Introducción al Cómputo

Templates

Reutilización de código

- Mejorar el copy/paste/change.
- Creación de nuevas clases en base a las ya existentes, sin necesidad de modificar su código.
  - Composición: crear nuevas clases compuestas por objetos de clases ya existente. Reuso de implementación.
  - Herencia: crear nuevas clases del tipo de una clase ya existente, sin modificarla. Reuso de interface.
- Programación parametrizada por tipos. Templates. Generic programming. Reuso de conceptos y algoritmos.

Vector

```
class Vector
{
public:
    Vector(int s);
    ~Vector();

    Vector(const Vector& v);

    Vector& operator=(const Vector& v);

    double& operator[](int idx);
    double  operator[](int idx) const;

    int size();

private:
    int sz;
    double *elem;
};
```

- Ahora necesito un vector de **Punto2D**
  - ¿Uso composición?
  - ¿Uso herencia?
  - ❑ No funcionan...
- Ahora necesito un Vector de **string**
  - ¿Hago (copy&paste) un **StrVector**?

✓ La solución es hacer una implementación de **Vector** parametrizada con el tipo de sus elementos.

✓ Template **Vector**. Plantilla.

Templates

- Proveen soporte para programación genérica.
- Utilizan tipos de datos como parámetros.

- Permite reutilización de código a nivel de código fuente.
- Método sencillo de representar conceptos generales como algoritmos, contenedores, iteradores, etc. y de combinarlos.
- Eficiencia en velocidad y tamaño.

- La abstracciones de la biblioteca estándar están basadas en templates.

Template **Vector**

```
template <class EltT>
class Vector
{
public:
    Vector(int s);
    ~Vector();

    Vector(const Vector &v);

    Vector& operator=(const Vector& v);

    EltT& operator[](int idx);
    EltT  operator[](int idx) const;

    int size();

private:
    int   sz;
    EltT* elem;
};
```

```
void f()
{
    Vector<double> dvec(5);
    dvec[0] = 3.14;
    double d = dvec[0];

    Vector<string> strvec(10);
    strvec[0] = "hola";
    cout << strvec[0] << endl;

    Vector<Punto2D> p2dvec(3);
    p2dvec[0] = Punto2D{ 1.1, 0.2 };
    Punto2D p2d = p2dvec[0];
    cout << p2d.x << endl;
}

template<class EltT>
void printVector(Vector<EltT>& v) {
    for( int i = 0; i < v.size(); ++i )
        cout << v[i] << ", ";
}
```

Templates: String

- Conceptos básicos de string:
  - Contenedor de caracteres.
  - Operación de comparación.
  - Operación de concatenación.
  - Operación de subscripting.
- Distintos tipos posibles de caracteres.
  - 8 bits, 16 bits, 32 bits.
- Representación con mínima dependencia del tipo específico de carácter.

Templates: String

```
template <class C> class String {
        struct Srep;
        Srep *rep;
    public:
        String();
        String(const C*);
        String(const String &s);
        C read(int i) const;
        C &operator[](int n);
        String &operator+=(C c);
        // ...
};
```

```
void f() {
    String<char> cs;
    String<unsigned char> us;
    String<wchar_t> ws;

    struct Jchar { /* */ };

    String<Jchar> js;
}
```

- En la standard library

```
using string = basic_string<char>;
```

Templates: String

```
template <class C> struct String<C>::Srep {
    C *s;
    int sz;
    int n;
    // ...
};

template<class C> C String<C>::read(int i) const {
    return rep->s[i];
}

template<class C> String<C>::String() {
    rep  = new Srep(0, C());
}
```

Templates: Instanciación

- El proceso de generar una clase o una función a partir de una definición template y sus argumentos se llama *template instantiation*.
- La versión de un template para un argumento en particular se llama *specialization*.
- Es responsabilidad de la implementación, no del programador, asegurar que se instancien todas las versiones de funciones y clases necesarias.

```
String<char> cs;
void f() {
    String<Jchar> js;
    cs = "que código genera?";
}
```

Templates: Parámetros

- Un template puede tener como parámetros:
  - tipos
  - parámetros de tipos nativos
  - parámetros templates

```
template<class T, T def_val> class Cont { /* ... */ };
template<class T, int N> class Buffer {
        T v[N];
        const int sz;
    public:
        Buffer() : sz(N) { }
};
Buffer<char, 127> cbuf;   // el entero debe ser constante
Buffer<Record, 8> rbuf;
```

Template function

- Teniendo tipos templates, aparece la necesidad de funciones template:

```
template<class T> void sort(vector<T> &);
void f(vector<int> &vi, vector<string> &vs) {
    sort(vi);     // sort(vector<int> &);
    sort(vs);     // sort(vector<string> &);
};
```

- Que función template: se determina de los argumentos.

```
template<class T> void sort(vector<T> &v) {  // Shell sort, Knuth
    const size_t n = v.size();
    for( int gap = n/2; 0 < gap; gap /= 2 )
        for( int i = gap; i < n; i++ )
            for( int j = i-gap; 0 <= j; j -= gap )
                if( v[j+gap] < v[j] )
                    swap(v[j], v[j+gap]);    // std::swap()
}
```

Template function

- Cambiando la política de comparación:

```
template<class T, typename Compare=less<T>> void sort(vector<T> &v) {
    Compare cmp;
    const size_t n = v.size();
    for( int gap = n/2; 0 < gap; gap /= 2 )
        for( int i = gap; i < n; i++ )
            for( int j = i-gap; 0 <= j; j -= gap )
                if( cmp(v[j+gap], v[j]) )
                    swap(v[j], v[j+gap]);    // std::swap()
}
```

Template function

- Las funciones template son esenciales para escribir algoritmos genéricos sobre la variedad de contenedores.
- La habilidad para deducir los argumentos es crucial.

```
template<class T, int i> T lookup(Buffer<T,i> &b, const char *p);
class Record {
    const char r[12];
};
Record f(Buffer<Record, 128> &buf, const char *p) { return lookup(buf, p);}
```

  - Se deducen **`T`** como **`Record`** e **`i`** como 128.

- Los argumentos de clases template nunca se deducen.
- La especialización provee un mecanismo para elegir implícitamente entre diferentes implementaciones.

Template function

- Si los argumentos templates de las funciones no pueden ser deducidos, se deben especificar explícitamente.

```
template<typename T>
T *create();

void f()
{
    vector<int> v;              // class, template argument int
    int *p = create<int>();     // function, template argument int
    int *q = create();          // error: can’t deduce template argument
}
```

Template function

- Un caso común es la especificación explícita del tipo de retorno de una función template.

```
template<typename To, typename From> To convert(From f);

void g(double d)
{
    int i = convert<int>(d);    // calls convert<int, double>(double)
    char c = convert<char>(d);  // calls convert<char, double>(double)
    int(*ptr)(float) = convert; // instantiates convert<int, float>(float)
}
```

Template function overloading

- Es posible declarar funciones template con el mismo nombre que funciones no template.

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);
void f(complex<double> z) {
    sqrt(2);                    // sqrt<int>(int);
    sqrt(2.0);                  // sqrt(double);
    sqrt(z);                    // sqrt<double>(complex<double>);
}
```

- Varias reglitas para resolución de overloading para funciones template y no template. Básicamente se busca la mejor especialización para el conjunto de argumentos template.

Templates: Especializacón

- Por default, un template da una única definición para ser utilizada con todos los distintos argumentos del template.
- Se puede querer especializar el comportamiento para ciertos casos:
  - El argumento es un puntero
  - Dar error salvo que el argumento sea derivado de una determinada clase.
- Se pueden proveer definiciones alternativas para la definición del template, dejando que el compilador utilice la más adecuada.
- Estas alternativas se llaman *user specializations*.

Especializacón: ejemplo

```
template<class T> class Vector {
        T *v;
        int sz;
    public:
        Vector();
        Vector(int);
        T &operator[](int i);
        void swap(Vector &);
        // ...
};
```

```
Vector<int> vi;

Vector<Shape *> vps;

Vector<string> vs;

Vector<char *> vpc;

Vector<Node *> vpn;
```

- Se replica el código para cada vector de tipo de puntero.

Especializacón: ejemplo

```
template<> class Vector<void *> {
    void **p;
    // ...
    void *& operator[](int i);
};
```

Especialización total: no quedan template arguments

```
template<class T> class Vector<T *>
    : private Vector<void *> {
    public:
        using Base = Vector<void *>;
        Vector() : Base() { }
        explicit Vector(int i) : Base(i) { }
        T* &operator[](int i) {
            return static_cast<T* &>(Base::operator[](i)); }
        // ...
};
```

Implementación para todos los **`Vector`**s de punteros

Orden de especializacón

```
template<class T> class Vector<T>;        // general
template<class T> class Vector<T *>;      // para cualquier tipo de puntero
template<> class Vector<void *>           // para void *
```

- La versión más especializada es preferida sobre las otras.

- Especialización de funciones template:

```
template<class T>
void swap(T &x, T&y)
{
    T t = x;
    x = y;
    y = t;
}
```

```
template<class T>
void swap(Vector<T> &a, Vector<T> &b)
{
   a.swap(b);
}
```

Template y herencia

- Derivación de una clase template de una no template:
  para dar una implementación común.

```
template<class T> class list<T *> : private list<void *> { /* ... */ };
```

- Derivación de una clase de una base template:
  para cambio de comportamiento de métodos.

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

- Se puede pasar la clase derivada como clase base

```
template<class C> class Basic_ops {    // basic operations for containers
    bool operator==(Const C&) const;
    bool operator!=(Const C&) const;
};
template<class T> class Math_container : public Basic_ops<Math_container<T>> {
// ...
};
```

Member templates

- Las clases o clases template pueden tener miembros que son a su vez templates.

```
template<class Scalar> class complex {
        Scalar re, im;
    public:
        template<class T>
        complex(const complex<T> &c) : re(c.re), im(c.im) { }
        // ...
};
complex<float> cf(0,0);
complex<double> cd = cf;      // ok, usa conversión de float a double
```

- Se puede construir un **`complex<T1>`** con un **`complex<T2>`** si y sólo si se puede construir un **`T1`** a partir de un **`T2`**.
- Los miembros template no pueden ser virtuales.

Templates y jerarquías

- Los templates definen una interface. Las implementaciones de especializaciones pueden ser muy distintas y agregar funcionalidad. El código fuente es el mismo para todos los tipos parametrizados.
  - Polimorfismo paramétrico o en compile-time.

- Las clases base definen una interface. La implementación de la clase base y sus derivadas pueden ser accedidas por funciones virtuales con implementaciones distintas. Las clases derivadas pueden agregar funcionalidad.
  - Polimorfismo en run-time.

Templates y jerarquías

- Programación Orientada a Objetos hace foco en el diseño de jerarquías de clases.
- Programación Genérica hace foco en el diseño de algoritmos y argumentos templates para manejar distintos tipos.
- Cada técnica debe ser utilizada cuando sea más apropiada. En general un diseño óptimo contiene elementos de ambas.

```
std::vector<Shape *> v;
```