Errores más comunes en C++

Sergio Luján Mora

16 de mayo de 2003

1. Introducción

En este documento se han recogido los errores más habituales que comete una persona cuando comienza a programar con el lenguaje de programación C++. Los errores se han clasificado en las siguientes 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.

Los mensajes de error pueden cambiar de una versión de compilador a otro. Por ejemplo, el siguiente código contiene dos errores, porque se confunde el manejo de objetos y punteros a objetos:

```
UnaClase a, *b;

b = &a;
cout << a->propiedad;
cout << b.propiedad;
```

Si se compila con el compilador `g++` 2.95.4 de GNU se obtiene el siguiente mensaje de error:

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:12: base operand of ‘->’ has non-pointer type ‘UnaClase’
prueba.cc:13: request for member ‘propiedad’ in ‘b’, which is of non-aggregate
              type ‘UnaClase *’
```

Si se compila con el compilador `bcc32` 5.5.1 de Borland se obtiene el siguiente mensaje de error:

```
Error E2288 prueba.cc 12: Pointer to structure required on left side of -> or ->*
in function main()
Error E2294 prueba.cc 13: Structure required on left side of . or .* in function
main()
```

Como se ha comentado previamente, el formato de los mensajes de error cambia de un compilador a otro, aunque suelen tener un estructura similar. En general, cada mensaje de error consta de:

- El nombre del archivo donde se ha detectado el error.
- El número de línea dentro del archivo donde se ha detectado el error.
- Un número de referencia de un tipo de error.
- Una breve descripción del error.

Por otro lado, el mensaje de error también puede cambiar si previamente han aparecido otros errores durante el proceso de compilación (pueden aparecer errores donde realmente no hay errores o pueden desaparecer errores verdaderos). Por ejemplo, el siguiente código:

```
...
UnaClase&
UnaClase::operator=(Unaclase& a)
{
  ...
}
...
```

produce los siguientes mensajes de error, de los cuales sólo el primero es válido (el resto son falsos errores inducidos por el primer error):

```
unaclase.cc:43: ‘Unaclase’ was not declared in this scope
unaclase.cc:43: ‘a’ was not declared in this scope
unaclase.cc:44: declaration of ‘operator =’ as non-function
unaclase.cc:44: invalid declarator
unaclase.cc:44: syntax error before ‘{’
unaclase.cc:46: ANSI C++ forbids declaration ‘dim’ with no type
unaclase.cc:46: ‘a’ was not declared in this scope
unaclase.cc:47: ANSI C++ forbids declaration ‘v’ with no type
unaclase.cc:47: invalid use of member ‘UnaClase::dim’
unaclase.cc:48: parse error before ‘for’
unaclase.cc:48: invalid use of member ‘UnaClase::dim’
unaclase.cc:48: parse error before ‘;’
unaclase.cc:48: syntax error before ‘++’
```

El error se produce porque `Unaclase` no está definido: cambiando `Unaclase` por `UnaClase` se resuelve el problema y desaparecen todos los mensajes de error. Conclusión: los errores hay que resolverlos desde el principio hasta el final del código, ya que al resolver un error pueden desaparecer mensajes de error "falsos".

Los mensajes de error que se recogen en este documento han sido generados por el compilador `g++` de GNU versión 2.95.4. Se ha intentado buscar ejemplos sencillos para mostrar la esencia del problema. Debido a ello, en algunos casos pueden parecer "estúpidos" los ejemplos.

Los errores más difíciles de detectar son los relacionados con el manejo de memoria. En especial los de violación de segmento (`Segmentation fault`), ya que no siempre se produce el mensaje de error (lo que no significa que no exista un error). Como la memoria se reserva por trozos (por ejemplo, de 16 en 16 bytes), una operación incorrecta que invada una zona de memoria no válida puede pasar siempre inadvertida. Otras veces, el mensaje de error se produce después del punto donde existe el error (el error se manifiesta en un punto donde realmente no existe). Para resolver este tipo de errores, lo mejor es emplear un depurador y realizar una traza del código desde el principio.

2. Sobre el fichero makefile y la compilación

- Compilar código en C++ con el compilador de C (`gcc`). Ejemplo:

```
gcc -c unaclase.cc
gcc -c prueba.cc
gcc -o prueba prueba.o unaclase.o
```

  Mensaje de error: En el proceso de compilación no se genera un mensaje de error. En el proceso de enlazado (última ejecución de `gcc`) se genera un mensaje de error porque no se enlazan las librerías con el código de C++.

```
prueba.o: In function ‘main’:
prueba.o(.text+0x28): undefined reference to ‘endl(ostream &)’
prueba.o(.text+0x35): undefined reference to ‘cout’
prueba.o(.text+0x3a): undefined reference to ‘ostream::operator<<(char const *)’
```

  Solución: Emplear el compilador de C++ (`g++`).

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

- Poner espacios en blanco en vez de emplear tabulador en el fichero `makefile`. Ejemplo: El subrayado (_) representa un espacio en blanco.

```
prueba: prueba.o unaclase.o
________g++ -o prueba prueba.o unaclase.o
```

  Mensaje de error:

```
makefile:2: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.
```

  Solución: Emplear el tabulador en vez de espacios en blanco.

```
prueba: prueba.o unaclase.o
        g++ -o prueba prueba.o unaclase.o
```

- No existe un fichero de una regla de dependencia (por ejemplo, porque se ha escrito mal el nombre del fichero). Ejemplo:

```
prueba.o: prueba.cc UnaClase.h
        g++ -c prueba.cc
```

  Mensaje de error:

```
make: *** No rule to make target ‘UnaClase.h’, needed by ‘prueba.o’. Stop.
```

Solución: Verificar que los nombres de los ficheros están bien escritos y que los ficheros existen.

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

- Compilar un fichero cuyo nombre una vez compilado no coincide con el esperado. Ejemplo: El compilador genera el fichero `pruebaa.o`, pero lo que se quiere obtener es el fichero `prueba.o`.

```
prueba: prueba.o unaclase.o
        g++ -o prueba prueba.o unaclase.o

prueba.o: pruebaa.cc unaclase.h
        g++ -c pruebaa.cc
```

Mensaje de error:

```
g++: prueba.o: No such file or directory
```

Solución: Verificar que los nombres de los ficheros están bien escritos y que los ficheros existen.

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

- 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
```

3. Sobre las directivas de inclusión

- Los espacios en blanco son significativos dentro de los < > o los " " de una directiva de inclusión. Ejemplo:

```
#include < iostream >
```

  Mensaje de error:

```
unaclase.cc:1:  iostream : No such file or directory
```

  Solución: Eliminar los espacios en blanco.

```
#include <iostream>
```

- Emplear < > en vez de " " para un archivo de cabecera que no es estándar. Ejemplo:

```
#include <unaclase>
// O también
#include <unaclase.h>
```

  Mensaje de error:

```
unaclase.cc:3: unaclase: No such file or directory
// O también
unaclase.cc:3: unaclase.h: No such file or directory
```

  Solución: Emplear " " y poner siempre el nombre completo del fichero de cabecera. Sólo para las cabeceras de la biblioteca estándar se puede emplear indistintamente < > o " " y con o sin `.h`.

```
#include "unaclase.h"
```

- Incluir múltiples veces el mismo fichero de cabecera. Ejemplo:

```
#include "unaclase.h"
#include "unaclase.h"
```

  Mensaje de error:

```
In file included from prueba.cc:3:
unaclase.h:5: redefinition of ‘class UnaClase’
unaclase.h:16: previous definition here
```

  Solución: Emplear las guardas de inclusión, que evita el problema de la redefinición cuando se incluye el mismo fichero de cabecera múltiples veces. Para evitar conflictos de nombres en las guardas, es conveniente elegir nombres largos y "extraños". Todas las cabeceras estándar tienen guardas de inclusión, así que no hay que preocuparse en incluirlas varias veces.

```
#ifndef __UNACLASE__
#define __UNACLASE__

class UnaClase
{
  ...
};

#endif
```

- Emplear un punto (.) en el nombre de una guarda de inclusión. Ejemplo:

```
#ifndef __UNACLASE.H__
#define __UNACLASE.H__
```

Mensaje de error: No se produce un mensaje de error. Se trata de una advertencia: el identificador definido se trunca hasta el carácter no válido.

```
unaclase.h:1: warning: garbage at end of ‘#ifndef’ argument
unaclase.h:2: warning: missing white space after ‘#define __UNACLASE’
```

Solución: Emplear únicamente los mismos caracteres que se emplean en el nombre de una función o de una variable.

```
#ifndef __UNACLASE__
#define __UNACLASE__
```

- No separar el nombre de una guarda de inclusión de las instrucciones de compilación condicional. Ejemplo:

```
#ifndef__UNACLASEH__
#define__UNACLASEH__
```

Mensaje de error:

```
In file included from unaclase.cc:3:
unaclase.h:31: unbalanced ‘#endif’
```

Solución: Separar con un espacio en blanco como el nombre de la guarda de inclusión de las instrucciones de compilación condicional.

```
#ifndef __UNACLASE__
#define __UNACLASE__
```

4. Sobre las clases

- Emplear el nombre de la clase y el operador de ámbito (`::`) incorrectamente al definir el código de una función. Ejemplo:

```
UnaClase::UnaClase& operator=(UnaClase& a)
{
  ...
}
```

  Mensaje de error:

```
unaclase.cc:14: ‘operator =(UnaClase &)’ must be a nonstatic member function
```

  Solución: Primero se tiene que poner el tipo del valor de retorno de la función y a continuación el nombre de la clase y el operador de ámbito.

```
UnaClase& UnaClase::operator=(UnaClase& a)
{
  ...
}
```

- No colocar el modificador de visibilidad `public:` antes de la declaración de los constructores, el destructor y otras funciones miembro. Ejemplo:

```
class UnaClase
{
  UnaClase();
  ~UnaClase();
  ...
};
```

  Mensaje de error: No se produce un mensaje de error. Se trata de una advertencia: una clase con todas sus funciones miembro privadas normalmente no tiene sentido.

```
In file included from unaclase.cc:1:
unaclase.h:7: warning: all member functions in class ‘UnaClase’ are private
```

  Solución: Incluir el modificador de visibilidad `public:` donde haga falta.

```
class UnaClase
{
  public:
    UnaClase();
    ~UnaClase();
    ...
};
```

- 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();
    ...
};
```

- Declarar una función miembro y un atributo de una clase con el mismo nombre. Ejemplo:

```
class UnaClase
{
  public:
    UnaClase();
    int n();
    ...

  private:
    int n;
    ...
};
```

Mensaje de error:

```
unaclase.h:12: declaration of `int UnaClase::n`
unaclase.h:9: conflicts with previous declaration `int UnaClase::n()`
```

Solución: Cambiar el nombre a la función miembro o al atributo. Normalmente, como el nombre de la función miembro está establecido de cara al exterior de la clase, se debe cambiar el nombre del atributo.

```
class UnaClase
{
  public:
    UnaClase();
    int n();
    ...

  private:
    int nn;
    ...
};
```

- Múltiples definiciones de la misma función. Ejemplo: Se ha definido el constructor por defecto de una clase y al definir el destructor de la clase se han olvidado de poner el símbolo ~.

```
UnaClase::UnaClase()
{
  ...
}
...
UnaClase::UnaClase()
{
  ...
}
```

Mensaje de error:

```
unaclase.cc:28: redefinition of ‘UnaClase::UnaClase()’
unaclase.cc:6: ‘UnaClase::UnaClase()’ previously defined here
```

Solución: Eliminar o modificar las definiciones múltiples.

```
UnaClase::UnaClase()
{
  ...
}
...
UnaClase::~UnaClase()
{
  ...
}
```

- No escribir el mismo tipo de valor de retorno en la declaración y la definición de una función de una clase. Ejemplo: Se declara una función con un tipo de valor de retorno (`int`) y luego se define con otro tipo (`void`).

```
class UnaClase
{
  public:
    UnaClase();
    int Algo(int);
};

UnaClase::UnaClase()
{
  ...
}

void UnaClase::Algo(int a)
{
  ...
}
```

Mensaje de error:

```
unaclase.cc:21: prototype for ‘void UnaClase::Algo(int)’ does not match
any in class ‘UnaClase’
unaclase.cc:9: candidate is: int UnaClase::Algo(int)
unaclase.cc:21: ‘void UnaClase::Algo(int)’ and ‘int UnaClase::Algo(int)’
cannot be overloaded
```

Solución: Verificar los tipos de retorno en la declaración y la definición de las funciones.

```
class UnaClase
{
  public:
    UnaClase();
    int Algo(int);
};

UnaClase::UnaClase()
{
  ...
}

int UnaClase::Algo(int a)
{
  ...
}
```

- Olvidarse el punto y coma (;) al final de la declaración de una clase. Ejemplo:

```
class UnaClase
{
  ...
}
```

Mensaje de error:

```
prueba.cc:5: semicolon missing after declaration of ‘UnaClase’
prueba.cc:6: two or more data types in declaration of ‘main’
prueba.cc:6: semicolon missing after declaration of ‘class UnaClase’
```

En otros casos, el mensaje de error es confuso y no ayuda a encontrar el error:

```
unaclase.cc:6: return type specification for constructor invalid
```

Solución: Poner el punto y coma al final de la declaración de la clase.

```
class UnaClase
{
  ...
};
```

- Acceder a los miembros privados de una clase. Ejemplo:

```
class UnaClase
{
  public:
    ...

  private:
    int dim;
    ...
};

/*** CÓDIGO DE EJEMPLO ***/
UnaClase a;

cout << a.dim;
```

Mensaje de error:

```
prueba.cc: In function ‘int main(...)’:
unaclase.h:16: ‘int UnaClase::dim’ is private
prueba.cc:10: within this context
```

Solución: Tener claro lo que se desea público o privado. Si se desea acceder a algún miembro declarado privado, se puede proporcionar un interfaz público.

```
class UnaClase
{
  public:
    ...
    int Dim(void);
```

```
  private:
    int dim;
    ...
};

/*** CÓDIGO DE EJEMPLO ***/
UnaClase a;

cout << a.Dim();
```

- En el destructor, liberar la memoria dinámica (`delete`), pero no modificar el valor de otros atributos que indican si el objeto contiene algo. Ejemplo:

```
UnaClase::~UnaClase()
{
  if(v != NULL)
  {
    delete v;
    v = NULL;
  }
  // No se hace lo siguiente
  // dim = 0;
}

/*** CÓDIGO DE EJEMPLO ***/
UnaClase a(5);
...
a.~UnaClase();
...
for(i = 0; i < a.dim; i++)
  cout << a.v[i];
```

Mensaje de error:

```
Segmentation fault
```

Solución: Definir correctamente el destructor de un objeto.

```
UnaClase::~UnaClase()
{
  if(v != NULL)
  {
    delete v;
    v = NULL;
  }

  dim = 0;
}
```

- No saber cuándo se trabajo con un objeto o con un puntero a un objeto: confundir los operadores `.` y `->`. Ejemplo:

```
UnaClase a, *b;

b = &a;
cout << a->propiedad;
cout << b.propiedad;
```

Mensaje de error:

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:12: base operand of ‘->’ has non-pointer type ‘UnaClase’
prueba.cc:13: request for member ‘propiedad’ in ‘b’, which is of non-aggregate
              type ‘UnaClase *’
```

Solución: Distinguir correctamente los objetos y los punteros a objeto.

```
UnaClase a, *b;

b = &a;
cout << a.propiedad;
cout << b->propiedad;
```

- No saber cuándo se trabajo con un objeto o con un puntero a un objeto: intentar asignar un objeto a un puntero o vice versa. Ejemplo:

```
UnaClase a, *b;

b = a;
```

Mensaje de error:

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:10: cannot convert ‘a’ from type ‘UnaClase’ to type ‘UnaClase *’
```

Solución: Distinguir correctamente los objetos y los punteros a objeto.

```
UnaClase a, *b;

b = &a;
```

- Crear un objeto a partir de una clase que no está perfectamente definida (por ejemplo, declarar una clase que contiene un objeto de la propia clase). Ejemplo:

```
class UnaClase {
  public:
    UnaClase();

  private:
    UnaClase a;
};
```

Mensaje de error:

```
unaclase.cc:11: field ‘a’ has incomplete type
```

Solución: En C++ no se puede crear un objeto si su tamaño y contenido son desconocidos (tipo incompleto). Con un tipo incompleto sólo se pueden hacer cosas que no requieran conocer su tamaño y disposición en memoria, como declarar punteros o referencias.

```
class UnaClase {
  public:
    UnaClase();

  private:
    UnaClase *a;
};
```

- Crear un objeto a partir de una clase que aún no ha sido declarada. Ejemplo:

```
class UnaClase
{
  public:
    UnaClase();

  private:
    OtraClase a;
};

class OtraClase
{
  public:
    OtraClase();
};
```

Mensaje de error:

```
unaclase.cc:11: ’OtraClase’ is used as a type, but is not defined as a type.
```

Solución: Declarar las clases antes de crear objetos a partir de ellas.

```
class OtraClase
{
  public:
    OtraClase();
};

class UnaClase
{
  public:
    UnaClase();

  private:
    OtraClase a;
};
```

- Establecer una referencia mutua entre dos clases: la clase A contiene un objeto de la clase B y la clase B contiene un objeto de la clase A. Ejemplo:

```
class UnaClase
{
  public:
    UnaClase();

  private:
    OtraClase a;
};

class OtraClase
{
  public:
    OtraClase();

  private:
    UnaClase b;
};
```

Mensaje de error:

```
unaclase.cc:11: 'OtraClase' is used as a type, but is not defined as a type.
```

Solución: Esta situación no se puede solucionar modificando el orden de declaración de las clases (como en el ejemplo anterior), ya que existen referencias mutuas. Además, desde un punto de vista lógico no tiene sentido: un objeto de la clase `UnaClase` contiene un objeto de la clase `OtraClase` que a su vez contiene un objeto de la clase `UnaClase` y así sucesivamente hasta el infinito.

- Establecer una referencia mutua entre dos clases (la clase A contiene un objeto de la clase B y la clase B contiene un objeto de la clase A), con una declaración adelantada (*forward*) para evitar el problema planteado en la situación anterior. Ejemplo:

```
class OtraClase;
```

```
class UnaClase
{
  public:
    UnaClase();

  private:
    OtraClase a;
};

class OtraClase
{
  public:
    OtraClase();

  private:
    UnaClase b;
};
```

Mensaje de error:

```
unaclase.cc:13: field ‘a’ has incomplete type
```

Solución: La declaración adelantada indica al compilador que OtraClase es de tipo `class`. Este tipo de declaración únicamente permite declarar punteros o referencias, pero no es suficiente para crear objetos, ya que no se conoce el tamaño del objeto o la disposición de su contenido en memoria.

```
class OtraClase;

class UnaClase
{
  public:
    UnaClase();

  private:
    OtraClase *a;
};

class OtraClase
{
  public:
    OtraClase();

  private:
    UnaClase b;
};
```

5. Sobre la sobrecarga de los operadores

- Emplear la declaración `friend` cuando se está definiendo el código (en el fichero `.cc`) de una función que es amiga. Ejemplo:

```
friend ostream& operator<<(ostream& os, UnaClase& uc)
{
  ...
}
```

Mensaje de error:

```
unaclase.cc:50: can't initialize friend function `<<'
unaclase.cc:50: friend declaration not in class definition
```

Solución: Eliminar la palabra clave `friend`. Una función se declara como `friend` dentro de la clase donde es amiga, pero cuando se define su código no se tiene que indicar.

```
ostream& operator<<(ostream& os, UnaClase& uc)
{
  ...
}
```

- En el operador asignación, no borrar el elemento de la izquierda de la asignación (es decir, el objeto sobre el que se invoca el operador asignación). Ejemplo:

```
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  dim = a.dim;
  v = new int[dim];
  for(int i = 0; i < dim; i++)
    v[i] = a.v[i];

  return *this;
}
```

Mensaje de error: No produce un mensaje de error. Este problema no se manifiesta en forma de error. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: Antes de realizar la copia en la asignación, hay que borrar el elemento de la izquierda de la asignación. Se puede realizar directamente o llamando al destructor de la clase.

```
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  this->~UnaClase();
  dim = a.dim;
  v = new int[dim];
  for(int i = 0; i < dim; i++)
    v[i] = a.v[i];
```

```
  return *this;
}
```

- En el operador asignación, no se protege el código de la autoasignación. Ejemplo:

```
// Sobrecarga del operador asignación
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  this->~UnaClase();
  dim = a.dim;
  v = new int[dim];
  for(int i = 0; i < dim; i++)
    v[i] = a.v[i];

  return *this;
}

/*** CÓDIGO DE EJEMPLO ***/
//
UnaClase a(5);
// Al realizar una autoasignación, se borra el objeto
a = a;
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. Únicamente se borra el objeto que realiza la autoasignación.

Solución: Comprobar antes de realizar la copia en la asignación que no se trata del mismo objeto. Si es así, la función no tiene que hacer nada.

```
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  // Más rápido que if(*this != a)
  if(this != &a)
  {
    this->~UnaClase();
    dim = a.dim;
    v = new int[dim];
    for(int i = 0; i < dim; i++)
      v[i] = a.v[i];
  }

  return *this;
}
```

- En los operadores de entrada >> y salida <<, emplear los flujos de entrada (`cin`) o salida estándar (`cout`) en vez de los recibidos. Ejemplo:

```
ostream& operator<<(ostream& os, UnaClase& uc)
{
  cout << "Dimension: " << uc.dim << endl;

  for(int i = 0; i < uc.dim; i++)
    cout << "[" << uc.v[i] << "]";

  return os;
}
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución.

Solución: Para la entrada y salida se tienen que emplear los flujos que se reciben como argumento. El mismo código se puede emplear con distintos tipos de flujo (por ejemplo, para flujos de fichero).

```
ostream& operator<<(ostream& os, UnaClase& uc)
{
  os << "Dimension: " << uc.dim << endl;

  for(int i = 0; i < uc.dim; i++)
    os << "[" << uc.v[i] << "]";

  return os;
}
```

- En los operadores de entrada >> y salida <<, no devolver el flujo recibido (primera variante). Ejemplo:

```
ostream& operator<<(ostream& os, UnaClase& uc)
{
  os << "Dimension: " << uc.dim << endl;

  for(int i = 0; i < uc.dim; i++)
    os << "[" << uc.v[i] << "]";
}

/*** CÓDIGO DE EJEMPLO ***/
UnaClase a;

cout << a << endl;
```

Mensaje de error: Se produce un error al realizar operaciones de salida sobre el mismo flujo en una sola instrucción.

```
Segmentation fault
```

Solución: La sobrecarga de los operadores de entrada y salida deben de devolver el flujo recibido.

```
ostream& operator<<(ostream& os, UnaClase& uc)
{
  os << "Dimension: " << uc.dim << endl;

  for(int i = 0; i < uc.dim; i++)
    os << "[" << uc.v[i] << "]";

  return os;
}
```

- En los operadores de entrada >> y salida <<, no devolver el flujo recibido (segunda variante). Ejemplo:

```
void operator<<(ostream& os, UnaClase& uc)
{
  os << "Dimension: " << uc.dim << endl;

  for(int i = 0; i < uc.dim; i++)
    os << "[" << uc.v[i] << "]";
}

/*** CÓDIGO DE EJEMPLO ***/
UnaClase a;

cout << a << endl;
```

Mensaje de error: Se está intentado emplear un valor de tipo `void` para realizar una operación.

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:24: void value not ignored as it ought to be
```

Solución: La sobrecarga de los operadores de entrada y salida deben de devolver el flujo recibido.

```
ostream& operator<<(ostream& os, UnaClase& uc)
{
  os << "Dimension: " << uc.dim << endl;

  for(int i = 0; i < uc.dim; i++)
    os << "[" << uc.v[i] << "]";

  return os;
}
```

- En el operador corchete `[]`, devolver por valor en vez de por referencia. Ejemplo:

```
int
UnaClase::operator[](int i)
```

```
{
   ...
}

/*** CÓDIGO DE EJEMPLO ***/
int i;
UnaClase a;

i = a[1];
a[1] = 10;
```

Mensaje de error: El error se produce cuando se emplea el operador corchete en la parte izquierda de una asignación, porque no se tiene una dirección de memoria (una referencia). Cuando se emplea en la parte derecha (`i = a[i]`) no se produce un error.

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:14: non-lvalue in assignment
```

Solución: El operador corchete tiene que devolver por referencia.

```
int&
UnaClase::operator[](int i)
{
   ...
}
```

6. Sobre la memoria

- Un objeto creado de forma dinámica con `new` existe hasta que se destruye explícitamente mediante `delete`. Ejemplo:

```
void unaFuncion(void)
{
  UnaClase *a;

  a = new UnaClase();
  ...
}
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: El programador se tiene que encargar de liberar toda la memoria que haya reservado de forma dinámica.

```
void unaFuncion(void)
{
  UnaClase *a;

  a = new UnaClase();
  ...
  delete a;
}
```

- Liberar la memoria reservada poniendo un puntero a `NULL`. Ejemplo:

```
void unaFuncion(void)
{
  UnaClase *a;

  a = new UnaClase();
  ...
  a = NULL;
}
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: El programador se tiene que encargar de liberar toda la memoria que haya reservado de forma dinámica. Al poner un puntero a `NULL` no se libera memoria reservada: únicamente se cambia el valor de un puntero.

```
void unaFuncion(void)
{
  UnaClase *a;

  a = new UnaClase();
  ...
  delete a;
}
```

- Para destruir un objeto, llamar directamente a su destructor, pero luego no liberar la memoria con `delete`. Ejemplo:

```
void unaFuncion(void)
{
  UnaClase *a;

  a = new UnaClase();
  ...
  a->~UnaClase();
}
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: El programador se tiene que encargar de liberar toda la memoria que haya reservado de forma dinámica. El destructor por sí solo no libera la memoria que ocupa un objeto: sólo libera la memoria que haya reservado el objeto. No hace falta llamar explícitamente al destructor, ya que al realizar `delete` se invoca automáticamente.

```
void unaFuncion(void)
{
  UnaClase *a;

  a = new UnaClase();
  ...
  a->~UnaClase();
  delete a;
  // O solamente (mejor)
  delete a;
}
```

- Liberar la memoria a la que apunta un puntero (`delete`), pero no poner el puntero a `NULL`. Ejemplo:

```
char *a;

a = new char[20];
...
delete a;
...
if(a != NULL)
  strcpy(a, "Una cadena");
```

Mensaje de error: Como se está accediendo a una zona de memoria que ya no es válida, lo más seguro es que se produzca un mensaje de violación de segmento.

```
Segmentation fault
```

Solución: Siempre que se libere la memoria a la que apunta un puntero, es recomendable ponerlo a `NULL` para evitar que se vuelva a usar esa zona de memoria.

```
char *a;

a = new char[20];
...
delete a;
a = NULL;
...
if(a != NULL)
  strcpy(a, "Una cadena");
```

- Liberar un array de objetos con `delete` en vez de con `delete []` o liberar un objeto individual con `delete []` en vez de con `delete`. Ejemplo:

```
UnaClase *a, *b;

a = new UnaClase;
b = new UnaClase[5];
...
delete [] a;
delete b;
```

Mensaje de error: La manera exacta en la que se asigna la memoria de los arrays y de los objetos individuales es dependiente de la implementación. El uso incorrecto del operador `delete` puede producir mensajes de violación de segmento.

```
Segmentation fault
```

Solución: El programador siempre debe decir si se va a destruir un array o un objeto individual.

```
UnaClase *a, *b;

a = new UnaClase;
b = new UnaClase[5];
...
delete a;
delete [] b;
```

- Liberar la misma zona de memoria dos veces. Ejemplo:

```
char *a, *b;

a = new char[20];
strcpy(a, "Una cadena");
b = a;
delete a;
a = NULL;
delete b;
b = NULL;
```

Mensaje de error: Como se está intentado liberar una zona de memoria que ya no es válida, lo más seguro es que se produzca un mensaje de violación de segmento.

```
Segmentation fault
```

Solución: En este ejemplo, el error se ha producido por tener dos punteros apuntando a la misma zona de memoria. Hay que evitar situaciones de este estilo. En el caso de objetos, este error se suele producir al copiar un objeto que contiene punteros y no se realiza una copia correcta (reservando una zona de memoria y copiando el contenido). Por tanto, suele ocurrir si el constructor de copia o el operador asignación no están correctamente programados.

- No liberar toda la memoria cuando un objeto contiene punteros a otros objetos. Ejemplo:

```
class OtraClase
{
  public:
    OtraClase();
    ~OtraClase();
    ...

  private:
    UnaClase *a;
    ...
};

OtraClase::OtraClase()
{
  a = new UnaClase();
}

OtraClase::~OtraClase()
{
}
```

  Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

  Solución: En el destructor de un objeto hay que liberar toda la memoria que tenga reservada el objeto.

```
OtraClase::~OtraClase()
{
  if(a != NULL)
  {
    delete a;
    a = NULL;
  }
}
```

7. Sobre las cadenas

- Realizar una copia de cadenas mediante una asignación de punteros. Ejemplo:

```
char a[] = "Una cadena";
char *b;
```

```
b = a;
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo `Segmentation fault` durante la ejecución, ya que existen dos punteros que hacen referencia a la misma zona de memoria.

Solución: Realizar una copia mediante `strdup` o mediante `new`, `strlen` y `strcpy`.

```
char a[] = "Una cadena";
char *b;

b = strdup(a);
// O también
b = new char[strlen(a) + 1];
strcpy(b, a);
```

- Realizar una copia de cadenas mediante `new` y `strcpy` y no obtener la longitud de la cadena. Ejemplo:

```
char a[] = "Una cadena";
char *b;

b = new char[a + 1];
strcpy(b, a);
```

Mensaje de error:

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:9: size in array new must have integral type
```

Solución: Emplear `strlen` para obtener la longitud de la cadena que se va a copiar.

```
char a[] = "Una cadena";
char *b;

b = new char[strlen(a) + 1];
strcpy(b, a);
```

- Realizar una copia de cadenas mediante `new`, `strlen` y `strcpy` y no reservar espacio para el carácter final de cadena `‘\0’`. Ejemplo:

```
char a[] = "Una cadena";
char *b;

b = new char[strlen(a)];
strcpy(b, a);
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo `Segmentation fault` durante la ejecución, ya que el carácter `'\0'` se ha copiado en una zona de memoria no válida.

Solución: Realizar una copia reservando una posición más para el carácter `'\0'`.

```
char a[] = "Una cadena";
char *b;

b = new char[strlen(a) + 1];
strcpy(b, a);
```

- Realizar una copia de cadenas mediante `new`, `strlen` y `strcpy` y reservar espacio para el carácter final de cadena `'\0'` de forma incorrecta. Ejemplo:

```
char a[] = "Una cadena";
char *b;

b = new char[strlen(a + 1)];
strcpy(b, a);
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo `Segmentation fault` durante la ejecución, ya que no se ha reservado suficiente espacio para copiar la cadena (se ha reservado dos posiciones menos de las necesarias).

Solución: Realizar una copia reservando una posición más para el carácter `'\0'`.

```
char a[] = "Una cadena";
char *b;

b = new char[strlen(a) + 1];
strcpy(b, a);
```

- Realizar una copia de una cadena sobre otra que ya existía previamente, sin destruir la primera y reservar espacio para la nueva cadena. Ejemplo:

```
char *a, *b;
a = strdup("Una cadena");
b = strdup("Otra cadena más larga");

strcpy(a, b);
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo `Segmentation fault` durante la ejecución, ya que al copiar una cadena sobre otra de longitud menor se invaden zonas de memoria reservadas o utilizadas en otro sitio.

Solución: Liberar la memoria que ocupa la cadena sobre la que se va a copiar y realizar una copia mediante `strdup` o mediante `new`, `strlen` y `strcpy`.

```
char *a, *b;
a = strdup("Una cadena");
b = strdup("Otra cadena más larga");

delete a;
a = strdup(b);
```

- Reservar memoria para una cadena mediante el operador `new` y confundir los paréntesis `()` por los corchetes `[]`. Ejemplo:

```
char *a;

a = new char(65);
strcpy(a, "Una cadena de caracteres");
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo `Segmentation fault` durante la ejecución, ya que la cadena se ha copiado en una zona de memoria no válida (se ha reservado espacio para un carácter y se ha inicializado con el código ASCII 65).

Solución: Para reservar una cadena (un array de caracteres) se tienen que emplear los corchetes.

```
char *a;

a = new char[65];
strcpy(a, "Una cadena de caracteres");
```

- Intentar liberar una cadena que no ha sido creada mediante memoria dinámica (`new`). Ejemplo:

```
char a[] = "Una cadena";

delete a;
```

Mensaje de error: El compilador es capaz de detectar esta situación y genera un mensaje de error.

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:15: warning: deleting array ‘char a[11]’
```

Solución: No se puede liberar la memoria estática.

- Intentar liberar una cadena que no ha sido creada mediante memoria dinámica (`new`). Similar a la anterior situación, pero la cadena se pasa a una función. Ejemplo:

```
void unaFuncion(char *c)
{
  delete c;
}

/*** CÓDIGO DE EJEMPLO ***/
char a[] = "Una cadena";

unaFuncion(a);
```

  Mensaje de error: En este caso, el compilador no es capaz de detectar esta situación. Durante la ejecución, lo más seguro es que se produzca un mensaje de violación de segmento.

```
Segmentation fault
```

  Solución: No se puede liberar la memoria estática.

- Emplear una cadena nula (el puntero apunta a `NULL`) con las funciones de manejo de cadena (`strlen`, `strcmp`, `strdup` y `strcpy`). Ejemplo:

```
char *a = NULL, *b;

cout << strlen(a);
// O también
b = strdup(a);
// O también
strcpy(b, a);
```

  Mensaje de error: Durante la ejecución, lo más seguro es que se produzca un mensaje de violación de segmento.

```
Segmentation fault
```

  Solución: Tener cuidado al manejar las cadenas. Comprobar siempre que no apuntan a `NULL`.

- Comparar dos cadenas comparando directamente los punteros. Ejemplo:

```
bool compara(char *a, char *b)
{
  if(a == b)
    return true;
  return false;
}
```

```
/*** CÓDIGO DE EJEMPLO ***/
char a[] = "Esto es una cadena";
char b[] = "Esto es una cadena";

// Escribe false
cout << compara(a, b);
```

Mensaje de error: No produce un mensaje de error. Se trata de un error lógico, ya que no se están comparando las cadenas, sino los punteros. En el ejemplo anterior, la función `compara()` devuelve `false`, aunque las dos cadenas sean iguales.

Solución: Emplear la función `strcmp()`, que devuelve `0` cuando dos cadenas son iguales.

```
bool compara(char *a, char *b)
{
  if(!strcmp(a, b))
    return true;
  return false;
}

/*** CÓDIGO DE EJEMPLO ***/
char a[] = "Esto es una cadena";
char b[] = "Esto es una cadena";

// Escribe true
cout << compara(a, b);
```

- Imprimir una cadena carácter a carácter. Ejemplo:

```
char a[] = "Esto es una cadena";

for(int i = 0; i < strlen(a); i++)
    cout << a[i];
```

Mensaje de error: No produce un mensaje de error. Es ineficiente, ya que una cadena se puede imprimir directamente.

Solución: Se puede imprimir una cadena directamente.

```
char a[] = "Esto es una cadena";

cout << a;
```

8. Varios

- Confundir la asignación (=) con el operador igualdad (==). Ejemplo:

```
void unaFuncion(char *a)
{
  if(a = NULL)
    cout << "La cadena está vacía" << endl;
  else
    cout << a << endl;
}
```

  Mensaje de error: No se produce un mensaje de error. Siempre se va a ejecutar el código de la parte `else` (muestra `(null)`, que es la representación en pantalla de un puntero a cadena que vale `NULL`), porque se está realizando una asignación en vez de una comparación.

  Solución: No confundir los dos operadores.

```
void unaFuncion(char *a)
{
  if(a == NULL)
    cout << "La cadena está vacía" << endl;
  else
    cout << a << endl;
}
```

- Confundir el preincremento (`++a`) con el postincremento (`a++`). Ejemplo:

```
a = b++;
cout << "a y b tienen el mismo valor" << endl;
```

  Mensaje de error: No se produce un mensaje de error. `a` y `b` no tienen el mismo valor porque primero se realiza la asignación y luego se incrementa `b`, ya que se está empleando el operador postincremento.

  Solución: Cuando se emplea el operador preincremento, primero se realiza el incremento y luego el resto de operaciones. Cuando se emplea el operador postincremento, primero se realizan el resto de operaciones y luego el incremento.

```
a = ++b;
cout << "a y b tienen el mismo valor" << endl;
```

- Poner un punto y coma (;) después de la sentencia de un bucle. Ejemplo:

```
for(i = 0; i < 10; i++);
{
  ...
}
```

Mensaje de error: No se produce un mensaje de error. Las sentencias que se supone que se ejecutan dentro del bucle sólo se ejecutan una vez, ya que no están realmente dentro del bucle.

Solución: Tener cuidado con las sentencias de repetición (`for`, `do ... while` y `while`).

```
for(i = 0; i < 10; i++)
{
  ...
}
```

- Acceder a una posición no válida de un array. Ejemplo:

```
int vector[10];

for(int i = 0; i <= 10; i++)
  vector[i] = 1;
```

Mensaje de error: Como se está accediendo a una zona de memoria que no es válida, lo más seguro es que se produzca un mensaje de violación de segmento.

```
Segmentation fault
```

Solución: Tener cuidado con los índices de los arrays. En C++ los arrays comienzan en `0` y terminan en `dimensión - 1`.

```
int vector[10];

for(int i = 0; i < 10; i++)
  vector[i] = 1;
```

- Devolver una referencia a una variable u objeto local. Ejemplo:

```
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  UnaClase u;

  ...
  return u;
}
```

Mensaje de error:

```
unaclase.cc: In method ‘class UnaClase & UnaClase::operator =(UnaClase &)’:
unaclase.cc:45: warning: reference to local variable ‘u’ returned
```

Solución: No devolver referencias a una variable u objeto local.

- Devolver un puntero a una variable local u objeto local. Ejemplo:

```
UnaClase*
f(void)
{
  UnaClase u;

  ...
  return &u;
}
```

Mensaje de error:

```
prueba.cc: In function ‘class UnaClase * f()’:
prueba.cc:8: warning: address of local variable ‘a’ returned
```

Sólo en aquellos casos (como el anterior) donde el compilador puede detectar que se está devolviendo un puntero a una variable u objeto local se genera un mensaje de error. Por ejemplo, el siguiente código presenta el mismo problema, pero el compilador no lo detecta:

```
UnaClase*
f(void)
{
  UnaClase a, *b;

  b = &a;

  return b;
}
```

Solución: No devolver punteros a una variable u objeto local.

- Emplear para una variable de una función el mismo identificador que un argumento de la función. Ejemplo:

```
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  UnaClase a;
```

```
  ...
}
```

Mensaje de error:

```
unaclase.cc: In method ‘class UnaClase & UnaClase::operator =(UnaClase &)’:
unaclase.cc:45: declaration of ‘a’ shadows a parameter
```

Solución: Usar identificadores distintos para los argumentos y las variables locales.

```
UnaClase&
UnaClase::operator=(UnaClase& a)
{
  UnaClase a1;

  ...
}
```

- Usar incorrectamente cualquier puntero y, en especial, el puntero `this`. Ejemplo:

```
UnaClase::UnaClase()
{
  (this*).dim=0;
  (this*).v=NULL;
}
```

Mensaje de error:

```
unaclase.cc: In method ‘UnaClase::UnaClase()’:
unaclase.cc:7: parse error before ‘)’
unaclase.cc:8: parse error before ‘)’
```

Solución: El operador `*` se pone delante del puntero y no detrás.

```
UnaClase::UnaClase()
{
  (*this).dim=0;
  (*this).v=NULL;
}
```

También se puede emplear el operador `->`.

```
UnaClase::UnaClase()
{
  this->dim=0;
  this->v=NULL;
}
```

- No tener en cuenta la precedencia de los operadores. Ejemplo:

```
UnaClase::UnaClase()
{
  *this.dim=0;
  *this.v=NULL;
}
```

Mensaje de error:

```
unaclase.cc: In method ‘UnaClase::UnaClase()’:
unaclase.cc:7: request for member ‘dim’ in ‘this’, which is of non-aggregate
type ‘UnaClase *’
unaclase.cc:8: request for member ‘v’ in ‘this’, which is of non-aggregate
type ‘UnaClase *’
```

Solución: El operador `.` tiene mayor precedencia que el operador `*`. Si primero se tiene que aplicar el operador `*`, se tiene que encerrar la operación entre paréntesis.

```
UnaClase::UnaClase()
{
  (*this).dim=0;
  (*this).v=NULL;
}
```

- Suponer que el indicador de puntero (`*`) se aplica a toda una lista de variables en una declaración. Ejemplo:

```
char *a, b, c;

a = strdup("Una cadena");
b = strdup("Otra cadena");
c = strdup("La última cadena");
```

Mensaje de error:

```
prueba.cc: In function ‘int main(...)’:
prueba.cc:10: assignment to ‘char’ from ‘char *’ lacks a cast
prueba.cc:11: assignment to ‘char’ from ‘char *’ lacks a cast
```

Solución: El indicador de puntero tiene que acompañar a cada variable que se desee declarar como puntero.

```
char *a, *b, *c;

a = strdup("Una cadena");
b = strdup("Otra cadena");
c = strdup("La última cadena");
```