An introduction to C++ template programming

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

March 2015
Templates and parametric polymorphism

Template parameters

Member functions of template classes

Object oriented patterns

Lambda expressions in C++11

Void pointer polymorphism in C

Template specialization
**C++ polymorphism**

<table>
<thead>
<tr>
<th></th>
<th>Templates</th>
<th>Dynamic polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>When</strong></td>
<td>compile-time</td>
<td>run-time</td>
</tr>
<tr>
<td><strong>Typing</strong></td>
<td>Type parameters</td>
<td>Subtyping</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>+ no runtime overhead</td>
<td>- indirection via pointers</td>
</tr>
<tr>
<td></td>
<td>- potential code bloat</td>
<td>at runtime</td>
</tr>
<tr>
<td><strong>Related to</strong></td>
<td>OCAMLL and Haskell polymorphism</td>
<td>Objective C messages</td>
</tr>
<tr>
<td></td>
<td>Java generics</td>
<td>Java methods</td>
</tr>
<tr>
<td></td>
<td>ML functors</td>
<td></td>
</tr>
</tbody>
</table>

*Over the last two decades, templates have developed from a relatively simple idea to the backbone of most advanced C programming.* (Stroustrup 2012, section 25.1)
C++ templates

- templates are important: Standard Template Library
- type-safe collections
- templates interact/collide with other C++ features
- templates allow compile-time computation $\Rightarrow$ zero overhead
- templates are a different language design dimension from object-orientation
- one way to use them is similar to polymorphism in functional languages
- In this module, we will build on your knowledge of functional programming
Templates and polymorphism

There are two kinds of templates in C++:

1. Class templates
2. Function templates

These correspond roughly to polymorphic data types and polymorphic functions in functional languages.
Polymorphism in functional languages

# [1; 2; 3];;
- : int list = [1; 2; 3]

type 'a bt = Leaf of 'a
   | Internal of 'a bt * 'a bt;;

# let twice f x = f(f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
Templates: keyword `template`

template<typename T>
struct s {
    ... T ... T ...
};

Then instantiating the template with argument A in s<A> is like
struct sA {
    ... A ... A ...
};

Compare: λ calculus.
Templates: type parameter

template<typename T>
struct S
{
    // members here may depend on type parameter
    T
    T data; // for example a data member
    void f(T); // or a member function
    using t = T; // or making t an alias for
    T
};
Class template example

```cpp
template< typename T>
struct Linked
{
    T head;
    Linked<T>* tail;
};
```

Class template - other keywords

```cpp
template< class T>
class Linked
{
public:
    T head;
    Linked<T>* tail;
};
```
Telling a template what to do

- We can pass types to templates
- We may also configure its behaviour
- Sometimes called “policy”, “callbacks”, etc
- Like higher-order functions
- There are many ways of doing this in C++
- Classes with static member functions
- Function pointers
- Function objects, “functors”
- Lambda expressions (new in C++11)
Function template example

We pass a type $T$ and a class $\text{Ops}$ that provides two operations.

```cpp
template <typename T, typename Ops>
T fold(Linked<T> *p) {
    T acc = Ops::initial();
    while (p) {
        acc = Ops::bin(acc, p->head);
        p = p->tail;
    }
    return acc;
}
```

Note: we pass the class, not an object.
This class provides integer operations:

```cpp
struct IntOps {

    static int initial() { return 0; };

    static int bin(int x, int y) { return x + y;

        }

};
```

In essence, this class is just a pair of functions.
Using the templates on int lists

```c++
int main(int argc, const char* argv[])
{
    auto sumup = fold<int, IntOps>;

    Linked<int> x {3, nullptr};
    Linked<int> y {2, &x};

    std::cout << sumup(&y);

    return 0;
}
```
Another class: string operations

This provides string operations:

```cpp
class StrOps {
public:
    static std::string initial() { return ""; }
    static std::string bin(const std::string x, const std::string y) {
        return x + y;
    }
};
```
Using the template on string lists

```cpp
int main(int argc, const char * argv[])
{
    Linked< std::string > b = { "bar", nullptr ];
    Linked< std::string > a = { "foo ", &b };

    auto sumupstr = fold< std::string, StrOps >;

    std::cout << sumupstr(&a) << "\n";

    return 0;
}
```
Function pointer as template argument

Here we pass a type T, a value of type T and a binary operation on T

```cpp
template< typename T, T init , T (*bin)(T, T) >
T fold2(Linked<T> *p)
{
    T acc = init;
    while (p != nullptr) {
        acc = bin(acc, p->data);
        p = p->next;
    }
    return acc;
}
```
Function pointer as template argument: using it

```cpp
inline int sum(int x, int y)
{
    return x + y;
}

auto sumup2 = fold2<int, 0, sum>;
```
Member functions of template classes

```cpp
template<typename T>
struct Linked {
    T data;
    Linked<T>* next;
    void apply_each(void (*f)(T));
};

template<typename T>
void Linked<T>::apply_each(void (*f)(T))
{
    T* p = data;
    while (p) {
        f(p->data);
        p = p->next;
    }
}
```
Functors/function objects in C++

▶ A function object is an object that can be used like a function.
▶ One of its member functions overloads the function call syntax ()
▶ Such an object can have its own data members.
▶ We can create function object dynamically, unlike C functions.
▶ In functional programming terms, function objects simulate closures.
▶ Function objects are often used with templates.
▶ Objects cannot be template parameters, only function parameters.
Function object example

```cpp
// class for curried addition function
class cadd {
private:
    int n;
public:
    cadd(int n) { this->n = n; }

    int operator()(int m) { return n + m; }
};

int main(int argc, const char * argv[]) {
    // make a new function object in local var
    cadd addfive(5);

    cout << addfive(7);
}

This prints 12.
```
Function objects and templates

template<typename T, typename Op>
T twice(T x, Op f)
{
    return f(f(x));
}

int main(int argc, const char *argv[])
{
    cadd addfive(5); // create function object
    cout << twice<int, cadd>(10, addfive) << endl;
    cout << twice(10, addfive) << endl;
}

This prints 20 twice.
Virtual member function templates?

Suppose we want a worker that computes XML from a tree of ints.

```cpp
template<typename T>
class polyworker {
public:
    virtual T workplus(T, T);
    virtual T workconstant(int);
};
```

See our paper on A Type-theoretic Reconstruction of the Visitor Pattern
Virtual member function templates?

Suppose we want a worker that computes XML from a tree of ints.

```cpp
template<typename T>
class polyworker {
public:
    virtual T workplus(T, T);
    virtual T workconstant(int);
};

class Base {
public:
    template<typename T>
    virtual T employ(polyworker<T>*) = 0;
    // compilation error
};
```

See our paper on "A Type-theoretic Reconstruction of the Visitor Pattern"
Visitor pattern

- The Visitor pattern is one of the classic patterns from the “Gang of Four” OO Patterns book *Design patterns : elements of reusable object-oriented software*

- Related patterns are Composite and Interpreter

- Worker and employ are like visit visitor and accept in GoF

- GoF visitors use local state in the object rather than return types; they have void return types

- The GoF book is from 1995

- There is a lot of emphasis on inheritance

- Since them, C++ has taken on more ideas from functional programming (e.g., lambda, auto)
Some object-oriented patterns

Behavioural patterns are related

- **Composite** = object-oriented idiom to define trees
- **Visitor** = tree walker, internal vs external visitor stateful vs functional
- **Interpreter**, special case of Composite for a grammar of a language
- **Iterator**: internal visitors from lambda expression with reference
Visitor pattern as per Gang of Four 1

class gofvisitor {
public:
    virtual void visitplus(class plus*) = 0;
    virtual void visitconstant(class constant*) = 0;
};

class plus : public gofbase {
    class gofbase *p1, *p2;
public:
    plus(gofbase *p1, gofbase *p2) {
        this->p1 = p1;
        this->p2 = p2;
    }
    void accept(gofvisitor *v) {
        p1->accept(v);
        p2->accept(v);
        v->visitplus(this);
    }
};
Visitor pattern as per Gang of Four 2

class plus : public gofbase {
    gofbase *p1, *p2;
public:
    plus(gofbase *p1, gofbase *p2) {
        this->p1 = p1;
        this->p2 = p2;
    }
    virtual void accept(gofvisitor *v) {
        p1->accept(v);
        p2->accept(v);
        v->visitplus(this);
    }
};
Because the return type is void, the visitor must use internal state to accumulate its result:

```cpp
class countplusvisitor : public gofvisitor {
    int count;

public:
    void visitconstant(class constant *p) {}
    void visitplus(class plus *p) { count++; }
    int getcount() { return count; }
    countplusvisitor() { count = 0; }
};
```
Now suppose we want to do some work only at the leaf nodes on the data, not the internal nodes that determine the shape of the data structure.
We do not need a whole class.
A function to be called on each leaf.
Lambda expressions in C++11

Lambda from the λ-calculus.
Here is a “lambda expression”:

\[ \text{[=]} (\text{int } x) \{ \text{return } x + a; \} \]

It is like

\[ \text{fun } x \rightarrow x + a \]

in OCAMLR or

\[ \lambda x. x + a \]

Variable a needs to be in scope.
Variables can be captured by reference:

\[ \text{[&]} (\text{int } x) \{ \text{return } x + a; \} \]
Lambda expressions in C++11

Lambda expressions in C++ provide two different new features:

1. anonymous functions
2. closures (capturing variables from a surrounding context)

Compare: gcc nested functions.
Lambda expressions in C++11

Lambda from the $\lambda$-calculus. Here is a “lambda expression”:

\[
[=] \ ( \text{int} \ x) \ \{ \ \text{return} \ x + a; \ \}
\]

It is like

\[
\text{fun} \ x \rightarrow x + a
\]

in OCAML or

\[
\lambda x.x + a
\]

Variable $a$ needs to be in scope. Variables can be captured by reference:

\[
[&] \ ( \text{int} \ x) \ \{ \ \text{return} \ x + a; \ \}
\]
Lambda calculus: anonymous functions

$(\lambda a. a + a)(2)$
Lambda calculus: anonymous functions

\((\lambda a. a + a)(2)\)

\(\leadsto 2 + 2\)

\(\leadsto 4\)
Lambda calculus 2

\((\lambda f. f(3))(\lambda a. a + a)\)
(\lambda f. f(3)) (\lambda a. a + a) 
\leadsto (\lambda a. a + a)(3)
Lambda calculus 2

\[(\lambda f. f(3)) \ (\lambda a. a + a) \]
\[\leadsto \ (\lambda a. a + a) \ (3) \]
\[\leadsto \ 3 + 3 \]
\[\leadsto \ 6 \]
Lambda calculus 3

\[(\lambda x. (\lambda y. x + y))(2)(3)\]
Lambda calculus 3

\[(\lambda x. (\lambda y. x + y) ) (2) (3) \]
\[\leadsto (\lambda y. 2 + y) (3) \]
(\lambda x. (\lambda y. x + y))(2)(3)
\rightarrow (\lambda y. 2 + y)(3)
\rightarrow 2 + 3
\rightarrow 5
((\lambda x. (\lambda y. x + y)) (2) (3)) 
\leadsto (\lambda y. 2 + y) (3) 
\leadsto 2 + 3 
\leadsto 5

Real programming languages do not actually copy parameters into the function. Closures are used to implement lambda expressions. In C++, we need to be aware of what happens in memory.
Lambda expressions are implemented by closures

\[
[=] \ (\text{int} \ x) \ { \ \text{return} \ x + x; \ }
\]
Lambda expressions are implemented by closures

\[
[=] \ (\text{int } x) \ {\ return \ x + x; } \]

Closed function: no free variables.
Easy to implement, like a function pointer in C.
Lambda expressions are implemented by closures

\[
\lambda \text{ (int } x \text{) } \{ \text{ return } x + x; \}
\]

Closed function: no free variables.
Easy to implement, like a function pointer in C.

\[
\lambda \text{ (int } x \text{) } \{ \text{ return } x + a; \}
\]
Lambda expressions are implemented by closures

\[
[=] \ (\text{int } x) \ { \ return \ x + x; } \]

Closed function: no free variables.
Easy to implement, like a function pointer in C.

\[
[=] \ (\text{int } x) \ { \ return \ x + a; } \]

Not closed due to \(a\): must build a closure containing a
Lambda expressions are implemented by closures

\[
[=] \ (\text{int } x) \ \{ \ \text{return} \ x + x; \ \}
\]
Closed function: no free variables.
Easy to implement, like a function pointer in C.

\[
[=] \ (\text{int } x) \ \{ \ \text{return} \ x + a; \ \}
\]
Not closed due to \( a \): must build a closure containing \( a \)

\[
[&] \ (\text{int } x) \ \{ \ \text{return} \ x + a; \ \}
\]
Lambda expressions are implemented by closures

\[
[=] \ (\text{int} \ x) \ \{ \ \text{return} \ x + x; \ \}
\]

Closed function: no free variables.
Easy to implement, like a function pointer in C.

\[
[=] \ (\text{int} \ x) \ \{ \ \text{return} \ x + a; \ \}
\]

Not closed due to \(a\): must build a closure containing \(a\)

\[
[&] \ (\text{int} \ x) \ \{ \ \text{return} \ x + a; \ \}
\]

Closure with pointer to \(a\)
The template `std::function` gives general function types. Example: type of functions taking two integers and returning a float is

```
function<float(int, int)>
```

Useful for typing lambda expressions. The same type can be used for C-style function pointers and C++ lambda expressions.
Lambdas as function parameters

```cpp
int twice(function<int(int)> g, int n)
{
    return g(g(n));
}

What does this print?
cout << twice([] (int m) { return m + 1; }, 10) << endl;
```

It prints 12.
int twice(function<int(int)> g, int n) {
    return g(g(n));
}

What does this print?

cout << twice([] (int m) { return m + 1; }, 10) << endl;

It prints 12.
function<int(int)> f(int x)
{
    return [=] (int y) { return x + y; };
}

int main(int argc, const char * argv[])
{
    auto g = f(2);
    cout << g(3) << endl;
    cout << g(4) << endl;
}

What does this print?
Lambdas as function results

```cpp
function<int(int)> f(int x)
{
    return [=](int y) { return x + y; };
}

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

    auto g = f(2);
    cout << g(3) << endl;
    cout << g(4) << endl;

    What does this print?
    It prints 5 and 6.
```
Lambdas and automatic variables

```c++
function<int()> seta()
{
    int a = 11111;
    return [=]() { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};

What does this print:
    cout << geta(seta()) << endl;
```

It prints 11111.
Lambdas and automatic variables

```cpp
function<int()> seta()
{
    int a = 11111;
    return [=] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```cpp
    cout << geta(seta()) << endl;
```

It prints 11111.
Lambdas and automatic variables, by reference

```cpp
function<int()> seta()
{
    int a = 11111;
    return [=]() { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
}

What does this print:

    cout << geta(seta()) << endl;
```

It prints 22222 when I tried it. Undefined behaviour.
Lambdas and automatic variables, by reference

```cpp
function<int()> seta()
{
    int a = 11111;
    return [=] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:
```cpp
cout << geta(seta()) << endl;
```

It prints 22222 when I tried it. Undefined behaviour.
Lambdas and capture by reference

Lambda expressions bring some more functional programming into C++. Lambda are usefully combined with templates and type inference.
Lambda expressions bring some more functional programming into C++. Lambda are usefully combined with templates and type inference. But C is lurking underneath. C++ references are implemented much like C pointers. Capture by reference

```
[&] (...) { ... };
```

requires understanding of object lifetimes, e.g. stack.
Currying in C++

In OCaml:

```ocaml
let curry f x y = f(x, y);;
```

Using C++ templates and lambda, the above becomes:

```cpp
template <typename A, typename B, typename C>
function<function<C(B)>(A)> curry(function<C(A,B)> f) {
  return ([=] (A x) { return [=] (B y) { return f(x, y); });
}
```

What does this print:

```cpp
auto cadd = curry<int, int, int>(add);
cout << cadd(100)(10) << endl;
```

It prints 110.
Currying in C++

In OCaml:

```ocaml
let curry f x y = f(x, y);
```

Using C++ templates and lambda, the above becomes:

```cpp
template <typename A, typename B, typename C>
function <function <C(B)>(A)> curry(function <C(A,B)> f)
{
    return ([=] (A x) { return ([=] (B y)
    { return f(x, y); }); });
}
```

What does this print:

```cpp
auto cadd = curry<int,int,int>(add);
cout << cadd(100)(10) << endl;
```

It prints 110.
Currying in C++

In OCaml:

```ocaml
let curry f x y = f(x, y);;
```

Using C++ templates and lambda, the above becomes:

```cpp
template< typename A, typename B, typename C>
function< function< C(B) >(A)>
curry(function< C(A,B) > f)
{
    return ( [=] ( A x ) { return ( [=] ( B y )
        { return f(x, y); } ); } );
}
```

What does this print:

```cpp
auto cadd = curry< int, int, int >(add);
cout << cadd(100)(10) << endl;
```

It prints 110.
template<typename T>
class bintree {
public:
    virtual void employ(std::function<void(T)>)
        = 0;
};

template<typename T>
class leaf : public bintree<T> {
    T data;
    public :
    leaf(T x) { data = x; }

    void employ(std::function<void(T)> f) {
        f(data);
    }
};
template<typename T>
class internal : public bintree<T> {
    class bintree<T> *left, *right;
public:
    internal(bintree<T> *p1, bintree<T> *p2) {
        left = p1; right = p2;
    }
    void employ(std::function<void(T)> f) {
        left->employ(f);
        right->employ(f);
    }
};
Lambda expression as internal iterator 3

```cpp
int sum1;
void sumfun(int n) { sum1 += n; }

int main(int argc, const char *argv[]) {
    int sum2;
    class bintree<int> *p = new internal<int>(
        new leaf<int>(4), new leaf<int>(3));

    sum1 = 0;
p->employ(sumfun); // employ a C function
    std::cout << sum1 << std::endl;

    sum2 = 0;
p->employ([&] (int x) { sum2 += x; });
// employ a C++ lambda
    std::cout << sum2 << std::endl;
}
```
The above is a good use of lambda expressions.

```
sum2 = 0;
p->employ([&] (int x) { sum2 += x; });
```

A small piece of code made on the fly.
In this case, by reference is safe.
Templates and lambda calculus

\[(\lambda x. (\lambda y. x + y)) (2) (3)\]
Templates and lambda calculus

\[(\lambda x. (\lambda y. x + y))(2)(3)\]

\[\rightsquigarrow (\lambda y. 2 + y)(3)\]
Templates and lambda calculus

$(\lambda x. (\lambda y. x + y)) (2) (3) 
\rightsquigarrow (\lambda y. 2 + y) (3) 
\rightsquigarrow 2 + 3 
\rightsquigarrow 5$
Templates and lambda calculus

\[(\lambda x. (\lambda y. x + y))(2)(3) \]
\[\leadsto (\lambda y. 2 + y)(3) \]
\[\leadsto 2 + 3 \]
\[\leadsto 5 \]

```cpp
template<int x>
int templatecurry (int y) { return x + y; }
...
    cout << templatecurry<2>(3) << endl;
```
Void pointer polymorphism in C

Quicksort from C library:

```c
void qsort (void* base, size_t num, size_t size,
            int (*compar)(void*, void*));
```

Comparison function using void pointers:

```c
int comparefloat (void *p, void *q)
{
    if ( *(float*)p < *(float*)q ) return -1;
    if ( *(float*)p == *(float*)q ) return 0;
    if ( *(float*)p > *(float*)q ) return 1;
}
```
void pointer polymorphism example 1

```c
struct Linked
{
    void* data;
    struct Linked* next;
};
```
void* fold(struct Linked *p, void *initial, 
    void (*bin)(void *x, void *y))
{
    void *acc;
    acc = initial;
    while (p) {
        acc = bin(acc, p->data);
        p = p->next;
    }
    return acc;
}
void *ptr2sum(void *x, void *y)
{
    int *p;
    p = malloc(sizeof(int));
    *p = *(int *)x + *(int *)y;
    return p;
}
Template specialization

- Template *specialization* is like pattern matching in functional languages
- \textit{specialization} \neq \text{instantiation}
- We can pattern-match on types or values
- Templates can be recursive
- One possibility: compute functions at compile-time, e.g. factorial
- More serious: optimize templates for particular type parameters.
- We can write dependent types, like in Agda
Values and types parameterized on values and types

<table>
<thead>
<tr>
<th>↓ parameterized on</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Function</td>
<td>Polymorphic function</td>
</tr>
<tr>
<td>Type</td>
<td>Dependent type</td>
<td>Polymorphic type</td>
</tr>
</tbody>
</table>

Dependent type example:

```cpp
template<int n>
struct s {
    // structure may depend on int parameter
};
```

Arrays are also dependent types, but they decay into pointers in C/C++.
Template specialization example: parsing C types

```cpp
template<typename T>
struct NameofType;

template<typename T, typename S>
struct NameofType<T (*)(S)> {
    static void p() {
        std::cout << "pointer to function returning a ";
        NameofType<T>::p();
        std::cout << " and taking an argument of type ";
        NameofType<S>::p();
    }
};
```
template <>
struct NameofType<int> {  
    static void p()  
    {
        std::cout << "int";
    }
};

template <>
struct NameofType<float> {  
    static void p()  
    {
        std::cout << "float";
    }
};
template<typename T>
struct NameofType<T*> {
    static void p() {
        std::cout << "pointer to ";
        NameofType<T>::p();
    }
};

template<typename T>
struct NameofType<T[]> {
    static void p() {
        std::cout << "array of ";
        NameofType<T>::p();
    }
};
N-dimensional matrix template example

```
template<typename T, int n>
struct ndimMatrix;

template<typename T>
struct ndimMatrix<T, 0>
{
    T m[];
};

template<typename T, int n>
struct ndimMatrix
{
    ndimMatrix<T,n - 1> m[];
};
```
template<template<
typename> class Cont>
struct UseContainer {
    Cont<int> key;
    Cont<float> value;
};
...
UseContainer<vector> uc;