Debug vs. release

En **cómputo de alto rendimiento** es importante distinguir los objetivos del código generado en modo debug o release.

La idea es que una vez que el código está probado en modo debug desactivar las verificaciones.

En el modo debug se requiere:	En modo release se requiere:
• Buscar fallas en la lógica del programa.	• **No** buscar fallas en las lógica del programa.
• Verificar de límites de valores.	• **No** verificar límites de valores.
• Imprimir mensajes de debug.	• **No** imprimir mensajes de debug.
• Insertar información para el debugger (en GCC se hace con **-g**).	• **No** insertar información para el debugger.
• Detectar errores catastróficos (falta al reservar memoria, falla en acceso a disco, etc).	• Detectar errores catastróficos (falta al reservar memoria, falla en acceso a disco, etc). • Que se ejecute lo más rápidamente posible. • Compilar habilitando la optimización del compilador (en GCC con **-O3**).

assert.h

La forma estandar de hacer verificaciones que sólo existan en modo debug es por medio de la función **assert** y del macro **NDEBUG**.

La función **assert** permite verificar condiciones dentro del código. Por ejemplo, en el siguiente código se puede asegurar que no habrá división entre cero.

```
#include <assert.h>

int main() {
    double a = 1;
    double b = 0;

    assert(b != 0);
    double c = a/b;
}
```

En caso de que no se cumpla la condición el programa terminará y se generará un mensaje de error.

Debido a que el programa termina **assert** no debe usarse para verificar entradas del usuario.

La función **assert** debe usarse para verificar la lógica del programa o para buscar bugs.

Una carácterística importante de la función **assert** es que puede desactivarse en el momento de la compilación. Esto se hace definiendo el macro **NDEBUG**.

```
#include <assert.h>

int main() {
    double a = 1;
    double b = 0;

    assert(b != 0);
    double c = a/b;
}
```

```
g++ -o example1 example1.cpp
```

```
#include <assert.h>

int main() {
    double a = 1;
    double b = 0;

    assert(b != 0);
    double c = a/b;
}
```

```
./example1
```

```
g++ -D NDEBUG -o example1 example1.cpp
```

```
#include <assert.h>

int main() {
    double a = 1;
    double b = 0;

    ((void) 0);
    double c = a/b;
}
```

```
./example1
```

Se puede además utilizar el macro NDEBUG para identificar cuando se está compilando en modo debug o en modo release.

```
#include <stdio.h>

int main() {
    printf("Test 1\n");

#if !defined(NDEBUG)
    printf("Test 2\n");
#endif

    printf("Test 3\n");
}
```

```
g++ -o example2 example2.cpp
```

```
#include <stdio.h>

int main() {
    printf("Test 1\n");

    printf("Test 2\n");

    printf("Test 3\n");
}
```

```
./example2
```

```
g++ -D NDEBUG -o example2 example2.cpp
```

```
#include <stdio.h>

int main() {
    printf("Test 1\n");



    printf("Test 3\n");
}
```

```
./example2
```

Ejemplo con matrices

Por ejemplo, para llenar una matriz

```
#include <assert.h>

#define ROWS 10000000
#define COLS 100

void Set(char* matrix, int i, int j, char value) {
    assert((i >= 0) && (i < ROWS));
    assert((j >= 0) && (j < COLS));
    matrix[i*COLS + j] = value;
}

int main() {
    char* matrix = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
        for (int j = 0; j < COLS; ++j)
            Set(matrix, i, j, 0);

    delete [] matrix;
    return 0;
}
```

```
g++ -o example3 example3.cpp
time ./example3
```

```
g++ -D NDEBUG -o example3 example3.cpp
time ./example3
```

Contenido

Makefiles

Se utilizan cuando hay que compilar varios archivos de código fuente.

Es análogo a un *project file* de Visual C++ o Builder.

Asigna un órden de compilación a un proyecto.

Sirve para establecer dependencias entre archivos de código.

Es el estandard “de facto” para compilar programas de C, C++ y fortran en sistemas POSIX (Unix, Linux, FreeBSD, etc.).

El Makefile mínimo

Queremos compilar un proyecto

- **funciones.h**
- **funciones.c**
- **main.c**
- **Makefile**

El archivo makefile contendría:

```
all:
   → gcc -c funciones.c
   → gcc -c main.c
   → gcc -o program funciones.o main.o

clean:
   → rm -f funciones.o main.o program
```

Es estandard que el archivo se llame “Makefile”

Se invoca con:

- **make**

El símbolo ‘→’ indica que se inserta un tabulador.

```
all:
   → gcc -c funciones.c
   → gcc -c main.c
   → gcc -o program funciones.o main.o

clean:
   → rm -f funciones.o main.o program
```

Un texto al inicio de una línea, seguido de dos puntos define un target:

```
all:
```

define al target “all”. Las lineas siguientes que inicien con un tabulador serán las instrucciones para completar este target.

Un target puede tener cualquier nombre. Se puede ejecutar un target en particular con:

`make clean`

El target que se ejecuta al utilizar el comando **make** es el primero que aparece en el Makefile.

En este caso

`make`

es equivalente a

`make all`

Los targets que siempre se suelen definit en un Makefile son: “all” y “clean”.

El Makefile con dependencias

Un target puede además indicar dependencias a otros targets o archivos:

```
all: program

program: funciones.o main.o
   → gcc -o program funciones.o main.o

funciones.o: funciones.c funciones.h
   → gcc -c funciones.c

main.o: main.c funciones.h
   → gcc -c main.c

clean:
   → rm -f funciones.o main.o program
```

Si el target es un archivo, entonces éste se ejecutará solo si el archivo no existe o si es más antiguo que sus dependencias.

Compilando en paralelo

Si se trata de un proyecto grande y se cuenta con una computadora *multi-core* es útil paralelizar la compilación, por ejemplo:

```
make -j 3
```

Compilará los targets del Makefile ejecutando tres renglones de instrucciones a la vez.

¿Preguntas?

Para saber más...

```
man make
```

En la WWW:

http://en.wikipedia.org/wiki/Makefile

http://www.gnu.org/software/make/manual

http://www.google.com/#&q=makefile+tutorial

El siguiente es un ejemplo de como hacer un makefile que sirva para debug o release.

```
#.SILENT:
.PHONY: release debug clean

CXX=g++
RM=rm -f
STRIP=strip
OUTPUT=example3
OBJS=example3.o
RELEASE_CPPFLAGS=-Wall -Wextra -pedantic -DNDEBUG
RELEASE_CXXFLAGS=-flto -O3
RELEASE_LDFLAGS=-flto
DEBUG_CPPFLAGS=-Wall -Wextra -pedantic
DEBUG_CXXFLAGS=-g
DEBUG_LDFLAGS=

release: CPPFLAGS=$(RELEASE_CPPFLAGS)
release: CXXFLAGS=$(RELEASE_CXXFLAGS)
release: LDFLAGS=$(RELEASE_LDFLAGS)
release: $(OUTPUT)
	$(STRIP) $(OUTPUT)

debug: CPPFLAGS=$(DEBUG_CPPFLAGS)
debug: CXXFLAGS=$(DEBUG_CXXFLAGS)
debug: LDFLAGS=$(DEBUG_LDFLAGS)
debug: $(OUTPUT)

clean:
	$(RM) $(OUTPUT) $(OBJS)

$(OUTPUT): $(OBJS)
	$(CXX) -o $@ $(OBJS) $(LDFLAGS)

example3.o: example3.cpp
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -o $@ -c $<
```

```
make debug
time ./example3
```

```
make release
time ./example3
```

Explicación paso a paso del Makefile para C++ (Debug/Release)

Encabezado del Makefile

.SILENT: (comentado): Si se descomenta, suprime la impresión de los comandos ejecutados.

.PHONY: release debug clean: Declara estas reglas como objetivos "falsos", lo que evita conflictos si existen archivos con esos nombres.

Variables generales

CXX=g++: Define el compilador.

RM=rm -f: Comando para borrar archivos.

STRIP=strip: Quita símbolos de depuración del ejecutable.

OUTPUT=example3: Nombre del ejecutable final.

OBJS=example3.o: Lista de objetos a generar.

Flags de compilación y enlace (release)

RELEASE_CPPFLAGS: Advertencias comunes + DNDEBUG para desactivar asserts.

RELEASE_CXXFLAGS: Optimización fuerte y Link Time Optimization (-O3 -flto).

RELEASE_LDFLAGS: Optimización de enlace con -flto.

Flags de compilación y enlace (debug)

DEBUG_CPPFLAGS: Igual que release pero sin -DNDEBUG.

DEBUG_CXXFLAGS: Usa -g para incluir símbolos de depuración.

DEBUG_LDFLAGS: Vacío, sin optimizaciones de enlace.

Regla 'release'

Configura flags de release, compila el ejecutable, y ejecuta strip para reducir el tamaño.

Comando final: strip example3

Regla 'debug'

Compila el programa con símbolos de depuración y sin optimización.

No se hace strip.

Explicación paso a paso del Makefile para C++ (Debug/Release)

Regla 'clean'

Borra el ejecutable y los archivos objeto:

rm -f example3 example3.o

Compilación y enlace final

$(OUTPUT): $(OBJS): Enlaza el ejecutable a partir de los objetos.

example3.o: example3.cpp: Compila el archivo fuente en un objeto.

¿Cómo usar este Makefile?

make release: Compilación optimizada.

make debug: Compilación para depuración.

make clean: Limpieza de archivos generados.