Template meta-programming in C++

git clone git@github.com:NCCA/Templates.git
Textbooks

C++ Templates
The Complete Guide
David Vandevoorde
Nicolai M. Josuttis

C++ Template Metaprogramming
Concepts, Tools, and Techniques from Boost and Beyond
David Abrahams
Aleksey Gurtovoy

C++ In-Depth Series • Bjarne Stroustrup
Introduction

• “Code-generating programs are sometimes called metaprograms; writing such programs is called metaprogramming”.

• There are many different forms of meta programs depending upon the language being used.

• In General this can be approached either by having a program generate new code at runtime

• Or by using some form of pre-processor.
Macros in C/C++

• The C/C++ pre-processor is responsible for macro expansion.

• This happens before the code is compiled.

• Usually we limit the use of Macros to the C programming language, and try to use them sparingly.

• In C++ we will avoid and use templates.
C Macros

• The advantage of a macro is that it can be type neutral

• It's inlined directly into the code, so there isn't any function call overhead.

• They are defined as follows

```c
#define MACRO_NAME(arg1, arg2, ...) [code to expand to]
```
#include <stdio.h>
#include <stdlib.h>

#define min(X, Y)  ((X) < (Y) ? (X) : (Y))

int main()
{
    int a=5;
    int b=6;
    printf("min(a,b)=%d\n", min(a,b));

    return EXIT_SUCCESS;
}
```c
#include <stdio.h>
#include <stdlib.h>

#define MULT(x, y) x * y

#define MULTB(x, y) ((x) * (y))

int main()
{
    int z = MULT(3 + 2, 4 + 2);
    int x = MULTB(3 + 2, 4 + 2);
    printf("z=\%d\n", z);
    printf("x=\%d\n", x);

    return EXIT_SUCCESS;
}
```
X Macros

• x macros are a technique to generate compile time code structures

• They consist of two parts,
  • a list
  • an execution of a list

• Can be useful for generating types and lookup tables for error checking etc.

• Also allows for easy expansion / maintenance of enums and code
```c
#include <stdio.h>
#include <stdlib.h>

#define COLOR_TABLE
X(red, "red")
X(green, "green")
X(blue, "blue")

#define X(a, b) a,
enum COLOR
{
    COLOR_TABLE
};
#undef X

#define X(a, b) b,
char *color_name[] =
{
    COLOR_TABLE
};
#undef X

int main()
{
    enum COLOR c = red;
    printf("c=%s\n", color_name[c]);
    return EXIT_SUCCESS;
}
```
#include <stdio.h>
#include <stdlib.h>

#define COLOR_TABLE
X(green, , "green") \nX(red, =3, "red") \nX(blue, , "blue")

#define X(a, b, c) a b,
enum COLOR
{
  COLOR_TABLE
};
#undef X

#define X(a, b, c) [a]=c,
char *color_name[] =
{
  COLOR_TABLE
};
#undef X

int main()
{
  enum COLOR c = red;
  printf("c=%s\n", color_name[c]);
  return EXIT_SUCCESS;
}
macros

• In general we will try not to use macros

• They are difficult to debug (however clang -E will give us the output of the pre-process as an aid)

• Can sometimes cause weird side effects and substitution issues

• Some debuggers will also have issues with the expanded code vs the source code.
Options for generic programming

• Implement the the same behaviour again and again for each type that needs it (OpenGL ?)

• Write generic code using void * and force the programmer to use coercion

• Use a special pre-processor (or 3rd party tools to generate your code)

• This usually leads to mistakes and can cause problems of maintenance and re-use.
templates

• C++ requires us to declare variables, function and most other entities using specific types.

• When we write code it usually looks the same for many different functions.

• This is very obvious if we write an algorithm to sort a list of data, depending upon the data type we would need a different implementation of essentially the same thing.

• This becomes more problematic if the language your are using doesn’t support generics.
templates in C++

• C++ supports 3 types of templates
  • function templates
  • class templates
  • variable templates (C++ 14 so don’t worry yet!)

```cpp
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```
template terminology

image from C++ common knowledge
template terminology

• precise terminology is important, especially when discussing templates.

• In the previous diagram we show the different names for the elements.

• The most important distinction is between the template parameter used in the definition of the template and the template argument which is used in the specialisation of the template.
// T is the template parameter
template <typename T>
class Heap{.....};

// double is the template argument
Heap<double> dHeap;
template terminology

• template name :
  • the simple identifier for the template

• template id
  • the template name with an appended template argument list. <T ....>
Function Templates

• Function templates provide a functional behaviour that can be called on for different types.

• This can be though of as a family of functions.

• We basically declare a function but unlike a normal function our parameters and / or return types are abstracted so they have no initial type

• Then the compiler will replace the types with the concrete types at compile time.
simple template

- max.h

```cpp
#ifndef MAX_H_
#define MAX_H_

template <typename T>
inline T const &max(T const & a, T const & b) {
    // if a<b then use b else use a
    return a < b ? b : a;
}
#endif
```
#include <iostream>
#include <string>
#include <cstdlib>
#include "max.h"

int main()
{
    std::cout << ::max(2, 5) << "\n";
    std::cout << ::max(0.1, 0.02) << "\n";
    std::cout << ::max('a', 'c') << "\n";
    std::cout << ::max("hello", "world") << "\n";

    return EXIT_SUCCESS;
}
global namespace

• In the previous example the max template was prefixed with :: to indicate it was in the global namespace

• This is due to potential issues with std::max

• If one argument type is in the std name space (string) according to the lookup rules both the local and std::max will be found as per the next example
#include <iostream>
#include <string>
#include <cstdlib>
#include "max.h"

int main()
{
    std::string a("hello");
    std::string b("templates");
    std::cout<< max(a,b)<<"\n";

    return EXIT_SUCCESS;
}
The template must be able to expand for the type and any operators

```cpp
#include <iostream>
#include <complex>
#include <cstdlib>
#include "max.h"

int main()
{
    std::complex<float> c1, c2;
    std::cout << ::max(c1, c2) << "\n";

    return EXIT_SUCCESS;
}
```

/max.h:9:11: error: invalid operands to binary expression ('const std::__1::complex<float>' and 'const std::__1::complex<float>')
    return a < b ? b : a;
~ ~ ~
template compilation

• templates are effectively compiled twice

• Without instantiation they are checked for syntax (e.g.; missing etc)

• At the time of instantiation (i.e. where it is used) it is checked to see if all calls are valid.

• For ease of use and to avoid this for simple templates we can use a header and an inline function
Argument deduction

• When we call a template function for some arguments, the template parameters determined by the arguments we pass.

• If we pass to ints to the parameter types T const & the C++ compiler must conclude that T must be an int.

• No automatic type conversion is allowed. Each T must match exactly.
```cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "max.h"

int main()
{
    std::cout << ::max(4, 7) << "\n";
    std::cout << ::max(4, 4.2) << "\n";

    return EXIT_SUCCESS;
}
```

error: no matching function for call to 'max'
    std::cout << ::max(4, 4.2) << "\n";
  ~~~~~
./max.h:6:17: note: candidate template ignored: deduced conflicting types for parameter 'T'
 ('int' vs. 'double')
inline T const &max(T const & a, T const & b)
Overloading function templates

- Like ordinary functions, function templates can also be overloaded.

- The compiler must then decide which ones to use.

- The rules of these decisions and how the compiler does this can get very complicated and lead to issues.

- The rules are similar to normal functions (for an in-depth discussion see Appendix B of “C++ templates the complete guide”
/* The following code example is taken from the book
 */
/* "C++ Templates - The Complete Guide"
 * by David Vandevoorde and Nicolai M. Josuttis, Addison-Wesley, 2002
 */

// maximum of two int values
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}

// maximum of two values of any type
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// maximum of three values of any type
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);   // calls the template for three arguments
    ::max(7.0, 42.0);  // calls max<double> (by argument deduction)
    ::max('a', 'b');   // calls max<char> (by argument deduction)
    ::max(7, 42);      // calls the nontemplate for two ints
    ::max<>()(7, 42);   // calls max<int> (by argument deduction)
    ::max<double>(7, 42); // calls max<double> (no argument deduction)
    ::max('a', 42.7);  // calls the nontemplate for two ints
}
**function template summary**

- Define a family of functions for different arguments
- When arguments are passed templates are instantiated for those arguments
- You can overload function templates
- When overloading limit changes by explicitly specifying parameters ( `<int,int>` )
- Make sure all overloaded versions are visible before being called
Class templates

- Like functions, classes may also be parameterised with one or more types.
- This is how STL works for most containers.
- This allows the type of the element being contained to be left open.
- The following example shows a simple stack template.
class template declaration

• Declaring class templates is similar to declaring functions.

• Before the declaration a statement declares an identifier as the type parameter(s)

• Usually T is used for a single simple parameter.

```cpp
template <typename T> class NAME
{
    .....}
```
#include <vector>
#include <stdexcept>

template <typename T>

class Stack
{
  private :
   std::vector <T> m_elements;

  public :
    void push(T conts &);
    void pop();
    T top() const;
    bool empty() const { return m_elements.empty();}
};
template <typename T>
void Stack<T>::push(T const & _e)
{
    m_elements.push_back(_e);
}

template <typename T>
void Stack<T>::pop()
{
    if(m_elements.empty())
    {
        throw std::out_of_range("Stack<>::pop()<>:empty_stack");
    }
    m_elements.pop_back();
}

template <typename T>
void Stack<T>::top() const
{
    if(m_elements.empty())
    {
        throw std::out_of_range("Stack<>::top()<>:empty_stack");
    }
    return m_elements.back();
}
The type of the class is Stack<T> with T being the template parameter.

Thus you have to use Stack<T> whenever you use the class definition.

This can be important for copy and assignment operators.

template <typename T> class Stack
{
    public:
    // copy ctor
    Stack(Stack<T> const &);
    // assignment operator
    Stack<T> & operator= (Stack<T> const &);
};
member functions

• To define a member function of a class template you have to specify that it is a function template.

• You also have to use the full template qualification as shown in the previous example.
#include <iostream>
#include <string>
#include <cstdlib>
#include "Stack.h"

int main()
{
    try
    {
        Stack <int> intStack;
        Stack <std::string> stringStack;

        intStack.push(7);
        std::cout<< intStack.top() << std::endl;

        stringStack.push("hello");
        std::cout<< stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch(std::exception const &ex)
    {
        std::cerr<< "Exception"<< ex.what()<<"\n";
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
Using the Stack<T>

• In the previous example we use Stack<int> and Stack<std::string> to instantiate two different versions (specialisation) of the Stack

• Code is only created (instantiated) for the methods that are called on the stack for that type.

• As pop is only used on the std::string stack it will only be created for that version.
An instantiated class template can be used as a type.

The following examples show this:

```c++
void foo(Stack<int> const &s)
{
    Stack<int> istack[10]; // array of int stacks
    ....
}

typedef Stack<int> IntStack;
IntStack a;

typedef Stack<float *> floatPtrStack;
typedef Stack<int> IntStack;
Stack <Stack<int> > intStackStack;

// error
Stack <Stack<float>> floatStackStack;
```
template specialisation

• Class templates can be specialised for certain arguments.

• This is similar to normal function overloading, however could result in a complete new class implementation.

• The following example overloads the Stack for the std::string
```cpp
#include <deque>

template <>
class Stack<std::string>
{
private:
    std::deque<std::string> m_elements;

public:
    void push(std::string const &);
    void pop();
    std::string top() const;  // This function is not defined in the class declaration.
    bool empty() const { return m_elements.empty(); }

};

void Stack<std::string>::push(std::string const & _e)
{
    m_elements.push_back(_e);
}

void Stack<std::string>::pop()
{
    if(m_elements.empty())
    {
        throw std::out_of_range("Stack<std::string>::pop() empty stack");
    }
    m_elements.pop_back();
}

std::string Stack<std::string>::top() const
{
    if(m_elements.empty())
    {
        throw std::out_of_range("Stack<std::string>::top() empty stack");
    }
    return m_elements.back();
}
```
partial specialisation

- Class templates can be partially specialised

- You can specify special implementations for particular circumstances, however some template parameters must still be defined by the user

- given the following class template

```cpp
template <typename T1, typename T2>
class MyClass
{
    ...
};
```
// both have the same type
template <typename T>
class MyClass<T, T>
{
    ....
};

// second type is int
template <typename T>
class MyClass<T, int>
{
    ....
};

// both are pointer types
template <typename T1, typename T2>
class MyClass<T1*, T2*> 
{
    ....
};
deduction for specialisation

```cpp
MyClass<int, float> mif; // uses MyClass<T1, T2>
MyClass<float, float> mff; // uses MyClass<T, T>
MyClass<float, int> mfi; // uses MyClass<T, int>
MyClass<int*, float*> mp; // uses MyClass<T1*, T2*>
// these are ambiguous and cause Errors
MyClass<int, int> m; // matches MyClass<T, T> and MyClass<T, int>
MyClass<int *, int *> m; // matches MyClass<T, T> and MyClass<T1*, T2*>

// the last error can be removed by doing this partial specialisation
template <typename T>
class MyClass<T*, T*> {
    ....
};
```
Default Template Arguments

• In the previous example we created a specialisation for std::string

• We can modify our class template to pass in a container type (which is also a template)

• By setting this to default to std::vector we can for most times use this but specialise if we wish.
template <typename T, typename CONTAINER = std::vector<T> >
class Stack
{
private:
    CONTAINER m_elements;

public:
    void push(T const &);
    void pop();
    T top() const;
    bool empty() const { return m_elements.empty(); }
};
template<typename T, typename CONTAINER>
void Stack<T,CONTAINER>::push(T const & _e)
{
    m_elements.push_back(_e);
}

template<typename T, typename CONTAINER>
void Stack<T,CONTAINER>::pop()
{
    if(m_elements.empty())
    {
        throw std::out_of_range("Stack<>::pop()\n         : empty stack");
    }
    m_elements.pop_back();
}

template<typename T, typename CONTAINER>
T Stack<T,CONTAINER>::top() const
{
    if(m_elements.empty())
    {
        throw std::out_of_range("Stack<>::top()\n         : empty stack");
    }
    return m_elements.back();
}

#include <iostream>
#include <string>
#include <cstdlib>
#include <deque>
#include "Stack.h"

int main()
{
    try
    {
        Stack <int> intStack;
        Stack <std::string, std::deque<std::string>> stringStack;

        intStack.push(7);
        std::cout << intStack.top() << std::endl;

        stringStack.push("hello");
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (std::exception const &ex)
    {
        std::cerr << "Exception: " << ex.what() << "\n";
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
Nontype parameters

• template parameters don’t have to be types
• The can also be ordinary values.
• The value will need to be declared explicitly.
• The following example show this in action to create a simple Array class (which is now in C++ 11)
template<typename T, size_t SIZE=0>
class Array
{
    private:
        // our array data
        T m_data[SIZE];
        // function to check for a valid array range.
        void rangeCheck(std::size_t idx);

    public:
        // ctor will set the data to zero type
        Array();
        // this is constant so can be static
        static size_t size() { return SIZE; }
        // access data elements
        T& at(std::size_t idx);
        // subscript operators
        T& operator[](std::size_t idx);
        const T& operator[](std::size_t idx) const;

        // read only data access
        const T* data() const { return m_data; }
        T* data() { return m_data; }

        Array<T,SIZE>& operator= (const Array<T,SIZE>& rhs);

        // types for iterators (boost style definitions)
        typedef T*      iterator;
        typedef const T* const_iterator;
        // begin iterators
        iterator begin() { return m_data; }
        const_iterator begin() const { return m_data; }
        const_iterator cbegin() const { return m_data; }
        // end iterators
        iterator end() { return m_data+SIZE; }
        const_iterator end() const { return m_data+SIZE; }
        const_iterator cend() const { return m_data+SIZE; }
};
template <typename T, size_t SIZE>
Array<T,SIZE>& Array<T,SIZE>::operator= (const Array<T,SIZE>& rhs)
{
    std::copy(rhs.begin(), rhs.end(), begin());
    return *this;
}

template <typename T, size_t SIZE>
Array<T,SIZE> Array<T,SIZE>::Array()
{
    // set to the default ctor value of T
    for(size_t i=0; i<SIZE; ++i)
        m_data[i]=T();
}

template <typename T, size_t SIZE>
void Array<T,SIZE>::rangeCheck(std::size_t idx)
{
    if(idx>SIZE)
        throw std::out_of_range("index_out_of_range_in_Array<T>::at");
}

template <typename T, size_t SIZE>
T& Array<T,SIZE>::operator[](std::size_t idx)
{
    rangeCheck(idx);
    return m_data[idx];
}

template <typename T, size_t SIZE>
T& Array<T,SIZE>::at(std::size_t idx)
{
    rangeCheck(idx);
    return m_data[idx];
}

template <typename T, size_t SIZE>
const T& Array<T,SIZE>::operator[](std::size_t idx) const
{
    rangeCheck(idx);
    return const_cast<T&>(m_data[idx]);
}
```cpp
#include <iostream>
#include <cstdlib>
#include <string>
#include "Array.h"

int main()
{
    Array<int, 10> intArray;
    for (int i=0; i<intArray.size(); ++i)
        intArray[i]=i;
    // copy array
    Array<int, 10> array2=intArray;
    Array<int>::const_iterator begin=array2.begin();
    Array<int>::const_iterator end=array2.end();
    while (begin != end)
    {
        std::cout<<* (begin++) <<"\n";
    }
    std::cout<<"\n";
    try
    {
        std::cout<<"try to access element at -1"<<intArray.at(-1)<<"\n";
    }
    catch (std::exception const &ex)
    {
        std::cerr<<"Exception"<<ex.what()<<std::endl;
    }
    try
    {
        std::cout<<intArray.at(6)<<"\n";
    }
    catch (std::exception const &ex)
    {
        std::cerr<<"Exception"<<ex.what()<<std::endl;
    }
    try
    {
        std::cout<<"try to access element at 99"<<array2.at(99)<<"\n";
    }
    catch (std::exception const &ex)
    {
        std::cerr<<"Exception"<<ex.what()<<std::endl;
    }
}
```
typename

• The keyword typename was introduced during the C++ 11 standardisation process

• It helps to clarify that an identifier within a template is a type.

• It should be used instead of Class (which could be used in the previous versions of C++)

```cpp
template <typename T>
class MyClass
{
    typename T::SubType *ptr;
};
```
typename

- In the previous example typename is used to clarify that SubType is a type defined within class T.

- Thus ptr is a pointer to the type T::SubType

- Without typename SubType would be considered a static member.

- A typical application of this would be to give access to iterators within STL containers
// use C++ 11
#include <iostream>
#include <vector>
#include <list>

template <typename T>
void print(T const &_c)
{
    typename T::const_iterator pos;
    typename T::const_iterator end(_c.end());
    for(pos=_c.begin(); pos!=end; ++pos)
    {
        std::cout<< *pos <<'\n';
    }
    std::cout<<std::endl;
}

int main()
{
    // note c++11 init lists
    std::vector <int> v1={1,2,3,4,5};
    std::list <float> l1={0.1,0.2,0.5,0.3};
    print(v1);
    print(l1);
}
typename

• In this function the parameter is an STL container of type T

• To iterate over the elements we need to access the iterators.

• We need to qualify it with the leading typename as internally it would be specified thus

```cpp
class stlcontainer
{
   ....
   typedef ... iterator;
   typedef ... const_iterator;
   ....
};
```
the .template construct

• A simple problem was discovered after the introduction of typename.

```cpp
template <size_t N>
void printBitset(std::bitset<N> const &b) {
    std::cout << b.template to_string < char, std::char_traits<char>,
               std::allocator<char> >() << "\n";
}

int main() {
    std::bitset<8>b(std::string("10101010"));
    printBitset(b);
    return EXIT_SUCCESS;
}
```

• This only happens if the construct before the period depends upon the template argument.
Member Templates

- Class Members can also be templates.
- This is possible for both nested classes and member functions.
- The following example show this in action with an a templated assignment operator.
- This will allow us to assign and implicitly convert types that allow this (such as int to float)
template <typename T>

class Stack
{
private:
    std::deque <T> m_elements;

public:
    void push(T const &);
    void pop();
    T top() const;
    bool empty() const { return m_elements.empty();}
    template <typename T2>
    Stack<T> & operator=(Stack<T2> const &);
};

Note T2 inner template parameter
template <typename T>

template <typename T2>
Stack<T> & Stack<T>::operator=(Stack<T2> const &_rhs)
{
    // self assignment?
    if((void *)this == (void *)&_rhs)
    {
        return *this;
    }
    // copy stack with type 2
    Stack<T2> tmp(_rhs);
    // remove existing data
    m_elements.clear();
    while(!tmp.empty())
    {
        m_elements.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
Stack<T>::operator =

• In this example an inner template parameter T2 is defined.

• The member function has to make a temporary copy of the parameter passed in.

• This is due to them being different types and thus only being able to use the public interface of the 2nd type to access the data.

• We now use a std::deque as we need to access the data via top() which necessitates the pushing of the data to the front of the stack.
int main() {
    Stack <int> intStack;
    for(int i=0; i<5; ++i) {
        intStack.push(i);
    }
    Stack <float> fStack;
    fStack = intStack;
    while(!fStack.empty()) {
        std::cout<<fStack.top()<<\n;  
        fStack.pop();
    }
}

This will work as int converts to float
non conversion

```cpp
Stack <std::string> sStack;
sStack = fStack;
```

- In this case the conversion is not possible, however the error message from the compiler will appear where the conversion takes place.
Template Template Parameters

- It is also possible to allow a template parameter itself to be a class template

- For example if we wished to use different internal containers for the stack, we would have to specify the internal container and it’s type.

- This means we can specify the container without the type of the elements.

```
Stack<int,std::vector<int>> > // declare int twice
Stack<int,std::vector> // just use same type
```
Argument matching

```
template < class T, class Alloc = allocator<T> > class deque;
```

- In the following example we are going to use the std::deque as the default template.

- The deque has the template parameters as shown above and we will need to match these when we create our template template class.

- To do this we use the std::allocator template.
Allocators

- Encapsulates a memory allocation and deallocation strategy

- Allocators are classes that define memory models to be used by some parts of the Standard Library, and most specifically, by STL containers.

- Usually we use the default std::allocators however by providing this template std allows us to define our own if we wish.

- This allows us to create our own memory pools, shared memory allocations etc for stl containers if we wish.


```cpp
template <typename T,
    template <typename CELEM,
            typename = std::allocator<CELEM> >
    class CONTAINER = std::deque>

class Stack
{
private:
    CONTAINER<T> m_elements;

public:
    void push(T const&);
    void pop();
    T top() const;
    bool empty() const { return m_elements.empty(); }

    template<typename T2,
        template<typename CELEM2,
                typename = std::allocator<CELEM2> >
        class CONTAINER2>
    Stack<T,CONTAINER>& operator= (Stack<T2,CONTAINER2> const&);  
};
```

note space (C++11 fixes this)

container must be a class
```cpp
template <typename T, template <typename, typename> class CONTAINER>
void Stack<T, CONTAINER>::push (T const& CELEM)
{
    m_elements.push_back(CELEM);
}

template<typename T, template <typename, typename> class CONTAINER>
void Stack<T, CONTAINER>::pop ()
{
    if (m_elements.empty())
    {
        throw std::out_of_range("Stack>::pop(): empty stack");
    }
    m_elements.pop_back();
}

template <typename T, template <typename, typename> class CONTAINER>
T Stack<T, CONTAINER>::top () const
{
    if (m_elements.empty())
    {
        throw std::out_of_range("Stack>::top(): empty stack");
    }
    return m_elements.back();
}
```
Note that we now have two template parameters as they may be different types

```cpp
template <typename T, template <typename,typename> class CONTAINER>
    template <typename T2, template <typename,typename> class CONTAINER2>
    Stack<T,CONTAINER>&
    Stack<T,CONTAINER>::operator= (Stack<T2,CONTAINER2> const& _rhs)
    {
        if ((void*)this == (void*)&_rhs)
        {
            return *this;
        }

        Stack<T2,CONTAINER2> tmp(_rhs);
        m_elements.clear();
        while (!tmp.empty())
        {
            m_elements.push_front(tmp.top());
            tmp.pop();
        }
        return *this;
    }
```
int main()
{
    Stack <int> intStack;
    for(int i=0; i<5; ++i)
    {
        intStack.push(i);
    }
    Stack <float> fStack;
    fStack=intStack;
    while(!fStack.empty())
    {
        std::cout<<fStack.top()<<"\n";
        fStack.pop();
    }
    std::cout<<"using Vector stack\n";
    // stack for ints using a vector as an internal container
    Stack<int, std::vector> vStack;
    vStack.push(42);
    vStack.push(7);
    std::cout << vStack.top() << std::endl;
    vStack.pop();
}

using string literals

• String literals can throw up interesting problems with template parameters

• Consider the following template

```cpp
template <typename T>
inline T const &max(T const &a, T const &b)
{
    return a<b ? b : a;
}
```
using string literals

```cpp
int main()
{
    std::string s;
    ::max("apple","peach");
    ::max("apple","tomato");
    ::max("apple",s);
    return EXIT_SUCCESS;
}
```

- The first compare will work as the size of the strings is the same.
- The other will fail as they are effectively const char data of different sizes.
- Best way to overcome this is to specialise for std::string
templates in practice

• Normal code is usually placed in the following formats

  • Classes and other types in header files.

  • For global variables and (non-inline) functions only the declaration is put in the .h header file.

  • The definition is put in the .c(pp) file

• This works well and helps the linker.
Problem 1

Type.h

```cpp
#ifndef TYPE_H__
#define TYPE_H__

template <typename T>
void printType(T const &);

#endif
```

Type.cpp

```cpp
#include "Type.h"

#include <iostream>
#include <typeinfo>

template<typename T>
void printType(T const &_t)
{
    std::cout<<"Type is:"
        << typeid(_t).name()<<"\n";
}
```
Linker has problems as printType has not been instantiated. Due to the fact that two files have been used.
Problem 1

• The compiler is happy and will compile both modules.

• In order for the template to be instantiated, the compiler must know which definition should be instantiated and for which template arguments.

• In the previous example these two pieces of information are in different files and the compiler assumes it will be resolved by the linker.

• However it is the compilers job to do the instantiation.
The inclusion model

• Typical practice is to place the template definition and declaration in the same header file

• This does have some issues

  • Increased cost of compilation due to having to include other headers `<iostream>`

  • This can significantly increase the compilation time

• Still the best method for most simple cases
explicit instantiation

- The C++ standard allows us to manually instantiate a template.

- This is advantageous as we can generate all the types we may need in advance

- can cause problems with maintenance etc when using libraries.

- Can be tuned to speed compilation in some case.

```cpp
template void printType<double>(double const &);
template void printType<float>(float const &);
```
parameter pack

• A function parameter pack is a function parameter that accepts zero or more function arguments

• A template parameter pack is a template parameter that accepts zero or more template arguments

• A template with at least one parameter pack is called a variadic template

• typename... Args is called a template parameter pack

• Args.. args is called a function parameter pack (Args is by convention only)
variadic templates

- These were introduced into the C++ 11 standard

- Prior to this all templates could only have a set number of arguments, set at declaration time

- When the ellipsis (...) operator appears to the left of a name it declares a parameter pack.

- When the ellipsis operator occurs to the right of a template or function call argument, it unpacks the parameter packs into separate arguments
recursion

- As it is difficult to expand the (comma) separated parameter packs we usually use recursion

- Typically a simple non templated function will be generated for the 0-1 argument version

- Then a templated parameter pack version will be generated which will expand and call the simple version.

- The following example shows a simple printf style function that uses the % as a place holder for arguments (Qt Style)
/ **base function**

```cpp
void print(const char* format)
{
    std::cout << format;
}
```

/** recursive variadic function */

```cpp
template<typename T, typename... Targs>
void print(const char* format, T value, Targs... Fargs)
{
    for ( ; *format != '\0'; format++)
    {
        // if we have a place holder expand
        if ( *format == '%' )
        {
            std::cout << value;
        // recursive call
            print(format+1, Fargs...);
            return;
        }
        std::cout << *format;
    }
}
```
int main()
{
    int a=10;
    float b=0.2f;
    std::string c="data";
    print("Values_int_%float_%string_%n",a,b,c);
    return EXIT_SUCCESS;
}
recursion in depth

• When the template expands it needs to write versions for the different function call signatures

• We can easily debug this to see what is happening in the expansion process using the

• __PRETTY_FUNCTION__ g++ and clang extension

• This is shown in the following example
template<
typename T>
T adder(T v)
{
    std::cout << __PRETTY_FUNCTION__ << "\n";
    return v;
}

template<
typename T, typename ... Args>
T adder(T first, Args... args)
{
    std::cout << __PRETTY_FUNCTION__ << "\n";
    return first + adder(args...);
}
int main()
{
    long sum = adder(1, 2, 3, 8, 7);

    std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
    std::string ssum = adder(s1, s2, s3, s4);
}

T adder(T, Args...) [T = int, Args = <int, int, int, int>]
T adder(T, Args...) [T = int, Args = <int, int, int>]
T adder(T, Args...) [T = int, Args = <int, int>]
T adder(T, Args...) [T = int, Args = <int>]
T adder(T) [T = int]
std::initializer_list

- This is a new C++ 11 feature that allows us to pass variable initialiser lists to functions and templates.

- According to the spec it is “is a lightweight proxy object that provides access to an array of objects of type const T.”

- A std::initializer_list object is automatically constructed when:
  - a braced-init-list is used in list initialisation, including function-call list initialisation and assignment expressions
  - a braced-init-list is bound to auto, including in a ranged for loop
```cpp
#include <iostream>
#include <cstdlib>
#include <initializer_list>

template <typename T>
void print(std::initializer_list<T> data)
{
    for (auto s : data)
    {
        std::cout << s << " ";
        std::cout << std::endl;
    }
}

int main()
{
    print({"this", "and", "that"});
    print({1, 2, 3, 4, 5});
    print({1.0f, 2.0f, 0.2f, 9.1f});
    print({'a', 'b', 'c'});
    return EXIT_SUCCESS;
}
```
References

- C++ templates the complete Guide, David Vandevoorde, Nicolai M. Josuttis. Addison Wesley 2003