Apoyo de programación
lo justo para hacer las prácticas

Jose Oncina

20 de marzo de 2023

1. algunas opciones del compilador g++

`g++ [opciones] infile`

Opciones:

`-o outfile`
: Nombre del ejecutable. Normalmente es igual a `infile`, pero sin la extensión.

`-O0`
: Optimiza. Reduce el tiempo de compilación y hace que el el depurado produzca los resutados adecuados. Es la opción por defecto.

`-O` **o** `-O1`
: Optimiza. Intenta reducir el tamaño del código y el tiempo de ejecución sin comprometer el tiempo de compilación.

`-O2`
: Optimiza más. Aplica todas las optimizaciones que no superen un compromiso entre espacio y velocidad.

`-O3`
: Optimiza aún más. Aplica todas las optimizaciones que son válidas para todas las aplicaciones que siguen el estándar.

`-g`
: Genera información para el depurado (`gdb`). No usar junto con `-O2` o `-O3`.

`-Wall`
: Activa mensajes de advertencia (*warnings*) acerca de construcciones que se consideran cuestionables y son fáciles de evitar. Algunas solo funcionan si está activado `-O2` o `-O3`.

  Ejemplos:

  - `if (x == "abc")`

- `if ( a == b );`

`-Wextra`
Activa más mensajes de advertencia (*warnings*) acerca de construcciones que se consideran cuestionables.

`-Wpedantic`
Activa todas las advertencias requeridas por el `ISO C++`

`-std=`***standard***
Determina el estándar de `C++` a utilizar. Algunas posibilidades son: `c++03`, `c++11` (o `c++0x`), `c++14` (o `c++1y`), `c++17` (o `c++1z`), `c++20` (o `c++2a`)

`-o outfile`
Fichero donde se escribirá el resultado de la compilación

`--version`
Muestra la versión del compilador

Para las prácticas recomiendo usarlo de la siguiente forma:

```
g++ -O3 -Wall -Wextra -std=c++17 -o outfile infile
```

Si vais a depurar cambiar la opción `-O3` por `-g`, o sea:

```
g++ -g -Wall -Wextra -std=c++17 -o outfile infile
```

2. Uso básico del `make`

El comando `make` fue diseñado para determinar automáticamente qué ficheros de un gran programa deben ser recompilados tras una serie de modificaciones.

Para ello se debe escribir un pequeño fichero llamado `makefile` (o `Makefile`) que describe la relación entre los ficheros del programa y los comandos que hay que ejecutar para obtenerlos. Cada una de estas relaciones reciben el nombre de reglas.

Una regla está compuesta de varias líneas. La primera línea indica el objetivo (lo que se desea obtener) y, tras el catacter '`:`' una lista de dependencias. Las siguientes líneas indican como se obtiene el objetivo a partir de las dependencias.

IMPORTANTE: las líneas que indican cómo se obtienen los objetivos han de empezar con un **tabulador** (y no una serie de espacios).

Ejemplo 2.1. *Queremos obtener el ejecutable* ***`main`*** *que se puede obtener a partir del fuente* ***`main.cc`*** *compilándolo como:*

```
g++ -Wall -Wextra -O3 -std=c++17 -o main main.cc
```

El ***`makefile`*** *quedaría como:*

```
main: main.cc
————→g++ -Wall -Wextra -O3 -std=c++17 -o main main.cc
```

En el ejemplo anterior, al ejecutar `make`, el comando mirará si la fecha de modificación del fichero `main` es posterior a la de sus dependencias (`main.cc`). Si no hay ningún fichero que se llame como el objetivo (`main`), se supone que su fecha es un pasado remoto. Si es posterior (es mas nuevo que sus dependencias), deducirá que `main` está actualizado y no hará nada. En otro caso, ejecutará el (los) comando(s) de las líneas siguientes para actualizarlo. O sea, **solo compila si es necesario**.

En el `makefile` se pueden definir variables. El ejemplo anterior se podría escribir como:

```
OPTS = -Wall -Wextra -O3 -std=c++17

main: main.cc
————→g++ ${OPTS} -o main main.cc
```

El `make` se puede usar para mas cosas que para compilar.

Ejemplo 2.2. *Usando el ejemplo anterior, queremos que al hacer* ***`make`*** *no solo se compile el programa (si es necesario) sino que lo ejecute y escriba su resultado en el fichero* ***`salida`****.*

```
OPTS = -Wall -Wextra -O3 -std=c++17

salida: main
————→main > salida

main: main.cc
————→g++ ${OPTS} -o main main.cc
```

En este caso tenemos dos objetivos, si ejecutamos `make main` solo actualizará el programa `main`. Si ejecutamos `make salida` primero actualizará la dependencia de `salida` (o sea `main`), y cuando esté actualizada, actualizará `salida`. Ejecutar `make` (sin ningún objetivo detrás) es equivalente a ejecutar `make salida` ya que `salida` es el objetivo de la primera regla.

A veces se añaden reglas para hacer otras cosas utiles.

Ejemplo 2.3. *Añadid al* ***`makefile`*** *anterior una regla* ***`clean`*** *para borrar los ficheros que no son necesarios (****`main`*** *y* ***`salida`****).*

```
OPTS = -Wall -Wextra -O3 -std=c++17

salida: main
————→main > salida

main: main.cc
————→g++ ${OPTS} -o main main.cc

clean:
————→rm -f main salida
```

Ejemplo 2.4. *Añadid al* `makefile` *anterior una regla* `tar` *para empaquetar los ficheros que sean necesarios para llevarse a otro equipo nuestro trabajo (*`main.cc` *y* `makefile`*).*

```
OPTS = -Wall -Wextra -O3 -std=c++17

salida: main
————————→main > salida

main: main.cc
————————→g++ ${OPTS} -o main main.cc

clean:
————————→rm -f main salida

tar:
————————→tar -czvf main.tar.gz main.cc makefile
```

3. Uso básico del `gnuplot`

`gnuplot` es un programa interactivo para dibujar gráficas que funciona mediante comandos.

Para comenzar a trabajar con él hay que ejecutar `gnuplot` desde un terminal. para salir hay que ejecutar `quit`.

3.1. representación de funciones

Ejemplo 3.1. *Dibujad la función* $\sin(x)$

```
plot sin(x)
```

Ejemplo 3.2. *Dibujad la función* $\sin(x)$ *entre* $x = 0$ *y* $x = 10$

```
plot [0:10] sin(x)
```

Ejemplo 3.3. *Dibujad la función* $\sin(x)$ *junto con la función* $\cos(x)$

```
plot [0:10] sin(x), cos(x)
```

Ejemplo 3.4. *Dibujad la función* $\sin(x)$ *entre* $x = 0$ *y* $x = 10$ *y que solo salga el rango de* $y = -0.5$ *a* $y = 1.5$

```
plot [0:10][-0.5:1.5] sin(x)
```

3.2. representación a partir de datos

Supongamos que tenemos un fichero de texto `data` que contiene:

```
# QuickSort CPU-times in milliseconds:
# Size Average CPU time (ms.)
   32768            2.20
   65536            4.76
  131072            9.92
  262144           20.32
  524288           43.84
 1048576           92.04
 2097152          190.76
 4194304          395.12
```

Cuando se lea este fichero, las líneas que empiezan por '`#`' se consideran comentarios y se saltarán. Los datos deben ir separados por uno o varios caracteres en blanco o tabuladores.

Ejemplo 3.5. *Dibujad los puntos representados en el fichero '`data`'*

```
plot "data"
```

Ejemplo 3.6. *Dibujad los puntos representados en el fichero '`data`' unidos por una línea*

```
plot "data" with lines
```

Ejemplo 3.7. *Dibujad los puntos representados en el fichero '`data`' unidos por una línea pero resaltando los puntos experimentales.*

```
plot "data" with linespoints
```

Ejemplo 3.8. *Dibujad los puntos representados en el fichero '`data`' pero de forma que el eje de las x sea la primera columna y el de das y la segunda.*

```
plot "data" using 2:1 with linespoints
```

El primer número de la opción `using` indica la columna del eje de las x y el segundo la columna del eje de las y. Si el fichero tiene varias columnas puede usarse números mayores que 2.

3.3. títulos y etiquetas

Ejemplo 3.9. *Añadid al ejemplo anterior una etiqueta en el eje de las x que ponga 'time (ms)' y otra, en el eje de las y que ponga 'size'.*

```
set xlabel "time (x)"
set ylabel "size"
plot "data" with linespoints
```

Ejemplo 3.10. *Añadid un título que ponga 'QuickSort CPU-time'.*

```
set title "QuickSort CPU-time"
plot "data" with linespoints
```

Observad que el `xlabel` y `ylabel` anterior se mantienen.

Ejemplo 3.11. *Quitad los títulos y etiquetas de los ejemplos anteriores.*

```
unset title
unset xlabel
unset ylabel
plot "data" with linespoints
```

Ejemplo 3.12. *Haced que la serie lleve por título 'QuickSort'.*

```
plot "data" with linespoints title "QuickSort"
```

3.4. `gnuplot` como calculadora

El `gnuplot` también se puede usar para realizar cálculos.

Ejemplo 3.13. *Sumar* $2+2$.

```
print 2+2
```

Ejemplo 3.14. *Calcular* 2^{16}.

```
print 2**16
```

También se pueden definir variables

Ejemplo 3.15. *Asignad* 3 *a una variable* a*, incrementarla en uno e imprimirla.*

```
a = 3
a = a + 1
print a
```

Se pueden definir funciones sencillas.

Ejemplo 3.16. *Definid una función* double(x) *que devuelva el doble de su argumento y calculad* double(3)

```
double(x) = 2*x
print double(3)
```

En las funciones se pueden usar variable globales.

Ejemplo 3.17. *Definid una función* $\mathrm{lin}(x)$ *que devuelva* $ax+b$ *donde* a *y* b *son otras variables. Calculad el valor de* $\mathrm{lin}(2)$ *cuando* $a=1$ *y* $b=3$ *y cuando* $a=4$ *y* b *no cambia.*

```
lin(x) = a*x + b
a=1
b=3
print lin(x)
a=4
print lin(x)
```

Notad que la función lin$(\cdot)$ no se evalúa cuando se define (por eso las variables a y b pueden estar indefinidas).

3.5. Ajustes por mínimos cuadrados

`gnuplot` también permite hacer ajustes por mínimos cuadrados.

Ejemplo 3.18. *Ajustad los parámetros a y b de la función* $\mathrm{lin}(x) = ax + b$ *a los datos que hay en el fichero '`data`' utilizado mas arriba.*

```
lin(x) = a*x + b
fit lin(x) "data" via a,b
```

Después de hacer ciertos cálculos, `gnuplot` muestra por pantalla, entre otras cosas, la suma de los errores al cuadrado (lo que da una idea de la calidad del ajuste) y los valores para las variables a y b que hacen que la función lin$(\cdot)$ se ajuste mejor a los datos.

Ademas, también asigna a las variables a y b los valores óptimos, por lo que si ejecutamos `print a, b` nos escribirá dichos valores.

Ejemplo 3.19. *representad los datos del fichero '`data`' junto con el ajuste por mínimos cuadrados de de la función lin$(\cdot)$.*

```
plot lin(x) title "linear fit", \
     "data" title "raw data" with linespoint
```

La barra invertida (backslash) es solo para partir líneas demasiado largas.

Ejemplo 3.20. *Ajustad los parámetros a, b y c de la función* $\mathrm{nlogn}(x) = a + bx + cx\log(x)$ *a los datos del fichero '`data`' y representad la función ajustada junto con los datos.*

```
nlogn(x) = a + b*x + c * x * log(x)
fit nlogn(x) "data" via a, b, c
plot nlogn(x) title "n*log(n) fit", \
     "data" title "raw data" with linespoint
```

3.6. Trabajos por lotes

Todo lo que hemos visto hasta ahora se puede hacer de forma que `gnuplot` lea todas las instrucciones de un fichero de texto.

Supongamos que tenemos un fichero de texto llamado `lin.gp` que contiene:

```
lin(x) = a*x + b
fit lin(x) "data" via a,b
plot lin(x) title "linear fit", \
     "data" title "raw data" with linespoint
pause -1 "press a key to continue"
```

Para ejecutarlo y mostrar la gáfica basta con ejecutar

```
gnuplot lin.gp
```

El comando `pause` sirve para que el lote pare en ese punto hasta que se pulse una tecla. De no hacerlo, la gráfica se mostrará y se borrará (al terminar el lote) tan rápido que no se verá.

Ejemplo 3.21. *modificad el lote anterior para que la gráfica la escriba en el fichero* `lin.png` *en formato* `png`

```
set terminal png
set output "lin.png"

lin(x) = a*x + b
fit lin(x) "data" via a,b
plot lin(x) title "linear fit", \
     "data" title "raw data" with linespoint
```

Notad que en este caso no se debe usar el comando `pause` ya que no se va a mostrar nada en la pantalla.

4. Consejos y modismos (*idioms*) en C++

Podeis encontrar muchas reglas de como se debe programas en C++ en: `https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md`

Esta lista esta editata por Bjarne Stroustrup (diseñador del `C++`) y Herb Sutter (destacado experto).

A continuación voy a transcribir algunas de estas reglas.

- la regla `ES.21` indica que no hay que declarar una variable (o constante) hasta que necesites usarla.

  No hacer esto:

  ```
  int x = 7;
  // ... no use of x here ...
  ++x;
  ```

- La `ES.22` indica que no hay que declarar una variable hasta que se tenga un valor para inicializarla.

  No hacer esto:

```
string s;
// ... no use of s here ...
s = "what a waste";
```

Mejor:

```
string s = "what a waste";
```

- La regla NR.2 indica que no hay que empeñarse en que las funciones tengan un solo `return`.

4.1. Palabra clave `auto`

La palabra clave `auto` declara una variable cuyo tipo se deduce de la expresión de inicialización en su declaración.

Ejemplo 4.1. *Intercambia el valor de las variables `a` y `b` (sabemos que son del mismo tipo).*

```
auto tmp = a;
a = b;
b = tmp;
```

`auto` declara la variable `tmp` del mismo tipo que `a`. Si la variable `a` fuese del tipo `int`, `auto tmp = a;` sería equivalente a `int tmp = a;`. Esto funciona independientemente del tipo de `a`.

Ejemplo 4.2. *hacer un bucle que recorra una lista de enteros declarada como: `list<int> l;`*

Sin utilizar `auto`, sería:

```
for( list<int>::iterator p = l.begin(); p != l.end(); ++p ) {
  ...
}
```

Usando `auto`:

```
for( auto p = l.begin(); p != l.end(); ++p ) {
  ...
}
```

Además, si por cualquier motivo cambiamos la declaración de la lista a `list<double> l;`, al usar `auto` no habría que cambiar el resto del código.

4.2. Medición del tiempo

Ejemplo 4.3. *Medir el tiempo que tarda en ejecutarse la función `long_lasting()` en segundos, milisegundos y microsegundos.*

```
using namespace std;
using namespace chrono;
...
auto begin = steady_clock::now();
long_lasting();
auto end = steady_clock::now();
...
cout << "Seconds"
     << duration_cast<seconds>(end-begin).count()
     << endl;
cout << "Milliseconds: "
     << duration_cast<milliseconds>(end-begin).count()
     << endl;
cout << "Microseconds: "
     << duration_cast<microseconds>(end-begin).count()
     << endl;
```

Nota importante: normalmente no se recomienda usar la directiva `using namespace ... ;`, sobre todo si se trata de cabeceras de librerías que se van a incluir en otros programas. El problema es que si las usas estas obligando al programa que incluya ese fichero a usar ese espacio de nombres. Sin embargo, aquí lo vamos a utilizar por hacer los programas mas legibles.

Podéis consultar mas posibilidades en:
`https://en.cppreference.com/w/cpp/chrono/duration`

Otra forma (menos recomendable) de hacerlo :

```
#include <ctime>

using namespace std;
using namespace chrono;
...
auto begin = clock();
long_lasting();
auto end = clock();
...
cout << "Seconds: "
     << (end-start) / CLOCKS_PER_SEC  << endl;
cout << "Millieseconds: "
     << 1000.0 * (end-start) / CLOCKS_PER_SEC << endl;
cout << "Microseconds: "
     << 1E6 * (end-start) / CLOCKS_PER_SEC << endl;
```

4.3. Analizando argumentos

Ejemplo 4.4. *Hacer un programa que busque coches que cumplan ciertas condiciones en una base de de datos almacenada en un fichero. El programa se*

llamará `search_cars` *y admitirá los siguientes argumentos:*

- `-f file`*: fichero donde está la base de datos (obligatorio)*
- `-l length`*: longitud mínima del coche (*`double`*, obligatorio)*
- `-a age`*: edad mínima del coche (*`int`*, opcional, 0 por defecto)*
- `-s`*: muestra solo coches deportivos (opcional)*
- `-h`*: imprime un mensaje de ayuda y termina*

```
#include <fstream>
#include <string>
#include <limits>

using namespace std;

const double SENTINEL = numeric_limits<double>::max();

void show_usage() {
    cout << "Usage: search_car [-s] [-a age] -l length -f file\n"
            "\n"
            "Search for a car in a database.\n"
            "\n"
            "  -s          print only sport cars\n"
            "  -a age      print only sports older than 'age'\n"
            "  -l length   print only cars longer than 'length'\n"
            "  -f file     set database to 'file'\n"
            "  -h          print this help message\n" ;
}

int main( int argc, char *argv[] ) {

    bool is_sport_car = false;
    int age = 0;       // default value for age
    double length = SENTINEL;
    string file_name;

// Parsing arguments

    for( int i = 1; i < argc; i++ ) {
        string arg = argv[i];

        if( arg == "-s" ) {
            is_sport_car = true;
```

```
        } else if( arg == "-a" ) {
            i++;
            if( i >= argc ) {
                cerr << "ERROR: can't read the age." << endl;
                exit(EXIT_FAILURE);
            }
            try {
                age = stoi(argv[i]);
            } catch( ... ) {
                cerr << "ERROR: invalid age" << endl;
                exit(EXIT_FAILURE);
            }

        } else if( arg == "-l" ) {
            i++;
            if( i >= argc ) {
                cerr << "ERROR: can't read the length." << endl;
                exit(EXIT_FAILURE);
            }
            try {
                length = stod(argv[i]);
            } catch( ... ) {
                cerr << "ERROR: invalid length" << endl;
                exit(EXIT_FAILURE);
            }

        } else if( arg == "-f" ) {
            i++;
            if( i >= argc ) {
                cerr << "ERROR: can't read file name." << endl;
                exit(EXIT_FAILURE);
            }
            file_name = argv[i];

        } else if( arg == "-h" ) {
            show_usage();
            exit(EXIT_SUCCESS);
         } else {
             cerr << "ERROR: unknown option '" << arg << "'." << endl;
             show_usage();
             exit(EXIT_FAILURE);
         }

    }

// Processing parameters
```

```
    if( length == SENTINEL ) {
        cout <<  "ERROR: mandatory car length not specified" << endl;
        exit(EXIT_FAILURE);
    }

    if( file_name.empty() ) {
        cerr << "ERROR: missing file name." << endl;
        exit(EXIT_FAILURE);
    }

    ifstream is(file_name);

    if( !is ) {
            cerr << "ERROR: can't open file: " << file_name << endl;
            exit(EXIT_FAILURE);
    }

//  Searching cars ...

    cout << "Searching in data file: " << file_name << endl;
    if( is_sport_car )
        cout << "Searching for sport cars" << endl;

    cout << "Searching for cars older than "
         << age << " years." << endl;
    cout << "Searching for cars longer than "
         << length << " meters." << endl;

    return 0;
}
```

4.4. Leyendo datos de un fichero de texto

Ejemplo 4.5. *Leer una matriz de enteros de un fichero de texto. La primera línea del fichero contiene dos enteros, separados por espacios en blanco, que indican las filas y columnas de la matriz. En líneas sucesivas vienen los elementos de las filas de la matriz, también separados por espacios en blanco.*

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;
```

```
int main() {
    const string file_name = "matrix.dat";

    ifstream is(file_name);

    if( !is ) {
        cerr << "ERROR: can't open file '" << file_name << "'." <<endl;
        exit(EXIT_FAILURE);
    }

    int rows, cols;
    is >> rows >> cols;
    vector< vector<int>> mat( rows, vector<int>( cols ));
    for( int i = 0; i < rows; i++ )
        for( int j = 0; j < cols; j++ )
            is >> mat[i][j];


    for( int i = 0; i < rows; i++ ) {
        for( int j = 0; j < cols; j++ )
            cout << mat[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```