¿Programación basada en contratos para C++17?

using std::cpp 2015

J. Daniel Garcia
josedaniel.garcia@uc3m.es

Grupo ARCOS
Universidad Carlos III de Madrid

18 de noviembre de 2015

Aviso

Reconocimientos

- Trabajo realizado antes de C++11.
  - Thorsten Ottosen, Lawerence Crowl, Dave Abrahams, . . .

Reconocimientos

- Trabajo realizado antes de C++11.
  - Thorsten Ottosen, Lawerence Crowl, Dave Abrahams, . . .
- Y después de C++11.
  - Bloomberg: John Lakos, Alisdair Meredith, Nathan Myers, . . .
  - Microsoft: Gabriel Dos Reis, Herb Sutter, . . .
  - Morgan Stanley: Bjarne Stroustrup.
  - Otros: Lawrence Crowl, Walter Brown, . . .
  - UC3M: J. Daniel García.

Reconocimientos

- Trabajo realizado antes de C++11.
  - Thorsten Ottosen, Lawerence Crowl, Dave Abrahams, ...
- Y después de C++11.
  - Bloomberg: John Lakos, Alisdair Meredith, Nathan Myers, ...
  - Microsoft: Gabriel Dos Reis, Herb Sutter, ...
  - Morgan Stanley: Bjarne Stroustrup.
  - Otros: Lawrence Crowl, Walter Brown, ...
  - UC3M: J. Daniel García.
- Y por supuesto, todo empezó con:
  - C.A.R Hoare.

Dos propiedades relacionadas

- En el diseño de una biblioteca hay una tensión entre dos propiedades: *robustez* y *corrección*.

Dos propiedades relacionadas

- En el diseño de una biblioteca hay una tensión entre dos propiedades: *robustez* y *corrección*.

- **Robustez**: Capacidad de un componente software para reaccionar en condiciones anormales.
  - ¿Qué pasa si al añadir un elemento a un contenedor no se puede obtener memoria dinámica?

Dos propiedades relacionadas

- En el diseño de una biblioteca hay una tensión entre dos propiedades: *robustez* y *corrección*.
- **Robustez**: Capacidad de un componente software para reaccionar en condiciones anormales.
  - ¿Qué pasa si al añadir un elemento a un contenedor no se puede obtener memoria dinámica?
- **Corrección**: Grado en el que un componente software cumple con sus especificaciones.
  - ¿Qué pasa si se accede a un contenedor fuera de rango?
  - ¿Qué pasa si paso un puntero nulo a **strcat**?

Robustez en la biblioteca estándar de C++

- La *robustez* requiere la identificación y tratamiento de situaciones **anormales**.
  - Estas situaciones ocurren en programas totalmente correctos.

Robustez en la biblioteca estándar de C++

- La *robustez* requiere la identificación y tratamiento de situaciones **anormales**.
  - Estas situaciones ocurren en programas totalmente correctos.
  - Buen ejemplo: No se puede obtener memoria dinámica.

Robustez en la biblioteca estándar de C++

- La *robustez* requiere la identificación y tratamiento de situaciones **anormales**.
  - Estas situaciones ocurren en programas totalmente correctos.
  - Buen ejemplo: No se puede obtener memoria dinámica.
  - Abominación: Fin de fichero.

Robustez en la biblioteca estándar de C++

- La *robustez* requiere la identificación y tratamiento de situaciones **anormales**.
  - Estas situaciones ocurren en programas totalmente correctos.
  - Buen ejemplo: No se puede obtener memoria dinámica.
  - Abominación: Fin de fichero.
    - ¿De verdad creías que eso no iba a pasar?

Robustez en la biblioteca estándar de C++

- La *robustez* requiere la identificación y tratamiento de situaciones **anormales**.
  - Estas situaciones ocurren en programas totalmente correctos.
  - Buen ejemplo: No se puede obtener memoria dinámica.
  - Abominación: Fin de fichero.
    - ¿De verdad creías que eso no iba a pasar?

Robustez en la biblioteca estándar de C++

- La *robustez* requiere la identificación y tratamiento de situaciones **anormales**.
  - Estas situaciones ocurren en programas totalmente correctos.
  - Buen ejemplo: No se puede obtener memoria dinámica.
  - Abominación: Fin de fichero.
    - ¿De verdad creías que eso no iba a pasar?
- ¿Qué hace la biblioteca estándar en C++?
  I. Especifica la condición que causa la situación.
  II. La excepción que se lanzará para notificar de la situación.

Throws: **bad_alloc** if the storage cannot be obtained.

Corrección

- La *corrección* está relacionada con encontrar defectos en un programa.

Corrección

- La *corrección* está relacionada con encontrar defectos en un programa.
- Un programa correcto no tiene defectos.

Corrección

- La *corrección* está relacionada con encontrar defectos en un programa.
- Un programa correcto no tiene defectos.

- ¿Qué hacen los matemáticos?
  - Definen el *dominio* y el *recorrido* de la función.

Corrección

- La *corrección* está relacionada con encontrar defectos en un programa.
- Un programa correcto no tiene defectos.

- ¿Qué hacen los matemáticos?
  - Definen el *dominio* y el *recorrido* de la función.

- ¿Qué podemos hacer nosotros?
  - Definir las *precondiciones* bajo las cuales se puede ejecutar de forma segura la operación.
  - Definir las *postcondiciones* que se cumplirán tras la ejecución de la operación.

Aproximaciones frente a la corrección

- Entonces ¿Qué podemos hacer con la corrección?

Aproximaciones frente a la corrección

- Entonces ¿Qué podemos hacer con la corrección?
  - No hacer nada.
    - Más habitual de lo que uno pueda creer.

Aproximaciones frente a la corrección

- Entonces ¿Qué podemos hacer con la corrección?
  - No hacer nada.
    - Más habitual de lo que uno pueda creer.
  - Documentar las precondiciones y postcondiciones.
    - Muy habitual.
    - Es lo que hace la biblioteca estándar de C++.
    - No deja de ser documentación (ni más, ni menos)

Aproximaciones frente a la corrección

- Entonces ¿Qué podemos hacer con la corrección?
  - No hacer nada.
    - Más habitual de lo que uno pueda creer.
  - Documentar las precondiciones y postcondiciones.
    - Muy habitual.
    - Es lo que hace la biblioteca estándar de C++.
    - No deja de ser documentación (ni más, ni menos)
  - Comprobar de forma defensiva las precondiciones y postcondiciones.
    - ¡No te fíes de tus clientes!
    - Llevada al extremo puede suponer un sobrecoste importante.
    - Se puede hacer condicionalmente solamente en modo de depuración. (todo lleno de **assert()**).

Aproximaciones frente a la corrección

- Entonces ¿Qué podemos hacer con la corrección?
  - No hacer nada.
    - Más habitual de lo que uno pueda creer.
  - Documentar las precondiciones y postcondiciones.
    - Muy habitual.
    - Es lo que hace la biblioteca estándar de C++.
    - No deja de ser documentación (ni más, ni menos)
  - Comprobar de forma defensiva las precondiciones y postcondiciones.
    - ¡No te fíes de tus clientes!
    - Llevada al extremo puede suponer un sobrecoste importante.
    - Se puede hacer condicionalmente solamente en modo de depuración. (todo lleno de **assert()**).
  - Dar soporte al análisis estático de código.
    - Éxito variable en distintos lenguajes (Eiffel, Spark, Ada2012, C#, . . .).

¿Qué hace la biblioteca estándar de C++?

- Sección 17.4.6.11:

> Violation of the preconditions specified in a function's Requires: paragraph results in undefined behavior unless the functions Throws: paragraph specifies throwing an exception when the precondition is violated.

- Efecto práctico → dos enfoques.
  - Algunas veces se lanza una excepción.
  - Otras veces se llega a comportamiento no definido.

¿Qué es un contrato?

- Conjunto de precondiciones, postcondiciones e invariantes asociadas a las operaciones de un programa.
  - Precondiciones: Deben satisfacerse antes de invocar una operación.
    - Para el cliente: Requisitos que debe cumplir antes de la invocación.
    - Para la operación: Asunciones que puede hacer la implementación.
  - Postcondiciones: Deben satisfacerse tras la ejecución de la operación.
    - Para el cliente: Asunciones que puede hacer el cliente tras la invocación.
    - Para la operación: Requisitos que debe cumplir después de finalizar.
  - Invariantes: Deben satisfacerse siempre.

¿Qué puede pasar si no se cumple el contrato?

- El programa finaliza de forma anormal (program crash versus system crash).
  - Podría ser peor.

¿Qué puede pasar si no se cumple el contrato?

- El programa finaliza de forma anormal (program crash versus system crash).
  - Podría ser peor.
- Mala prensa (p.ej.: filtración de seguridad).
  - Podría ser peor.

¿Qué puede pasar si no se cumple el contrato?

- El programa finaliza de forma anormal (program crash versus system crash).
  - Podría ser peor.
- Mala prensa (p.ej.: filtración de seguridad).
  - Podría ser peor.
- Desastre económico.
  - Podría ser peor.

¿Qué puede pasar si no se cumple el contrato?

- El programa finaliza de forma anormal (program crash versus system crash).
  - Podría ser peor.
- Mala prensa (p.ej.: filtración de seguridad).
  - Podría ser peor.
- Desastre económico.
  - Podría ser peor.
- Pérdida de vidas.

¿Qué puede pasar si no se cumple el contrato?

- El programa finaliza de forma anormal (program crash versus system crash).
  - Podría ser peor.
- Mala prensa (p.ej.: filtración de seguridad).
  - Podría ser peor.
- Desastre económico.
  - Podría ser peor.
- Pérdida de vidas.
- Y además: comportamiento no definido.

Principios de diseño

- Soporte de lenguaje para programación basada en contratos.
  - Demasiado tiempo intentando una solución de biblioteca.
  - Una solución de lenguaje permite análisis estático, optimizaciones y anti-optimizaciones.

Principios de diseño

- Soporte de lenguaje para programación basada en contratos.
  - Demasiado tiempo intentando una solución de biblioteca.
  - Una solución de lenguaje permite análisis estático, optimizaciones y anti-optimizaciones.

- Un programa correcto al que se le eliminan todos los contratos debería comportarse exactamente igual.

Principios de diseño

- Soporte de lenguaje para programación basada en contratos.
  - Demasiado tiempo intentando una solución de biblioteca.
  - Una solución de lenguaje permite análisis estático, optimizaciones y anti-optimizaciones.
- Un programa correcto al que se le eliminan todos los contratos debería comportarse exactamente igual.
- Debe poder usarse especialmente en sistemas críticos con restricciones.
  - Nos guste o no hay sistemas donde no se pueden usar excepciones

Elementos de diseño

- ¿Cómo se especifican los contratos? ¿En la declaración? ¿En la definición?
- ¿Cuándo se comprueban los contratos?
- ¿Puede haber múltiples niveles de comprobación?
- ¿Qué pasa cuando se rompe un contrato?
- ¿Qué ocurre con las funciones **noexcept**?

Idea general

- Sintaxis basada en atributos.
  - No alteran semánticamente el programa.

Idea general

- Sintaxis basada en atributos.
  - No alteran semánticamente el programa.

- Precondiciones:

  [[expects: condicion]]

Idea general

- Sintaxis basada en atributos.
  - No alteran semánticamente el programa.

- Precondiciones:

  [[expects: condicion]]

- Postcondiciones:

  [[ensures: condicion]]

Idea general

- Sintaxis basada en atributos.
  - No alteran semánticamente el programa.
- Precondiciones:

  [[expects: condicion]]
- Postcondiciones:

  [[ensures: condicion]]
- Invariantes: No consideradas.

Precondiciones en la declaración

- La declaración de una operación puede incluir precondiciones.

```
template <typename T>
T & mi_vector::operator[](int  i )
   [[ expects: 0<=i && i<size() ]]
{
   return v[ i ];
}
```

Precondiciones en la declaración

- La declaración de una operación puede incluir precondiciones.

```
template <typename T>
T & mi_vector::operator[](int  i )
   [[ expects: 0<=i && i<size() ]]
{
   return v[ i ];
}
```

- La precondición se evalúa lógicamente en el contexto del llamante.
  - Solamente puede usar la interfaz pública.
  - No debería tener efectos laterales.

Postcondiciones en la declaración

- Las declaración de una operación puede incluir postcondiciones.

```
void queue::push(int val)
   [[ expects: ! full () ]]
   [[ ensures: !empty() ]];
```

Postcondiciones en la declaración

- Las declaración de una operación puede incluir postcondiciones.

```
void queue::push(int val)
  [[ expects: ! full () ]]
  [[ ensures: !empty() ]];
```

- La postcondición se evalúa lógicamente en el contexto del llamante.
  - Solamente puede usar la interfaz pública.
  - No debería tener efectos laterales.

Comprobaciones en la implementación

- El cuerpo de una operación puede incluir comprobaciones.

```
int f(const vector<int> & v) {
  int r=0;
  for (auto x : v) {
    [[check: x>=0]];
    r = max(r,x);
  }
}
```

Comprobaciones en la implementación

- El cuerpo de una operación puede incluir comprobaciones.

```
int f(const vector<int> & v) {
  int r=0;
  for (auto x : v) {
    [[check: x>=0]];
    r = max(r,x);
  }
}
```

- Razones para usar comprobaciones en la implementación:
  - Las comprobaciones requieren acceso a detalles de implementación no visibles en la interfaz pública.
  - La comprobación en la implementación requiere una menor complejidad computacional.

Niveles

- Se puede realizar la compilación con distintos niveles de comprobación:
  - **off**: No se realiza ninguna comprobación.
  - **min**: Comprobaciones críticas y poco costosas.
  - **default**: La mayoría de las comprobaciones.
  - **max**: Incluye todas las comprobaciones.

Niveles

- Se puede realizar la compilación con distintos niveles de comprobación:
  - **off**: No se realiza ninguna comprobación.
  - **min**: Comprobaciones críticas y poco costosas.
  - **default**: La mayoría de las comprobaciones.
  - **max**: Incluye todas las comprobaciones.

- El nivel de comprobación es un parámetro de compilación que se pude fijar por unidad de traducción.
  - Se podría hacer por módulo (si/cuando tengamos módulos).

Especificando el nivel del contrato

- Se puede especificar el nivel de comprobación de un contrato.
  - Si no se indica nada, el nivel es **default**

```
double * find(double * v, int  sz)
   [[ expects: v!=nullptr;  level :min]]
   [[ expects: sz>0]]
   [[ expects: is_sorted(v,v+sz);  level :max]];
```

Efectos de la violación de un contrato

- Terminación rápida (*fail-fast*) o terminación (*terminate*).
  - Lo único que puedes hacer en algunos sistemas.

Efectos de la violación de un contrato

- Terminación rápida (*fail-fast*) o terminación (*terminate*).
  - Lo único que puedes hacer en algunos sistemas.
- Tratar y continuar.
  - Registrar defectos menores en archivo de log.
  - Gestión de defectos en plug-ins.
  - Guardar estado en un fichero y terminar limpiamente.

Efectos de la violación de un contrato

- Terminación rápida (*fail-fast*) o terminación (*terminate*).
  - Lo único que puedes hacer en algunos sistemas.
- Tratar y continuar.
  - Registrar defectos menores en archivo de log.
  - Gestión de defectos en plug-ins.
  - Guardar estado en un fichero y terminar limpiamente.
- Informar lanzando una excepción predefinida (**broken_contract**).

Efectos de la violación de un contrato

- Terminación rápida (*fail-fast*) o terminación (*terminate*).
  - Lo único que puedes hacer en algunos sistemas.
- Tratar y continuar.
  - Registrar defectos menores en archivo de log.
  - Gestión de defectos en plug-ins.
  - Guardar estado en un fichero y terminar limpiamente.
- Informar lanzando una excepción predefinida (**broken_contract**).
- Informar lanzando una excepción especificada por el contrato.

Efectos de la violación de un contrato

- Terminación rápida (*fail-fast*) o terminación (*terminate*).
  - Lo único que puedes hacer en algunos sistemas.
- Tratar y continuar.
  - Registrar defectos menores en archivo de log.
  - Gestión de defectos en plug-ins.
  - Guardar estado en un fichero y terminar limpiamente.
- Informar lanzando una excepción predefinida (**broken_contract**).
- Informar lanzando una excepción especificada por el contrato.
- El efecto es una propiedad del sistema de construcción y no se puede alterar desde el código.

Comprobación de precondiciones

- Se aplica en el contexto del código cliente.

Comprobación de precondiciones

- Se aplica en el contexto del código cliente.

1. Se evalúan los argumentos de la función.

Comprobación de precondiciones

- Se aplica en el contexto del código cliente.

1. Se evalúan los argumentos de la función.
2. Se evalúa cada condición cuyo nivel es igual o inferior que el de la unidad de traducción.

Comprobación de precondiciones

- Se aplica en el contexto del código cliente.

1. Se evalúan los argumentos de la función.
2. Se evalúa cada condición cuyo nivel es igual o inferior que el de la unidad de traducción.
3. Si alguna condición se viola se opta por el efecto establecido para la unidad de traducción.

Comprobación de precondiciones

- Se aplica en el contexto del código cliente.

1. Se evalúan los argumentos de la función.
2. Se evalúa cada condición cuyo nivel es igual o inferior que el de la unidad de traducción.
3. Si alguna condición se viola se opta por el efecto establecido para la unidad de traducción.
4. Se ejecuta el cuerpo de la función.

Comprobación de postcondiciones

- Se aplica en el contexto del código cliente.

Comprobación de postcondiciones

- Se aplica en el contexto del código cliente.

1. Se ejecuta el cuerpo de la función y se evalúa al expresión de **return**.

Comprobación de postcondiciones

- Se aplica en el contexto del código cliente.

1. Se ejecuta el cuerpo de la función y se evalúa al expresión de **return**.

2. Se evalúa cada condición cuyo nivel es igual o inferior que el de la unidad de traducción.

Comprobación de postcondiciones

- Se aplica en el contexto del código cliente.

1. Se ejecuta el cuerpo de la función y se evalúa al expresión de **return**.
2. Se evalúa cada condición cuyo nivel es igual o inferior que el de la unidad de traducción.
3. Si alguna condición se viola se opta por el efecto establecido para la unidad de traducción.

Comprobación de postcondiciones

- Se aplica en el contexto del código cliente.

1. Se ejecuta el cuerpo de la función y se evalúa al expresión de **return**.
2. Se evalúa cada condición cuyo nivel es igual o inferior que el de la unidad de traducción.
3. Si alguna condición se viola se opta por el efecto establecido para la unidad de traducción.
4. Se continúa la ejecución en el código cliente.

Funciones **noexcept**

- C++11 trajo el marcado **noexcept** que resuelve problemas con la semántica de movimiento.
  - Además abre la puerta a optimizaciones.

Funciones **noexcept**

- C++11 trajo el marcado **noexcept** que resuelve problemas con la semántica de movimiento.
  - Además abre la puerta a optimizaciones.
- ¿Podemos especificar el contrato de una función **noexcept**?

Funciones **noexcept**

- C++11 trajo el marcado **noexcept** que resuelve problemas con la semántica de movimiento.
  - Además abre la puerta a optimizaciones.
- ¿Podemos especificar el contrato de una función **noexcept**?

```
template <typename T>
T & my_vector::operator[](int i) noexcept
   [[ expects: 0<=i && i<size() ]]
{
   return buf[ i ];
}
```

Funciones **noexcept**

- C++11 trajo el marcado **noexcept** que resuelve problemas con la semántica de movimiento.
  - Además abre la puerta a optimizaciones.
- ¿Podemos especificar el contrato de una función **noexcept**?

```
template <typename T>
T & my_vector::operator[](int i) noexcept
   [[ expects: 0<=i && i<size() ]]
{
   return buf[ i ];
}
```

- La violación de un contrato en una función **noexcept** siempre termina la aplicación (**terminate()** o *fail-fast*).

Resumen

- Visión general de contratos.

Resumen

- Visión general de contratos.
  - Se pueden añadir muchos detalles una vez se alcance consenso en la visión general.

Resumen

- Visión general de contratos.
  - Se pueden añadir muchos detalles una vez se alcance consenso en la visión general.
- Ventajas de una solución basada en el lenguaje.

Resumen

- Visión general de contratos.
  - Se pueden añadir muchos detalles una vez se alcance consenso en la visión general.

- Ventajas de una solución basada en el lenguaje.
  - Permite eliminar comprobaciones innecesarias.

Resumen

- Visión general de contratos.
  - Se pueden añadir muchos detalles una vez se alcance consenso en la visión general.

- Ventajas de una solución basada en el lenguaje.
  - Permite eliminar comprobaciones innecesarias.
  - Permite prevenir optimizaciones con problemas de seguridad.

Resumen

- Visión general de contratos.
  - Se pueden añadir muchos detalles una vez se alcance consenso en la visión general.

- Ventajas de una solución basada en el lenguaje.
  - Permite eliminar comprobaciones innecesarias.
  - Permite prevenir optimizaciones con problemas de seguridad.
  - Mejora importante para el análisis estático de código.

Resumen

- Visión general de contratos.
  - Se pueden añadir muchos detalles una vez se alcance consenso en la visión general.

- Ventajas de una solución basada en el lenguaje.
  - Permite eliminar comprobaciones innecesarias.
  - Permite prevenir optimizaciones con problemas de seguridad.
  - Mejora importante para el análisis estático de código.
  - Compatible con sistemas que no pueden usar excepciones.

Lista incompleta de referencias

- C.A.R Hoare. **Hints on programming language design**. Technical Report, Stanford University, CA, USA. 1973.
- J. Daniel García. **Exploring the design space of contracts specification for C++**. N4110. Julio 2014.
- J. Daniel García. **C++ language support for contract programming**. N4293. Noviembre 2014.
- Gabriel Dos Reis, J. Daniel García, Francesco Logozzo, Manuel Fahndrich, Shuvendu Lahri. **Simple contracts for C++**. N4415. Abril 2015.
- J. Daniel García. **Three interesting questions about contracts**. P0166R0. Noviembre 2015.

¿Programación basada en contratos para C++17?

using std::cpp 2015

J. Daniel Garcia
josedaniel.garcia@uc3m.es

Grupo ARCOS
Universidad Carlos III de Madrid

18 de noviembre de 2015