PUBLICACIÓ DOCENT

MANUAL DE LABORATORI D'ESIN Sessió 3

AUTOR: Bernardino Casas, Jordi Esteve
ASSIGNATURA: Estructura de la Informació (ESIN)
CURS: Q3
TITULACIONS: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació
ANY: 2019

Vilanova i la Geltrú, 15 de setembre de 2019

3

Exercici

L'objectiu d'aquest exercici és implementar la classe `mcj_enters` que ens permet representar i manipular multiconjunts d'enters sense una restricció de mida màxima del multiconjunt. Recorda que un multiconjunt és un conjunt on els elements poden estar repetits (i com succeeix en els conjunts, no importa l'ordre dels elements).

La declaració d'aquesta classe amb els mètodes que has d'implementar és idèntica a la de `mcj_enters.hpp` de la sessió 1, exceptuant que ja no és necessària (i per tant no hi és) la constant `MAX_SIZE`. Ara estàs obligat a implementar la constructora per còpia, la destructora i l'operador d'assignació, donat que els corresponents mètodes d'ofici no són adequats.

Els passos a seguir són els següents:

1. troba una representació adequada pels objectes de la classe i escriu els atributs necessaris en la part `private` de `mcj_enters.hpp`. La representació que has d'utilitzar és la d'una llista enllaçada dinàmica.

2. comença amb una implementació trivial per tots els mètodes en `mcj_enters.cpp`, i després implementa i prova els diferents mètodes paulatinament. Els primers mètodes que hauries d'implementar són la constructora, `insereix`, `conte` i `print`.

3. un cop provats aquests mètodes continua implementant i provant tota la resta fins a tenir la classe feta.

En cas que no es tingui clar el passos a seguir per crear un classe en `C++` es recomana llegir el Decàleg de la sessió 1.

3.1 Implementació i testeig

Un cop completada la part privada de la classe `mcj_enters` en el fitxer `mcj_enters.hpp` i implementats els mètodes públics i privats en el fitxer `mcj_enters.cpp` caldria fer un programa principal en el fitxer `main.cpp` que crei alguns objectes de la classe `mcj_enters`, insereixi dades i testegi els diferents mètodes aplicant-los sobre aquests objectes. Després compilar, linkar i testejar.

Per facilitar la feina de testeig, pots usar el mateix problema "Classe multiconjunt d'enters" (https://jutge.org/problems/X39772_ca) de la sessió 1 on només cal enviar l'especificació i implementació de la classe `mcj_enters`. Degut a que jutge.org només permet l'enviament d'un fitxer amb la solució del problema, en el mateix fitxer hi ha d'haver l'especificació i la implementació de la classe (el que normalment estarien separats en els fitxers .hpp i .cpp). I també cal eliminar la directiva `#include "mcj_enters.hpp"` per no tenir problemes de precompilació. Ho pots fer tot a la vegada amb la comanda:

```
cat mcj_enters.hpp mcj_enters.cpp | sed '/include "mcj_enters.hpp"/d' > solucio.cpp
```

i enviar a jutge.org el fitxer `solucio.cpp`.

Aquest problema de jutge.org ja disposa d'un programa principal i de jocs de prova públics i privats que automatitza la feina de testeig. Un cop enviïs la teva solució, jutge.org farà la compilació i linkat de tot i testejarà que passin tots els jocs de prova.

Recorda que implementarem el multiconjunt d'enters usant memòria dinàmica (tots els elements del multiconjunt d'enters estaran guardats en una llista dinàmica, has de decidir si ha d'estar simplement o doblement enllaçada, si ha de contenir un element fantasma inicial o no cal, ...). Et pots inspirar en les implementacions que trobaràs en el "Tema 4. Estructures lineals dinàmiques" de teoria. No es poden usar les classes `stack`, `queue`, `list` o `set` de la STL.

Si els mètodes d'unir, intersectar, restar, igualtat i diferència no es programen de forma eficient (cost lineal) no passaran els jocs de prova privats degut a un excés en el temps d'execució. Per tant has de calcular el cost temporal que té cadascun dels mètodes que has implementat i, en cas que sigui pitjor que un cost lineal, caldrà pensar com millorar la implementació, potser retocant com es guarden els elements del multiconjunt d'enters en la llista dinàmica.

Envia la solució a jutge.org amb l'anotació "Fet amb memòria dinàmica" perquè el professor sàpiga quina versió mirar quan te la corregeixi.

C

Compilació, muntatge i execució en C++

Un compilador és un programa d'ordinador que permet traduir un programa escrit (un llenguatge d'alt nivell) a un altre llenguatge de programació (normalment llenguatge màquina), generant un programa equivalent que l'ordinador pot entendre.

Aquesta eina permet al programador desconèixer el llenguatge que utilitza l'ordinador i escriure en un llenguatge més universal i més proper a com pensa un ésser humà.

En aquest capítol veure'm com és el procés de compilació, muntatge i execució en C++, i diferents eines que ens poden ser útils en cadascuna d'aquestes etapes.

C.1 Compilació separada i muntatge

C.1.1 Compilació i muntatge bàsic

La compilació és el procés durant el qual es tradueixen les instruccions escrites en un determinat llenguatge de programació a llenguatge màquina.

Per compilar programes en C++ utilitzarem el compilador GNU gcc, en concret utilitzarem l'ordre g++.

Per compilar un fitxer font per separat s'usa l'ordre g++ (el text en cursiva indica un argument) amb l' *opció -c* :

```
$ g++ -c nom_fitxer.cpp
```

Aquesta ordre produeix un fitxer objecte *nom_fitxer.o*.

Per generar el fitxer executable cal muntar ("linkar") un o més fitxers objectes. Només cal posar els noms dels fitxers objecte (sense que importi l'ordre) darrera de g++:

```
$ g++ nom_fitxer1.o nom_fitxer2.o ...
```

Aquest procés genera el fitxer executable per defecte que s'anomena a.out. Si es vol que el fitxer executable tingui un nom diferent llavors s'ha d'usar l' *opció -o* :

```
$ g++ -o nom_executable.e nom_fitxer1.o nom_fitxer2.o ...
```

També es pot compilar i muntar varis fitxers en una única ordre:

```
$ g++ -o nom_executable.e f1.cpp f2.o ...
```

Per exemple, donades dues classes (a i b) que són utilitzades per un programa principal ($prog$) (veure la la representació en la figura C.1) es podria generar l'executable d'aquesta forma:

```
$ g++ -o prog.e a.o prog.cpp b.o
```

Es genera un fitxer executable anomenat `prog.e`, que es forma a partir de la compilació del fitxer `pr.cpp` i el fitxer objecte resultant es munta amb els fitxers `a.o` i `ba.o`. El fitxer intermig `pr.o` no es conserva.

C.1.2 Compilació i muntatge de classes genèriques

Els mòduls que defineixen classes o funcions genèriques no es compilen MAI per separat. Una forma adequada d'organitzar el codi consisteix en escriure la classe en dos fitxers:

- *declaració de la classe*: en el fitxer capçalera amb extensió `.hpp` com estem acostumats.
- *implementació de la classe*: en un fitxer que per conveni li donarem l'extensió `.t`.

Cal tenir en compte que el fitxer `.hpp` ha d'incloure al fitxer `.t`. Així doncs, si un programa usa a la classe genèrica *X* llavors haurà d'incloure el fitxer `X.hpp` (i indirectament inclourà a `X.t`).

Es pot obtenir una comprovació sintàctica de la classe genèrica mitjançant la inclusió del fitxer `.hpp` (i per tant del fitxer `.t`) en un fitxer `.cpp`. Aquest fitxer només cal que tingui la línia d'inclusió.

Per exemple, donades dues classes a i b, que són utilitzades per un programa principal ($prog$) i la classe a és genèrica (veure la representació en la figura C.2) es podria generar l'executable després de fer:

```
$ g++ -c prog.cpp
$ g++ -c b.cpp
$ g++ -o prog.e prog.o b.o
```

Com es pot comprovar la classe *a* no s'hauria de compilar.

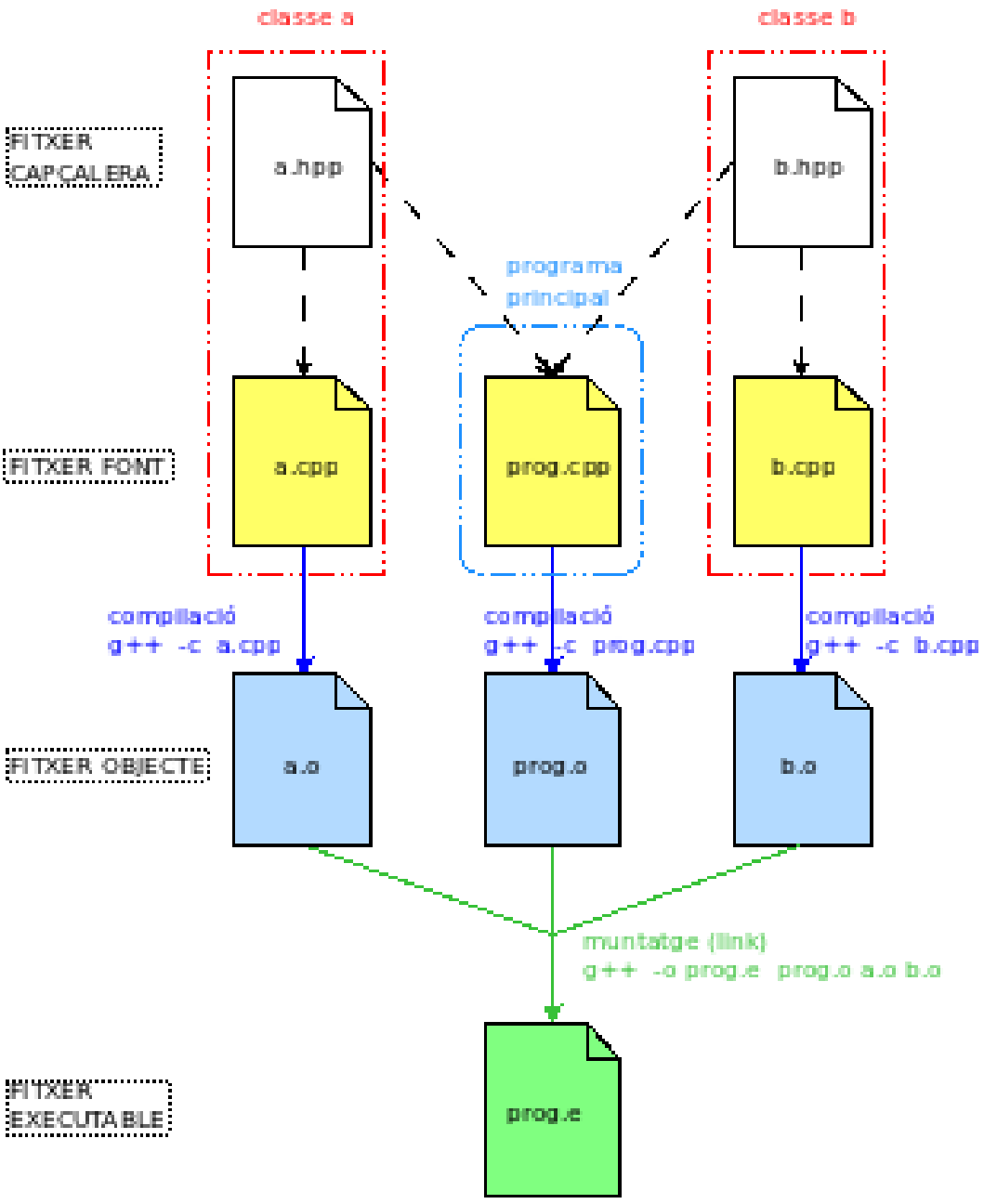

Figura C.1: Compilació i muntatge de les classes a i b, i el programa principal $prog$.

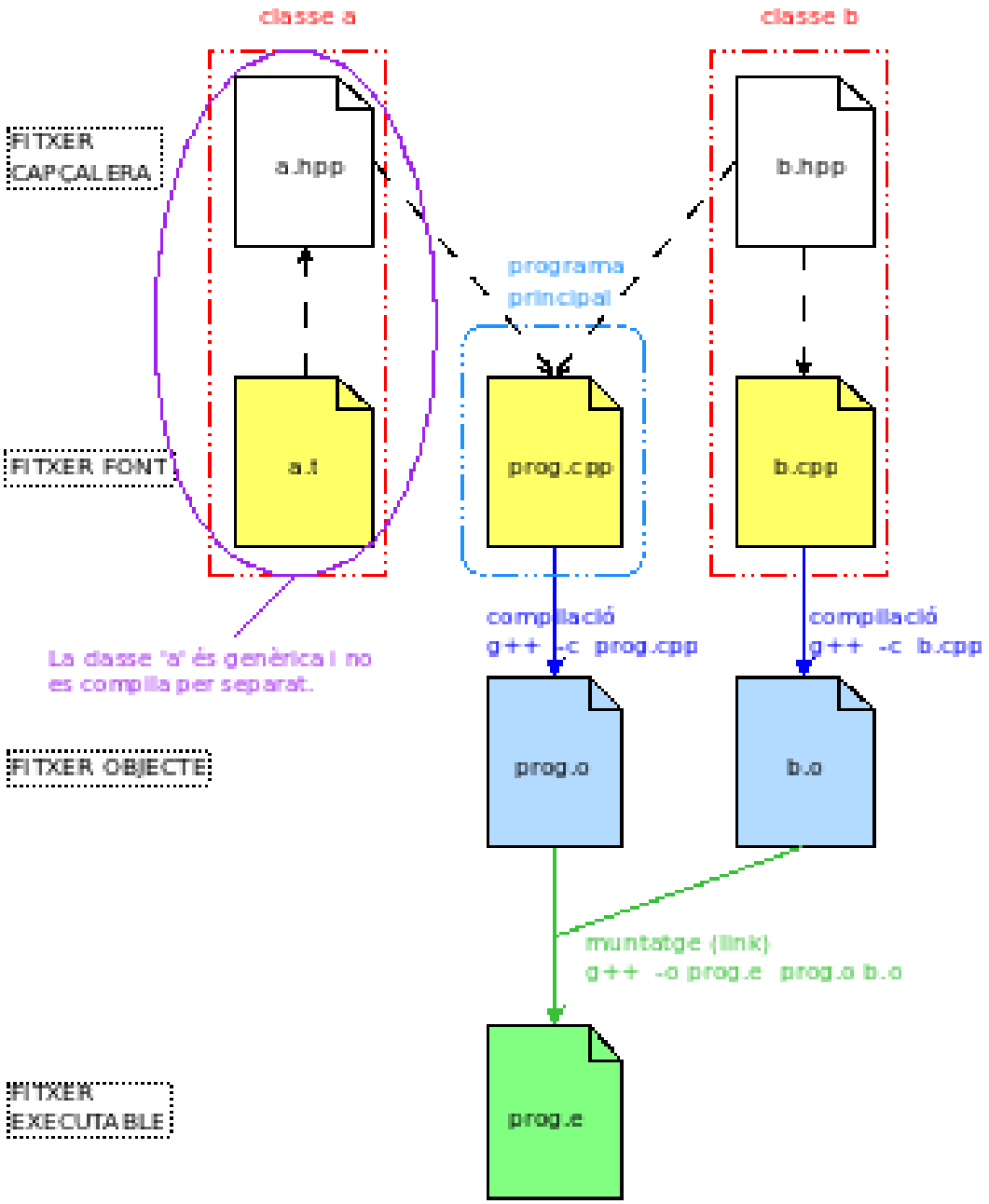

Figura C.2: Compilació i muntatge de les classes a (genèrica) i b, i el programa principal $prog$.

C.1.3 Compilació i muntatge amb biblioteques

Molts programes només utilitzen classes, mètodes i funcions definides a la biblioteca estàndard (per exemple, `string`, `iostream`, `list`, ...). El muntador (o *linker*) sempre empra per defecte aquesta biblioteca.

Si volem emprar una altra biblioteca caldrà indicar-ho explícitament en l'ordre de muntatge mitjançant l' *opció* `-l` .

A Unix el conveni és anomenar a les biblioteques `lib`*xxxx* amb extensió `.a` o `.so` (depenent de si la biblioteca és estàtica o `dinàmica`. Després de l'opció `-l` es posa la part variable del nom de la biblioteca, és a dir, *xxxx*.

Per exemple, per usar les biblioteques `libB1` i `libB2` caldria executar la següent ordre:

```
$ g++ -o nom_executable.e f1.o f2.o ... -lB1 -lB2
```

C.1.4 Altres opcions

Opció -I

Si necessitem utilitzar fitxers de capçalera (`.hpp`) que no es troben en el mateix directori on estem compilant o en un directori estàndard d'inclusió (per exemple, `/usr/include`) caldrà utilitzar l' *opció* `-I` . Posarem tantes opcions `-I` com directoris volguem afegir a la llista de directoris d'inclusió.

```
$ g++ -c -I /home/users/adam/headers f1.cpp
```

Per especificar un camí amb l'opció `-I` es pot usar el camí absolut o el camí relatiu.

Si l'ordre de l'exemple anterior s'estigués executant en el directori `/home/users/eva/pract` podríem haver escrit:

```
$ g++ -c -I ../../adam/headers f1.cpp
```

Opció -L

Un problema similar a l'anterior es dóna si necessitem usar una biblioteca que no estigui en el directori en curs o en un directori estàndard (por exemple, `/usr/lib`). En aquest cas s'utilitza l' *opció* `-L` per indicar el camí.

Per exemple, suposem que volem utilitzar `libB1.a` que es troba a `/home/users/esin`. Llavors escriurem:

```
$ g++ -L /home/users/esin -o prog.e prog.cpp f1.cpp -LB1
```

Opció -Wall

Per aconseguir que el compilador verifiqui instruccions dubtoses i avisi del màxim nombre possible de fonts d'error cal utilitzar l' *opció* `-Wall` :

```
$ g++ -c -Wall  nom_fitxer.cpp
```

Opció -g

Si es vol utilitzar un *debugger* caldrà afegir l' *opció* `-g` al compilar:

```
$ g++ -g -c -Wall fitx.cpp
```

o si es crea directament l'executable (sense compilació separada) llavors:

```
$ g++ -g -Wall -o nom_executable.e f1.cpp f2.cpp ...
```

C.2 Make

`make` és un programa d'Unix que simplifica notablement el treball de compilació i muntatge. A més a més, es pot instruir adequadament a `make` per tal que recompili només aquells fitxers que facin falta.

El programa `make` utilitza un fitxer anomenat `Makefile` que s'ha de trobar en el mateix lloc on s'executa el `make`. Aquest programa té un argument, el *target*, que és el que volem que es construeixi. Si escrivim `make` sense cap argument, el *target* és el primer que aparegui en el `Makefile`.

Per veure el funcionament del `make` escriurem el Makefile per l'exemple de la figura C.1. Per complicar-ho una mica suposarem que els fitxers `.hpp` es troben en el directori: `/home/users/yo/elsmeusincludes`, i a més a més necessitem la biblioteca `/home/users/yo/elsmeuslibs/libB1.a`.

```
1  CC = g++
2  INCL = /home/users/yo/elsmeusincludes
3  COMPILE = $(CC) -c -Wall -I $(INCL)
4  LIBS = /home/users/yo/elsmeuslibs
5  LINK = $(CC) -L $(LIBS)
6  OBJS = prog.o a.o b.o
7
8  prog.e: $(OBJS) $(LIBS)/libB1.a
9          $(LINK) -o prog.e $(OBJS) -lB1
10 prog.o: prog.cpp $(INCL)/a.hpp $(INCL)/b.hpp
11         $(COMPILE) prog.cpp
12 a.o: a.cpp $(INCL)/a.hpp
13         $(COMPILE) a.cpp
14 b.o: b.cpp $(INCL)/b.hpp
15         $(COMPILE) b.cpp
```

Les primeres sis línies defineixen variables. Es poden declarar tantes variables com vulguem. Si `X` és una variable, es pot accedir al seu valor mitjançant `$(X)`. Després de la declaració de variables vénen una sèrie de blocs de la forma:

```
target: dependències
        ordre1
        ordre2
        ...
```

Cada línia en la que escrivim una ordre ha de començar obligatòriament amb un **tabulador**.

Les dependències són llistes que indiquen quins elements intervenen directament en la construcció del *target* i poden canviar.

Per exemple, `prog.e` depèn dels fitxers `prog.o`, `a.o`, `b.o` i també de `libB1.a`, per això apareixen aquests fitxers en la llista de dependències. Sota la llista apareix l'ordre per generar `prog.e` a partir de les dependències. I el mateix passa amb la resta.

Es poden fer moltes altres coses mitjançant `make`. No només facilita el procés de compilació i muntatge.

Per exemple, si afegim aquestes línies al final del `Makefile`:

```
1 clean:
2         rm -rf *.o; rm prog.e
```

cada cop que escrivim `make clean` esborrarem tots el fitxers `.o` i l'executable.

C.3 Execució

Per executar un programa només cal escriure el nom de l'executable després del *prompt* de Unix:

$ *nom_executable*

Si el programa llegeix i escriu pels canals estàndard d'entrada (`cin`) i sortida (`cout`) es pot redirigir l'entrada per tal que vingui d'un fitxer i no del teclat, i la sortida per tal que s'escrigui en un fitxer enlloc de la pantalla. Per fer això utilitzarem els operadors de redireccionament (< per l'entrada i > per la sortida) seguits del nom del fitxer corresponent. Es pot redireccionat només l'entrada, només la sortida, o ambdues:

$ *nom_executable* < *fitxer_entrada* > *fitxer_sortida*

Es pot connectar la sortida estàndard d'un programa amb l'entrada estàndard d'un altre programa mitjançant una *pipe*. L'operador corresponent és la barra vertical `|`.

$ *nom_executable1* | *nom_executable2*

Molts programes escriuen els seus missatges d'error pel canal estàndard d'error (`cerr`) que normalment està associat a la pantalla. Si utilitzem l'operador > els missatges continuen apareixent per pantalla. Per redirigir el canal `cerr` s'utilitza l'operador `>&` seguit del nom del fitxer. Per exemple:

$ `g++ -c pr.cpp >& errores.txt`