# Introduction to C++ Templates

Lecture 9

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

# Let's Start with a Problem...

## Sorting an array of **integer**

**insertion_sort.cpp**

```
void InsertionSort( int *v, int n ) {
    int i, j, x;

    for ( j = 1; j < n; ++j ) {
        x = v[j];
        for ( i = j-1; i >= 0 && v[i] > x; --i )
            v[i+1] = v[i];
        v[i+1] = x;
    }
}
...
```

**Sorting an array of integer**

```
...
int main(void) {
    int vector[5] = { 2, 1, 3, 0, 4 };
    InsertionSort( vector, 5 );

    for( int i = 0; i < 5; i++ )
        std::cout << "[" << i << "] :" << vector[i] << std::endl;

    return 0;
}
```

What if we want to sort an array of **double**?

# Let's Start with a Problem...

## Sorting an array of **double**

**insertion_sort.cpp**

```
void InsertionSort( int *v, int n ) {
    int i, j, x;

    for ( j = 1; j < n; ++j ) {
        x = v[j];
        for ( i = j-1; i >= 0 && v[i] > x; --i )
            v[i+1] = v[i];
        v[i+1] = x;
    }
}
...
```

In principle, we **do not** have to **modify** the **algorithm**...

**Sorting an array of double**

```
...
int main( void ) {
    double vector[5] = { 1.3, 1.1, 1.8, 1.6, 1.9 };
    InsertionSort( ( int* ) vector, 5 );

    for( int i = 0; i < 5; i++ )
        std::cout << "[" << i << "] :" << vector[i] << std::endl;

    return 0;
}
```

**Oops!** What happened? Is it possible to **reuse** an portion of **source code** for **different data types**?

**Yes! With C++ Templates!**

# Types of Templates

- Function Templates
- Class Templates
- Variable Templates (C++14) – we won't cover this!

# Function Templates

## Simple offset function

**simple_template_f.cpp**

```
template< class TYPE >
TYPE OffSet( TYPE value, int offset ) {
    return value + offset;
}

int main(void)
{
    int i = OffSet( 1, 10 );

    float f = OffSet( 1.0f, 10 );

    double d = OffSet( 1.0, 10 );

    long double dd = OffSet( 1.0L, 10 );

    return 0;
}
```

int

float

double

long double

Template type parameter

```
template< class TYPE >
...
```

```
template< typename TYPE >
...
```

In the context of specifying a template, these two forms are equivalent.

# Function Templates

## Simple offset function

**simple_template_f.cpp**

```
template< class TYPE >
TYPE OffSet( TYPE value, int offset ) {
    return value + offset;
}

int main(void)
{
    int i = OffSet( 1, 10 );

    float f = OffSet( 1.0f, 10 );

    double d = OffSet( 1.0, 10 );

    long double dd = OffSet( 1.0L, 10 );

    return 0;
}
```

```
int OffSet( int value, int offset ) {
    return value + offset;
}
```

```
float OffSet( float value, int offset ) {
    return value + offset;
}
```

```
double OffSet( double value, int offset ) {
    return value + offset;
}
```

```
long double OffSet( long double value, int offset ) {
    return value + offset;
}
```

# Function Templates

## Generic sorting function

**temp_insertion_sort.cpp**

```
template< class TYPE >
void InsertionSort( TYPE *v, int n ) {
    int i, j;
    TYPE x;

    for ( j = 1; j < n; ++j ) {
        x = v[j];
        for ( i = j-1; i >= 0 && v[i] > x; --i )
            v[i+1] = v[i];
        v[i+1] = x;
    }
}
```

**Sorting an array of generic type**

```
...
int main(void) {
    double vector[5] = { 1.3, 1.1, 1.8, 1.6, 1.9 };
    InsertionSort( vector, 5 );

    for( int i = 0; i < 5; i++ )
        std::cout << "[" << i << "] :" << vector[i] << std::endl;

    return 0;
}
```

**Template type argument is inferred**

# Function Templates

## Generic sorting function

temp_insertion_sort.cpp

```
template< class TYPE >
void InsertionSort( TYPE *v, int n ) {
    int i, j;
    TYPE x;

    for ( j = 1; j < n; ++j ) {
        x = v[j];
        for ( i = j-1; i >= 0 && v[i] > x; --i )
            v[i+1] = v[i];
        v[i+1] = x;
    }
}
```

How about the **code generated** by the **compiler**?

**Calling the same function template with different inferred template type arguments**

```
...
int main(void) {
    int vector_i[5] = { 3, 1, 8, 6, 9 };
    double vector_d[5] = { 1.3, 1.1, 1.8, 1.6, 1.9 };

    InsertionSort( vector_i, 5 );
    InsertionSort( vector_d, 5 );
    ...
}
```

# Template Instantiation

**temp_insertion_sort.cpp**

```
...
int main(void) {
    int vector_i[5] = { ... };
    double vector_d[5] = { ... };

    InsertionSort( vector_i, 5 );
    InsertionSort( vector_d, 5 );
    ...
}
```

**temp_insertion_sort.s**

```
    ...
main:
    ...
    movq   ...
    ...
    movl   ...
    ...
    call   _Z13InsertionSortIiEvPT_i
    ...
    call   _Z13InsertionSortIdEvPT_i
    ...
    ret
    ...
_Z13InsertionSortIiEvPT_i:
    ...
    ret
    ...
_Z13InsertionSortIdEvPT_i:
    ...
    ret
    ...
```

The compiler **instantiates** the **function template** for **each** call with a distinct **template argument**!

# Template Instantiation

**temp_insertion_sort.cpp**

```
template< class TYPE >
void InsertionSort( TYPE *v, int n ) {
  int i, j;
  TYPE x;

  for ( j = 1; j < n; ++j ) {
    x = v[j];
    for ( i = j-1; i >= 0 && v[i] > x; --i )
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

**Function template instance for `InsertionSort( int ...)`**

```
void InsertionSort( int *v, int n ) {
  int i, j;
  int x;

  for ( j = 1; j < n; ++j ) {
    x = v[j];
    for ( i = j-1; i >= 0 && v[i] > x; --i )
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

**Function template instance for `InsertionSort( double ...)`**

```
void InsertionSort( double *v, int n ) {
  int i, j;
  double x;

  for ( j = 1; j < n; ++j ) {
    x = v[j];
    for ( i = j-1; i >= 0 && v[i] > x; --i )
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

If the **function template** is **not invoked**, it **does not generate code**!

# Sorting Objects of the **Record** Class

**temp_insertion_sort.cpp**

```
struct Record {
    int id_;
    int value_;
};

template< class TYPE >
void InsertionSort( TYPE *v, int n )
{
    int i, j;
    TYPE x;

    for ( j = 1; j < n; ++j )
    {
        x = v[j];
        for ( i = j-1; i >= 0 && v[i] > x; --i )
            v[i+1] = v[i];
        v[i+1] = x;
    }
}
...
```

```
...
int main(void) {

    Record vr[5];

    // initialize vr array

    InsertionSort( vr, 5 );
}
```

Try this code...

**What happened**?

The class **Record does not implement** the **operator '>'** (greater than)**!**

# Sorting Objects of the `Record` Class

**temp_insertion_sort.cpp (old)**

```
struct Record {
    int id_;
    int value_;
};

template< class TYPE >
void InsertionSort( TYPE *v, int n )
{
    ...
}
...
```

**temp_insertion_sort.cpp (new)**

```
struct Record {
    int id_;
    int value_;

    bool operator>( const Record &rhs ) const {
        return value_ > rhs.value_;
    }
};

template< class TYPE >
void InsertionSort( TYPE *v, int n )
...
```

**The behavior of the operator `'>'` is explicitly defined for the class `Record`. In this case, `Record` objects will be ordered according to the value of the `value_` data member.**

# Two or More Template Parameters

**more_params.cpp**

```
template< class T1, class T2 >
void vecScale( T1 *v, const T2 s )
{
    v[0] *= s;
    v[1] *= s;
    v[2] *= s;
}
...
```

**more_params.cpp**

```
...
int main( void ) {

    float v1[3] = { 1.0f, 2.0f, 3.0f };
    int s1 = 2;

    vecScale( v1, s1 );

    int v2[3] = { 1, 2, 3 };
    double s2 = 2.0;

    vecScale( v2, s2 );
}
```

```
void vecScale( float *v, const int s )
{...}
```

```
void vecScale( int *v, const double s )
{...}
```

**One function template instance will be created for each combination of template type parameters generated during function invocation.**

**more_params.s**

```
...
main:
    ...
    call    _Z8vecScaleIfiEvPT_T0_
    ...
    call    _Z8vecScaleIidEvPT_T0_
    ...
```

# Forcing a Specific Instantiation

**more_params.cpp**

```
template< class T1, class T2 >
void vecScale( T1 *v, const T2 s )
{
     v[0] *= s;
     v[1] *= s;
     v[2] *= s;
}
...
```

```
...
int main( void ) {

  float v1[3] = { ... };
  int s1 = 2;

  VecScale< float, float >( v1, s1 );

  int v2[3] = { ... };
  double s2 = 2.0;

  vecScale< int, float >( v2, s2 );

  float v3[3] = { ... };
  double s3 = 2.0;

  vecScale< float, float >( v3, s3 );
}
```

**more_params.s**

```
...
main:
     ...
     call    _Z8vecScaleIffEvPT_T0_
     ...
     call    _Z8vecScaleIifEvPT_T0_
     ...
     call    _Z8vecScaleIffEvPT_T0_
     ...
```

**If we force an specific template function instantiation, arguments will be cast if possible.**

**Same instance**

# A Class for Dynamic `float` Arrays

**fvector.cpp**

```
class vector {
public:

    ~vector( void ) {
        if ( elements_ptr_ ) {
            delete [] elements_ptr_;
            elements_ptr_ = nullptr;
        }
    }

    std::size_t size( void ) const {
        return num_elements_;
    }

private:

    float *elements_ptr_ = nullptr;
    std::size_t allocated_size_ = 0;
    std::size_t num_elements_ = 0;
}
```

**Release the dynamic array memory**

**Returns the number of elements stored in the dynamic array**

**Pointer to the dynamic array**

**Space actually allocated**

**Number of floats actually stored**

# A Class for Dynamic `float` Arrays

**fvector.cpp**

```
class vector {
public:

    ~vector( void ) ...

    std::size_t size( void ) ...

    void push_back( const float &a ) {
        if ( num_elements_ == allocated_size_ )
            reallocate_vector();

        elements_ptr_[ num_elements_++ ] = a;
    }

private:

    float *elements_ptr_ = nullptr;
    ...
}
```

**Insert new elements into the array**

# A Class for Dynamic `float` Arrays

**fvector.cpp**

```
class vector {
    ...

private:

    void reallocate_vector( void ) {
        if( num_elements_ == 0 ) {
            elements_ptr_ = new float;
            allocated_size_++;
        }
        else {
            float *tmp = new float[ allocated_size_ * 2 ];
            for ( std::size_t i = 0; i < allocated_size_; i++ )
                tmp[i] = elements_ptr_[i];
            delete [] elements_ptr_;
            elements_ptr_ = tmp;
            allocated_size_ *= 2;
        }
    }

    float *elements_ptr_ = nullptr;
    ...
}
```

**Reallocate the array when it is full**

# A Class for Dynamic `float` Arrays

**fvector.cpp**

```
#include <iostream>

class vector {
  ...

  ~vector( void ) ...

  std::size_t size( void ) ...

  void push_back( const float &a ) {

  float operator[]( std::size_t i ) const {
    return elements_ptr_[i];
  }

private:
  ...
}
...
```

But, it works **only** for **`floats`**.
**How** we could **modify** it to handle **any type**?

**Usage example**

```
...
int main(void) {
  vector x;
  x.push_back( 10.2f );
  x.push_back( 20.3f );
  x.push_back( 30.4f );

  for(std::size_t i = 0; i < x.size(); i++)
    std::cout << x[i] << std::endl;

  return 0;
}
```

# A Class Template for Dynamic Arrays

**tvector.cpp**

```
#include <iostream>

template< class T >
class vector {
  ...

  void push_back( const T &a ) ...

  T& operator[]( std::size_t i ) ...

private:

  void reallocate_vector( void ) ...

  T *elements_ptr_ = nullptr;
  ...
}
...
```

```
void reallocate_vector( void ) {
  if( num_elements_ == 0 ) {
    elements_ptr_ = new T;
    ...
  }
  else {
    T *tmp = new T[ allocated_size_ * 2 ];
    ...
  }
}
```

Any mention to type `float` is **removed** in favor of the `T` **generic type**.

# A Class Template for Dynamic Arrays

**tvector.cpp**

```
#include <iostream>

namespace MyStd {
    template< class T >
    class vector {
        ...
    }
}
...
```

And before we proceed with our tests, lets wrap up the `vector` class with the `MyStd` namespace.

```
...
int main(void) {
  MyStd::vector< float > x;
  x.push_back( 10.2f );
  x.push_back( 20.3f );
  x.push_back( 30.4f );

  for(std::size_t i = 0; i < x.size(); i++)
    std::cout << x[i] << std::endl;

  return 0;
}
```

This **class instantiation** does not **resemble** the one of **`std::vector`**?? :)

# Separate Compilation of Template Code

**vector.h**

```
#include <iostream>

namespace MyStd {

template< class T >
class vector {
public:

   ~vector( void );
   std::size_t size( void ) const;
   void push_back( const T &a );
   T& operator[]( std::size_t i );

private:

   void reallocate_vector( void );
   T *elements_ptr_ = nullptr;
   std::size_t allocated_size_ = 0;
   std::size_t num_elements_ = 0;
}
}
```

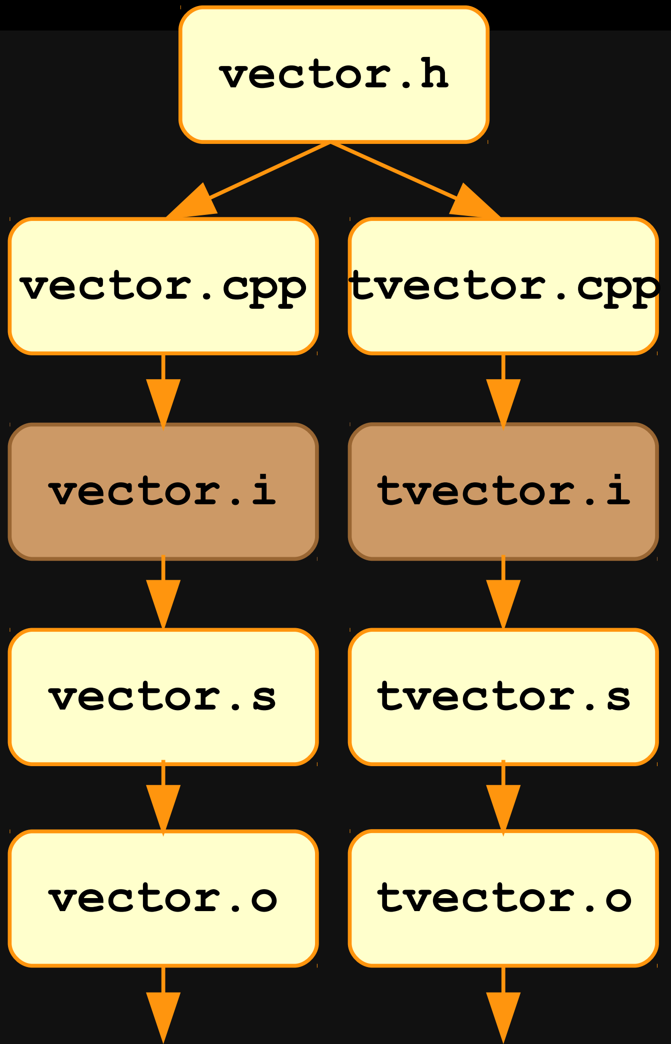
**vector.cpp**

```
#include "vector.h"

namespace MyStd {

   template< class T >
   vector< T >::~vector( void )
   {...}

   template< class T >
   std::size_t vector< T >::size( void ) const
   {...}

   template< class T >
   void vector< T >::push_back( const T &a )
   {...}

   template< class T >
   T& vector< T >::operator[]( std::size_t i )
   {...}

   template< class T >
   void vector< T >::reallocate_vector( void )
   {...}
}
```

# Separate Compilation of Template Code

**tvector.cpp**

```
#include <iostream>
#include "vector.h"

int main(void) {

  MyStd::vector< float > x;
  x.push_back( 10.2f );
  x.push_back( 20.3f );
  x.push_back( 30.4f );

  for(std::size_t i = 0; i < x.size(); i++)
    std::cout << x[i] << std::endl;

  return 0;
}
```




What happened here?

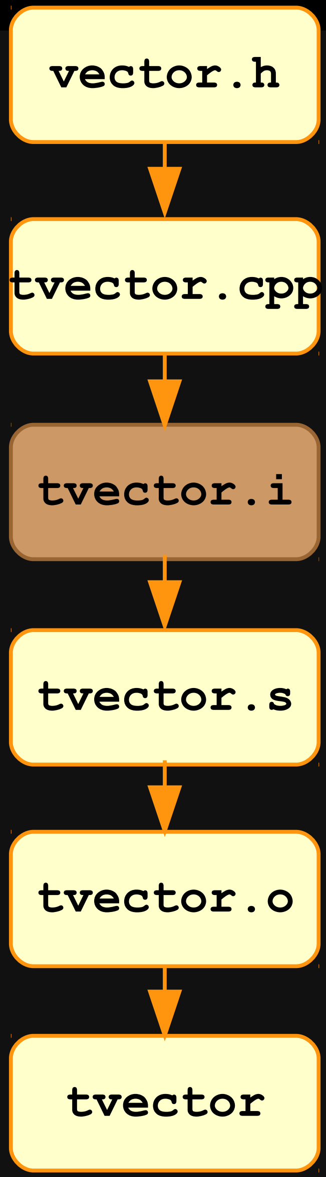
```
main.o: In function `main':
...undefined reference to `MyStd::vector<float>::push_back(float const&)'
...undefined reference to `MyStd::vector<float>::push_back(float const&)'
...
```

# Separate Compilation of Template Code

**vector.h**

```
#include <iostream>

namespace MyStd {

template< class T >
class vector {
public:

   ~vector( void );
   ...

private:
   ...
}

template< class T >
vector< T >::~vector( void )
{...}

...

}
```

**Templates** are resolved at **compile time**, and it is required that the **entire** class template **source** be available (in the current **translation unit**) at the **moment** of **compilation**.



# One Template Application: **`SmartPtr`**

**`smart_ptr.cpp`**

```
template< class T >
class SmartPtr {
public:
    SmartPtr( T *ptr = nullptr ) :
        ptr_( ptr )
    {}

    ~SmartPtr( void ) {
        delete ptr_;
    }

private:
    T *ptr_ = nullptr;
};
...
```

```
...
class Dummy{
};

int main(void)
{
    Dummy *x = new Dummy;

    Dummy *y = new Dummy;
    SmartPtr< Dummy > smart_ptr( y );

    return 0;
}
```

# References

- An Idiot's Guide to C++ Templates - Part 1. Ajay Vijayvargiya. 2013.
  - http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part
- An Idiot's Guide to C++ Templates - Part 2. Ajay Vijayvargiya. 2013.
  - http://www.codeproject.com/Articles/268849/An-Idiots-Guide-to-Cplusplus-Templates-Part

# References

- C++ Templates: The Complete Guide.
  - David Vandevoorde, Nicolai M. Josuttis.
- Moder C++ Design: Generic Programming and Design Patterns Applied.
  - Andrei Alexandrescu.