Templates en C++

Algoritmos y estructuras de datos II

31 de agosto de 2016

Clases en C++

Profundizando en clases...

- El const
- Constructor por defecto
- Constructor con parámetros
- Constructor por copia
- Listas de inicialización
- Destructor
- Operador de asignación

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de "sólo lectura" para esa función/método de clase.

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de "sólo lectura" para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado ("solo lectura" de nuevo);

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de "sólo lectura" para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado ("solo lectura" de nuevo);
- Un valor de retorno devuelto por referencia es de "sólo lectura" para el llamador de una función/método de clase.

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de "sólo lectura" para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado ("solo lectura" de nuevo);
- Un valor de retorno devuelto por referencia es de "sólo lectura" para el llamador de una función/método de clase.

¡const!

El const se usa para indicar que...

- Una variable local o de clase es una constante.
- Un parámetro pasado por referencia es de "sólo lectura" para esa función/método de clase.
- Un método de clase no modifica a **this**, es decir, a la instancia sobre la cual es llamado ("solo lectura" de nuevo);
- Un valor de retorno devuelto por referencia es de "sólo lectura" para el llamador de una función/método de clase.

El const es una manera de "especificar" el comportamiento del código con respecto a la lectura/escritura de valores. Conviene usarlo porque el compilador nos va a alertar de algunos errores comunes.

const, punteros y referencias

- El tipo **const T *** se leen como *puntero a un T constante*. El que es de sólo lectura es el T apuntado.

const, punteros y referencias

- El tipo **const T *** se leen como *puntero a un T constante*. El que es de sólo lectura es el T apuntado.
- El tipo **const T&** y se lee como *referencia constante a T*. El que es de sólo lectura es el T referenciado.

const, punteros y referencias

- El tipo **const T *** se leen como *puntero a un T constante*. El que es de sólo lectura es el T apuntado.
- El tipo **const T&** y se lee como *referencia constante a T*. El que es de sólo lectura es el T referenciado.
- Cuidado: **T** y **const T** son tipos distintos!!

const - conversiones

- Cosas del tipo **T** se pueden usar en lugares donde es necesario algo del tipo **const T** (el compilador hace la conversión)...

const - conversiones

- Cosas del tipo **T** se pueden usar en lugares donde es necesario algo del tipo **const T** (el compilador hace la conversión)...
- ... pero cosas del tipo **const T** NO se pueden usar en lugares donde es necesario algo del tipo **T**! Se les ocurre por qué?

const - conversiones

- Cosas del tipo **T** se pueden usar en lugares donde es necesario algo del tipo **const T** (el compilador hace la conversión)...
- ... pero cosas del tipo **const T** NO se pueden usar en lugares donde es necesario algo del tipo **T**! Se les ocurre por qué?
- **const T** es *más restrictivo* que **T**. El compilador no va a hacer la conversión de **const T** a **T** porque es inseguro (estaría convirtiendo algo de solo lectura en algo escribible).

const - ejemplos

```
// Una constante numerica
const double PI = 3.1416;

// Calcula el area: no modifica a c
double area(const Circulo & c);

// El punto devuelto se mira y no se toca.
// Tampoco modifica al circulo
const punto2d & Circulo::centro() const;

// este punto si se puede modificar! (ojo!)
// tampoco modifica al circulo
punto2d & Circulo::centro() const;
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() { return 7; }
  // mas codigo...
};

// esto no va a compilar... por?
void mostrarVidas(const GatoMontes& g) {
  cout << g.vidas() << endl;
}
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() const { return 7; }
  // mas codigo...
};

// Ahora si!
void mostrarVidas(const GatoMontes& g) {
  cout << g.vidas() << endl;
}
```

To const or not to const

```
class GatoMontes {
public:
  int vidas() { return 7; }
  // mas codigo...
};

// esto compila?
void mostrarVidas(GatoMontes g) {
  cout << g.vidas() << endl;
}

// cual es el sentido de esto?
void mostrarVidasBis(const GatoMontes g) {
  cout << g.vidas() << endl;
}
```

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.
- Estar atentos a las conversiones automáticas

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.
- Estar atentos a las conversiones automáticas
- Prestar especial atención cuando creamos nuestras propias clases:

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.
- Estar atentos a las conversiones automáticas
- Prestar especial atención cuando creamos nuestras propias clases:
  - qué métodos tienen const sobre el parámetro implícito **this**

Reflexiones sobre el const

- El const está aquí para ayudarnos a programar mejor y no para complicarnos la vida.
- Estar atentos a las conversiones automáticas
- Prestar especial atención cuando creamos nuestras propias clases:
  - qué métodos tienen const sobre el parámetro implícito **this**
  - y cuáles sobre los parámetros de entrada y/o de retorno.

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

- Constructor que toma cualquier tipo y cantidad de parámetros

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

- Constructor que toma cualquier tipo y cantidad de parámetros
- Si se define, el compilador no provee el constructor por defecto

Constructor por defecto

Constructor por defecto

- Constructor que no toma parámetros
- El compilador provee uno si no declaramos ningún constructor
- Ojo: se limita a usar los constructores por defecto de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por defecto, **no va a compilar**.

Constructor con parámetros

- Constructor que toma cualquier tipo y cantidad de parámetros
- Si se define, el compilador no provee el constructor por defecto
- Se pueden agregar tantos como se quiera a una clase

Ejemplo: constructor por defecto y con parámetros

```
class Test {
  // Aca tenemos Test() implicito
  int a;
};

class Test2 {
  // Al declarar este, no hay
  // constructor Test2() implicito
  Test2(long l);
};
```

Ejemplo: constructor por defecto y con parámetros

```
class Test {
  // Aca tenemos Test() implicito
  int a;
};

class Test2 {
  // Al declarar este, no hay
  // constructor Test2() implicito
  Test2(long l);
};
```

```
int main() {
  Test  unTest; // cual sera el valor de unTest.a ?
  Test2 elOtro; // compila?
  Test2 posta(200); // ahora si!
}
```

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar**.

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar**.
- Si alguno de los miembros es un puntero...

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar**.
- Si alguno de los miembros es un puntero...

Constructor por copia

- Constructor de la clase *T* que siempre toma un *const T&*
- Invocado cuando se pasan parámetros por copia.
- El compilador provee uno si no lo declaramos nosotros.
- Ojo: se limita a usar los constructores por copia de cada miembro de la clase (si esto es posible).
- Si alguno de los miembros no tiene constructor por copia, **no va a compilar**.
- Si alguno de los miembros es un puntero... ¡va a copiar el valor del puntero! Eso no puede terminar bien...

Constructor por copia - ejemplos

```
class Copiable {
  Copiable (const Copiable & c2) {
    this->a = NULL;
    // Lo tenemos que copiar
    // explicitamente (si corresponde)
    if (c2.a != NULL)
      this->a = new int(*(c2.a));
  }

  int* a;
};
```

Constructor por copia - ejemplos

```
class Copiable {
  Copiable (const Copiable & c2) {
    this->a = NULL;
    // Lo tenemos que copiar
    // explicitamente (si corresponde)
    if (c2.a != NULL)
      this->a = new int(*(c2.a));
  }

  int* a;
};
```

```
int main() {
  Copiable ccc; // como se construye ccc?
  Copiable otroC(ccc); // como se construye ccc?
}
```

Listas de inicialización

- Sintaxis usada en los constructores para construir variables miembro (algunas o todas). Importa el orden.
- Se vuelven indispensables cuando tenemos miembros sin constructor por defecto.

Listas de inicialización

- Sintaxis usada en los constructores para construir variables miembro (algunas o todas). Importa el orden.
- Se vuelven indispensables cuando tenemos miembros sin constructor por defecto.

```
// Esfera.h
class punto3d {
  double x,y,z;
  punto3d(double x0, double y0, double z0)
    : x(x0), y(y0), z(z0) {  }
};

class esfera {
  double radio; punto3d centro;
  esfera(); // necesita lista!
};
```

Listas de inicialización (2)

```
// Esfera.cpp
esfera::esfera()
: radio(1), centro(0,0,0)
{
}

// o equivalentemente:
esfera::esfera()
: centro(0,0,0)
{
  radio = 1;
}
```

Listas de inicialización - Otro ejemplo (3)

```
class Pp {
  private:
  int &valor;

  public:
  //Compila?
  Pp(int &v)
  {
    valor = v;
  }
  int sumarYRetornar();
};
```

Listas de inicialización - Otro ejemplo (3)

```
class Pp {
  private:
  int &valor;

  public:
  //Ahora s  .
  Pp(int &v) : valor(v) { };
  int sumarYRetornar();
};
```

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos?

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos?

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador "="
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador "="
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador “=”
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador "="
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.
- Si alguno de los miembros es un puntero...

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador "="
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.
- Si alguno de los miembros es un puntero...

Operador de asignación

- Es una función que se puede usar de manera *infija* mediante el operador "="
- Toma dos parámetros: el parámetro implícito *this* y un *const T&*. Devuelve un *T&*, cuál de los dos? Por convención **this*
- El compilador provee uno si no lo declaramos nosotros (déjà vu)
- Ojo: se limita a usar los operadores de asignación de cada miembro
- Si alguno de los miembros no tiene operador de asignación, **no va a compilar**.
- Si alguno de los miembros es un puntero... ¡va a copiar el valor del puntero! Eso no puede terminar bien...
- Se aplica sobre una instancia YA CONSTRUIDA: **¡hay que limpiar lo que ya estaba!**

Operador de asignación - ejemplos

```
class asignable {
  asignable& operator=(const asignable& a2) {
     if (this == &a2) return *this;  // a = a
     // HAY QUE BORRAR LO VIEJO!
     if (this->a != NULL) {
       delete this->a;
       this->a = NULL;
     }
     // copiamos si corresponde
     if (a2.a != NULL)
       this->a = new int(*(a2.a));
     return *this;
  }
  int* a;
};
```

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del *scope* de una variable

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del *scope* de una variable
- *No se lo llama explícitamente.*

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del *scope* de una variable
- *No se lo llama explícitamente.*
- Debe realizar todas las tareas de limpieza de memoria dinámica necesarias... ¡no queremos perder memoria!

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del *scope* de una variable
- *No se lo llama explícitamente.*
- Debe realizar todas las tareas de limpieza de memoria dinámica necesarias... ¡no queremos perder memoria!

Destructor

- Operación especial que se ejecuta al hacer *delete* de un puntero a un objeto, o al salir del *scope* de una variable
- *No se lo llama explícitamente.*
- Debe realizar todas las tareas de limpieza de memoria dinámica necesarias... ¡no queremos perder memoria!

```
class leaker {
  int * p;

  leaker(int tam){
    p = new int[tam];
   }

  ~leaker() {} // No hago nada
};
```

Destructor

```
class limpita {
  int * p;

  limpita(int tam) { p = new int[tam]; }
  ~limpita() { delete[] p; }
  // delete p hace lo mismo?
  // depende del compilador!!
  // puede borrar solo la primera posicion
  // delete[] es lo correcto
}
```

"Regla de tres"

Cuando nos veamos obligados a definir...

- ...el constructor por copia
- ...el operador de asignación
- ...o el destructor

...probablemente tengamos que definir los tres.

"Regla de tres"

Cuando nos veamos obligados a definir...

- ...el constructor por copia
- ...el operador de asignación
- ...o el destructor

...probablemente tengamos que definir los tres.

Atención

Estos tres se autogeneran por el compilador si no los declaramos, por lo que, si lo que el compilador autogenera no sirve en un caso, probablemente tampoco sirva en los demás.

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.
- Los constructores y destructores nos dan un forma de meternos con el scope de una variable y *manejar* su tiempo de vida.

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.
- Los constructores y destructores nos dan un forma de meternos con el scope de una variable y *manejar* su tiempo de vida.
- Podemos definir una función (el constructor) que se llama cuando *comienza* el scope de la variable y otra para cuando *finaliza* (el destructor)

Tiempo de vida de una variable (bis bis)

Constructores y destructor

- Nos permiten manejar explícitamente el tiempo de vida de las instancias de las clases que creo.
- Los constructores y destructores nos dan un forma de meternos con el scope de una variable y *manejar* su tiempo de vida.
- Podemos definir una función (el constructor) que se llama cuando *comienza* el scope de la variable y otra para cuando *finaliza* (el destructor)
- Respetar la convención: "El que lo crea, lo destruye".

Pasaje de parámetros por referencia o por copia

```
class Grandota {
  Grandota(){ a = new int[100000];}
  Grandota(const Grandota& gr) {
    this->a = new int[100000];
    for (int i=0; i < 100000, i++) {
      this->a[i] = gr.a[i];
    }
  }
  ~Grandota(){ delete [] a; }

  int* a;
};

void funPesada(Grandota g); //que recibe la funcion?
void funLiviana(Grandota& g); //y aca?
void funLivianaYSegura(const Grandota& g); //y aca?
```

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Destructor

- No usar new nos garantiza no perder memoria, y por lo tanto, no definir el destructor.
- Liberar con delete la memoria que pedimos con new, únicamente de nuestra clase y **no** en otras.

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes?

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes? Templates!

Preguntas existenciales al crear una clase en C++...

- Cómo sé si funciona bien? Testing!
- Cómo sé si no pierde memoria? Valgrind!
- Y si ahora quiero una lista de docentes? Templates!

Sugerencias y consejos adicionales

- Empezar con el testing a medida que van implementando, es decir **durante** la creación de la clase.

  Tengan en cuenta que algunos métodos pueden ser necesarios para muchas de las funciones de test, por lo tanto, es aconsejable empezar por esos test.
- Además, de los casos de test que puedan ser provistos por la cátedra les recomendamos fuertemente que realicen **siempre** sus propios tests.
- Dado que la implementación no debe perder memoria, es una buena práctica utilizar la herramienta *valgrind* **durante** el desarrollo.
- Si sus clases son *Template*, probar para distintas instancias de tipos (no sólo los tipos básicos).

Templates

- ¿Qué es un template?

- ¿Cómo lo usamos en C++?

Templates

- ¿Qué es un template?
  - Un template es un molde que sirve como base para producir muchas veces lo mismo.

- ¿Cómo lo usamos en C++?

Templates

- ¿Qué es un template?
  - Un template es un molde que sirve como base para producir muchas veces lo mismo.
  - En nuestro caso queremos muchos arreglos con la misma funcionalidad, pero con distintos contenidos.
- ¿Cómo lo usamos en C++?

Templates

- ¿Qué es un template?
  - Un template es un molde que sirve como base para producir muchas veces lo mismo.
  - En nuestro caso queremos muchos arreglos con la misma funcionalidad, pero con distintos contenidos.
- ¿Cómo lo usamos en C++?
  - Mediante la palabra clave `template` que nos permite parametrizar estructuras y funciones con distintos tipos.

Templates

- ¿Qué es un template?
  - Un template es un molde que sirve como base para producir muchas veces lo mismo.
  - En nuestro caso queremos muchos arreglos con la misma funcionalidad, pero con distintos contenidos.
- ¿Cómo lo usamos en C++?
  - Mediante la palabra clave `template` que nos permite parametrizar estructuras y funciones con distintos tipos.
  - De esta forma podemos hacer que la clase `ArregloDimensionable` tenga un parámetro que indique el tipo de su contenido.

Usando la palabra clave template

A la hora de declarar una clase:

```
// MyClass.h

template<typename T>
class MyClass {

  public:
    T myMethod(T parameter);

  private:
    T myVariable;

};
```

Usando la palabra clave template

A la hora de declarar una función:

```
// MyClass.cpp

#include "MyClass.h" // Ojo con esto!

template<typename T>
T MyClass<T>::myMethod(T parameter) {
  return parameter;
}
```

Usando la palabra clave template

```
// Main.cpp

#include <iostream>
#include "MyClass.h"

using namespace std;

int main(int argc, char *argv[])
{
  MyClass<char*> mcChars;
  cout << mcChars.myMethod("Hola mundo") << endl;

  MyClass<int> mcInt;
  cout << mcInt.myMethod(1) << endl;

  return 0;
}
```

Usando la palabra clave `template`

El *linker* nos tira el siguiente error al intentar compilar esto:

```
undefined reference to 'MyClass<char*>::myMethod(char*)'
undefined reference to 'MyClass<int>::myMethod(int)'
```

¿Cuál es el problema?

¿Cómo lo arreglamos?

Usando la palabra clave `template`

El *linker* nos tira el siguiente error al intentar compilar esto:

```
undefined reference to 'MyClass<char*>::myMethod(char*)'
undefined reference to 'MyClass<int>::myMethod(int)'
```

¿Cuál es el problema?

- Se compila la clase parametrizada una vez para cada parámetro. Para ello necesita toda la implementación (no sólo la definición de la clase que está en el .h), y al poner el código en el .cpp la implementación queda oculta.

¿Cómo lo arreglamos?

Usando la palabra clave `template`

El *linker* nos tira el siguiente error al intentar compilar esto:

```
undefined reference to 'MyClass<char*>::myMethod(char*)'
undefined reference to 'MyClass<int>::myMethod(int)'
```

¿Cuál es el problema?

- Se compila la clase parametrizada una vez para cada parámetro. Para ello necesita toda la implementación (no sólo la definición de la clase que está en el .h), y al poner el código en el .cpp la implementación queda oculta.

¿Cómo lo arreglamos?

- Escribimos toda la implementación en el .h a continuación de la declaración de la clase.

Usando la palabra clave template

```
// MyClass.h

template<typename T>
class MyClass {

  public:
    T myMethod(T parameter);

  private:
    T myVariable;

};

// Ponemos el codigo a continuacion

template<typename T>
T MyClass<T>::myMethod(T parameter) {
  return parameter;
}
```

¿Qué función cumple la palabra clave typename?

Significa que podemos dar como parámetro cualquier tipo, ya sean clases, *structs* o tipos primitivos.

Ejemplo:

```
MyClass<int> myInt;

struct Point {
  int x;
  int y;
};
MyClass<Point> myPoint;

class Human;
MyClass<Human> myHuman;
```

En lugar de typename se puede usar la palabra clave **class** al momento de declarar el template, son equivalentes.

¿Qué función cumple el keyword `typename`?

¿Por qué no lo agrega automáticamente el compilador?

Una razón es que se pueden usar otras cosas además de un tipo (pero esto no nos interesa en la materia).

¿Qué función cumple el keyword `typename`?

¿Por qué no lo agrega automáticamente el compilador?

Una razón es que se pueden usar otras cosas además de un tipo (pero esto no nos interesa en la materia).

Para que no se queden con la duda: el estándar permite usar `enums` e `ints`, pero los valores tienen que poder resolverse en tiempo de compilación. El ejemplo clásico es el de una estructura de datos de tamaño fijo, como un array, con templates podemos agregar un parámetro `int` que indique el tamaño (que se resuelve en tiempo de compilación).

ArregloDimensionable con templates

```
// ArregloDimensionable.h

// Pre: DATATYPE tiene constructor por defecto y operator=
template<typename DATATYPE>
class ArregloDimensionable {
  public:
    ArregloDimensionable();
    ~ArregloDimensionable();

    void insertarAtras(const DATATYPE& elem);
    int tamanio() const;
    const DATATYPE& iesimo(int i) const;
    DATATYPE& iesimo(int i);

  private:
    DATATYPE* _arreglo;
    int  _espacio;
    int  _ultimo;
};
```

ArregloDimensionable con templates

```
template<typename T>
ArregloDimensionable<T>::ArregloDimensionable(){
  _espacio = 1;
  _ultimo  = 0;
  _arreglo = new T[_espacio];
}

template<typename T>
int ArregloDimensionable<T>::tamanio() const {
  return _ultimo;
}

template<typename T>
const T& ArregloDimensionable<T>::iesimo(int i) const {
  return _arreglo[i];
}
```

ArregloDimensionable con templates

```
template<typename T>
void ArregloDimensionable<T>::insertarAtras(const T& elem) {

  if( _ultimo == _espacio ) {
    T* arregloViejo = _arreglo;
    _arreglo = new T[_espacio*2];        // usa T()

    for( int i = 0; i < _espacio; ++i )
      _arreglo[i] = arregloViejo[i];     // usa operator=

    _espacio *= 2;
    delete[] arregloViejo;
  }

  _arreglo[_ultimo] = elem;              // usa operator=
  _ultimo++;
}
```

ArregloDimensionable con templates

```
template<typename T>
void ArregloDimensionable<T>::insertarAtras(const T& elem) {

  if( _ultimo == _espacio ) {
    T* arregloViejo = _arreglo;
    _arreglo = new T[_espacio*2];        // usa T()

    for( int i = 0; i < _espacio; ++i )
      _arreglo[i] = arregloViejo[i];     // usa operator=

    _espacio *= 2;
    delete[] arregloViejo;
  }

  _arreglo[_ultimo] = elem;              // usa operator=
  _ultimo++;
}
```

La implementación del destructor queda como ejercicio. También habría que agregar un constructor por copia.

También podemos definir clases con más de un parámetro

```
template<typename Tizq, typename Tder>
class Par {

  public:
    Tizq izq;
    Tder der;

};
```

Y podemos pasar tipos que a su vez son paramétricos

```
ArregloDimensionable< Par<int, float> > arregloDePares;
```

Y podemos pasar tipos que a su vez son paramétricos

```
ArregloDimensionable< Par<int, float> > arregloDePares;
```

¿Y si queremos un `ArregloDimensionable` de punteros a pares?

Y podemos pasar tipos que a su vez son paramétricos

```
ArregloDimensionable< Par<int, float> > arregloDePares;
```

¿Y si queremos un `ArregloDimensionable` de punteros a pares?

```
ArregloDimensionable< Par<int, float>* > arregloDePunterosAPares;
```

Y podemos pasar tipos que a su vez son paramétricos

```
ArregloDimensionable< Par<int, float> > arregloDePares;
```

¿Y si queremos un `ArregloDimensionable` de punteros a pares?

```
ArregloDimensionable< Par<int, float>* > arregloDePunterosAPares;
```

En general si tenemos nombres largos conviene renombrar:

```
typedef Par<int, float>* ParPtr;
ArregloDimensionable< ParPtr > arregloDePunterosAPares;
```

Y podemos pasar tipos que a su vez son paramétricos

```
ArregloDimensionable< Par<int, float> > arregloDePares;
```

¿Y si queremos un `ArregloDimensionable` de punteros a pares?

```
ArregloDimensionable< Par<int, float>* > arregloDePunterosAPares;
```

En general si tenemos nombres largos conviene renombrar:

```
typedef Par<int, float>* ParPtr;
ArregloDimensionable< ParPtr > arregloDePunterosAPares;
```

Nota: El espacio entre el final del tipo interno y el externo es imprescindible, pues el compilador interpreta >> como un operador.

Algoritmos genéricos

Escribamos una función que devuelva el mejor elemento de un arreglo (donde, por el momento, el mejor es el mayor).

Algoritmos genéricos

Escribamos una función que devuelva el mejor elemento de un arreglo (donde, por el momento, el mejor es el mayor).

```
// Pre: ad.tamanio() > 0 y que el tipo T implmente operator>
template<typename T>
const T& mejor(const ArregloDimensionable<T>& ad) {
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (ad.iesimo(i) > ad.iesimo(mejor)) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```

Algoritmos genéricos

Escribamos una función que devuelva el mejor elemento de un arreglo (donde, por el momento, el mejor es el mayor).

```
// Pre: ad.tamanio() > 0 y que el tipo T implmente operator>
template<typename T>
const T& mejor(const ArregloDimensionable<T>& ad) {
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (ad.iesimo(i) > ad.iesimo(mejor)) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```

¿Y si quiero usar un criterio específico para cada tipo T?

Algoritmos genéricos

Escribamos una función que devuelva el mejor elemento de un arreglo (donde, por el momento, el mejor es el mayor).

```
// Pre: ad.tamanio() > 0 y que el tipo T implmente operator>
template<typename T>
const T& mejor(const ArregloDimensionable<T>& ad) {
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (ad.iesimo(i) > ad.iesimo(mejor)) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```

¿Y si quiero usar un criterio específico para cada tipo T?
¿Y si ahora quiero que el mejor sea el menor?

Algoritmos genéricos

Agreguemos un criterio a los pares como una clase interna.

```
template<typename Tizq, typename Tder>
class Par {

  public:
    Tizq izq;
    Tder der;

    class Criterio {
      public:
        bool esMejor(Par<Tizq, Tder>& a, Par<Tizq, Tder>& b);
    };
};

template<typename Tizq, typename Tder>
bool Par<Tizq, Tder>::Criterio::esMejor(
        Par<Tizq, Tder>& a, Par<Tizq, Tder>& b)
{
    return a.izq > b.izq;
}
```

Algoritmos genéricos

```
// Pre: ad.tamanio() > 0 y T tiene una clase interna Criterio
//      Criterio tiene un metodo 'bool esMejor(a, b)'
template<typename T>
const T& mejor(const ArregloDimensionable<T>& ad) {
  T::Criterio criterio;
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (criterio.esMejor(ad.iesimo(i), ad.iesimo(mejor))) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```

Algoritmos genéricos

```
// Pre: ad.tamanio() > 0 y T tiene una clase interna Criterio
//      Criterio tiene un metodo 'bool esMejor(a, b)'
template<typename T>
const T& mejor(const ArregloDimensionable<T>& ad) {
  T::Criterio criterio;
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (criterio.esMejor(ad.iesimo(i), ad.iesimo(mejor))) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```

Ooops, error al compilar:

```
Mejor.h: In function 'T mejor(ArregloDimensionable<DATATYPE>&)':
Mejor.h:5: error: expected ';' before "criterio"
Mejor.h:10: error: 'criterio' undeclared (first use this function)
Mejor.h: In function 'T mejor(ArregloDimensionable<DATATYPE>&)
Mejor.h:5: error: dependent-name 'T::Criterio' is parsed as a non-type,
          but instantiation yields a type
Mejor.h:5: note: say 'typename  T::Criterio' if a type is meant)
```

Algoritmos genéricos

```
// Pre: ad.tamanio() > 0 y T tiene una clase interna Criterio
//      Criterio tiene un metodo 'bool esMejor(a, b)'
template<typename T>
const T& mejor(const ArregloDimensionable<T>& ad) {
  typename T::Criterio criterio;
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (criterio.esMejor(ad.iesimo(i), ad.iesimo(mejor))) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```

El problema es que al usar la clase interna de T (T::Criterio) el compilador interpreta que se trata de un *valor* (y queremos que lo interprete como un *tipo*).

La solución es usar el keyword **typename**.

Algoritmos genéricos

¿Qué valor termina en la variable `elMejorPar`?

```
int main(){

    Par<int, float> p1(1, 2.0);
    Par<int, float> p2(2, 2.0);
    Par<int, float> p3(3, 2.0);
    ArregloDimensionable< Par<int, float> > arregloDePares;
    arregloDePares.insertarAtras(p1);
    arregloDePares.insertarAtras(p2);
    arregloDePares.insertarAtras(p3);

    Par<int, float> elMejorPar = mejor(arregloDePares);

    return 0;
}
```

Algoritmos genéricos

¿Y si en vez de un único criterio por tipo queremos poder elegir?

```
// Pre: ad.tamanio() > 0 y Comparador tiene un mtodo
//      esMejor : T x T -> bool
template<typename T, typename Comparador>
const T& mejor(
     const ArregloDimensionable<T>& ad, Comparador comparador)
{
  int mejor = 0;
  for (int i = 1; i < ad.tamanio(); ++i) {
    if (comparador.esMejor(ad.iesimo(i), ad.iesimo(mejor))) {
      mejor = i;
    }
  }
  return ad.iesimo(mejor);
}
```