Templates en C++

Algoritmos y estructuras de datos II

8 de abril de 2016

Hasta ahora vimos:

- Tipos primitivos.
- Estructuras de control (`if`, `while`, `for`).
- Cómo declarar clases.
- Cómo construirlas y destruirlas.
- Cómo reservar memoria dinámica.

Ejemplo: arreglo dimensionable

```
class ArregloDimensionable {
  public:
    ArregloDimensionable();
    ~ArregloDimensionable();

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

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

Esto sólo sirve si necesitamos arreglos de `ints`.

¿Qué hacemos si necesitamos arreglos de `floats`?

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

```cpp
// 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?

- 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).

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++;
}
```

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 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).

```
// 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);
}
```

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);
}
```

Uso avanzado de templates

En algunas ocasiones es necesario tratar un tipo de forma especial.
Para eso tenemos en C++ "*template specialization*".
Por ejemplo la función mejor (que utiliza el criterio interno) normalmente no podría ser usada con `int` pues no podemos definir una clase `Criterio` para los `int`s.

```
// Pre: ad.tamanio() > 0
template<>
const int& mejor(const ArregloDimensionable<int>& 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);
}
```

Nota No se espera que tengan que usar esto en el TP.

Uso avanzado de templates

Esto es utilizado para programar al estilo funcional en tiempo de compilación con "*template metaprogramming*".

```
struct Zero {
  static const int val = 0;
};

template<typename Nat>
struct Suc {
  static const int val = 1 + Nat::val;
};

template<typename Nat> struct Fact {};

template<>
struct Fact<Zero> {
  static const int val = 1;
};

template<typename Nat>
struct Fact< Suc<Nat> > {
  static const int val = Suc<Nat>::val * Fact<Nat>::val;
};
```

Uso avanzado de templates

Por completitud veamos un main que use esto.

```
int main(int argc, char *argv[]) {

  int three = Suc< Suc< Suc< Zero > > >::val;

  std::cout << three << std::endl;

  int fact = Fact< Suc< Suc< Suc< Zero > > > >::val;

  std::cout << fact << std::endl;

  return 0;
}
```

Nota Definitivamente **NO** tienen que usar esto en el TP.

Manos a la obra

Traten de implementar una lista genérica usando templates.

Implemente una lista simplemente enlazada que permita consultar su tamaño, agregar un elemento al principio y remover el primer elemento.