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 };
```

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

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 };
```

Invocación de código distintos: operador asignación

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 };
```

Operador copia también (y hay otros más...)

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];
   }
```

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];
   }
```

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];
   }
```

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...

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;

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...
tenemos que castear!

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...

tenemos que dereferencear;

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
```

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];
   }
};
```

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];
   }
};
```

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];
   }
};
```

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

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;
}
```

- Múltiples parámetros

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;
}
```

- Valores por default

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;
}
```

- Funciones templates

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)
```

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    }
```

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    }
```

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    }
```

Polimorfismo en tiempo de compilación

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

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  }
```

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");
```

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`

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.

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?

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).

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 };
```

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 };
```

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 };
```

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 };
```

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 };
```

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?

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!?

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.

Resumen - Templates

- Jamás implementar un template

Resumen - Templates

- Jamás implementar un template al primer intento. Crear una clase prototipo (`Array_ints`, testearla y luego pasarla a template `Array<T>`

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*>`).

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.