Tema VI
Novedades ANSI/ISO-C++'98 (R-1.0)

Programación en Entornos Interactivos.

14 de marzo de 2011

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Alicante

Resumen

Novedades introducidas en C++ por este estándar. Adecuación al mismo de los compiladores.

Ficheros de cabecera

- Los ficheros de cabecera estándar de C++ que hacen uso del espacio de nombres std se llaman igual que antes pero sin extensión '.h', por ejemplo:
  En lugar de <iostream.h>. . . <iostream>
- Los ficheros de cabecera de C se llaman como antiguamente pero sin extensión '.h' y precedidos por el prefijo 'c', por ejemplo: cstdio en lugar de stdio.h.
  De este modo podemos saber si estamos usando un fichero de C o de C++.

STL y clase string

- La librería `STL` (Standard Template Library) forma parte del estándar del lenguaje.
- La clase string también, aunque no tiene nada que ver con `STL`.
- Para hacer uso de la clase `string` se debe incluir el fichero `<string>`.
- Se prefiere usar datos de tipo `string` en lugar de '`char *`'. Si hace falta disponer de un dato de tipo '`char *`' se puede usar el mensaje `c_str()` enviado a un objeto de la clase `string`.

Tipos de datos nuevos

- Tipo bool.
- Se definen los identificadores false y true.

Nombres para operadores

- Determinados operadores ahora pueden tener un nombre...

Operadores

`&&`	`and`	`&=`	`and_eq`
`||`	`or`	`|=`	`or_eq`
`!`	`not`	`^=`	`xor_eq`
`!=`	`not_eq`	`&`	`bit_and`
`|`	`bit_or`	`^`	`xor`
`~`	`compl`		

- Importante: el compilador g++ necesita la opción -foperator-names o -ansi para reconocerlos, así se evitan conflictos con código escrito que pudiera hacer uso de alguno de estos identificadores.

Espacios de nombres (I)

- Mediante el uso de la palabra reservada **namespace** se permiten construcciones como la siguiente:

namespace

```
namespace ACME {                    ACME::Widget w;
  class Widget {...};              ACME::i = 3;
  class string {...};
  int i;
}   // No necesita ';' final
```

- Incluso se permite crear alias de namespaces ya existentes:

alias

```
namespace lib = ACME;  // Ahora se puede cambiar asi...
// namespace lib = RogueWave;
// namespace lib = std;
// Y nunca hace falta cambiar esto:
lib::string texto;
```

Espacios de nombres (II)

- Podemos crearlos sin nombre. . . (internamente el compilador asignará uno que sea único).
- Los identificadores declarados en uno de ellos, son globales al fichero donde se declara este espacio de nombres.
- En un futuro, esto hará que desaparezca el uso del modificador static a nivel de fichero, por ejemplo:

namespace sin nombre

```
namespace {
  int j = 3; // Solo es visible en esta
             // unidad de compilación
}

void main(void) {
  cout << "j= " << j << endl;
}
```

Espacios de nombres (III)

Para facilitar su uso, se permiten las Declaraciones de uso:

Using-declarations

```
namespace ACME {
  class Widget ...
  int i;
}
using ACME::Widget;

Widget w;      // Se refiere a ACME::Widget
ACME::i = 3;   // Es necesario indicar su namespace
```

Espacios de nombres (IV)

Y también están permitidas las Directivas de uso:

Using-directives

```
namespace ACME {
  class Widget ...
  int i;
}
using namespace ACME;
Widget w;  // Se refiere a ACME::Widget
i = 3;     // Se refiere a ACME::i
```

Como se puede ver, las Declaraciones de uso son más selectivas que las Directivas.

Espacios de nombres (V)

- Son abiertos, en cualquier momento (después de haber terminado su declaración) podemos añadir elementos a un `namespace`, al contrario de como ocurre con las clases.
- Existe uno predefinido que no se puede modificar: std, representa el espacio de nombres estándar, en el se declaran los identificadores de la librería estandar que hasta ahora eran globales.
- Se pueden anidar sus declaraciones, es decir, declarar un `namespace` dentro de otro.

Operadores de conversión de tipo (I)

Existen 4 nuevos operadores de conversión de tipo que reemplazan al 'conversor de tipo' (`cast` o `molde`) heredado de 'C':

- static_cast: Se usa para conversiones entre tipos estándar (y compatibles entre si, p.e. `int<-->float`) del lenguaje o entre tipos relacionados por herencia `public`:

static_cast

```
int i = static_cast<int>(2.3)
```

- const_cast: Se debe usar sólo para conversiones en las que se quiere eliminar el atributo `const` o `volatile` de un identificador:

const_cast

```
const MiClase objeto_constante;
MiClase *obj_no_const = const_cast<MiClase *>
                        (&objeto_constante);
```

Operadores de conversión de tipo (II)

- reinterpret_cast: Se puede usar para realizar cualquier conversión de tipo, por tanto **se debe emplear con cuidado**:

reinterpret_cast

```
char *c;
void *p;
...
c = reinterpret_cast<char *>(p);
```

Operadores de conversión de tipo (III)

- dynamic_cast: Se debe usar para realizar conversiones de tipo en tiempo de ejecución sobre todo entre punteros a 'clases base' y objetos de 'clases derivadas':

dynamic_cast

```
class Base {...};
class Derivada : public Base {...};

Derivada unD, *pD;
Base *pB = static_cast<Base *>(&unD);
// En tiempo de compilacion
...
pD = dynamic_cast<Derivada *>(pB);
// En tiempo de ejecucion
```

- Si no puede realizar la conversión devuelve el valor 0 o se lanza una excepción.

RTTI (I)

Identificación de tipos en tiempo de ejecución.

- Algunos compiladores necesitan un determinado 'flag' para activarla.
- Se debe incluir la cabecera <typeinfo>.
- Se proporciona el operador typeid, al aplicarlo sobre un dato, devuelve un objeto de la clase type_info.

RTTI (II)

- Los objetos type_info se pueden comparar con == y !=.
- A los objetos `type_info` se les puede preguntar el nombre del tipo que representan mediante el mensaje name(), por ejemplo:

RTTI

```
if (typeid(*p) == typeid(Frame)) {
  // Es un puntero a Frame
}
const type_info& tir = typeid(*p);
std::cout << "p es un puntero a " << tir.name() << std::endl;
```

Nuevas palabras reservadas (I)

- **explicit**: Modifica el constructor de una clase para que no pueda usarse como mecanismo de conversión de tipos:

explicit

```
class Vector_entero {
  explicit Vector_entero(int i) {...}
  ...
};

int suma_componentes(const Vector_entero& v) {...}
...
int a = suma_componentes(10);
//Error de compilacion por uso de ‘‘explicit’’
```

Nuevas palabras reservadas (II)

- **mutable**: Permite calificar miembros de datos de una clase como modificables incluso si pertenecen a un objeto const:

mutable

```
class Objeto {
  public:
    void f() { m1 = m2 =0; }
    // 'f()' puede modificar a 'm1' y 'm2'

    void g() const { m2 = 1; }
    // 'g()' No puede modificar 'm1', si 'm2'

  private:
  int          m1;
  mutable int m2;
};
```

Nuevas palabras reservadas (III)

- Nuevos operadores de reserva de memoria dinámica especiales para vectores:
  - new[]
  - delete[]

Tratamiento de excepciones (I)

- Hace uso de las palabras reservadas try, throw y catch.
- Dispone de una jerarquía de clases de excepciones estándar:
  1. logic_error, domain_error, invalid_argument
  2. length_error, out_of_range, runtime_error, range_error
  3. overflow_error, underflow_error
- Para poder utilizarlas es necesario incluir la cabecera estándar <stdexcept>.

Tratamiento de excepciones (II)

- Se puede usar en la fase de iniciación de miembros en un constructor o incluso en una función normal:

excepciones

```
C::C() try : m(10)
{
  // cuerpo del constructor o funcion
}
catch (...) {
  cout << "Excepción en la fase de iniciación\n";
  // No se puede volver, hay que terminar
  terminate();
}
```

Templates (I)

- Admiten argumentos de tipos estándar –Non-type–:

Non-type

```
template<int N>
class Array
{
  // ...
  char  data[N];
};

Array<256> a256_char;
```

Templates (II)

- Valores por defecto para los parámetros del template:

Valores por defecto

```
template<int N = 42, class T = char>
class Array
{
  // ...
  T data[N];
};
Array<100, int> a100_ints;
Array<256>      a256_char;
Array<>         a42_char;
```

Templates (III)

- Calificación explícita de funciones template:

Calificación explícita

```
template<class T, class U> T make(U u);
Thing t = make<Thing>(1.23);
```

Templates (IV)

- Templates como argumentos de templates:

Argumentos template

```
template<class T> class List;
template<class T> class Vector;
template<class T, template<class U> class C = List>
class Group {
  // ...
  C<T>    container;
};
Group<int>                  group_int_list;
Group<int, Vector>          group_int_vector;
```

Templates (V)

- Nueva palabra reservada **typename**, permite ayudar al compilador a saber lo que es un 'tipo' perteneciente a un template:

2 usos de typename

```
// 1
template<typename T> class List;
...
// 2
template<class T>
class Wrapper
{
  // ...
  typename T::X data;   // X es un tipo del parametro T
};
```

Templates (VI)

- Instanciación explícita, ahora existe una sintaxis estándar para hacerla, antes cada compilador lo hacía (si podía) a su manera:

Instanciación explícita

```
template<typename T>
class List
{
  // ...
};

template List<int>;  // La instanciacion
```

Clase auto_ptr (I)

- Es una clase template declarada en la cabecera estándar <memory>.
- Los objetos de esta clase se crean con un puntero a un objeto de otra clase reservado dinámicamente, aunque también se pueden crear vacíos y asignar el puntero posteriormente.
- Elimina la necesidad de liberar memoria:

auto_ptr

```
#include <memory>
void f() {
  auto_ptr<double> dptr(new double(0.0));
  ...
  *dptr = 4.7;
}
```

Clase auto_ptr (II)

- Dispone de los métodos get, release y reset.
- No se debe usar un objeto `auto_ptr` como elemento de un contenedor de `STL`.

Funciones virtuales

- Las funciones virtuales de una clase derivada de otra no tienen que tener exáctamente el mismo prototipo que la función virtual de la clase base.
- Pueden diferir en el tipo del resultado si éste en la función de la clase derivada es un puntero o una referencia a una clase derivada del tipo del resultado de la función en la clase base:

Funciones virtuales

```
class Base {
  public:
    virtual Base* clonar();
};
class Derivada : public Base {
  public:
    virtual Derivada* clonar(); // Con enlace dinámico.
};
```

Compilador de C++ de GNU (I)

- Este compilador se adhiere al estándar de C++ además de proporcionar actualizaciones del mismo cada poco tiempo.
- La mayoría de distribuciones soportan la instalación en 'paralelo' de distintas versiones del compilador.
- La versión actualmente instalada la puedes consultar con la orden: `g++ -v`.
- Hay un cambio importante en la forma de nombrar los identificadores generados —**ABI**— por parte de los compiladores de la versión '`2.x.y`' y de la '`3.x.y`'.

Compilador de C++ de GNU (II)

- La última versión estable es la `4.5.2`, liberada el 16 de diciembre de 2010 y se trabaja en la siguiente versión de desarrollo, la `4.6.x`.
- Está disponible para la mayoría de plataformas, Unix, Linux, Windows y también...
- Para la mayoría de microprocesadores `IA32/x86`, `IA64`, `AMD32/AMD64`, `HPA-RISC`, `SPARC`, `PowerPC-32/PowerPC-64`, `MC-680XX`, `Alpha`, etc...
- Puedes conseguirlo en http://gcc.gnu.org .
- Se distribuye bajo la licencia de GPL.

Esto... ¿Qué hay de nuevo viejo?

- Actualmente usamos C++03, el cual es una ligera corrección sobre C++98.
- Durante mucho tiempo se pensó que habría un C++0x... casi, casi fue $x = 9$, hoy día se cree que sí habrá un C++1x.
- Los cambios son más notables a nivel de bibliotecas que en el núcleo del lenguaje.
- Boost .
- Más información en la página de Stroustrup o en la wikipedia .
- Versiones recientes de g++ van incorporando poco a poco estos añadidos, debes usar la opción de compilación: '`-std=c++0x`'. Consulta esta página para saber lo que soporta la versión de tu compilador g++.