

```
template<class T = ResaCON>
class Presentation
{
    static const char *author = "Pplux" ;
};
```


templates?

- ✦ Ventajas:

- ✦ blah

- ✦ blah

- ✦ Problemas:

- ✦ blah blah

- ✦ slack no lleva razón blah blah

Especialización

```
struct NullType  
{  
};
```

```
template<int i, class T1, class T2, class T3>  
struct select  
{  
    typedef NullType Result;  
};
```


Especialización

```
struct NullType  
{  
};
```

```
template<int i, class T1, class T2, class T3>  
struct select  
{  
    typedef NullType Result;  
};
```


Especialización

```
struct NullType  
{  
};
```

```
template<int i class T1, class T2, class T3>  
struct select  
{  
    typedef NullType Result;  
};
```


Especialización (...Parcial !!!)

```
template<class T1, class T2, class T3>
struct select<0,T1,T2,T3>
{
    typedef T1 Result;
};
```

```
template<class T1, class T2, class T3>
struct select<1,T1,T2,T3>
{
    typedef T2 Result;
};
```

```
template<class T1, class T2, class T3>
struct select<2,T1,T2,T3>
{
    typedef T3 Result;
};
```


Especialización (...Parcial !!!)

```
template<class T1, class T2, class T3>
```

```
struct select<0,T1,T2,T3>
```

```
{
```

```
    typedef T1 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<1,T1,T2,T3>
```

```
{
```

```
    typedef T2 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<2,T1,T2,T3>
```

```
{
```

```
    typedef T3 Result;
```

```
};
```



Especialización (...Parcial !!!)

```
template<class T1, class T2, class T3>
```

```
struct select<0,T1,T2,T3>
```

```
{
```

```
    typedef T1 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<1,T1,T2,T3>
```

```
{
```

```
    typedef T2 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<2,T1,T2,T3>
```

```
{
```

```
    typedef T3 Result;
```

```
};
```



Especialización (...Parcial !!!)

```
template<class T1, class T2, class T3>
```

```
struct select<0,T1,T2,T3>
```

```
{
```

```
    typedef T1 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<1,T1,T2,T3>
```

```
{
```

```
    typedef T2 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<2,T1,T2,T3>
```

```
{
```

```
    typedef T3 Result;
```

```
};
```



Especialización (...Parcial !!!)

```
template<class T1, class T2, class T3>
```

```
struct select<0,T1,T2,T3>
```

```
{
```

```
    typedef T1 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<1,T1,T2,T3>
```

```
{
```

```
    typedef T2 Result;
```

```
};
```

```
template<class T1, class T2, class T3>
```

```
struct select<2,T1,T2,T3>
```

```
{
```

```
    typedef T3 Result;
```

```
};
```



Especialización (...¿?...)

```
int main()  
{  
    select<2, float, float, int>::Result obj = 2;  
}
```


Especialización (cálculo)

```
template<unsigned int D>
struct fib
{
    enum { value = fib<D-1>::value + fib<D-2>::value };
};

template<>
struct fib<0>
{
    enum { value = 0 };
};

template<>
struct fib<1>
{
    enum { value = 1 };
};
```


Especialización (cálculo)

```
template<unsigned int D>
struct fib
{
    enum { value = fib<D-1>::value + fib<D-2>::value };
};

template<>
struct fib<0>
{
    enum { value = 0 };
};

template<>
struct fib<1>
{
    enum { value = 1 };
};
```


Especialización (cálculo)

```
template<unsigned int D>
struct fib
{
    enum { value = fib<D-1>::value + fib<D-2>::value };
};
```

```
template<>
struct fib<0>
{
    enum { value = 0 };
};
```

```
template<>
struct fib<1>
{
    enum { value = 1 };
};
```


Especialización (cálculo)

```
template<unsigned int D>
struct fib
{
    enum { value = fib<D-1>::value + fib<D-2>::value };
};
```

```
template<>
struct fib<0>
{
    enum { value = 0 };
};
```

```
template<>
struct fib<1>
{
    enum { value = 1 };
};
```


Especialización (cálculo)

```
int main()  
{  
    std::cout << fib<8>::value << std::endl;  
  
    char a[fib<15>::value];  
    return 0;  
}
```


Especialización (cálculo)

```
int main()  
{  
    std::cout << fib<8>::value << std::endl;
```

Constante
(tiempo de Compilación)

```
char a[fib<15>::value];  
return 0;  
}
```


Especialización (cálculo)

```
int main()  
{  
    std::cout << fib<8>::value << std::endl;
```

Constante
(tiempo de Compilación)



```
char a[fib<15>::value];  
return 0;  
}
```


Más usos de las constantes

```
template<class T, unsigned int D>
struct vec
{
    T& operator[] (unsigned int p)
    {
        return data[p];
    }

    T data[D];
};

int main()
{
    vec<float, 2> v2;
    vec<float, 3> v3;

    return 0;
}
```


Más usos de las constantes

```
template<class T, unsigned int D>
struct vec
{
    ...
    vec<T,D-1> reduce()
    {
        vec<T,D-1> r;
        for(unsigned int i = 0; i < D-1; ++i)
        {
            r[i] = data[i]/data[D-1];
        }
        return r;
    }
    ...
};
```


Más usos de las constantes

```
template<class T, unsigned int D>
struct vec
{
    ...
    vec<T,D-1> reduce()
    {
        vec<T,D-1> r;
        for(unsigned int i = 0; i < D-1; ++i)
        {
            r[i] = data[i]/data[D-1];
        }
        return r;
    }
    ...
};
```


Más usos de las constantes

```
template<class T, unsigned int D>
struct vec
{
    ...
    vec() {}
    vec(T a) { data[0] = a; }
    vec(T a, T b) { data[0] = a; data[1] = b; }
    vec(T a, T b, T c) { data[0] = a; data[1] = b; data[2] = c; }
    ...
};
```


Más usos de las constantes

```
template<class T, unsigned int D>
struct vec
{
    ...
    vec() {}
    vec(T a) { data[0] = a; }
    vec(T a, T b) { data[0] = a; data[1] = b; }
    vec(T a, T b, T c) { data[0] = a; data[1] = b; data[2] = c; }
    ...
};
```



Más usos de las constantes

```
template<int n>
struct check_vector_size
{
    check_vector_size() {}
    char b[n-1];
};
template<class T, unsigned int D>
struct vec
{
    vec<T,D-1> reduce()
    {
        check_vector_size<D-1>();
        vec<T,D-1> r;
        ....
    }
    vec() {}
    vec(T a) { check_vector_size<D>(); data[0] = a; }
    vec(T a, T b) { check_vector_size<D-1>(); data[0] = a; data[1] = b; }
    vec(T a, T b, T c) { check_vector_size<D-2>(); data[0] = a; data[1] = b; data[2] = c; }
    ...
};
```


Más usos de las constantes

```
template<int n>
struct check_vector_size
{
    check_vector_size() {}
    char b[n-1];
};
template<class T, unsigned int D>
struct vec
{
    vec<T,D-1> reduce()
    {
        check_vector_size<D-1>();
        vec<T,D-1> r;
        ....
    }
    vec() {}
    vec(T a) { check_vector_size<D>(); data[0] = a; }
    vec(T a, T b) { check_vector_size<D-1>(); data[0] = a; data[1] = b; }
    vec(T a, T b, T c) { check_vector_size<D-2>(); data[0] = a; data[1] = b; data[2] = c; }
    ...
};
```


Más usos de las constantes

```
template<int n>
struct check_vector_size
{
    check_vector_size() {}
    char b[n-1];
};
template<class T, unsigned int D>
struct vec
{
    vec<T,D-1> reduce()
    {
        check_vector_size<D-1>();
        vec<T,D-1> r;
        ....
    }
    vec() {}
    vec(T a) { check_vector_size<D>(); data[0] = a; }
    vec(T a, T b) { check_vector_size<D-1>(); data[0] = a; data[1] = b; }
    vec(T a, T b, T c) { check_vector_size<D-2>(); data[0] = a; data[1] = b; data[2] = c; }
    ...
};
```

Inválido si $n == 0$

Más usos de las constantes

```
template<int n>
struct check_vector_size
{
    check_vector_size() {}
    char b[n-1];
};

template<class T, unsigned int D>
struct vec
{
    vec<T,D-1> reduce()
    {
        check_vector_size<D-1>();
        vec<T,D-1> r;
        ....
    }
    vec() {}
    vec(T a) { check_vector_size<D>(); data[0] = a; }
    vec(T a, T b) { check_vector_size<D-1>(); data[0] = a; data[1] = b; }
    vec(T a, T b, T c) { check_vector_size<D-2>(); data[0] = a; data[1] = b; data[2] = c; }
    ...
};
```

Inválido si $n == 0$

Check en tiempo de Compilación

The diagram illustrates the use of constant expressions for compile-time checks. A blue box labeled 'Check en tiempo de Compilación' has three arrows pointing to the `check_vector_size` calls in the `vec` struct: `check_vector_size<D-1>()` in the `reduce` method, `check_vector_size<D>()` in the `vec(T a)` constructor, and `check_vector_size<D-1>()` in the `vec(T a, T b)` constructor. Another blue box points to the `char b[n-1];` line in the `check_vector_size` struct, with the text 'Inválido si n == 0' indicating that this is invalid when `n` is zero.

sizeof

sizeof es realmente potente, se puede aplicar a cualquier expresión, sin importar lo compleja que esta sea, sizeof devolverá el tamaño de esta sin necesidad de llegar a evaluar la expresión en tiempo de ejecución.

Esto significa que sizeof tiene presente la sobrecarga, la especialización de plantillas, reglas de conversión ... **absolutamente todo** lo que pueda aparecer en una expresión de C++ .

En realidad, sizeof es capaz de deducir el tipo de una expresión; desechando la expresión pero devolviendo su tamaño.

sizeof

```
template <class A, class B>
class Convert
{
    typedef char S_True;
    class S_False { char dummy[2]; };
    static A CreateInstance();
    static S_True Test(B);
    static S_False Test(...);

public:
    enum {
        exists = ( sizeof(Test(CreateInstance())) == sizeof
(S_True) )
    };
};
```


sizeof

```
template <class A, class B>
class Convert
{
    typedef char S_True;
    class S_False { char dummy[2]; };
    static A CreateInstance();
    static S_True Test(B);
    static S_False Test(...);

public:
    enum {
        exists = ( sizeof(Test(CreateInstance())) == sizeof
(S_True) )
    };
};
```


sizeof

```
template <class A, class B>
class Convert
{
    typedef char S_True;
    class S_False { char dummy[2]; };
    static A CreateInstance();
    static S_True Test(B);
    static S_False Test(...);

public:
    enum {
        exists = ( sizeof(Test(CreateInstance())) == sizeof
(S_True) )
    };
};
```



sizeof

```
int main()  
{  
    std::cout << Convert<Liso,People>::exists << std::endl;  
    std::cout << Convert<float,int>::exists << std::endl;  
}
```


sizeof

```
int main()  
{  
    std::cout << Convert<Liso,People>::exists << std::endl;  
  
    std::cout << Convert<float,int>::exists << std::endl;  
}
```

```
static S_True Test(B);  
static S_False Test(...);
```


sizeof

```
int main()  
{  
    std::cout << Convert<Liso,People>::exists << std::endl;  
    std::cout << Convert<float,int>::exists << std::endl;  
}
```



```
static S_True Test(B);  
static S_False Test(...);
```


Ejemplo: Signals

```
class Sender
{
public:
    signal<float, float> sendFloats;
    signal<int> sendInt;

    void fire()
    {
        sendFloats(2.0, 3.0);
        sendInt(5);
    }
};
```

```
class Receiver
{
public:
    void onSignal(float a, float b)
    {...}
    void onSignal(int i)
    {...}
};
```

```
int main()
{
    Sender s;
    Receiver r;
    connect(&s, &Sender::sendFloats, &r, &Receiver::onSignal);
    s.fire();
    return 0;
}
```


Ejemplo: Signals

```
template<class T1 = NullType , class T2 = NullType, class T3 = NullType>
class signal : private BaseSignal<T1,T2,T3>
{
    ...
};
```

```
template<class T1, class T2>
class signal<T1,T2, NullType> : private BaseSignal<T1,T2>
{
public:
    void operator()(T1 a,T2 b) { BaseSignal<T1,T2>::operator()(a,b); }

    template<class C>
    void connect(C *a, typename slot<C,T1,T2>::Prototipe a_m)
    {
        BaseSignal<T1,T2>::_slot = new slot<C,T1,T2>(a,a_m);
    }
};
```


Ejemplo: Signals

```
template<class T1 = NullType , class T2 = NullType, class T3 = NullType>
class signal : private BaseSignal<T1,T2,T3>
{
    ...
};
```

```
template<class T1, class T2>
class signal<T1,T2, NullType> : private BaseSignal<T1,T2>
{
public:
    void operator()(T1 a,T2 b) { BaseSignal<T1,T2>::operator()(a,b); }

    template<class C>
    void connect(C *a, typename slot<C,T1,T2>::Prototipe a_m)
    {
        BaseSignal<T1,T2>::_slot = new slot<C,T1,T2>(a,a_m);
    }
};
```


Ejemplo: Signals

```
template<class C, class T1 = NullType, class T2 = NullType, class T3 = NullType>
struct slot : public BaseSlot<T1,T2,T3>
{
    typedef void (C::*Prototype)(T1,T2,T3);
    ...
};
```

```
template<class C, class T1, class T2>
struct slot<C,T1,T2,NullType> : public BaseSlot<T1,T2>
{
    typedef void (C::*Prototype)(T1,T2);

    slot(C *obj, Prototype m) : _obj(obj), _m(m) {}
    void operator()(T1 a,T2 b) { (_obj->*_m)(a,b); }
private:
    C *_obj;
    Prototype _m;
};
```


Ejemplo: Signals

```
template<class T1, class T2, class S, class R>
void connect_bad(
    S *sender,
    Signal<T1,T2> S::*s_sig,
    R *receiver,
    void (R::*r_Slot)(T1,T2) ) {...}
```


Ejemplo: Signals

```
template<class T1, class T2, class S, class R>
void connect_bad(
    S *sender,
    Signal<T1,T2> S::*s_sig,
    R *receiver,
    void (R::*r_Slot)(T1,T2) ) {...}
```

No generaliza en
número

Ejemplo: Signals

```
template<class T1, class T2, class T3, class S, class R>
void connect(
    S *sender,
    Signal<T1,T2,T3> S::*s_sig,
    R *receiver, typename Slot<R, T1,T2,T3>::Prototype r_Slot)
{
    (sender->*s_sig).connect(receiver, r_Slot);
}
```


Ejemplo: Signals

```
template<class T1, class T2, class T3, class S, class R>
void connect(
    S *sender,
    Signal<T1,T2,T3> S::*s_sig,
    R *receiver, typename Slot<R, T1,T2,T3>::Prototipe r_Slot)
{
    (sender->*s_sig).connect(receiver, r_Slot);
}
```


Ejemplo: Signals

```
template<class T1, class T2, class T3, class S, class R>
void connect(
    S *sender,
    Signal<T1,T2,T3> S::*s_sig,
    R *receiver, typename Slot<R, T1,T2,T3>::Prototipe r_Slot)
{
    (sender->*s_sig).connect(receiver, r_Slot);
}
```



```
template<class C, class T1, class T2>
struct slot<C,T1,T2,NullType> : public BaseSlot<T1,T2>
{
    typedef void (C::*Prototipe)(T1,T2);
};
```


Ejemplo: Signals

```
template<class T1, class T2, class T3, class S, class R>
void connect(
    S *sender,
    Signal<T1,T2,T3> S::*s_sig,
    R *receiver, typename Slot<R, T1,T2,T3>::Prototipe r_slot)
{
    (sender->*s_sig).connect(receiver, r_slot);
}
```

```
template<class C, class T1, class T2>
struct slot<C,T1,T2,NullType> : public BaseSlot<T1,T2>
{
    typedef void (C::*Prototipe)(T1,T2);
};
```



```
connect(&s, &Sender::sendFloats, &r, &Receiver::onSignal);
```


Otros Ejemplos...

```
SLB::Class<Animal, SLB::Instance::NoCopy >("Unit_004::Animal")
    .set("makeSound", &Animal::makeSound)
;
SLB::Class<Duck>("Unit_004::Duck")
    .constructor<bool>()
    .set("canFly", &Duck::canFly)
    .inherits<Animal>()
;
SLB::Class<Dog>("Unit_004::Dog")
    .constructor<bool>()
    .set("bites", &Dog::bites)
    .inherits<Animal>()
;
```

```
SLB::Class< SFH, SLB::Instance::NoCopy >("Unit_005::SFH")
    .inherits<SLB::Script>()
    .constructor()
    .set("doString", &SFH::doString)
    .set("calc", &SFH::calc)
    .hybrid()
;
```


Otros Ejemplos...

```
template<class V, bool N>
inline VertexAttribArray<V,N>::VertexAttribArray()
: ArrayBase( V::Size, C_to_GL<typename V::Type>::value,N )
{}
```


Otros Ejemplos...

```
template<class V, bool N>
inline VertexAttribArray<V,N>::VertexAttribArray()
: ArrayBase( V::Size, C_to_GL<typename V::Type>::value,N )
{}
```



```
template<class T>
struct C_to_GL
{
    //Default...
    enum { value = -1 };
};

template<>
struct C_to_GL<float>
{
    enum { value = GL_FLOAT };
};

template<>
struct C_to_GL<double>
{
    enum { value = GL_DOUBLE};
};
```


Otros Ejemplos...

```
template<class V, bool N>
inline VertexAttribArray<V,N>::VertexAttribArray()
: ArrayBase( V::Size, C_to_GL<typename V::Type>::value,N )
{}
```

```
template<class T, size_t D>
class Vec: public Vec<T, D-1>
{
public:
    enum { Size = D };
    typedef T Type;
    ...
}
```

```
template<class T>
struct C_to_GL
{
    //Default...
    enum { value = -1 };
};

template<>
struct C_to_GL<float>
{
    enum { value = GL_FLOAT };
};

template<>
struct C_to_GL<double>
{
    enum { value = GL_DOUBLE };
};
```



```
template<class T = ResaCON>
class Preguntas
{
    /* TODO */ /* HACK */ /*WTF!*/
};
```