    int qtdCompN, qtdCompP;
    float valorTotal;
    float valorN, valorP;
    struct simpleProd *left, *right;
};

struct simpleProd {
    Codigo codigo;
    int mes;
    int qtdCompradaTotal;
    int qtdCompN, qtdCompP;
    float valorTotal;
    float valorN, valorP;
    struct simpleProd *left, *right;
};
```

2.3.2. Desenho da estrutura de dados

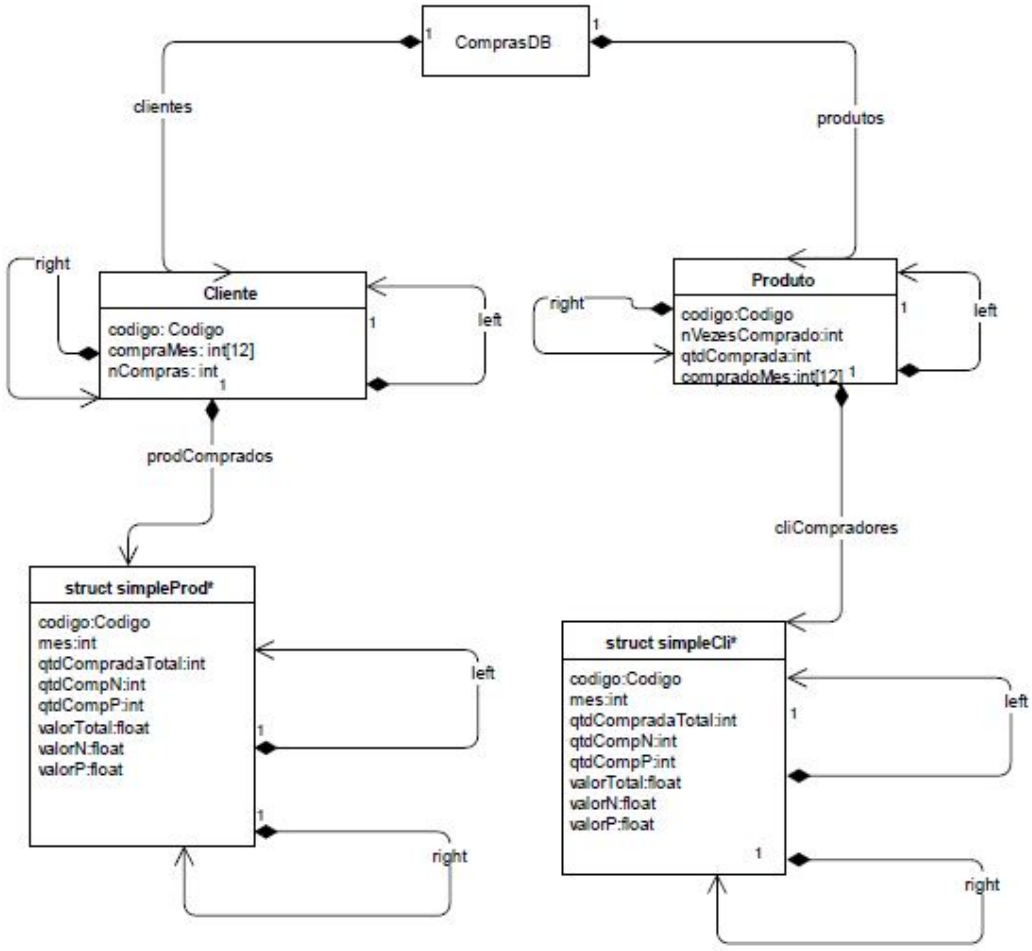


Figura 4- Representação da estrutura de dados do módulo Compras

2.3.3. Funções de inicialização

`Produto newProd(Codigo codigo)-` Função que recebe como argumento um código e aloca memória para uma estrutura de dados do tipo de dados Produto e nessa estrutura vai inserir o código recebido.

`Cliente newCli(Codigo codigo) -` Função que recebe como argumento um código e aloca memória para uma estrutura de dados do tipo de dados Cliente e nessa estrutura vai inserir o código recebido.

`ComprasDB newCDB()`–Função que aloca memória para uma estrutura de dados do tipo ComprasDB.

2.3.4. Funções de Inserção

`Cliente ClienT_insert(Cliente ct, Codigo codigoC) -` Função que recebe um código de cliente e vai inseri-lo na árvore de dados do tipo Cliente.

`Produto ProdT_insert(Produto pt, Codigo codigoP) -` Função que recebe um código de produto e vai inseri-lo na árvore de dados do tipo Produto.

```
ComprasDB CDB_insertCliente(ComprasDB cdb, Codigo codigoC);
```

```
ComprasDB CDB_insertProduto(ComprasDB cdb, Codigo codigoP);
```

```
ComprasDB CDB_registerSale(ComprasDB cdb, Codigo codigoP,
float valor, int qtd, char modo, Codigo codigoC, int mes);
```

2.3.5. Funções que libertam memória

`voidClienT_dispose(Cliente ct)-` Função que liberta a memória ocupada árvore de dados Cliente.

`voidCompT_dispose(Compra ct)-`Função que liberta a memória ocupada árvore de dados Compra.

`voidProdT_dispose(Produto pt)-`Função que liberta a memória ocupada árvore de dados Produto.

`voidCDB_dispose(ComprasDB cdb)-` Função que utiliza as três funções anteriores para cada um dos elementos da estrutura ComprasDB e por fim faz o free dessa estrutura.

2.3.6. Outras funções

`Produto ProdT_updateProd(Produto pt, Codigo codigoP, int qtd, float valor, char modo, Codigo codigoC, int mes)-`Função que recebe uma árvore do tipo Produto e uma compra e vai atualizar a árvore de Produtos, atualizando as variáveis conforme os valores recebidos como argumento.

`Cliente ClienT_updateCli(Cliente ct, Codigo codigoP, int qtd, float valor, char modo, Codigo codigoC, int mes)-`Função que recebe uma árvore do tipo Cliente e uma compra e vai atualizar a árvore de Clientes, atualizando as variáveis de acordo com os valores recebidos como argumento.

`Codigo ArrayProd_getCliCompradores(Produto p)-` Função que retorna os códigos de todos os clientes que compraram algum produto.

`int Prod_getVezesComprado(Produto p, int mes)`-função que retorna o numero de vezes que um dado produto foi comprado num mês.

`int Prod_getQuantidadeComprada(Produto p)`-função que retorna a quantidade comprada de um dado produto.

3. Interface com Utilizador e opções de navegação

Para que o utilizador possa navegar em queries que requerem resultados em forma de lista foi necessário criar o seguinte método de navegação:

`Apresenta 10 elementos;`

`Pergunta se pretende Continuar (Sim ou Não);`

`Caso já se esteja na 2ª, ou superior, "página" pergunta se pretende ir para a página anterior.`

`Baseado na opção realiza função desejada.`

A seguir encontra-se o ciclo de execução do programa desenvolvido (Antes deste menu o utilizador terá de inserir os nomes dos ficheiros de fonte):

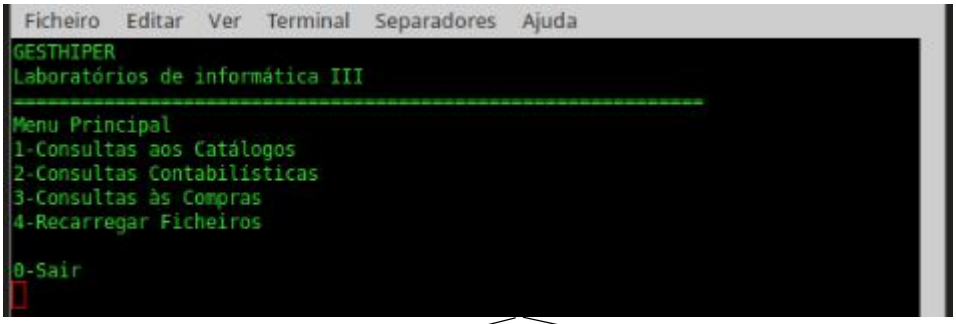


Figura 5 – Menu Principal

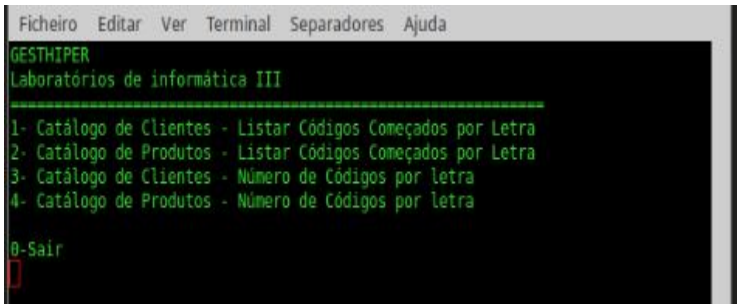


Figura 7- Menu_Consultas aos Catálogos

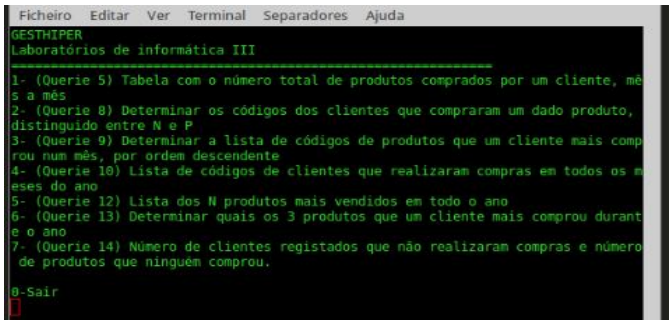


Figura 6 – Menu_Consultas às Compras

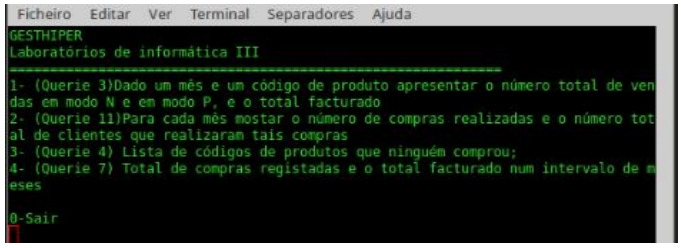


Figura 8 – Menu_Consultas Contabilísticas

4. Testes de Performance

4.1. Código utilizado nos testes

```
#include<time.h>
.
.
.
time_tstart, end;
.
.
start=time(NULL);
//Função a Chamar
end=time(NULL);
.
.
printf("Realizado em %f segundos\n",difftime(end,start));
```

4.2. Resultados e análise

Os resultados obtidos foram os seguintes, realizados em 2 máquinas diferentes:

Máquina 1: Intel® Core™ i5 CPU M480 @ 2.67GHz, Ubuntu 14.04 LTS Nativo

	Leitura e envio para os módulos	**Query 8**
FichClientes.txt	0.0 Segundos	
FichProdutos.txt	2.0 Segundos	
Compras.txt	13.0 Segundos	5 ms
Compras1.txt	28.0 Segundos	21 ms
Compras3.txt	88.0 Segundos	4 ms

- Para a query 8 foi utilizado o valor IE5572.

Máquina 2: Intel® Pentium™ Dual CPU T3200 @ 2.00 GHz, Xubuntu 12.04 LTS, a correr em Virtual Box

	Tempo
FichClientes.txt	0.0 Segundos
FichProdutos.txt	6.0 Segundos
Compras.txt	30.0 Segundos
Compras1.txt	62.0 Segundos
Compras3.txt	216.0 Segundos

Tempo de espera total:

Compras ≈36 segundos

Compras1≈68 segundos

Compras 3≈222 segundos

Devido a problemas na realização dos testes não é possível apresentar mais resultados.

Com os dados recolhidos acreditamos que há melhorias a fazer nas funções de inserção e registo dos módulos de Contabilidade e Compras, mas priorizando o segundo.

5. Makefile

5.1. Apresentação da Makefile

```
CC:=gcc

CFLAGS:=-ansi -O2 -Wall -pedantic

TARGET:=GESTHIPER

all: $(TARGET)

$(TARGET): Main.o Catalog.o  Contabilidade.o Compras.o EstruturasAux.o
CusTypes.o

	$(CC) $(CFLAGS) $^ -o $@

Main.o: Main.c

	$(CC) $(CFLAGS) -c $^

Contabilidade.o: Contabilidade.c

	$(CC) $(CFLAGS) -c $^

Catalog.o: Catalog.c

	$(CC) $(CFLAGS) -c $^

Compras.o: Compras.c

	$(CC) $(CFLAGS) -c $^

EstruturasAux.o: EstruturasAux.c

	$(CC) $(CFLAGS) -c $^

CusTypes.o: CusTypes.c

	$(CC) $(CFLAGS) -c $^

clean:

	@$(RM) *.o

	@echo Limpo
```

5.2. Grafo de dependências da Makefile

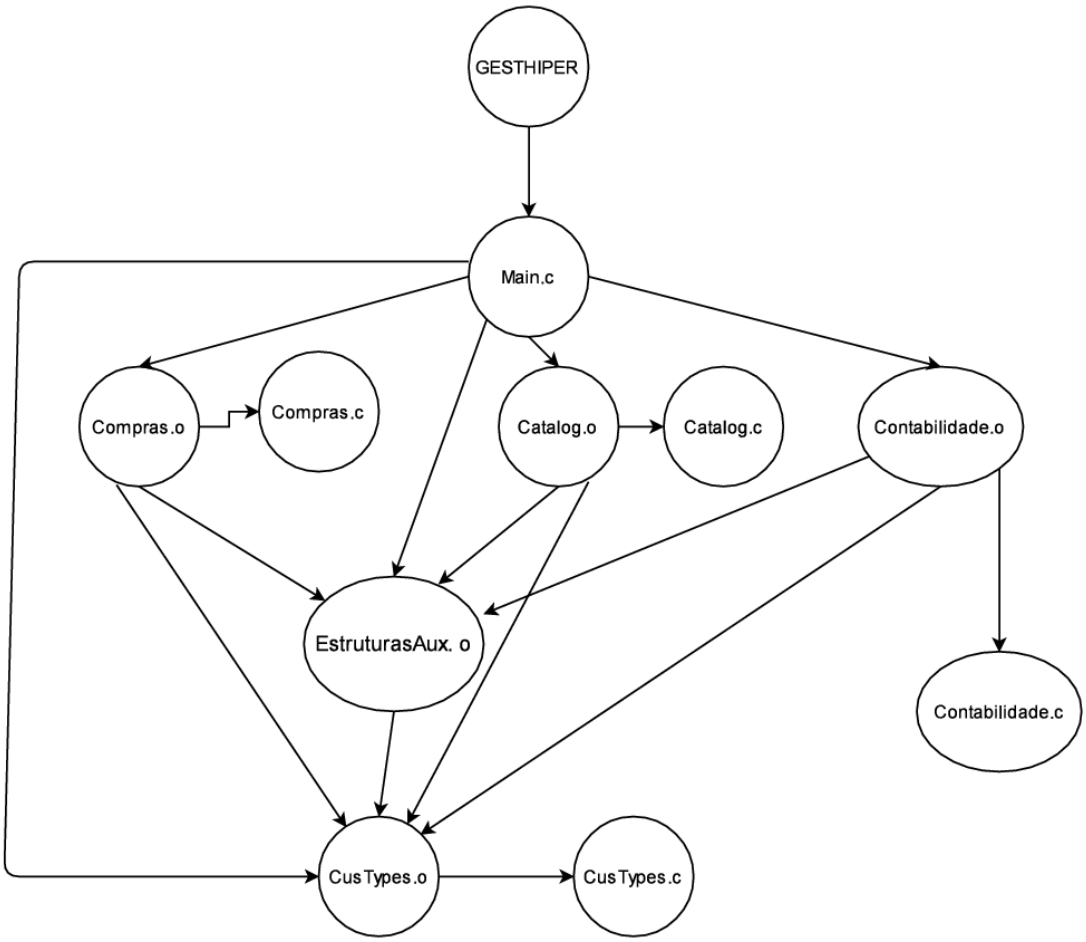


Figura 9 – Grafo de Dependências

6. Conclusão

6.1. Conclusões

O trabalho desenvolvido apresenta os requisitos que foram traçados no enunciado. Os módulos apresentados têm uma estrutura bem definida e podem ser utilizados separadamente, tanto em implementação como no programa em si, em que é possível utilizar sem ter os três ficheiros carregados, não podendo obviamente utilizar as queries relativas a esse módulo.

O número de estruturas diferentes para suportar a resposta das queries fez com que separássemos essas estruturas para um módulo extra. No entanto, foram detetados problemas, tanto de conteúdo como de gestão de memória, nas estruturas em forma de lista (array). Esta falha foi apenas colmatada no módulo de catálogo.

A fase de testes permitiu-nos detetar vários erros, muitos deles encontrados em funções de inserção em módulos e gestão de memória em listas de "objetos". Conseguimos corrigir com sucesso a maioria dos erros que encontrámos.

No que toca a queries, conseguimos resultados válidos na maioria das queries requeridas. Nas queries que não conseguimos apresentar resultados válidos o problema encontrou-se nas estruturas em forma de lista desenvolvidas no ficheiro EstruturasAux.c.

Os princípios de modularidade e encapsulamento, muito visados no decorrer das aulas práticas foram cumpridos e seguem as orientações e requisitos que foram apresentados.

Ao desenvolver uma aplicação modular em C é necessário adotar uma série de mecanismos e metodologias de desenvolvimento, como o caso dos Tipos Abstratos de Dados, que em outras linguagens de programação já se encontram implícitas, como o Java. Num ponto de vista geral avaliamos positivamente o trabalho realizado neste projeto e acreditamos que cumpre os requisitos enunciados, apesar das suas debilidades.

6.2. Trabalhos Futuros

De forma a aprimorar os resultados obtidos na realização deste projeto encontrámos os seguintes pontos:

- Refazer os "objetos" CodigoArray, ListaDePCM e ListaDePCQ já que na fase de testes, já bastante tardia e próxima da entrega eletrónica foi detetado que o segundo índice de variáveis deste tipo possuía campos indesejados e outras vulnerabilidades que suscitam a existência de falhas de segmentação. Uma possível reparação a estes objetos seria retirar-lhes a natureza dinâmica (utilizámos realloc sempre que fosse inserido um código novo, por exemplo), alterando, por exemplo, nos "construtores" destes objetos os parâmetros para que recebesse um tamanho máximo que haja uma só alocação de memória.
- Outro dos pontos a melhorar será a natureza recursiva que foi utilizada, a título de facilidade, nas inserções e atualizações nas árvores dos módulos de Compras e Contabilidade. Isso refletiu-se nos tempos de carregamento das estruturas de dados.