Templates

Taller de Programación I - FIUBA

Cuales son las diferencias?

```
class Array_ints {
  int data[64];
public:
  void set(int p, int v) {
    data[p] = v;
  }
  int get(int p) {
    return data[p];
  }
};
```

```
class Array_chars {
  char data[64];
public:
  void set(int p, char v) {
    data[p] = v;
  }
  char get(int p) {
    return data[p];
  }
};
```

Espacios distintos: 64***sizeof(int)** contra 64***sizeof(char)**

Llaman a código (operador asignación) distintos.

Otros también: el constructor por copia, etc

Templates

```
template<class T>
class Array {
  T data[64];
public:
  void set(int p, T v) {
    data[p] = v;
  }
  T get(int p) {
    return data[p];
  }
};
```

```
class Array_chars {
  char data[64];
public:
  void set(int p, char v) {
    data[p] = v;
  }
  char get(int p) {
    return data[p];
  }
};
```

Con sólo **Array<T>** se puede generar automáticamente variables del mismo código.

Array<int> crea un array de 64 **int** (como la clase Array_ints)

Array<char> crea un array de 64 **char** (como la clase Array_chars)

Templates: Fácil Uso

```
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.

Más Templates...

```
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;
}
```

¿Cómo se resuelve en tiempo de Compilación?

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

Paso a paso...

```
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`

Paso a paso...

```
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.

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

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

Paso a paso...

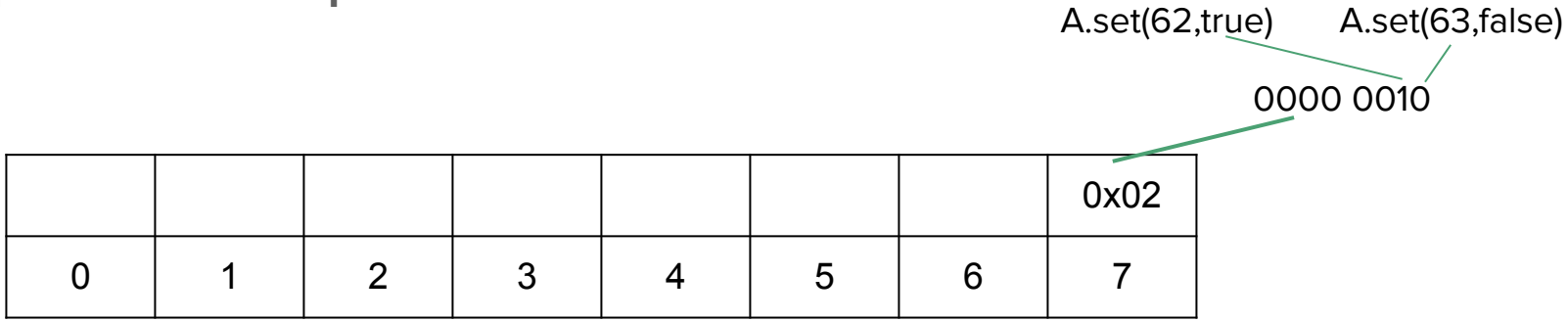
```
5 other_ints.set(0, 1);
```

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

Optimización por tipo: Ej. de necesidad - Array<bool>

Implementación Obvia/Automática:

Implementación Optimizada:



Optimización por tipo

```
template<>
class Array<bool> {
  unsigned char data[8];
public:
  void set(int p, bool v) {
    if (v) { data[p/8] = data[p/8] | (1 << (p%8));  }
    else { data[p/8] = data[p/8] & ~(1 << (p%8));  }
  }
  bool get(int p) {
    return (data[p/8] & (1 << (p%8))) != 0;
  }
};
```

Array<bool> usará el código especializado.

Con cualquier otro tipo se usará el código genérico **Array<T>**

Esto permite tener código eficiente para tipos particulares.

Se requiere implementar primero **Array<T>** antes de la especialización.

Especialización por tipo

```
template<class T>
bool cmp(T a, T b) {
    return a == b;
}

template<>
bool cmp(const char* a, const char* b) {
    return strncmp(a, b, SZ);
}
```

La especialización de templates es también para tener implementaciones más apropiadas para un tipo en particular.

cmp(*“foo”*, *“foo”*) podría dar **false** con la implementación genérica por comparar sólo punteros

La especialización para **const char*** es más apropiada.

Funciona como polimorfismo pero en tiempo de compilación.

Especialización parcial de templates

```
template<class T>
class Array<T*> : private Array<void*> {
public:
  void set(int p, T* v) {
    Array<void*>::set(p, v);
  }
  T* get(int p) {
    return (T*) Array<void*>::get(p);
  }
};
```

Cuando se quiera un array de punteros, se usará el código especializado para ellos (**Array<T*>**).

Esta especialización parcial reutiliza la especialización completa para **void***. El compilador directamente no genera código lo que evita el **code bloat**.

Se requiere implementar primero **Array<T>** y **Array<void*>** antes de la especialización.

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 };
```

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)
```

Resumen

- Jamás implementar un template al primer intento. Crear una clase prototipo (**Array_ints** , probarla 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>**)