Compilación, Depuración, y el "Preprocessor" de C/C++

TPV2

Samir Genaim

Las fases de compilación

Preprocessing ⇨ **Compilation** ⇨ **Linking**

La fase de preprocessing resuelve los "macros" para cada archivo file.cpp, p.ej., remplaza la directiva #include "file.h" por el contenido del archivo "file.h"

La fase de compilación compila cada archivo file.cpp, después de haber pasado la fase de preprocessing, a un archivo file.obj. Es código binario, pero tiene referencias no resueltas, p.ej., llamadas a métodos en otros archivos o librerías.

La fase de linking recibe todos los archivos .obj, y las librerías usadas (el código binario), y los pone en un archivo ejecutable resolviendo las referencias entre ellos (genera prog.exe en Windows). Las librerías estáticas se incluyen en el ejecutable, y las librerías dinámicas no se incluyen (para poder ejecutar el programa tenemos que tener las librerías dinámicas).

Preprocessor: la directiva #define

```
#define ID(a1,...,an) exp
```

El preprocessor remplaza todas la ocurrencias de ID(e1,...,en) por exp y también remplaza cada ai por ei dentro de exp

```
#define VEL 5
```

remplaza VEL por 5

```
#define CMPS A,B,C,D
```

remplaza CMPS por A,B,C,D

```
#define CMP(e,t) e->addC<t>()
```

Si en el programa tenemos CMP(x,Transfrom) lo remplaza por x->addC<Transfrom>()

Preprocessor: #ifdef, #ifndef

```
#define ID
#undef ID
```

Definer (o barrar la definición de) la "variable" ID

```
#ifdef ID
// parte A
…
#else
// parte B
…
#endif
```

Si ID está definido, incluye la "parte A" en el programa, en otro caso incluye la "parte B". El #else es opcional. También en lugar de #ifdef se puede usar #ifndef como condición "si ID no está definido".

```
…
#ifdef _DEBUG
std::cout << "x is equal to " << x << std::endl;
#endif
…
```

Preprocessor: uso clásico de #ifndef

Un uso clásico de #ifndef es para evitar la inclusion multiple de un archivo .h. Por ejemplo, si file.cpp incluye a file1.h y file2.h, pero file2.h también incluye file1.h, acabamos con 2 copias de file1.h en file.cpp.

Se puede evitar este problema si los archivos .h tienen la siguiente forma, porque la primera inclusión define la "variable" _FILENAME_H_ y así no se incluye otra vez porque la condición "#ifndef _FILENAME_H_" ya no cumple:

```
#ifndef _FILENAME_H_
#define _FILENAME_H_

// contenido del archivo filename.h

#endif
```

Otra solución es poner "#pragma once" al principio de cada .h

Preprocessor: #define via compilador

Se puede definir "variables" del preprocessor a través del compilador. Por ejemplo, si usamos el compilador g++ desde la linea de comandos:

```
> g++ -D_DEBUG main.cpp
```

define _DEBUG como se fuera "#define _DEBUG". En general la opción -DID del compilador es como poner la linea "#define ID" al principio de todos los archivos. En Visual Studio se pueden añadir "variables" del preprocessor en "Project Properties".

Es útil porque podemos activar/borrar código de depuración sin modificar nada, simplemente pasmos -D_DEBUG al compilador

```
#ifdef _DEBUG
std::cout << "x is equal to " << x << std::endl;
#endif
```

Debug mode y Release mode

- Los entornos de desarrollo, como Visual Studio o Eclipse, normalmente tienen la opción de compilar usando "Debug Mode" o "Release Mode".

- Entre otras cosas, en "Release Mode" definen la variable NDEBUG (es decir pasan -DNDEBUG al compilador) y en el caso de "Debug Mode" definen la variable _DEBUG (es decir pasan -D_DEBUG al compilador).

- Usamos "Debug Mode" durante el desarrollo, pero cuando queremos generar el ejecutable final usamos "Release Mode" para quitar todas las instrucciones de depuración, etc.

Debug mode y Release mode

Compila (y ejecuta) este programa usando Release y Debug mode para ver la deferencia

```
#include <iostream>

int main(int, char**) {
    int n = 0;
    int sum = 0;

    std::cout << "Enter an integer number: ";
    std::cin >> n;

    for (int i = 1; i < n; i++) {
        sum += i;
#ifndef NDEBUG
        std::cout << "iter= " << i << ", sum= " << sum << std::endl;
#endif
    }

    std::cout << "Final value of sum = " << sum << std::endl;
    return 0;
}
```

Preprocessor: la directiva #pragma

Aparte de "#pragma once" que se usa para evitar la inclusion multiple, la directiva #pragma se usa para controlar el comportamiento de algunos aspectos del compilador.

Ver más detalles en:

https://en.cppreference.com/w/cpp/preprocessor/impl

assert

```
#include <cassert>
…
assert(cond);
…
```

Es un macro que se usa para comprobar si la expresión "cond" evalúa a true, es decir que alguna condición cumple. Si no cumple avisa con un error correspondiente y sale del programa. Se usa como una manera de depuración.

Ejemplos

```
assert(x != nullptr)
assert(n>0).
```

En el "Release Mode" se eliminan todas las llamadas a assert.

assert: posible implementación

Usando # antes del nombre del parámetro cond, el preprocessor lo remplaza por una cadena de caracteres correspondiente para poder escribirlo ...

```
#ifndef NDEBUG
#define assert(cond) \
   if ( !(cond) ) { \
      std::cerr << "assert(" << #cond << ") failed: " << \
      "line " << __LINE__ << " file " << __FILE__ << std::endl; \
      exit(1); \
   }
#else
#define assert(cond) ((void)0)
#endif
```

El preprocessor los remplaza por el número de linea y el nombre del archivo (hay muchas más variables predefinidas, ver la documentación del preprocessor)

El backslash \ después de cada linea es necesario si la definición del macro se hace en varias lineas.

static_assert

C++ tiene la instrucción static_assert(cond), es como assert pero la condición se tiene que cumplir durante la compilación — cond tiene que ser una expresión constante, es decir se puede calcular durante la compilación.

Ejemplo:

```
template<typename T, typename ...Ts>
inline T* doSomething(Ts &&... args) {
  static_assert( sizeof(T) < 16 );
  static_assert( std::is_base_of_v<Student,T> );
    ...
}
```

T es un tipo de tamaño minor de 16 bytes

T tiene que ser una clase derivada de la clase Student

Preprocessor: más información

Ver más detalles en:

https://en.cppreference.com/w/cpp/preprocessor

Optimizaciones del Compilador

- Los compiladores de C++ son capaces de optimizar el código para mejorar el tiempo de ejecución o el tamaño del ejecutable (pueden generar código mucho más rápido!).

- Desde la línea de comandos se puede controlar el nivel de optimización usando las opciones -O1, -O2 y -O3. Ver la documentación de compilador para más detalles sobre cada nivel.

```
> g++ -O3 main.cpp
```

- En Visual Studio se puede controlar el nivel de optimizaciones en "Project Properties".