Programación genérica y templates en C++

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

Programación genérica

- Motivación
- Templates
- Internals

Programación genérica

Motivación

Juego de buscar diferencias

```
1  class Array_int {
2     int data[64];
3
4     public:
5     void set(int p, int v){
6        data[p] = v;
7     }
8
9     int get(int p) {
10       return data[p];
11    }
12 };
```

```
1  class Array_char {
2     char data[64];
3
4     public:
5     void set(int p, char v){
6        data[p] = v;
7     }
8
9     char get(int p) {
10       return data[p];
11    }
12 };
```

Reserva de espacio distintos Invocación de código distintos: operador asignación Operador copia también (y hay otros más...)

- Imaginemos que necesitamos un array de `64 int`s asi como también de `64 char`s. De las dos implementaciones, que diferencias hay?
- Aunque no lo parezca, hay diferencias importantes desde el punto de vista del compilador y del código máquina generado.
- Primero, reservan espacios distintos: `64*sizeof(int)` contra `64*sizeof(char)`
- Segundo, invocan a código (operadores) distintos: por ejemplo el operador asignación
- Otros códigos también: el constructor por copia, posiblemente el constructor por default y el destructor.
- Y todas estas diferencias por tan solo debido al cambio del tipo `int` por `char`

Alternariva I: void*

```
1  class Array {
2     void *data;
3     size_t sizeobj;
4
5     public:
6     void set(int p, void *v) {
7        memcpy(&data[p*sizeobj],
8               v,   sizeobj);
9     }
10
11    void* get(int p) {
12       return &data[p*sizeobj];
13    }
14
15    Array(size_t s) : sizeobj(s) {
16       data = malloc(64 * sizeobj);
17    }
```

```
class Array_int {
   int data[64];


   public:
   void set(int p, int v){
      data[p] = v;
   }


   int get(int p) {
      return data[p];
   }
```

- Una alternativa al código repetido es usar un `void*` y el heap.
- Tendremos que hacer la copia bit a bit (no se llama a ningun constructor por copia u operador asignación). Esto puede traer varios problemas para objetos que necesitan copiarse de manera no tan trivial.
- Con el `void*` ganamos generalidad, pero nos arriesgamos a castear manzanas con bananas.
- Tenemos que saber cual es el tamaño del objeto.

void* nightmare

```
1  // Array version void* (enjoy!)
2
3  Array my_ints(sizeof(int));
4
5  int i = 5;
6  my_ints.set(0, &i);
7
8  int j = *(int*)my_ints.get(0);
```

```
// Array original

Array_int my_ints;


my_ints.set(0, 5);

int j = my_ints.get(0);
```

La implementación con `void*` es genérica pero...
no podemos usar literales; tenemos que castear! tenemos que dereferencear;

- No podemos guardar un literal `set(0, 1)`; tenemos que usar una variable `set(0, &i)`
- La copia del objeto retornado es hecha en el caller
- Es trivial cometer un error de casteo!

Alternativa II: Precompilador mágico

```
1  #define MAKE_ARRAY_CLASS(TYPE)\\
2    class Array_##TYPE          \\
3       TYPE data[64];           \\
4                                \\
5       public:                  \\
6       void set(int p, TYPE v){\\
7          data[p] = v;          \\
8       }                        \\
9                                \\
10      TYPE get(int p) {        \\
11         return data[p];       \\
12      }                        \\
13   }//<- fin de la macro sin ;
```

```
class Array_int {
   int data[64];

   public:
   void set(int p, int v) {
      data[p] = v;
   }

   int get(int p) {
      return data[p];
   }
}; // aca si incluyo un ; !!
```

```
1  MAKE_ARRAY_CLASS(int);  // instanciacion de las clases
2  MAKE_ARRAY_CLASS(char); // Array_int y Array_char
```

- La idea es crear una macro para crear múltiples clases parecidas. No esta mal, pero es muy díficil de debugear.
- Requiere la instanciación explícita de las clases y es fácil que alguien instancie dos veces la misma clase (llame a la macro dos veces con los mismos argumentos).
- Observación: cuando se haga la macro, no poner el `;` al final de esta!

Programación genérica

Templates

Templates: un único código para gobernarlos a todos

```
1  template<class T>
2  class Array {
3     T data[64];
4
5     public:
6     void set(int p, T v) {
7        data[p] = v;
8     }
9
10    T get(int p) {
11       return data[p];
12    }
13 };
```

```
class Array_int {
   int data[64];

   public:
   void set(int p, int v) {
      data[p] = v;
   }

   int get(int p) {
      return data[p];
   }
};
```


- La idea es similar a la alternativa del precompilador: usar el mismo código pero reemplazando el tipo particular `int/char` por uno genérico `T`.
- Pero a diferencia del precompilador, el código template es procesado por el compilador: es más seguro, hay chequeo de tipos y los errores estan mejor explicados (bueno, hasta cierto punto)

Templates: un único código para gobernarlos a todos

```
1  Array<int> my_ints;
2
3  my_ints.set(0, 5);
4  int j = my_ints.get(0);
```

```
Array_int my_ints;

my_ints.set(0, 5);
int j = my_ints.get(0);
```

Usamos `Array<int>` para instanciar el array y la clase si no fue ya instanciada.

Programación genérica

```
template<class T, class U>
struct Dupla {
   T first;
   U second;
};

template<class T=char, int size=64>
class Array {
   T data[size];
};

Array<> a; // T = char, size = 64
Array<int, 32> b;
```

```
template<class T>
void swap(T &a, T &b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

- Containers y algoritmos
- Múltiples parámetros
- Valores por default
- Funciones templates

- No solo las clases pueden ser templates, los `struct` y las funciones también.
- Se puede parametrizar por tipo (`class T`) o por una constante (`int size`).
- Como todo parámetro, pueden tener un default.

Deducción automática de tipos

```
1  template<class T>
2  void foo(T i) {
3      // ...
4  }
5
6
7  foo<int>(1);  // T = int (explicit)
8  foo(2);       // T = int (automatic)
9
10 foo<char>(3); // T = char (explicit)
```

- Al llamar a una función template podemos especificar sobre que tipos estamos trabajando o podemos dejar que el compilador lo deduzca automáticamente basandose en los parámetros.
- Tener en cuenta que la deducción automática puede no tener el efecto que uno quiere pues no siempre es fácil determinar el tipo de los parámetros: el número `3` es un `int` o un `char`?

Optimización por tipo - Especialización de templates

```
1  template<class T>          // Template
2  class Array { /*...*/ };   // anterior
3
4  template<>
5  class Array<bool> {
6     char data[64/8];
7
8     public:
9     void set(int p, bool v) {
10       if (v)
11          data[p/8] = data[p/8] | (1 << (p%8));
12       else
13          data[p/8] = data[p/8] & ~(1 << (p%8));
14    }
15
16    bool get(int p) {
17       return (data[p/8] & (1 << (p%8))) != 0;
18    }
```

- Se permite definir una implementación específica para un tipo en especial.
- La especialización de templates es usado en casos de optimización o algun otro tipo de customización
- Cuando se instancie un array de tipo `Array<bool>`, se utilizará la implementación optimizada, mientras que el template `Array<T>` genérico se usara en el resto de los casos. El compilador siempre elegirá la especialización mas específica.
- La versión especializada debe definir los mismo métodos que su par genérico, pero la implementación puede ser completamente distinta.

Polimorfismo en tiempo de compilación

```
1  template<class T>
2  bool cmp(T &a, T &b) {
3     return a == b;
4  }
```

La especialización no solo sirve para optimizar sino para hacer código mas razonable.

```
5  template<>
6  bool cmp<const char*>(const char* &a, const char* &b) {
7     return strncmp(a, b, MAX);
8  }

9  cmp(1, 2);
10 cmp("hola", "mundo");
```


- No tiene mucho sentido comparar dos punteros a `char`, tal vez tiene más sentido hacer una comparación de strings.
- Es como una especie de polimorfismo en tiempo de compilación pues el código que se ejecuta depende del tipo de sus argumentos aunque esta decisión se toma en tiempo de compilación.

Programación genérica

Internals

Detras de la magia

Veamos las implicaciones de este código:

```
1 Array<int> my_ints;
2 my_ints.get(0);
3
4 Array<int> other_ints;
5 other_ints.set(0,1)
```


Detras de la magia: generación de código mínimo

```
1 Array<int> my_ints;
```

- No existe la clase `Array<int>`
  - Se busca . . .
    - un template especializado `Array<T>` con `T = int` (no hay)
    - un template parcialmente especializado (no hay)
    - un template genérico `Array<T>` (encontrado!)
  - Se instancia la clase `Array<int>`
  - Se crea solo código para el constructor y destructor.
- Se crea código para llamar al constructor e instanciaciar el objeto `my_ints`

```
2 my_ints.get(0);
```

- No está creado el código para el método `Array<int>::get`, se lo crea y compila.
- Se crea código para llamar al método.

- Código template no usado es código no compilado: es muy fácil creer que algo esta bien codeado y darnos cuenta al momento de usarlo que no lo está.
- C++ solo generara código desde un template si lo necesita y si solo no existe previamente.

Detras de la magia: generación de código mínimo

```
4 Array<int> other_ints;
```

- Ya existe la clase `Array<int>`
- Directamente se crea código para llamar al constructor.

```
5 other_ints.set(0, 1);
```

- No está creado el código para el método `Array<int>::set`, se lo crea y compila.
- Se crea código para llamar al método.

Copy Paste Programming automático

```
1  template<class T>
2  class Array { /*...*/ };
3
4  class A { /*...*/ };
5  class B: public A { /*...*/ };
6  class C { /*...*/ };
7
8  Array<A*> a;
9  Array<B*> b;
10 Array<C*> c;
11 Array<A>  d;
12 Array<B>  e;
13 Array<A>  f;
```

Cuántas clases `Array`s se construyeron? 5! Un `Array` para `A*`, otro para `B*`, ... Hay código copiado y pegado 5 veces (code bloat).

- C++ simplemente toma el template y de él genera código al mejor estilo copy and paste
- Como los tipos `A*`, `B*` y `C*` son tipos distintos, C++ generara 3 clases una para cada uno de esos tipos usando como `Array<T>` como template: esto termina en un ejecutable mucho mas grande de lo necesario (code bloat).

Especialización parcial de templates

```
1  template<class T> // Template generico
2  class Array { /*...*/ };
3
4  template<>        // Especializacion completa para void*
5  class Array<void*> { /*...*/ };
6
7  template<class T> // Especializacion parcial para T*
8  class Array<T*> : private Array<void*> {
9     public:
10    void set(int p, T* v) {
11       Array<void*>::set(p, v);
12    }
13
14    T* get(int p) {
15       return (T*) Array<void*>::get(p);
16    }
17 };
```


- El problema de la implementación original que usaba `void*` eran los peligrosos casteos y la pérdida de chequeos de tipos.
- Una especialización parcial nos permite encapsular los casteos en un template, liberando al usuario de ellos mientras que la mayoria de la implementación del container esta contenida en el `Array<void*>`

Especialización parcial de templates

```
8  Array<A*> a;
9  Array<B*> b;
10 Array<C*> c;
11 Array<A>  d;
12 Array<B>  e;
13 Array<A>  f;
```

Y ahora, cuántas clases `Array`s se construyeron? 6!

- 2 clases usando `Array<T>` con `T = A` y `T = B`
- 3 clases usando `Array<T*>` con `T = A`, `T = B` y `T = C`
- 1 clase más para `Array<void*>`

Más clases, es peor!? Como `Array<T*>` son puros casteos, el compilador se encargará de hacer inline y remover el código superfluo. Más compacto y más rápido.

- Para los tipos `A*`, `B*` y `C*` se crearan 3 clases con `Array<T*>` como su template y como este hereda de `Array<void*>` tambien se creara esa clase dandonos un total de 4 clases. Para los tipos A y B se crearan dos clases adicionales usando `Array<T>` como template dando un conteo total de 6 clases.
- Pero observando en detalle, `Array<T*>` solo tiene código de casteo (seguro) y métodos de una sola línea. Las clases instanciadas para los tipo `A*`, `B*` y `C*` no supone un aumento considerable del código (evitamos el code bloat).
- Mas aun, con métodos de una sola línea, puede que el compilador los haga inline, optimizando el tamaño del ejecutable y el tiempo de ejecución.

Resumen - Templates

- Jamás implementar un template al primer intento. Crear una clase prototipo (`Array_ints`, testearla y luego pasarla a template `Array<T>`
- Si se va a usar el templates con punteros, evitar el code bloat implementando la especialización `void*` (`Array<void*>`) y luego la especialización parcial `T*` (`Array<T*>`).
- Opcionalmente, implementar especializaciones optimizadas (`Array<bool>`)

Appendix

Referencias

Referencias I

- http://cplusplus.com
- Herb Sutter.
  Exceptional C++: 47 Engineering Puzzles.
  Addison Wesley, 1999.
- Bjarne Stroustrup.
  The C++ Programming Language.
  Addison Wesley, Fourth Edition.