Laboratorio de Programación 2
Guiones de las Sesiones (1ª edición)

C. Martínez

18 de septiembre de 2019

Prefacio

Estas transparencias se basan en las transparencias de las sesiones de laboratorio de Programación de Sistemas, asignatura de la extinta Ingeniería Técnica en Informática de Sistemas. Las transparencias escritas por diversos autores (B. Casas, R. M. Jiménez, C. Martínez y S. Pérez, en su 3ª y última edición) han sido actualizadas para adaptarlas a la asignatura Programación 2 del Grado en Ingeniería Informática.

1. Entorno de Trabajo y Repaso de C++
   - Repaso de C++: Funciones y Paso de Parámetros
   - Estructura de un programa en C++
   - Compilación separada y Montaje
   - Ejecución

Referencias

- Una referencia es un nombre alternativo a una variable ya declarada, es decir, un alias.
- Una referencia de una variable de tipo `T` es de tipo `T&`
- Toda referencia ha de ser inicializada en su declaración

```
char a; char& b = a;
```

excepto cuando la referencia es un parámetro de la función: en ese caso se inicializa al invocarse la función y efectuarse el paso de parámetros.

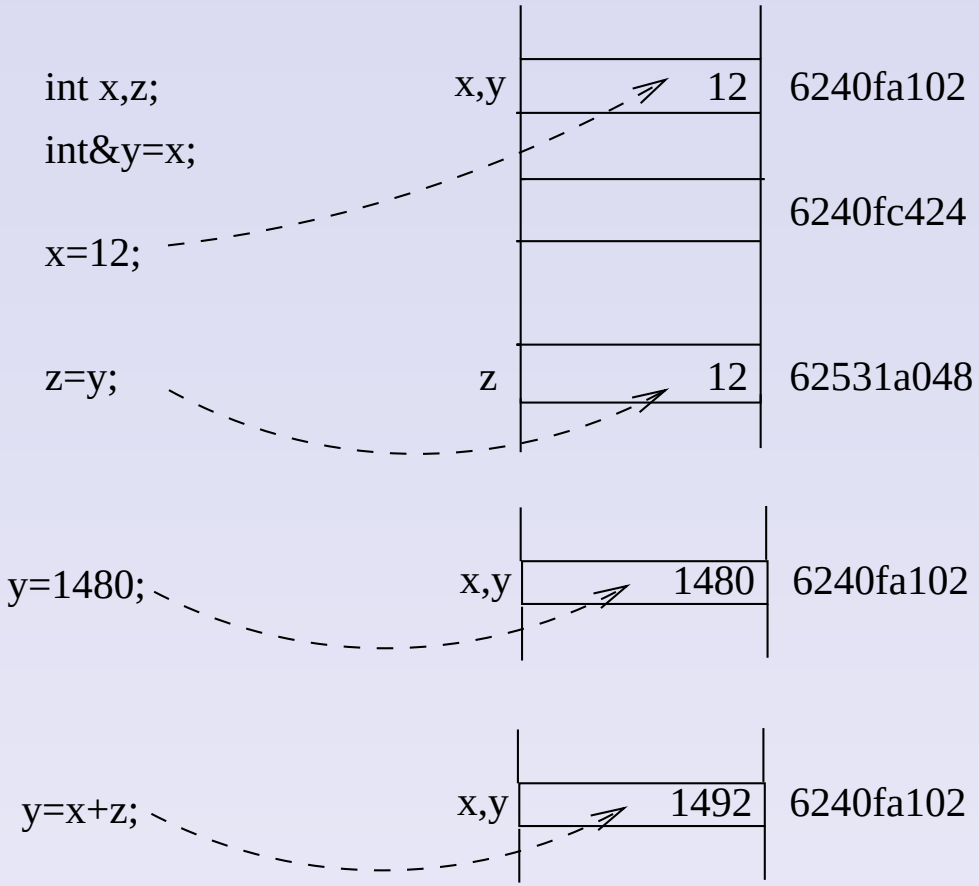
Referencias

- El valor de una referencia no se puede cambiar. Sólo se puede modificar el valor al que se refiere

```
a = 'b'; b = 'c'; // a vale 'c'
```

- La dirección de una referencia es la de la variable a la que se refiere.
- Las referencias están implementadas mediante punteros, pero no disponen de las operaciones sobre punteros.

Referencias



Ejemplo de manipulación de referencias

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue: (paso de parámetro por referencia)

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue: (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue: (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue: (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:
  1. El parámetro formal se comporta como una variable local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
     (paso de parámetro por valor)

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:
  1. El parámetro formal se comporta como una variable local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
     (paso de parámetro por valor)
     - Parámetro formal: `T` *`identificador`*

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:
  1. El parámetro formal se comporta como una variable local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
     (paso de parámetro por valor)
     - Parámetro formal: `T` *`identificador`*
     - Parámetro actual: *expresión*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*
3. El parámetro formal es una referencia al parámetro actual (no se hace copia) pero el parámetro formal no se puede modificar:
   (paso de parámetro por referencia constante)

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*
3. El parámetro formal es una referencia al parámetro actual (no se hace copia) pero el parámetro formal no se puede modificar:
   (paso de parámetro por referencia constante)
   - Parámetro formal: `const T&` *`identificador`*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*
3. El parámetro formal es una referencia al parámetro actual (no se hace copia) pero el parámetro formal no se puede modificar:
   (paso de parámetro por referencia constante)
   - Parámetro formal: `const T&` *`identificador`*
   - Parámetro actual: *variable*

Paso de Parámetros

- El paso de parámetros por valor (`T x`, `const T x`) se recomienda para los parámetros de entrada de tipos elementales predefinidos (`int`, `char`, `bool`, ...).

Paso de Parámetros

- El paso de parámetros por valor (`T x`, `const T x`) se recomienda para los parámetros de entrada de tipos elementales predefinidos (`int`, `char`, `bool`, ...).
- En cambio, para parámetros de entrada que sean objetos "complicados" es preferible el paso por referencia constante (`const T& x`), ya que se evita hacer copias costosas.

Paso de Parámetros

- El paso de parámetros por valor (`T x`, `const T x`) se recomienda para los parámetros de entrada de tipos elementales predefinidos (`int`, `char`, `bool`, ...).
- En cambio, para parámetros de entrada que sean objetos "complicados" es preferible el paso por referencia constante (`const T& x`), ya que se evita hacer copias costosas.
- El calificativo `const` permite que el compilador detecte errores en los que se modifica inadvertidamente un parámetro de entrada; no sería un problema si usamos paso por valor (`T x`), ya que trabajamos con una copia, pero es un problema muy serio si usamos paso por referencia, pues la función trabaja con el parámetro actual (simplemente le da un nombre o alias para referirse a él).

Paso de Parámetros

Parámetros de entrada/salida usando referencias

```
void ff(int& a, int& b) {
    if (a > 3) { a += 2; }
    else       { a -= 3; }
    b = b + 5 * a;
}

void f() {
    int x, y;
    cin >> x >> y;

    // p.e. x = 4, y = 7
    ff(x, y);
    // x = 6, y = 37
}
```

Paso de Parámetros

Parámetros de salida

```
void ff(int& a, int& b) {
    cin >> a >> b;

    if (a > 3) { a += 2; }
    else       { a -= 3; }
    b = b + 5 * a;
}

void f() {
    int x, y;

    // los valores de 'x' e 'y' no son
    // relevantes para 'ff'
    ff(x,y);
    // 'x' e 'y' se han modificado
}
```

Paso de Parámetros

Parámetros de entrada

```
void escribe_f(int a,const int b,
               const racional& r) {
   // 'b' y 'r' no pueden modificarse aqui
   cout << a + r.num() << endl;
   a = a + 2;
   cout << b - a + r.denom() << endl;
}

void f() {
   int x, y;
   racional r(1, 7);

   cin >> x >> y;

   // p.e. x = 5, y = 2 , r = 1/7}
   // el 3er parametro tiene que ser un objeto
   escribe_f(x + 3, 2 * y,r);
   // 'x', 'y', 'r' no han cambiado su valor
}
```

Devolución de Resultados

Una función puede devolver sus resultados por copia, por referencia o por referencia constante, i.e.,

tipo funcion(*lista_parámetros_formales*)

donde *tipo* es el nombre de un tipo (`T` o `void`), una referencia (`T&`), o una referencia constante (`const T&`).

Devolución de Resultados


```
int  f1(int x) { return x; }
int  f2(int& x) { return x; }
int& f3(int& x) { return x; }
int& f4(int x)  { return x; } // ← MAL!!

void f() {
    int w = 10;
    int  y = f1(w);
    // genera 3 copias del valor de 'w':
    // w → x, x → resultado, resultado → y
    int z = f2(w);
    // el parametro actual ha de ser una variable!
    // genera 2 copias: w ≡x → resultado, resultado → z
    int u = f3(w);
    // genera 1 copia: w ≡ x ≡ resultado → u
    ...
}
```

Devolución de Resultados

Una función no debe devolver un puntero o una referencia a una variable local o a un parámetro puesto que son destruidos al acabar la ejecución. El compilador detecta la mayor parte de errores como los que mostramos a continuación si activamos los *flags* adecuados. No obstante, a veces no son detectados, de manera que el error se produce en tiempo de ejecución.

```
float& calcula(int x, const float y) {
    float z = 0.0;
    while (x > 0) {
        z = z + y / x;
        x--;
    }
    return z;          // ERROR !!
}
```

Estructura de un programa C++

Hasta ahora (PRO1) nuestros programas se almacenaban en un único fichero *fuente* (con extensión `.cc` o `.cpp`)

```
// inclusiones
#include <iostream>
#include <vector>
...
// definiciones de constantes y de tipos
const int MAX_COLS = 100;
typedef vector<int> Fila;
typedef vector<Fila> Matriz;

// funciones y procedimientos
void lee_matrix(int nfil, int ncol, Matriz& M) {
    ...
}
...
// programa principal
int main() {
    ...
}
```

Estructura de un programa C++

Para el desarrollo de programas grandes y facilitar la reutilización de código los programas deben estructurarse en *módulos*.

- Cada módulo suele dar origen a dos ficheros
- Un fichero de cabecera (con extensión `.hh`) dónde se escribe la parte "declarativa" o especificación del módulo
- Un fichero fuente (con extensión `.cc`) dónde se escribe el código C++ de la implementación

Estructura de un programa C++

Matriz.hh

```
#include <vector>
// definiciones de constantes y de tipos
typedef vector<int> Fila;
typedef vector<Fila> Matriz;
// declaración de funciones y procedimientos
void lee_matriZ(int nfil, int ncol, Matriz& M);
void escribe_matriz(const Matriz& M);
...
```

prog.cc

```
#include "Matriz.hh"
#include <iostream>
...
// programa principal
int main() {
    ...
}
```

Estructura de un programa C++

Matriz.cc

```
#include <iostream>
#include <vector>
#include "Matriz.hh"

void lee_matriZ(int nfil, int ncol, Matriz& M) {
  ...
}
void escribe_matriz(const Matriz& M) {
  ...
}
```

Estructura de un programa C++

- Cada uno de los módulos que forma parte del programa se compilará de manera separada y para obtener el ejecutable se tendrá que hacer un paso adicional de *montaje*
- Esta "complicación" ofrece innumerables ventajas de todo tipo: desde ser imprescindible para el desarrollo de programas complejos a reducir el tiempo de generación de un nuevo ejecutable, pasando por la reutilización de módulos en varios programas distintos

Compilación separada y montaje

- El compilador GNU `gcc-`*`x.xx.x`* (instalado en el LCFIB) permite compilar programas en C y en C++. Para compilar programas en C++ los ficheros fuente deben tener extensión `.cc` y utilizarse el comando g++ (mejor aún: usar el alias p2++ que invoca a g++ con toda una serie de *flags* útiles).
- El alias podéis definirlo en vuestro ordenador con una línea

```
alias p2++='g++ -D_GLIBCXX_DEBUG -O2 -Wall -Wextra
            -Werror -Wno-uninitialized
            -Wno-sign-compare -std=c++11'
```

Es conveniente incluir esta línea en vuestro fichero `.tcshrc` (`.bashrc`, ...) de vuestro directorio raíz

Compilación separada y montaje

- Se puede consultar un resumen de la documentación con el comando `man gcc`

Compilación separada y montaje

- Se puede consultar un resumen de la documentación con el comando `man gcc`
- El comportamiento del comando g++ se puede controlar mediante diversos *flags*, como es habitual en Unix. Para averiguar los principales *flags* admitidos por g++ se usa el *flag* `--help`

Compilación separada y montaje

- Para compilar por separado un fichero fuente se usa el comando (las letras en cursiva indican un argumento)

```
% g++ -c nom_fich.cc
```

el cual produce un fichero objeto *nom_fich*.o.

- Para montar ("linkar") varios ficheros objeto simplemente se ponen sus nombres tras el comando g++, seguidos unos de otros y no importando el orden:

```
% g++ nom_fich1.o nom_fich2.o  ...
```

- El fichero ejecutable por defecto se llama a.out. Si se quiere que el ejecutable tenga un nombre diferente entonces se debe usar el *flag* -o:

```
% g++ -o nom_ejecutable f1.o f2.o ...
```

Compilación separada y montaje

- Es habitual poner extensión `.exe` a los ficheros ejecutables
- También se puede compilar y linkar varios ficheros en una única línea de comando:

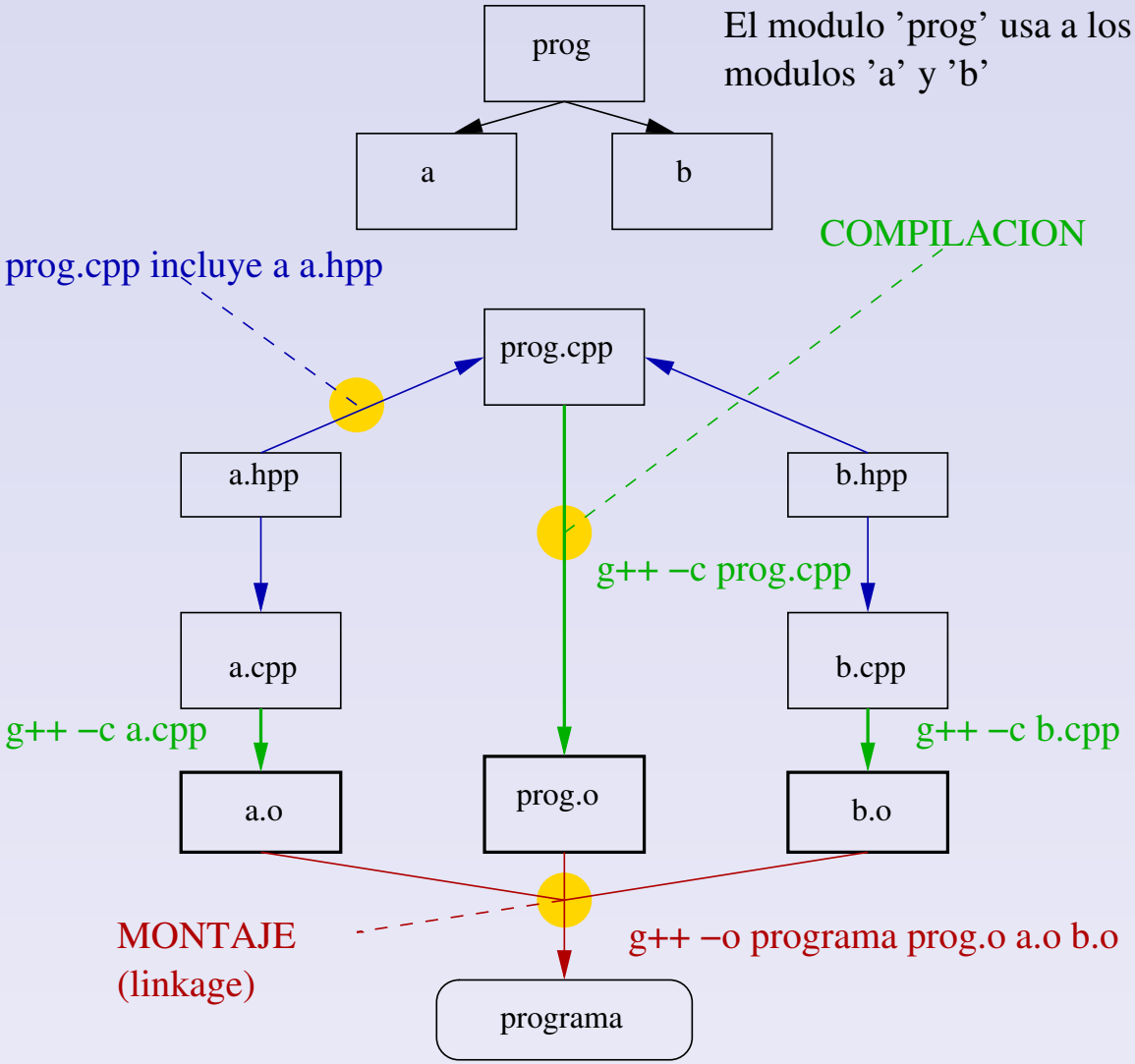
```
% g++ -o nom_ejecutable f1.cpp f2.o ...
```

Por ejemplo,

```
% g++ -o prog.exe cola.o pr.cc pila.o
```

genera un fichero ejecutable llamado `prog.exe`, para lo cual se compila `pr.cc` y el fichero objeto resultante se linka con los ficheros objeto `pila.o` y `cola.o`. No se conserva el fichero intermedio `pr.o`.

Compilación separada y montaje



Compilación separada y montaje

- Los módulos que definen clases o funciones genéricas no se compilan por separado. Lo habitual en dichos casos es escribir la especificación **y** la implementación en el fichero de cabecera `.hh`. Si un programa usa a la clase genérica X entonces deberá incluir el fichero `X.hh` (que incluye la implementación). No habrá un fichero `X.cc` con la implementación.

Compilación separada y montaje

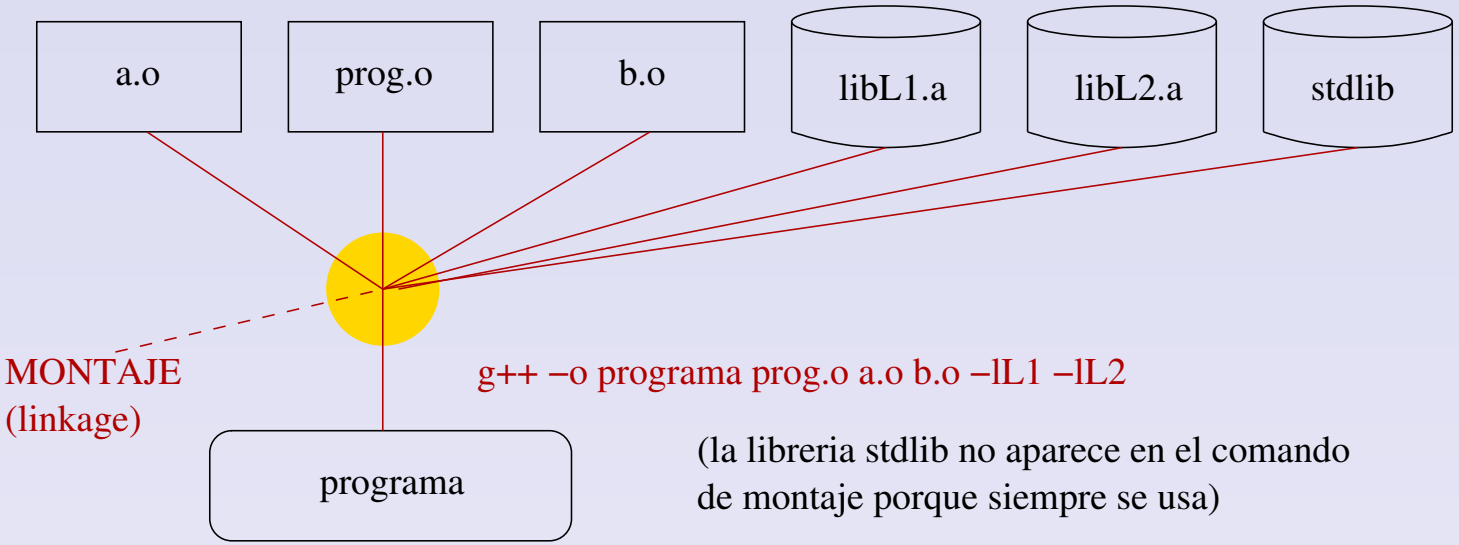
- Los módulos que definen clases o funciones genéricas no se compilan por separado. Lo habitual en dichos casos es escribir la especificación **y** la implementación en el fichero de cabecera `.hh`. Si un programa usa a la clase genérica X entonces deberá incluir el fichero `X.hh` (que incluye la implementación). No habrá un fichero `X.cc` con la implementación.
- Se puede obtener una comprobación sintáctica de la especificación e implementación de una clase genérica mediante la inclusión del `.hh` en un fichero `.cc`. Este fichero solamente debe tener la línea de inclusión.

Compilación separada y montaje

- Muchos programas sólo emplean clases, métodos y funciones definidos en la biblioteca estándar (p.e. `string`, `list`, `iostream`, ...).
  El montador (*linker*) emplea siempre (por defecto) dicha biblioteca.
- Si queremos usar otra biblioteca se habrá de indicar explícitamente en el comando de montaje, mediante el *flag* `-l`. En Unix el convenio es llamar a las bibliotecas `lib`*xxx* con extensión `.a` o `.so`. Después del flag `-l` se pone la parte variable del nombre, es decir *xxx*. Por ejemplo, para usar las librerias `libL1` y `libL2` escribiremos el comando

```
% g++ -o nom_ejecutable f1.o f2.o ... -lL1 -lL2
```

Compilación separada y montaje



(la libreria stdlib no aparece en el comando de montaje porque siempre se usa)

Compilación separada y montaje

- Si necesitamos utilizar ficheros de cabecera (*headers*) que no se encuentran en el mismo directorio donde estamos compilando o en un sudirectorio estándar de inclusión (p.e. `/usr/include`) debemos usar uno o más *flags* `-I`, tantos como subdirectorios queramos añadir a la lista de subdirectorios de inclusión. Por ejemplo:

```
% g++ -c -I /usr/users/fred/my_headers f1.cc
```

- Para especificar un camino en el flag `-I` puede usarse el camino absoluto o el camino relativo. Si el comando del ejemplo anterior se estuviera haciendo en el subdirectorio `/usr/users/wilma/pract` podríamos haber escrito:

```
% g++ -c -I ../../fred/my_headers f1.cc
```

Compilación separada y montaje

- Un problema similar al anterior se da si necesitamos usar una biblioteca que no está en el directorio en curso o en un subdirectorio estándar. Se usa el *flag* `-L` para indicar el camino. Supongamos que queremos usar `libL1.a` que se encuentra en `/usr/users/ps`. Entonces escribiremos:

```
% g++ -L /usr/users/ps -o pr pr.cc f1.cc f2.o -lL1
```

- Para conseguir que el compilador verifique instrucciones dudosas y alerte del máximo número posible fuentes de error debe utilizarse el *flag* `-Wall`:

```
% g++ -c -Wall nom_fich.cc
```

Compilación separada y montaje

- Si queremos emplear un *debugger* se habrá de poner el *flag* -g al compilar o crear el ejecutable a partir de los ficheros fuente:

```
% g++ -g -c -Wall fich.cc
% g++ -g -o nom_ejecutable -Wall f1.cc f2.cc ...
```

Bibliotecas externas

- Hasta ahora se ha visto que la compilación produce ficheros objeto *nom_fich*.o que se utilizan para montar el ejecutable final. También se ha comentado que el compilador utiliza bibliotecas externas para montar ese ejecutable.

Bibliotecas externas

- Hasta ahora se ha visto que la compilación produce ficheros objeto *nom_fich*.o que se utilizan para montar el ejecutable final. También se ha comentado que el compilador utiliza bibliotecas externas para montar ese ejecutable.
- Estas bibliotecas son generalmente del sistema, pero también se pueden crear unas propias y utilizarlas para futuros programas (o bien distribuirlas para su uso).

Bibliotecas externas

- Existen dos tipos de bibliotecas externas: estáticas y dinámicas. Las estáticas se adjuntan al ejecutable en el momento del enlazado contribuyendo con su código. Es decir, el fichero ejecutable contiene todo el código necesario y eventualmente podríamos prescindir de los ficheros de las bibliotecas sin problema

Bibliotecas externas

- Existen dos tipos de bibliotecas externas: estáticas y dinámicas. Las estáticas se adjuntan al ejecutable en el momento del enlazado contribuyendo con su código. Es decir, el fichero ejecutable contiene todo el código necesario y eventualmente podríamos prescindir de los ficheros de las bibliotecas sin problema
- Por el contrario, el ejecutable se enlaza a las bibliotecas dinámicas mediante una referencia, de modo que la correcta ejecución del programa requiere que la biblioteca esté accesible en el momento de la ejecución, en concreto, estar ubicada en unos directorios específicos, listados en la variable de entorno `LD_LIBRARY_PATH`

Bibliotecas externas estáticas

- Para formar una biblioteca estática, previamente hemos de compilarla y posteriormente formarla mediante el comando `ar` que es un archivador (al estilo de `tar` o `jar`):
  1. Compilación:

```
% g++ -c -Wall fich1.cc fich2.cc
```

  2. Creación de la biblioteca:

```
% ar -r nom_lib fich1.o fich2.o
```

  3. Esto produce un fichero `.a` que contiene los ficheros objeto de la biblioteca.

Bibliotecas externas dinámicas

- El caso de las bibliotecas dinámicas o compartidas, la compilación es algo más compleja. Es necesario solicitar código independiente de la posición (position-independent code (PIC)) mediante el modificador -fpic
  1. Compilación:

```
% g++ -c -Wall -fpic fich1.cc fich2.cc
```

  2. Creación de la biblioteca en Linux:

```
% g++ -o nom_lib -G -fpic fich1.o fich2.o
```

  3. Creación de la biblioteca en Solaris:

```
% g++ -o nom_lib -fpic -shared -Ur fich1.o fich2.o
```

  4. Esto produce un fichero *nom_lib*.so que contiene los ficheros objeto de la biblioteca. La extensión de estos ficheros indica una biblioteca dinámica: **s**hared **o**bject.

La ubicación de las bibliotecas dinámicas

- Los ejecutables enlazados con bibliotecas dinámicas dependen de la existencia de los ficheros `.so` correspondientes.
- Para saber qué bibliotecas dinámicas utiliza un determinado ejecutable y si éstas bibliotecas se encuentran disponibles existe el comando `ldd`:

```
% ldd ejecutable
```

- El resultado nos muestra la lista de bibliotecas que se requieren y si se encuentran o no disponibles. En caso de que alguna no se encuentre, el programa puede no funcionar correctamente.

La ubicación de las bibliotecas dinámicas

- Los directorios donde están ubicados los ficheros de bibliotecas dinámicas o compartidas han de estar includos en la variable de entorno `LD_LIBRARY_PATH`.
- Para añadir un directorio a la lista de directorios de esta variable:

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:nuevo_dir
```

- Es importante recalcar que los ejecutables y las bibliotecas han de ser compatibles en cuanto a arquitectura (Intel, Sparc, etc) como a sistema operativo (Unix, GNU/Linux, etc) como a versión del compilador utilizado.

Compilación separada y montaje

- Tener que escribir un comando de compilación y montaje en el que intervienen varios módulos, bibliotecas, *flags*, . . . acaba convirtiéndose en una tarea pesada y propensa al error. Una opción consiste en definir un alias en el fichero `~/.tcshrc` (el nombre dependerá del *shell* que se esté usando). Por ejemplo, si ponemos

```
alias compila 'g++ -c -I dir1 \!*'
```

  a partir de ese momento (mejor dicho, al hacer *login* la siguiente vez) podremos escribir:

```
% compila pr.cc
```

- Otra opción es definir un *makefile*.

Make

- `make` es un programa de Unix que simplifica notablemente el trabajo de compilación y montaje. Además, instruyendo adecuadamente a `make`, éste se limitará a recompilar los módulos que realmente haga falta.
- El programa `make` utiliza un fichero llamado `Makefile` que se debe encontrar en el mismo directorio donde se ejecuta el `make`. Tiene un argumento, el *target*, que es lo que queremos que se construya. Si se escribe el comando `make` a secas, el *target* es el primero que aparece en el `Makefile`.
- Ilustramos su funcionamiento mediante un ejemplo concreto, correspondiente (más o menos) al diagrama de módulos de la figura 1.

Ejemplo de Makefile

- El fichero `Makefile` que debe residir en el mismo directorio que los ficheros `.cc`.
- Supondremos que los ficheros `.hh` están en el subdirectorio `/usr/users/yo/mis_include`.
- Para construir `prog` supondremos que además se necesita la biblioteca `/usr/users/yo/mis_libs/libL1.a`.

Ejemplo de Makefile

Makefile

```
CC = g++
INCL = /usr/users/yo/mis_include
COMPILE = $(CC) -c -Wall -I $(INCL)
LIBS = /usr/users/yo/mis_libs
LINK = $(CC) -L $(LIBS)
OBJS = prog.o a.o b.o
prog.exe: $(OBJS) $(LIBS)/libL1.a
    $(LINK) -o prog $(OBJS) -lL1
prog.o: prog.cc $(INCL)/a.hh $(INCL)/b.hh
    $(COMPILE) prog.cc
a.o: a.cc $(INCL)/a.hh
    $(COMPILE) a.cc
b.o: b.cc $(INCL)/b.hh
    $(COMPILE) b.cc
```

Sintaxis de Makefile

- Las primeras seis líneas definen variables. Se pueden tener tantas como se quieran. Si `X` es una variable, se puede acceder a su valor mediante `$(X)`. Luego vienen una serie de bloques de la forma:

```
target: dependencias
        comando1
        comando2
        ...
```

- Cada línea en la que escribimos un comando debe necesariamente empezar con un tabulador.

Sintaxis de Makefile

- Las dependencias son listas que dicen que elementos intervienen directamente en la construcción del *target* (y están sujetas a cambio).
- Por ejemplo, `prog` depende de los ficheros objeto `prog.o`, `a.o`, `b.o` y también de `libL1.a`. Eso mismo indican la línea de dependencias. En el comando de abajo se indica como obtener `prog` a partir de las dependencias.
- Lo mismo con `prog.o` y los demás. Por ejemplo, `prog.o` **no** depende de `a.cc` ni `b.cc`, pero sí de sus especificaciones (`a.hh` y `b.hh`).

Sintaxis de Makefile

- Se pueden hacer **muchas** cosas mediante `make`. No sólo facilitar el proceso de compilación y montaje. Aquí sólo hemos arañado la superficie y dado una visión simplificada. Por ejemplo, si añadimos al final del `Makefile` las líneas

```
clean:
	rm -f *.o ; rm prog
```

bastará escribir `make clean` para borrar todos los ficheros objetos y el ejecutable.

Ejecución

- Para ejecutar un programa basta con escribir el nombre del ejecutable correspondiente como respuesta al *prompt* de Unix.

```
% nom_ejecutable
```

- Si el programa lee y escribe por los canales estándar de entrada (`cin`) y salida (`cout`) se puede redirigir la entrada para que provenga de un fichero y no del teclado, y la salida para que escriba en un fichero en lugar de en la pantalla.

Ejecución

- Para ello se utilizan los operadores de rediccionamiento (< para la entrada, > para salida) seguidos del nombre del fichero correspondiente. Se puede redireccionar sólo la entrada, sólo la salida, o ambas:

```
% nom_ejecutable < fichero_entrada > fichero_salida
```

- Se puede conectar la salida estándar de un programa con la entrada estándar de otro programa mediante una *pipe*. El operador correspondiente es la barra vertical |.

```
% nom_ejecutable1 | nom_ejecutable2
```

Ejecución

- Al combinar dos programas mediante una *pipe* el resultado se comporta como un único programa al cual se le puede redirigir la entrada, la salida, conectarlo con otra *pipe*, etc. Por ejemplo, `more` muestra su entrada por pantalla página a página esperando una orden del usuario para saltar de una página a la siguiente. Si tenemos un programa `prog` y queremos ver su salida poco a poco podemos escribir:

```
% prog.exe | more < entrada.in
```

Ejecución

- Muchos programas imprimen sus mensajes de error por el canal estándar de error (`cerr`) que normalmente está asociado a la pantalla. Si utilizamos el operador > los mensajes siguen apareciendo por pantalla. Para redirigir el `cerr` se usa el operador >& seguido de un nombre de fichero. Por ejemplo,

```
% g++ -c pr.cc >& errores.out
```

genera el fichero `errores.out` con un listado de todos los errores de compilación.