Programación genérica y templates en C++

[1] Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

1. Programación genérica
   - Motivación
   - Templates
   - Internals
2. Standard Template Library
   - Containers
   - Iteradores
   - Algoritmos

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...)

```
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 ints asi como también de 64 chars. 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 el 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];
   }
```

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...
- 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;

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
```

Alternativa II: Precompilador mágico

```
#define MAKE_ARRAY_CLASS(TYPE)\\
  class Array_##TYPE          \\
     TYPE data[64];           \\
                              \\
     public:                  \\
     void set(int p, TYPE v){\\
        data[p] = v;          \\
     }                        \\
                              \\
     TYPE get(int p) {        \\
        return data[p];       \\
     }                        \\
  }//<- 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 ; !!
```

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

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

Un único código para gobernarlos a todos

Templates en C++

```
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];
   }
};
```

```
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 (hasta cierto punto)

Programación genérica

```
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 (y la clase si no fue ya instanciada).

Containers y algorítmos genéricos

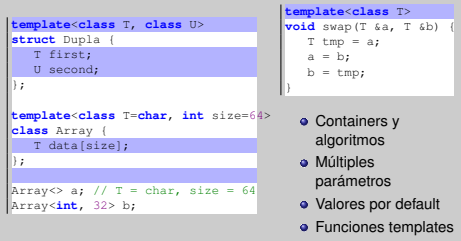
```
template<class T, class U>
struct Dupla {
    T first;
    U second;
};

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 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)
```

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 automaticamente 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

Optimización de código

```
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    }
```

```
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 use Array<bool>, se utilizara la implementación optimizada, el 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

La especialización no solo sirve para optimizar.

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

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");
```

Polimorfismo en tiempo de compilación
La especialización no solo sirve para optimizar.

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

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.

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
Instanciación de templates

```
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.

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 no usado es código no compilado: es muy fácil creer que algo esta bien codeado para luego usarlo y darse cuenta de que no es asi.
- C++ solo generara código desde un template si lo necesita y si solo no existe previamente.

Detras de la magia
Generación de código sólo si es usado

```
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).

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: esto termina en un ejecutable mucho mas grande de lo necesario (code bloat).

Especialización parcial

```
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

```
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
- 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*>

Especializacion parcial

```
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.

```
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.

- Al final, Array<T*> solo tendra código de casteo (seguro) y métodos de una sola línea. Crear instanciar las clases para A*, B*, 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 tamanõ 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>`
- Implementar la especialización `void*` (`Array<void*>`)
- Implementar la especialización parcial `T*` (`Array<T*>`) para evitar el code bloat.
- Opcionalmente, implementar especializaciones optimizadas (`Array<bool>`)

Containers - Programar en C++ y no en C con objetos

C es muy eficiente, pero tareas simples pueden resultar titánicas.

- Armar un vector que aumente de tamaño.
- Ordenar elementos.
- Remover duplicados.

C++ ofrece containers muy versátiles y eficientes. En C++, hay que programar en C++ y no en C!

Strings

Útil para el manejo de textos

```
1  std::string saludos = "Hola mundo!";
2
3  //Substring "ola"
4  std::string otro_string = saludos.substr(1, 3);
5
6  // Comparacion de strings
7  bool son_iguales = (saludos == otro_string);
8
9  // Concatenacion
10 otro_string = "H" + otro_string + " mundo!";
```

Strings
Útil para el manejo de textos

```
1  std::string saludos = "Hola mundo!";
2
3  //Substring "ola"
4  std::string otro_string = saludos.substr(1, 3);
5
6  // Comparacion de strings
7  bool son_iguales = (saludos == otro_string);
8
9  // Concatenacion
10 otro_string = "H" + otro_string + " mundo!";
```

- std::string es un container flexible y poderoso pensado en el procesamiento de texto.
- No usar este container para blobs binarios. En general es mejor std::vector

Vector
y adios al new[]

Por ser RAII, es una alternativa al `new[]` (excepto para `Vector<bool>`)

```
1 std::vector<char> data(256, 0);
2
3 char *buf = &data[0];
4 file.read(buf, 256);
```

- Para C++98 (depende del compilador)
- Para C++11 es un estándar

Arrays asociativos

En vez de indexarse por números, son indexados por claves arbitrarias

```
1 std::map<std::string, std::string> traductor;
2
3 traductor["hola"] = "hello";
4 traductor["adios"] = "bye";
```

Arrays asociativos

```
1  std::map<char, int> freq_de_caracteres;
2
3  std::string texto = "Lorem ipsum dolor sit amet, " /*...*/
4
5  for (int i = 0; i < texto.length(); ++i) {
6      char c = texto[i];
7      if (freq_de_caracteres.count(c)) {
8          freq_de_caracteres[c] += 1;
9      }
10     else {
11         freq_de_caracteres[c] = 1;
12     }
13 }
```

Remover duplicados con Set

```
1 void remover_duplicados(std::list<int> &lista) {
2     std::set<int> unicos(lista.begin(), lista.end());
3     std::list<int> filtrado(unicos.begin(), unicos.end());
4
5     lista.swap(filtrado);
6 }
```

Remover duplicados con Set

```
1 void remover_duplicados(std::list<int> &lista) {
2     std::set<int> unicos(lista.begin(), lista.end());
3     std::list<int> filtrado(unicos.begin(), unicos.end());
4
5     lista.swap(filtrado);
6 }
```

- Muchos containers pueden construirse a partir de otros a traves de dos iteradores que marcan desde donde y hasta donde se deben copiar los elementos.
- Dado que std::set ignora los duplicados esto es una forma interesante de resolver el problema.
- Nota: como side effect, std::set nos dejara ordenados los elementos que no fueron removidos.
- Para finalizar se podría haber hecho lista = filtrado; pero eso generaría otra copia mas. El método swap cambia los containers internos y resulta mas eficiente que una copia.

Y los clásicos de hoy y de siempre

```
1  std::list<int> lista;    // doubled "linked" list
2
3  lista.push_back(1);      lista.push_front(2);
4  lista.insert(...);       lista.erase(...);
5
6  std::stack<int> pila;
7
8  pila.push_back(1);    // push
9  pila.pop_back();      // pop (no devuelve nada!)
10
11 std::queue<int> cola;
12
13 cola.push_back(1);    // push
14 cola.pop_front();     // pull (no devuelve nada!)
15
16 // Para obtener el valor de un stack/queue
17 int i = pila.back(); int j = cola.front();
```

Y los clásicos de hoy y de siempre

```cpp
std::list<int> lista;   // doubled "linked" list

lista.push_back(1);     lista.push_front(2);
lista.insert(...);      lista.erase(...);

std::stack<int> pila;

pila.push_back(1);   // push
pila.pop_back();     // pop (no devuelve nada!)

std::queue<int> cola;

cola.push_back(1);   // push
cola.pop_front();    // pull (no devuelve nada!)

// Para obtener el valor de un stack/queue
int i = pila.back(); int j = cola.front();
```

- Los clásicos, lista, colas, pilas (incluso hay colas prioritarias)
- front() y back() retornan una copia del primer y último elemento mientras que pop_front() y pop_back() los remueven pero sin devolverlos.

Custom, custom, custom

Containers adapters, allocators

```
1 std::stack<int> pila;
2
3 std::stack<int, std::vector<int> > pila;
4
5 std::stack<int, std::vector<int, MyAlloc<int> > > pila;
```

- Algunos containers son en realidad adapters, podemos cambiar su implementación.
- Más aun, podemos cambiar en donde allocan los objetos (no usan new!)

Containers - Resumen

- Esto no es C. Hay una lib estándar más completa. Usarla
- Los containers pueden ser muy eficientes si los eligen correctamente.
- Y aun así los podemos configurar más con allocators y otros.

- Algunos containers son en realidad adapters. Esto significa que podemos cambiar el objeto interno que implementa realmente el container.
- Aun mas, los containers usan un objeto "allocator" por default para acceder a la memoria (no hacen "new" sino que llaman a un método allocate). Este objeto allocator se puede cambiar tambien! Se podria implementar un stack que en vez de usar la memoria use un archivo o una shared memory.

Iteradores - Abstracción del container

Muchos algoritmos son independientes del container sobre el que trabajan; sólo necesitan una forma de recorrerlos.

- Sumatoria de números de un container.
- Imprimir sus elementos.
- Búsqueda secuencial.

Los iteradores abstraen la forma de recorrer un container.
En C++, hay distintas clases de iteradores pero cada container sólo implementa aquellos que pueda hacerlos eficientemente.

Iteradores

```
1 | // Todos pueden
2 | ++it;          it++;
3 | it = itx;      Iter it(itx);
```

```
4 | // Input (mutables)
5 | *it = t;      *it++ = t;
6 | //
```

```
4 | // Output (inmutables)
5 | t = *it;      it->m;
6 | it == itx;    it != itx;
```

```
7 | // Bidirectional
8 | --it;          it--;
```

```
 9 | // RandomAccess (aka pointers)
10 | it + n;       it - n;       it[n];
11 | it += n;      it + itx;     it < n;
```

```
1 | // Todos pueden
2 | ++it;        it++;
3 | it = itx;    Iter it(itx);
```

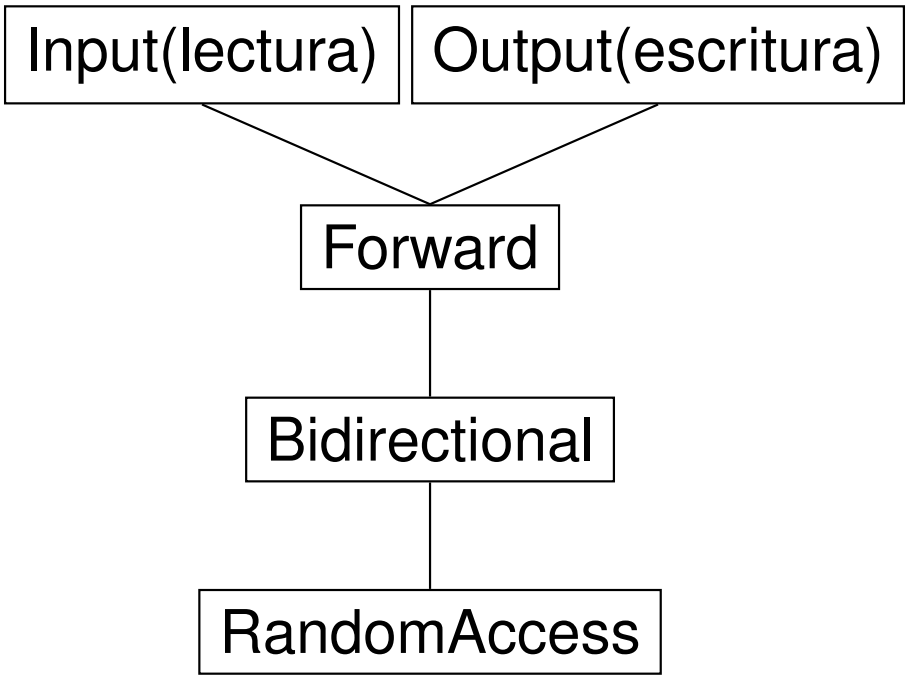
```
4 | // Input (mutables)          4 | // Output (inmutables)
5 | *it = t;     *it++ = t;     5 | t = *it;     it->m;
6 | //                          6 | it == itx;   it != itx;
```

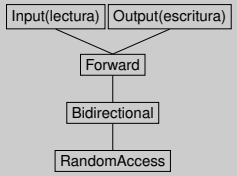
```
7 | // Bidirectional
8 | --it;        it--;
```

```
 9 | // RandomAccess (aka pointers)
10 | it + n;      it - n;      it[n];
11 | it += n;     it + itx;    it < n;
```

- Todos los iteradores pueden moverse hacia adelante (forward) y copiarse.
- Algunos soportan el dereferenciado para la asignación (input) mientras que otros lo soportan para la lectura del objeto apuntado (output). Estos últimos permiten realizar comparaciones por igualdad.
- Otros soportan moverse hacia atras (backward)
- Y finalmente existen algunos iteradores que permiten moverse de forma no secuencial como si fueran punteros.

Iteradores


Input(lectura)
Output(escritura)
Forward
Bidirectional
RandomAccess



- Existe una jerarquía de iteradores. Todos los iteradores con RandomAccess son iteradores Bidirectionales pero no todos los Bidirectionales tienen RandomAccess

Lifetime de los iteradores

Los iteradores son válidos mientras no se modifique el container

Mal:

```
1 std::list<int>::iterator it = lista.begin();
2 for (; it != lista.end(); ++it)
3    if (*it % 2 == 0)  // remover si es par
4       lista.erase(it);
```

Bien:

```
1 std::list<int> tmp;
2 std::list<int>::iterator it = lista.begin();
3 for (; it != lista.end(); ++it)
4    if (*it % 2 != 0) // copiar si no es par
5       tmp.push_back(*it);
6 lista.swap(tmp);
```

Mejor!

```
1 bool es_par(const int &i) { return i % 2 == 0; }
2 std::remove_if(lista.begin(), lista.end(), es_par);
```

Lifetime de los iteradores

Los iteradores son válidos mientras no se modifique el container

Mal:

```
1 std::list<int>::iterator it = lista.begin();
2 for (; it != lista.end(); ++it)
3    if (*it % 2 == 0)  // remover si es par
4       lista.erase(it);
```

Bien:

```
1 std::list<int> tmp;
2 std::list<int>::iterator it = lista.begin();
3 for (; it != lista.end(); ++it)
4    if (*it % 2 != 0) // copiar si no es par
5       tmp.push_back(*it);
6 lista.swap(tmp);
```

Mejor!

```
1 bool es_par(const int &i) { return i % 2 == 0; }
2 std::remove_if(lista.begin(), lista.end(), es_par);
```

- En general un iterador es válido mientras que su container no cambie: iterar un container para removerle algunos elementos suele ser el clásico bug.

Algortimos genéricos - Abstracción de código

```
1  // no compila por un mini detalle (typename)
2  template <class Container, class Val>
3  Container::iterator find(
4                           Container &v,
5                           const Val &val) {
6    Container::iterator it = v.begin();
7    Container::iterator end = v.end();
8
9    while (it != end and val != *it) {
10       ++it;
11   }
12
13   return it;
14 }
```

Intermezzo: typenames

Para diferenciar entre un método y un subtipo de clase

```
1  struct List {
2      struct iterator {
3          /* ... */
4      };
5
6      List::iterator begin() {
7          return List::iterator(/*...*/);
8      }
9  };
```

- `List::iterator` hace referencia a un tipo (el struct iterator dentro de List)
- `List::begin` hace referencia a un metodo de List

Intermezzo: typenames

Para diferenciar entre un método y un subtipo de clase

Cómo sabe el compilador que `Container::iterator` es un tipo y no un método si ni siquiera sabe que es `Container`?

```
1 template <class Container, class Val>
2 Container::iterator find(...) { ... }
```

La keyword **`typename`** permite diferenciar un método de un tipo.

```
1 template <class Container, class Val>
2 typename Container::iterator find(...) { ... }
```

Algortimos genéricos - Abstracción de código

```
1  // ahora si compila (siempre que Container y Val cumplan)
2  template <class Container, class Val>
3  typename Container::iterator find(
4                                    Container &v,
5                                    const Val &val) {
6    typename Container::iterator it = v.begin();
7    typename Container::iterator end = v.end();
8
9    while (it != end and val != *it) {
10       ++it;
11   }
12
13   return it;
14 }
```

Programar los algoritmos con iteradores y no con containers

```
1 template <class Iterator, class Val>
2 Iterator find(Iterator &it, Iterator &end, const Val &val) {
3     while (it != end and val != *it) {
4         ++it;
5     }
6
7     return it;
8 }
```

```
1 template <class Iterator, class Val>
2 Iterator find(Iterator &it, Iterator &end, const Val &val) {
3     while (it != end and val != *it) {
4         ++it;
5     }
6
7     return it;
8 }
```

- La mayoría de los algoritmos deberían escribirse en términos de iteradores: recibir y retornar iteradores, independizandose del container en cuestión.

Algortimos de la STL

Menos código, menos bugs!

For each, (también conocido como map)

```
1 | std::for_each(container.begin(), container.end(), func);
```

Algortimos de la STL

Menos código, menos bugs!

Como imprimir al stdout un container (útil para debug)

```
1 template<class T>
2 void print_to_cout(const T &val) {
3     std::cout << val << "␣";
4 }
5
6 std::list<int> l;
7 for_each(l.begin(), l.end(), print_to_cout<int>);
```

Algortimos de la STL

Menos código, menos bugs!

O con functors:

```
1  template<class T>
2  struct Printer {
3      std::ostream &out;
4
5      Printer(std::ostream &out) : out(out) {}
6
7      void operator()(const T &val) {
8          out << val << " ";
9      }
10 };
11
12 std::list<int> l;
13 for_each(l.begin(), l.end(), Printer<int>(std::cout));
```

Algortimos de la STL

Menos código, menos bugs!

Sorting

```
1 // usando el operador less < como ordenador
2 std::sort(container.begin(), container.end());
3
4 // usando la funcion/functor especifica
5 std::sort(container.begin(), container.end(), less_func);
6
7 // orden estable
8 std::stable_sort(container.begin(), container.end());
```

Searching (sobre containers ordenados)

```
1 // usando la misma funcion/functor que se uso para el
2 // ordenamiento (el operador less < es el default)
3 std::binary_seach(container.begin(), container.end(),
4                   val_to_be_found);
```

Algortimos de la STL

Con posibilidad de optimizaciones

Swap, con implementaciones especializadas para containers

```
1 int a = 1, b = 2;
2 std::swap(a, b);      // a == 2, b == 1 haciendo copias
3
4 std::list<int> a;
5 std::list<int> b;
6 std::swap(a, b);      // solo swap de punteros internos!
```

STL - Resumen

- Cuidado de no poner un `>>`, poner siempre un espacio entre ambos símbolos (en C++98, en C++11 ya arreglaron el problema). Por ejemplo `std::list<Array<int> >`
- El uso de templates, containers e iteradores puede dejar el código muy verbose, usar **`typedef`**
- Usar el operador de preincremento `++it` y no el de pos incremento para evitar copias.
- Busquen! `std::stack`, `std::queue`, `std::make_heap`, `std::set_intersection`, `std::set_union`, etc. Hay más contenedores y algoritmos listos para ser usados. Encuentrenlos y usenlos!

Referencias I

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