Un enfoque para la enseñanza de la depuración de errores en las asignaturas de programación

Sergio Luján-Mora
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante
Carretera de San Vicente del Raspeig s/n
E-03080 San Vicente del Raspeig, Alicante
e-mail: sergio.lujan@ua.es

Resumen

Uno de los aspectos más difíciles de la programación es la depuración de errores. Como dice el aforismo, "errar es de humanos": los errores de programación son muy comunes incluso en los productos comerciales que se "supone" que se ven sometidos a controles de calidad exhaustivos. Por ello, no es de extrañar que cuando se comience a programar por primera vez o se aprenda un nuevo lenguaje de programación se cometan muchos errores, lo que suele ser origen de muchas frustraciones. Sin embargo, aunque se trata de un aspecto importante de la programación, no se le presta mucha atención tanto en la bibliografía sobre programación (existen pocos libros cuyo tema central sea la depuración de errores) como en la docencia. En este artículo mostramos cómo afrontamos este problema en el contexto de la asignatura "Programación y estructuras de datos" en la Universidad de Alicante: comentamos las carencias y malos hábitos que presentan los alumnos, presentamos las soluciones que hemos planteado para que el alumno desarrolle su capacidad de depurar programas y valoramos los resultados obtenidos durante los primeros meses de aplicación.

1. Introducción

La depuración de los programas es un tema que se suele dejar de lado cuando se aprende/enseña a programar debido a diferentes factores: falta de tiempo, complejidad del proceso, etc. Sin embargo, parece algo inherente a la programación (y se asume) que cualquier código contenga errores, lo que puede producir en ocasiones graves problemas. Así, por ejemplo, en la industria del software se suele considerar que lo normal es que haya 15 errores de programación por cada 1000 líneas de programa. Puede parecer una pequeña cantidad, pero en un programa real pueden ser muchos fallos: un comunicado interno de Microsoft afirmaba que Windows 2000 (con unos 40 millones de líneas de código) se pondría a la venta con 63.000 fallos [12], lo que es 10 veces menos de lo que cabría esperar si se aplica el valor de 15 errores por cada 1000 líneas.

Por otro lado, la baja calidad del software supone un grave problema hoy en día: un estudio del Instituto Nacional de Estándares y Tecnología de los Estados Unidos [14] estima que en dicho país el empleo de software con errores supone un coste anual cercano a los 60 mil millones de dólares (daños ocasionados, acciones legales, recursos destinados a la depuración de errores, etc.). Otro dato que confirma está desastrosa situación es que el "60% de los desarrolladores de software en Estados Unidos están involucrados en resolver errores que se podrían haber evitado" [7]. El mismo autor también estima que "sólo 47 días del calendario laboral se destinan a desarrollar el software, mientras que 150 días se dedican a verificar y depurar el software".

Aunque los datos anteriores se refieren a la situación en Estados Unidos, no creemos que la situación en España sea muy distinta. Ello nos lleva a creer que se debería de prestar más atención a la depuración de los programas e integrarla en el proceso educativo. Además, se debería de incidir en lo importante que es obtener software de calidad.

En este artículo mostramos cómo afrontamos este problema en el contexto de la asignatura

"Programación y estructuras de datos" (PED) de la Universidad de Alicante (UA). La depuración de errores es uno de los principales problemas a los que se enfrenta un alumno cuando estudia un nuevo lenguaje de programación. En este artículo mostramos los principales consejos y materiales que ofrecemos a los alumnos para que aprendan a depurar sus programas.

El resto del artículo se ha estructurado de la siguiente forma: en la sección 2 se comenta el tratamiento que recibe la depuración de errores en la bibliografía más extendida sobre los lenguajes de programación C y C++; en la sección 3 se describe la asignatura PED en la que se ha realizado la experiencia que se presenta en este artículo; en la sección 4 se comentan algunas de las carencias y malos hábitos que suelen presentar los alumnos de cara a la depuración de errores; en la sección 5 se presenta cómo abordamos el aprendizaje de la depuración de errores en PED; finalmente, la sección 6 cierra el artículo con las conclusiones.

2. Cómo se enseña a depurar errores en los libros

Muchos libros que tienen como objetivo enseñar a programar en un lenguaje de programación dedican poco o nada de su contenido a la depuración de errores. Por ejemplo, si nos centramos en los lenguajes C y C++ y consultamos algún libro de la bibliografía más extendida:

- En el libro de referencia básico de C [8] no se trata el tema de la depuración de errores en sus casi 300 páginas.
- En [4] sólo se dedican 4 páginas de casi 600 a la depuración de errores.
- En [2] únicamente se dedican 3 páginas de casi 300 a tratar los errores más comunes relacionados con la gestión de memoria.
- En [3], en el capítulo dedicado al preprocesador se explica la aplicación de la compilación condicional a la depuración y de forma dispersa a lo largo de las casi 1000 páginas del libro se comentan los errores sintácticos, los errores de enlazado y los errores lógicos, pero sin profundizar.
- Finalmente, en el libro de referencia de C++ [15] existe un capítulo dedicado al tratamiento de errores y excepciones en tiempo de ejecución, pero no se trata la depuración de errores.

El único libro que conocemos que trata el tema de la depuración de errores es [13], aunque sólo el primer tercio del libro se centra en la depuración de programas en general, ya que el resto del libro trata la depuración de errores en entornos concretos como son los programas para Microsoft Windows, los controles ActiveX o los componentes COM.

En definitiva, la depuración de errores es un tema que se suele dejar de lado. En la mayoría de los libros, se remite al lector a que consulte las secciones sobre compilación y depuración de errores en los manuales de referencia de su compilador. Pero en estos últimos únicamente se explican las herramientas que posee el compilador y cómo emplearlas, pero no se explican los errores de programación que se pueden cometer.

3. Características de la asignatura

La asignatura PED es una asignatura troncal de segundo curso en el plan de estudios de Ingeniería en Informática, Ingeniería Técnica en Informática de Gestión e Ingeniería Técnica en Informática de Sistemas en la UA. Consta de 9 créditos anuales: 4,5 créditos teóricos y otros 4,5 créditos prácticos. Posee como prerrequisitos las asignaturas "Fundamentos de Programación I" y "Fundamentos de Programación II", ambas de primer curso. El descriptor de la asignatura que aparece en el plan de estudios de las tres titulaciones en la UA es el siguiente:

- Estructuras de datos y algoritmos de manipulación.
- Tipos abstractos de datos.
- Diseño recursivo.

En el primer curso, los alumnos estudian los principios básicos de la algoritmia, las estructuras usuales de programación y la representación de los datos en la asignatura "Fundamentos de Programación I" durante el primer cuatrimestre, mientras que en el segundo cuatrimestre estudian un lenguaje de programación concreto (C++ sin la orientación a objetos [10]) en la asignatura "Fundamentos de Programación II".

En la asignatura PED, los alumnos estudian las estructuras de datos básicas (pila, lista, árbol binario, etc.) y algunas más avanzadas (árbol

AVL, árbol 2-3-4, trie, etc.). En las prácticas se emplea el lenguaje C++ [15] para implementar algunas de las estructuras de datos estudiadas. En las últimas prácticas, la implementación de una estructura de datos puede ser muy compleja, con varios miles de líneas de código.

4. Carencias de los alumnos

Muchos alumnos de la asignatura PED presentan una serie de carencias o falta de habilidades:

- No comprenden realmente qué funciones realizan las herramientas de desarrollo (compilador, enlazador, etc.).
- No entienden o no los tienen en cuenta los mensajes de error que generan las herramientas de desarrollo.
- No tienen muy claras las diferencias entre los distintos tipos de errores que existen (de diseño, lógicos, sintácticos, etc.).
- No saben cómo emplear un depurador para corregir los errores de sus programas.
- No son sistemáticos ni tienen paciencia a la hora de corregir los errores.

Estas carencias además se agravan por el comportamiento que suelen presentar los alumnos respecto a los errores en el código:

- No le dan importancia a los errores: "No es importante, si sólo falta declarar la variable y poner un punto y coma".
- No leen los mensajes de error: simplemente se fijan en que línea se ha producido un error.
- Piensan que la tarea del profesor es actuar como depurador: en cuanto se presenta un problema, solicitan la ayuda del profesor para que resuelva el error.
- "Echan balones fuera": a veces el alumno no puede localizar el origen de un error y llega a la convicción de que el código fuente no contiene errores. En esas ocasiones, muchos alumnos piensan que el problema se encuentra en el hardware o en el software (y en especial en el compilador).

5. Aprender a depurar

Debido a la importancia que creemos que posee la depuración de errores y a los problemas que presentan los alumnos, en la asignatura PED hemos incorporado el aprendizaje de técnicas y habilidades relacionadas con la depuración de errores. En esta sección comentamos los principales aspectos que enseñamos a los alumnos.

5.1. No todos los errores son iguales

Antes de intentar solucionar un error hay que saber de qué tipo de error se trata. A los alumnos les presentamos una clasificación de los errores de programación en base al momento en que se presenta u origina el error:

- *Errores de diseño*. Este tipo de errores aparece en el momento de plantear el algoritmo que resuelve el problema al que el alumno se enfrenta. Estos errores son independientes del lenguaje de programación empleado.
- *Errores lógicos*. Estos errores se producen por una incorrecta codificación del algoritmo planteado. Ejemplos de este tipo de errores son: condiciones lógicas mal planteadas, incorrecta elección de los tipos de datos de las variables, etc.
- *Errores sintácticos* (también llamados *errores gramaticales*, *de compilación* o *en tiempo de compilación*). Este tipo de errores impide que el código fuente se compile con éxito (si existen errores no se podrá generar el código objeto). Ejemplos de estos errores son: palabras clave mal escritas, expresiones incompletas, referenciar una variable no declarada, puntuación incorrecta (ausencia de paréntesis, punto y coma), etc.
- *Avisos de compilación*. Además de los mensajes de error, el compilador también puede generar avisos (*warnings*), errores que no son suficientemente graves como para impedir la generación del código objeto. Ejemplos de estos avisos son: empleo de variables no inicializadas, conversiones automáticas del tipo de las variables con una posible pérdida de información, etc.
- *Errores de enlazado*. Estos errores impiden que se genere el ejecutable final a partir de los ficheros con el código objeto. Ejemplos de estos errores son: ausencia de algún fichero objeto, empleo de funciones no definidas, empleo incorrecto de funciones de las librerías del lenguaje, etc.
- *Errores de ejecución* (también llamados *errores en tiempo de ejecución*). Este tipo de

errores impide que el programa se ejecute con éxito. Ejemplos típicos de este tipo de errores son: desbordamiento en una operación aritmética, división por cero, calcular la raíz cuadrada de un número negativo, etc.

Algunos autores defienden otros tipos de clasificaciones. Así, por ejemplo, en [3] se distinguen los errores sintácticos, los errores de enlazado y los errores lógicos, que pueden ser fatales ("[…] hará que un programa falle y que termine de forma prematura") o no fatales ("[…] permitirá que continúe el programa, pero produciendo resultados incorrectos").

Los errores de sintaxis y de enlazado son los más fáciles de detectar y corregir, ya que se genera un mensaje de error que nos avisa del problema. Los errores de ejecución se pueden resolver fácilmente si se genera algún mensaje de error, ya que muchas veces el programa simplemente termina su ejecución bruscamente. Sin embargo, los errores de diseño y lógicos son más difíciles de localizar, ya que pueden pasar desapercibidos. Esta cuestión se la remarcamos repetidamente a los alumnos, ya que muchas veces los alumnos piensan que si han obtenido un programa ejecutable significa que su código no contiene errores.

Por otro lado, en la mayoría de los casos los mensajes de error que producen los compiladores son breves y muy crípticos, lo que no ayuda a corregir los errores. A veces, la interpretación correcta de los mensajes de error suele implicar un conocimiento profundo del lenguaje por parte del programador, lo cual no posee un estudiante que se inicia en el uso de un nuevo lenguaje.

5.2. Mentalízate

Como se puede decir que es "inevitable" que se cometa algún error al programar, aconsejamos a los alumnos que cuenten por anticipado con la necesidad de tener que depurar su código en algún momento.

Además, un objetivo que nos marcamos es que comprendan lo importante que es obtener código sin errores. A los alumnos les describimos numerosos ejemplos [5] sobre las consecuencias que pueden acarrear los errores en el software, pero les resaltamos tres casos por la notoriedad o impacto que tuvieron:

- Entre 1985 y 1987 se produjeron en Canadá y Estados Unidos seis accidentes, con muertes y heridos graves por sobredosis de radiación, debido a un error de software en el ordenador que controlaba una máquina de radioterapia Therac-25 [9]. Bajo ciertas condiciones se producían condiciones de carrera, lo que originaba un funcionamiento incorrecto de la máquina.
- El 4 de junio de 1996 y durante su vuelo inaugural, el nuevo cohete Ariane 5 de la Agencia Espacial Europea se destruyó 40 segundos después de despegar. El cohete (resultado de un proyecto con un coste de 7 mil millones de dólares y más de 10 años de duración) y su carga (no asegurada por ser el primer vuelo) estaban valorados en 500 millones de dólares. La causa del fallo fue un error de software: una conversión errónea de un número en coma flotante de 64 bits a un número entero de 16 bits produjo un desbordamiento en el sistema de guiado y su posterior apagado automático.
- El 23 de septiembre de 1999, la sonda espacial Mars Climate Orbiter de la NASA, valorada en 125 millones de dólares, oficialmente se dio por perdida. La causa de la perdida fue un error en las unidades de medida empleadas en el software: mientras que en un módulo se empleó el sistema de medidas anglosajón (millas), en otro módulo se empleó el sistema métrico decimal (kilómetros). Ambos módulos tenían que comunicarse entre sí, pero no se realizaba ninguna conversión de los valores de un sistema de medida al otro.

5.3. Inténtalo tú mismo

A los alumnos les aconsejamos que depuren su código sin solicitar la ayuda de un compañero o del profesor. Si siempre que tienen un problema piden ayuda a alguien, nunca serán capaces de desarrollar buen código. Programar requiere ciertas habilidades que sólo se pueden lograr con la práctica.

5.4. Estado mental

La depuración de errores es una actividad mental que requiere una gran concentración. Por ello, recomendamos a los alumnos que si no encuentran a veces un error después de varios intentos, a

veces es mejor dejar la búsqueda para un momento posterior. Muchas veces, la fatiga y la frustración impiden localizar un error obvio.

5.5. Localiza el error

Lo más importante en el proceso de depuración es localizar el punto exacto del error. Para ello, una de las técnicas más comunes consiste en modificar el código fuente para que escriba ciertos mensajes para saber por donde discurre el hilo de ejecución o mostrar resultados intermedios para comprobar que los cálculos son correctos y volver a ejecutar el programa. En nuestra asignatura les enseñamos esta técnica, pero les indicamos que aprovechen las ventajas que ofrece la compilación condicional para no tener que modificar el código fuente cada vez que se quiera depurar el código o se desee obtener el programa final. De este modo, a partir del mismo código fuente se puede obtener una versión de depuración y una versión final. Por ejemplo, en la Tabla 1 se muestra un ejemplo de código que contiene instrucciones que sólo se compilan si el identificador `__DEPURACION__` se encuentra definido.

```
#include <iostream>
#define __DEPURACION__

int unaFuncion(void) {
#ifdef __DEPURACION__
cout << "Entra en: int unaFuncion(void)" << endl;
#endif

  int algo;
  /* Instrucciones de la función */

#ifdef __DEPURACION__
cout << "Sale de: int unaFuncion(void)" << endl;
cout << "Devuelve: " << algo << endl;
#endif

  return algo;
}

void main(void) {
#ifdef __DEPURACION__
cout << "Inicio del programa" << endl;
#endif

  /* Instrucciones */
  unaFuncion();

#ifdef __DEPURACION__
cout << "Fin del programa" << endl;
#endif
}
```

Tabla 1. Ejemplo de compilación condicional

5.6. Prueba y error

Una de las técnicas que más emplean los alumnos para depurar su código es la de “prueba y error”, que consiste en el ciclo modificar el código fuente, compilar y probar (ejecutar), que se repite sucesivamente hasta que “parece” que se ha resuelto el problema. A los alumnos les explicamos que esta técnica es “peligrosa” si no se emplea con cuidado y de forma sistemática, ya que en el proceso de resolver un error puede ser que se hayan introducido nuevos errores.

5.7. Cuidado con los mensajes de error

Los mensajes de error que genera un compilador suelen ser crípticos y poco explicativos, lo que ayuda poco a la depuración. Pero si además no se tienen en cuenta las dos siguiente reglas, el proceso de depuración puede ser un "infierno":

- Cuando el compilador genera muchos mensajes de error, puede ser que haya tantos que no quepan en una pantalla y se produzca un desplazamiento. En esa situación, muchos alumnos tienen la mala costumbre de comenzar a resolver los últimos errores, ya que son los que pueden ver en la pantalla. Sin embargo, los errores hay que resolverlos desde el primero hacia el último, ya que al resolver un error pueden desaparecer mensajes de error posteriores que son falsos: "El primer mensaje de error mostrado siempre es un error real; sin embargo, los últimos errores puede ser que no sean reales" [1]. Por ello, aconsejamos a los alumnos que redirijan la salida de error del compilador a un fichero para poder consultarlos desde el primero.
- En muchas ocasiones, los alumnos se centran en la línea del código fuente que el mensaje de error indica que posee un error y se olvidan del resto del código. Sin embargo, un error se puede manifestar en una línea aunque su origen se encuentre en una línea distinta: "El origen de un mensaje de error se puede encontrar en cualquier línea por encima de la línea señalada en el mensaje de error; sin embargo, el origen no puede estar en una línea posterior" [1]. A los alumnos les aconsejamos que cuando depuren tengan una visión global del código fuente y que no se centren exclusivamente en una línea.

5.8. Tu amigo el depurador

Los alumnos suelen huir del depurador, ya que lo consideran "engorroso" de emplear y poco útil. En [3] se deja claro este problema: "[…] Sin embargo, con frecuencia los depuradores son difíciles de utilizar y de comprender, por lo que son rara vez utilizados por los estudiantes de un primer curso de programación".

El origen de esta situación suele ser el desconocimiento: normalmente a los alumnos nadie les ha enseñado con ejemplos cómo emplear un depurador, lo único que han recibido son explicaciones sobre su manejo. Así, en cuanto descubren las ventajas del empleo de los puntos de ruptura (*breakpoints*), la ejecución paso a paso o la visualización de los valores de las variables, no dudan en emplearlo a partir de entonces. Por ello, a los alumnos de PED les enseñamos con ejemplos reales de código cómo se emplea un depurador.

5.9. Conjunto de ficheros de prueba

Aun cuando con un programa se obtengan los resultados correctos (esperados), no se puede asegurar que el programa no contenga errores, ya que algunos errores sólo se producen en situaciones muy concretas. A los alumnos les indicamos que se deben crear sus propios ficheros de prueba que les permitan verificar automáticamente y de forma exhaustiva el correcto funcionamiento de sus programas. Les aconsejamos que empleen estos ficheros conforme finalicen módulos de sus programas y que cuando hayan finalizado e integrado todos los módulos, vuelvan a verificar todos los ficheros para comprobar que no existe algún error producido por la integración de los módulos.

5.10. Contempla todos los posibles casos

Una de las principales causas de error es no contemplar todos los posibles casos porque se cree que es "imposible" que se produzcan. A los alumnos les indicamos que siempre tienen que contemplar todos los posibles casos.

Por ejemplo, en las sentencias condicionales simples (`if`) o múltiples (`switch`) es conveniente proporcionar siempre un caso por defecto (`else` o `default`, respectivamente) para atrapar los posibles errores que se puedan producir, aun cuando se esté completamente seguro de que el programa no contiene errores y se crea que sólo se pueden dar los casos contemplados.

5.11. Los errores son evitables

En la industria del software existe la creencia de que los errores son inevitables en los programas. Debido a ello existe la política de verificar un programa cuando está terminado. Pero este

procedimiento se ha comprobado desde hace años que es muy costoso y poco eficiente [6]. Los principales problemas que presenta esta forma de trabajar son:

- El proceso de depuración consume mucho tiempo.
- El proceso de depuración no asegura que un programa esté libre de errores.

Por ello, a los alumnos les explicamos que con un buen análisis y diseño de su código se pueden evitar muchos errores desde el principio. Los errores no se tienen que dejar para el final: se tiene que abordar su resolución desde el principio.

Finalmente, a los alumnos les proporcionamos una guía [11] que explica los principales errores lógicos, sintácticos y de enlazado que suelen cometer los estudiantes cuando comienzan a trabajar con el lenguaje C++. En esta guía los errores se han clasificado en siete categorías:

- Sobre el fichero *makefile* y la compilación.
- Sobre las directivas de inclusión.
- Sobre las clases.
- Sobre la sobrecarga de los operadores.
- Sobre la memoria.
- Sobre las cadenas.
- Varios.

5.12. Errores más comunes

Al enlazar, no incluir un fichero necesario. Ejemplo:

```
g++ -c unaclase.cc
g++ -c prueba.cc
g++ -o prueba prueba.o
```

Mensaje de error:

```
prueba.o: In function 'main':
prueba.o(.text+0xe): undefined reference to 'UnaClase::UnaClase(void)'
prueba.o(.text+0x2f): undefined reference to 'UnaClase::~UnaClase(void)'
prueba.o(.text+0x4a): undefined reference to 'UnaClase::~UnaClase(void)'
```

Solución: Verificar que en el proceso de enlazado se tienen en cuenta todos los ficheros necesarios.

```
g++ -c unaclase.cc
g++ -c prueba.cc
g++ -o prueba prueba.o unaclase.o
```

Tabla 2. Ejemplo de error de compilación

Confundir la declaración de una función amiga (friend) con los modificadores de visibilidad. Ejemplo:

```
class UnaClase {
  friend:
    int funcionAmiga(void);
  public:
    UnaClase();
    ~UnaClase();
    ...
};
```

Mensaje de error:

```
In file included from unaclase.cc:1:
unaclase.h:9: parse error before '('
```

Solución: El modificador friend se tiene que poner a cada función que sea amiga.

```
class UnaClase {
  friend int funcionAmiga(void);
  public:
    UnaClase();
    ~UnaClase();
    ...
};
```

Tabla 3. Ejemplo de error sobre las clases

El objetivo de la guía es mostrar cómo reconocer y corregir errores comunes y otros poco conocidos de forma rápida y fácil. Para cada error se incluye un enunciado del error, un ejemplo que contiene el error, el mensaje de error que se produce (al compilar, al ejecutar, etc.) y una solución al problema. Por ejemplo, en la Tabla 2 se muestra un ejemplo de error perteneciente a la categoría de errores de compilación y en la Tabla 3 un ejemplo de error perteneciente a la categoría de errores sobre las clases.

6. Resultados obtenidos

Es difícil valorar si la incorporación del estudio de la depuración de errores influye en el rendimiento académico, ya que este es el primer año en que hemos incorporado este tema y no disponemos de datos cuantitativos. Sin embargo, sí que se aprecia un mayor interés en los alumnos por lograr programas sin errores y una disminución en el número de consultas que realizan referentes a su código. Por otro lado, la guía de errores más comunes en C++ [11] ha recibido una buena valoración por parte de los alumnos e incluso algunos alumnos nos han hecho llegar ejemplos que no se habían incluido en la guía.

7. Conclusiones

En este artículo hemos explicado cómo nos planteamos la enseñanza de la depuración de errores dentro de la asignatura "Programación y estructuras de datos". Creemos que la depuración de errores es un aspecto de la programación que no suele recibir la atención que debería. Por ello, en nuestra asignatura abordamos su estudio. Con este artículo pretendemos dar a conocer nuestro planteamiento e intercambiar ideas con la comunidad educativa universitaria.

Como trabajo futuro está pendiente la realización de un estudio para averiguar si la enseñanza de la depuración de errores influye en el rendimiento académico del alumno y el desarrollo de una metodología de depuración.

Referencias

[1] Carter, Paul. *How To Debug Programs*. 2001. Disponible en Internet: http://www.drpaulcarter.com/cs/debug.php.

[2] De Pereda, Miguel y Matero, Javier. *Programación Orientada a Objetos con C++*. Anaya Multimedia, 1994.

[3] Deitel, H. M. y Deitel, P. J. *Cómo programar en C/C++*. Prentice Hall Hispanoamericana, 2ª edición, 1995.

[4] Gottfried, Byron S. *Programación en C*. McGraw-Hill, 1991.

[5] Huckle, Thomas. *Collection of Software Bugs*. Disponible en Internet: http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html.

[6] Humphrey, Watts S. *Comments on Software Quality*. En National Conference of Commissioners on Uniform State Laws for their Annual Meeting, 1997. Disponible en Internet: http://www.2bguide.com/docs/whsq.html.

[7] Jones, Capers. *The Impact of Poor Quality and Canceled Projects on the Software Labor Shortage*. Informe técnico, Software Productivity Research, Inc., 1998.

[8] Kernighan, Brian W. y Ritchie, Dennis M. *El lenguaje de programación C*. Prentice Hall Hispanoamericana, 1986.

[9] Leveson, Nancy y Turner, Clark S. *An Investigation of the Therac-25 Accidents*. IEEE Computer, Vol. 26, No. 7, julio 1993, páginas 18-41.

[10] Llopis Pascual, Fernando y Pérez López, Ernesto. *C++-OO como lenguaje introductorio a la programación*. En VII Jornadas de Enseñanza Universitaria de la Informática, páginas 299-304, 2001.

[11] Luján Mora, Sergio. *Errores más comunes en C++*. Disponible en Internet: http://www.dlsi.ua.es/~slujan/files/errores.pdf.

[12] Minasi, Mark. *¿Sirven para algo las versiones betas?* Windows 2000 Magazine 47, 2000. Disponible en Internet: http://www.windowstimag.com/atrasados/2000/47_nov00/articulos/engarde.htm.

[13] Pappas, Chris H. y Murray, William H. *C++. Sin errores*. McGraw-Hill, 2001.

[14] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Informe técnico, National Institute of Standards & Technology (NIST), PR 02-3, 2002.

[15] Stroustrup, Bjarne. *El Lenguaje de Programación C++*. Addison Wesley, 2002.