Guía para las sesiones prácticas

Programación 2

Curso 2018-2019

Inicio de la sesión de trabajo

Después de encender la máquina, se debe entrar en Linux y, una vez haya arrancado el sistema, introducir el usuario y contraseña de UACloud en el ordenador. Aunque es posible usar entornos integrados, también se puede usar un *terminal*. De hecho, lo recomendable es abrir dos terminales, uno para editar el programa y otro para compilar y ejecutar el programa.

En UNIX/Linux, todos los datos de un usuario se almacenan en un directorio llamado "`/home/usuario`" (cambiando "`usuario`" por el usuario en cuestión, obviamente). El terminal se abre automáticamente en dicho directorio. En Internet existen muchos manuales y tutoriales de Linux, uno de los más interesantes es:

`http://www.linux-party.com/TutorialLinux/linux_files/contenidos.html`

IMPORTANTE: Para evitar que un compañero se copie tu trabajo (en casos de copia se suspenderá a *todos* los alumnos implicados), cuando termines de trabajar debes salir de la sesión de la máquina en la que hayas trabajado, de forma que vuelva a salir la pantalla en la que pide usuario y contraseña o se apague.

Edición de un programa fuente

Lo primero que hace un programador cuando le dicen que haga un programa es encender el ordenador, entrar en un editor de textos, y ponerse a teclear; este suele ser el primero de los errores que va a cometer a lo largo del desarrollo del programa. Lo primero que *debe* hacer un programador es apagar el ordenador y coger un lápiz y un papel para estudiar bien el problema a resolver, su solución, sus casos especiales y diseñar la solución al problema en forma de algoritmo, en papel, sin utilizar ningún lenguaje de programación (normalmente se usa pseudo-código).

Una vez se ha diseñado en papel el algoritmo, el programa se puede escribir en el lenguaje de programación elegido mediante un editor de textos. Por ejemplo, supongamos que queremos escribir un programa que imprima un saludo por pantalla. Suponiendo que ya se ha diseñado el algoritmo en papel (tampoco es muy complicado), el programa resultante sería:

```
#include <iostream>

using namespace std;

int main()
{
   cout << "Hola, mundo" << endl ;
}
```

A modo de ejemplo, utiliza el editor que prefieras para teclear este programa y guardarlo en un fichero con nombre `hola.cc`. En general, es recomendable elegir un editor que proporcione ayudas para visualizar mejor el código, sangrado automático, etc. La elección de un editor es una tarea a la que merece la pena dedicar un poco de tiempo, y depende de los gustos personales de cada programador. La diferencia entre usar un buen editor (o un editor adaptado a los gustos del programador) y usar uno malo puede ser enorme en cuanto a productividad a la hora de escribir código. Como norma general, cuanto menos se

tenga que utilizar el ratón, mayor será la productividad. Además, la mayoría de los editores tienen atajos de teclado que conviene aprender, y muchos de ellos permiten definir atajos al programador, que pueden ahorrar mucho tiempo.

Algunos editores recomendables son los siguientes:

- Windows (no recomendado para Programación 2)
  - CodeLite (`http://codelite.sourceforge.net`)
  - Notepad++ (`http://notepad-plus.sourceforge.net`)
- Linux
  - CodeLite (`http://codelite.sourceforge.net`)
  - Anjuta (`http://anjuta.sourceforge.net`)
  - Joe
  - Gedit
  - Geany
  - Kate
- OS X (no recomendable para P2, pero mejor que Windows)
  - TextWrangler (`http://www.barebones.com/products/textwrangler/`)
  - Xcode (integrado en OS X)

Además de estos editores, podemos trabajar con Eclipse (`https://www.eclipse.org/`), que funciona en cualquiera de los anteriores sistemas operativos.

Compilación de un programa fuente

El compilador es un programa (o conjunto de programas) que se encarga de traducir un programa fuente en un programa ejecutable; el programa fuente está escrito en un lenguaje de alto nivel (C/C++, Pascal, Java, . . .), y el programa ejecutable está escrito en el lenguaje que entiende la máquina, el código máquina.

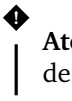
Si durante el proceso de traducción el compilador encuentra algún error, produce un mensaje (indicando la fila en la que está el error) y no genera el programa ejecutable. A veces, el compilador produce un *aviso* (*warning* en inglés), indicando que hay algo raro, pero que no es incorrecto; en este caso sí se genera el programa ejecutable, pero lo más probable es que no funcione correctamente. Un buen programador (tenga más o menos experiencia) debe considerar errores todos los avisos que produce el compilador, y corregirlos como si fueran errores.

En C/C++ (y en otros muchos lenguajes) un programa fuente puede (y suele) estar dividido en varios ficheros con código (acabados en `.cc`) y ficheros con declaraciones o *ficheros cabecera* (del inglés *header*), acabados en `.h`.

El comando que debes utilizar para compilar un programa desde un terminal en Linux es:

Terminal

```
$ g++ -Wall -g progFuente.cc -o progEjecutable
```



Atención: El símbolo `$` que aparece en el ejemplo anterior simboliza el punto de entrada de la línea de comando. No debes escribirlo.

El programa que invoca al compilador es `g++`, y las opciones que aparecen sirven para que se generen todos los avisos (`-Wall`) y para incluir información para depurar el programa (`-g`). Por ejemplo, para compilar el programa fuente que has escrito antes debes introducir el siguiente comando:

Terminal

```
$ g++ -Wall -g hola.cc -o hola
```

Este comando compila el programa fuente `hola.cc` y genera el programa ejecutable `hola`, si no tiene errores, por supuesto; si el programa tiene errores hay que volverlo a editar, corregir los errores y volver a compilar. Si hay muchos errores y no se pueden ver todos en la pantalla normalmente se debe al primero de ellos, que es el que hay que corregir antes de volver a compilar. Para poder estudiar detenidamente los errores (o ver el primero), el comando que se debe utilizar es:

Terminal

```
$ g++ -Wall -g progFuente.cc -o progEjecutable 2> errores
```

Este comando guarda los errores y *warnings* en el fichero `errores` (borrando su contenido previo, si lo hubiera); si el fichero `errores` está vacío es porque no hay errores, obviamente. Ese fichero se puede editar con el editor de textos o consultar con el comando:

Terminal

```
$ less errores
```

Una vez hayas corregido todos los errores y *warnings* puedes ejecutar el programa generado por el compilador, para lo cual tienes que escribir lo siguiente (suponiendo que el programa ejecutable se llama `hola`):[1]

Terminal

```
$ ./hola
```

Cuando el programa fuente está formado por más de un fichero (como por ejemplo en las prácticas de la asignatura), lo normal es que utilices un fichero `makefile`, para compilar todos los ficheros y construir el programa ejecutable.[2]

IMPORTANTE: No se debe esperar a tener escrito todo el programa para compilarlo, porque entonces el número de errores puede ser importante, y puede que haya que diseñar de nuevo el programa. Cada vez que se termine una parte del programa se debe compilar, corregir los errores que pudiera tener e incluso hacer alguna prueba parcial del programa antes de seguir tecleando el resto del programa.

Depuración y búsqueda de errores en un programa

La localización de errores de ejecución en un programa es una tarea complicada que se debe abordar con paciencia y de forma sistemática. La alternativa más simple cuando se produce un error de compilación es la de utilizar la salida por pantalla para mostrar valores de variables y seguir el flujo del programa. En más ocasiones de las que se suele pensar, es la mejor alternativa para encontrar un error.

[1] Si escribes `PATH=$PATH:.` en el terminal, ya no será necesario poner `./` delante del nombre del ejecutable cada vez que lo ejecutes.

[2] Verás más información sobre cómo crear un fichero `makefile` en el *Tema 5* de teoría.

A mano

Una herramienta indispensable para la prueba y depuración de un programa que trabaja en modo texto (con entrada y salida por pantalla), es la redirección de entrada y salida que proporciona el sistema operativo.

La técnica consiste en diseñar varios conjuntos de valores de entrada para el programa a probar, y almacenar cada conjunto en un fichero de texto normal, con cada dato en una línea (si se leen con `cin >>` se pueden poner varios datos en la misma línea separados por blanco). A continuación se ejecuta el programa con cada fichero de entrada, y se redirige la salida a otro fichero de texto. La salida se puede comprobar a mano o (mejor) con un programa. Los correctores de las prácticas de la asignatura utilizan esta técnica.

Por ejemplo, dado el siguiente programa en C/C++:

```
#include <iostream>
#include <math.h>

using namespace std;

int main()
{
  double a,b,c;

  cout << "a x^2 + b x + c = 0" << endl;
  cout << "a=" ; cin >> a;
  cout << "b=" ; cin >> b;
  cout << "c=" ; cin >> c;
  cout << (-b + sqrt(b*b-4*a*c))/(2*a) << endl;
  cout << (-b - sqrt(b*b-4*a*c))/(2*a) << endl;
}
```

Para probar este programa, diseñamos por ejemplo tres (deberían ser más) conjuntos de entradas, y los almacenamos en tres ficheros:

```
entrada1.txt : 1 7 2
entrada2.txt : 1 3 -1
entrada3.txt : 1 2 2
```

A continuación, ejecutamos el programa (que supongamos que se llama `ec2g`) de la siguiente manera:

Terminal

```
$ ./ec2g < entrada1.txt > salida1.txt
$ ./ec2g < entrada2.txt > salida2.txt
$ ./ec2g < entrada3.txt > salida3.txt
```

De esta manera, el programa `ec2g` no pedirá los datos por pantalla y los leerá del fichero. Para el primer conjunto, los valores que almacenará en `a`, `b` y `c` son `1`, `7` y `2`, respectivamente. Una vez ejecutadas las pruebas, es necesario comprobar que las salidas son correctas, bien a mano o utilizando otro programa (las salidas están en los ficheros `salida1.txt`, `salida2.txt` y `salida3.txt`).

IMPORTANTE: la elección de los conjuntos de entradas debe hacerse de forma cuidadosa, eligiendo los casos que el probador piense que pueden producir fallos. No es tan importante la cantidad de pruebas como la calidad de dichas pruebas. En muchos proyectos, se diseñan antes las pruebas (a partir de la especificación del problema a resolver) que el programa. En este ejemplo hay un caso que produce un error, pero hay más casos conflictivos (p.ej. "`0 2 2`", "`1 0 3`").

El depurador `gdb`

Otra alternativa que puede resultar más cómoda a veces es utilizar un depurador de código, que es un programa que permite visualizar los valores de las variables mientras se ejecuta el programa a depurar, y permite la ejecución paso a paso o parar la ejecución en un punto determinado del programa. El depurador habitual en Linux es el `gdb`. La forma de ejecutar el `gdb` con un programa es:

Terminal

```
$ gdb ./hola
```

A partir de ese momento se entra en el depurador, que tiene su propia línea de comandos (como la del terminal pero con otros comandos, obviamente), que empieza con el prompt `(gdb)`.[3] Los comandos más habituales son:

`(gdb) quit`	salir del depurador
`(gdb) run` *argumentos*	ejecutar el programa (`hola`) con esos argumentos
`(gdb) step`	ejecuta una instrucción
`(gdb) next`	igual que `step`, pero si la instrucción es una llamada a una función, pasa a la instrucción siguiente a la llamada (no *entra* en la función)
`(gdb) print` *expr*	visualiza el valor de *expr* (normalmente *expr* será una variable o vector)
`(gdb) display` *expr*	como `print`, pero se visualiza el valor de *expr* en cada paso de la depuración
`(gdb) break` *lugar*	pone un punto de parada (*breakpoint*), que para la ejecución del programa al llegar a ese *lugar*; *lugar* puede ser un número de línea o el nombre de una función
`(gdb) delete` *n*	borrar el *breakpoint* número *n* (a cada *breakpoint* se le asigna automáticamente un número)
`(gdb) list` *lugar*	visualiza por pantalla un trozo de código. *lugar* puede ser un número de línea o un rango de líneas como `10-25`, y puede llevar delante un nombre de fichero: `list hola.cc:3`

Un tutorial interesante sobre la herramienta `gdb`, de los muchos que se pueden encontrar en Internet, es: `http://www.cs.cmu.edu/~gilpin/tutorial/`.

El *memory checker* `valgrind`

En muchas ocasiones, un programa funciona aparentemente bien en una máquina y falla en otra máquina. Normalmente, se debe a una o más de las siguientes causas:

- Accesos incorrectos en vectores y matrices
- Uso de variables sin inicializar
- Accesos a punteros sin memoria asignada
- En general, errores en el código

El programa `valgrind` es un *memory checker*, es decir, un programa que controla los accesos a la memoria que hace otro programa, vigilando entre otras muchas cosas las causas más habituales de fallos de memoria.

Aunque `valgrind` no encuentra todos los fallos de un programa (a veces depende del flujo de ejecución que un fallo se produzca o no), sí permite encontrar muchos fallos, y es recomendable utilizarlo una vez se hayan corregido todos los errores *visibles*. El programa `valgrind` se ejecuta de esta manera:

[3] También es posible utilizar `gdb` con un interfaz gráfico mediante el programa Nemiver.

Terminal

```
$ valgrind ./hola
```

El programa `hola` se ejecuta bajo la supervisión del `valgrind`, que produce un informe en caso de encontrar un fallo. Como en el caso de los errores de compilación, lo recomendable es redirigir la salida de error a un fichero y corregir los fallos uno a uno (a veces el primer fallo provoca todos los demás).

Detalles de implementación

- Cuando se lee un dato con `cin >>` (por ejemplo la opción del menú) y luego se lee una cadena con `getline(cin,cadena)`, puede producirse un comportamiento a priori extraño. Por ejemplo:

  ```
  int a; string s;
  cin >> a; // Entrada: 27\n
  getline(cin,s); // Entrada: una cadena cualquiera\n
  ```

  El problema es que la lectura con `>>` se para en cuanto encuentra un *blanco*, por lo que el `\n` que hay después del 27 se queda sin leer, y el `getline` lee hasta que encuentre un `\n`, así que en ese ejemplo la cadena `s` valdría `""` (tendría tamaño 0) y se quedaría sin leer de la entrada "`una cadena cualquiera\n`".

  La solución a este problema es eliminar siempre el `\n` después de leer con `>>`:

  ```
  int a; string s;
  cin >> a;
  cin.get(); // Eliminar el '\n' (si va pegado al 27)
  getline(cin,s); // Ahora ya va bien
  ```

  Podría haber más caracteres entre el 27 y el `\n`, y ahí las soluciones son unas cuantas,[4] pero en las prácticas no va a suceder, el `\n` estará siempre pegado al número o carácter anterior. Para más información sobre la lectura con `»` y `getline`, consulta el *Tema 2* de teoría.

- En las prácticas es necesario utilizar `string`, que son cadenas de caracteres muy similares a los vectores de caracteres, pero más fáciles de manejar.

  Ejemplo:

  ```
  string s;         // Declaración (se inicializa a cadena vacía, "")
  getline(cin,s);   // Lectura hasta el final de la línea
  cout << s << endl;  // Mostrar por pantalla

  s="hola"; // Asignación

  for (unsigned int i=0; i<s.length(); i++) // Recorrido, s.length() es la longitud
    cout << s[i];  // imprimir un carácter

  s[3]='A';  // Asignación de un carácter
  s=s+',';       // concatena un carácter    "holA,"
  s=s+" mundo";  // concatena una cadena     "holA, mundo"


  if (s <= "hola") // Comparacion de strings
    if (s == "adios")
       ...
  ```

[4] Por ejemplo: `while(cin.get()!='\n');`

- También es necesario utilizar en las prácticas los vectores, que se definen usando un *template* `vector<>`, con el que se pueden declarar vectores de cualquier tipo (del que se ponga entre < y >). Por ejemplo, `vector<Manzana> manzanas;` declara un vector de elementos de tipo `Manzana`.

  Ejemplo:

```
#include <vector>

vector<int> v; // Declaración, en este caso de un vector de enteros

v.push_back(3); // Añadir un elemento al final del vector, redimensionándolo
v[1]=3; // Error, ya que el vector sólo tiene un elemento actualmente
v[0]=2; // Correcto

// Recorrido (en realidad se deberían usar iteradores, pero se puede hacer así en Prog2)
for (unsigned int i=0; i<v.size(); i++)
      v[i]=23;  // Asignación

// Borrado de un elemento en la posición 'pos'

v.erase( v.begin()+pos );  // suponiendo que 'pos' es una posición correcta

// Borrado de todo el vector

v.clear();


int ultimo = v.back();  // obtener el último elemento
v.pop_back();           // eliminar el último elemento
```