Alice's adventures in Template Land

Whoo are youuu?

Jonathan O'Connor

The Cluster Company GmbH

Email: jonathan.oconnor@theclustercompany.de
or: ninkibah@eircom.net

Twitter: @ninkibah

Mobile: +353 86 824 9736

'Begin at the beginning'

C++ 17 only

A sample Model class

```
struct Person {
  std::string firstName;
  std::string lastName;
  int age;
  int birthYear() const;
  std::string fullName() const;
  void setAge(int age);
};
```

'Why?', said the Caterpillar

In-memory database

```
template<typename Model, typename... Indexes>
struct Table {
  std::map<int, Model> data;
  std::tuple<Indexes...> indices;
  int insert(Model const& model) {
    //...
  }
};
```

'Why?', said the Caterpillar

Table::insert function

```
template<typename Model, typename… Indexes>
struct Table {

  int insert(Model const& model) {
    int id = generateID();
    data.insert(std::pair(id, model));
    for_each(indices, [&](auto& index) {
      index.insert(index.extractKey(model), id);
    });
  }
};
```

'Why?', said the Caterpillar

Templated Index class

```
template<typename Key>
struct Index {
  std::map<Key, int> index_data;

  void insert(Key const& key, int id) {
    index_data.insert(std::pair(key, id));
  }
};
```

'Why?', said the Caterpillar

Every index class needs a static extractKey method

```
class ByNameIndex :
  public Index<std::string>
{
  static std::string extractKey(Person const& p) {
    return p.name;
  }
};
```

'That is not said right,' said the Caterpillar

Boost.MultiIndex container has a nice way of defining indexes using:

```
multi_index_container<Person,
  indexed_by<
    hashed_non_unique<
      member<Person, std::string, &Person::firstName>
    >,
    hashed_non_unique<
      member<Person, int, &Person::age>
    >
  >
>
```

'That is not said right,' said the Caterpillar

Boost.MultiIndex container could be better:

```
multi_index_container<Person,
  indexed_by<
    hashed_non_unique<&Person::firstName>,
    hashed_non_unique<&Person::age>
  >
>
```

'That is not said right,' said the Caterpillar

We'll look at how to declare and define such a class:

```
Index<&Person::firstName> myFirstNameIndex;
```

Pointers to data members

Not your normal pointer!

Declaration:

```
MemberType SomeClass::* pDMem;

int Person::* pPersonIntMember;
```

Pointers to data members

Not your normal pointer!

Declaration:

```
MemberType SomeClass::* pDMem;
```

```
int Person::* pPersonIntMember;
```

Pointer to data member values:

```
pPersonIntMember = &Person::age;
```

Pointers to data members

```
MemberType SomeClass::* pDMem;
int Person::* pPersonIntMember;
pPersonIntMember = &Person::age;
```

Invocation:

```
Person alice;
alice.*pPersonIntMember = 8;
Person* pAlice = &alice;
pAlice->*pPersonIntMember = 9;
```

Pointers to member functions

Declaration:

```
Ret SomeClass::* pMemFunc(argTypes);

int (Person::*)() pIntMemFn =
      &Person::getBirthYear;

void (Person::*)(int) pSetter =
      &Person::setAge;
```

Pointers to member functions

Declaration:

```
int (Person::*)()
  pIntMemFn = &Person::birthYear;
```

Invocation:

```
Person alice;
int birth = alice.*pIntMemFn();
alice.*pSetter(10);
```

C++ Templates - What can be templated?

Function templates

Class templates

Alias templates (C++11)

Variable templates (C++14)

C++ Templates - Function templates

```
template <int N>
int times(int x) {
  return N * x;
}

int fourty_two  = times<6>(7);
```

C++ Templates - Function templates

```
template <typename T>
T* addressOf(T& obj) {
  return &obj;
}


std::string alice{"Alice"};
std::string* pAlice = addressOf(alice);
```

Template type deduction

C++ Templates - Class templates

```
template <typename T, int N>
class MadHattersArray {
  T data[N];
public:
  T operator[](int i) const {
    return data[N - i - 1];
  }
};
MadHattersArray<Dormouse, 1> teapot;
```

C++ Templates - Alias templates

Stephen Dewhurst's favourite feature of C++11

```
template <typename T>
using remove_cv_t = typename remove_cv<T>::type;

static_assert(is_same_v<int*, remove_cv_t<const int*>);
```

C++14 added lots of helper types ending in "_t" to the standard library

C++ Templates - Variable templates

```
template <typename T>
constexpr T pi = 3.1415926535;

template <typename T>
auto tau = pi<T> * 2;
constexpr auto piOnWater = pi<float>;
```

C++ Template Parameters - typename parameters

```
template <typename T>
struct ThatsNotWhatIMeanByPair {
  T one;
  T two;
};
```

C++ Template Parameters - Non-type Parameters

```
template <int N>
constexpr const int Twice{N * 2};
```

Allowed types:

- {signed | unsigned} char, short, int, long, long long
- Fixed size arrays of above
- function pointers
- pointers to members

Restrictions: No strings :-(

C++ Template Parameters - Non-type Parameters

```
template <int N>
constexpr const int Twice{N * 2};
```

Allowed types:

- {signed | unsigned} char, short, int, long, long long
- Fixed size arrays of above
- function pointers
- pointers to members

Restrictions: No strings :-(yet :-)

C++ Template Parameters - template template parameters

Go talk to Nikolai Josuttis!

Down the rabbit hole

"Would you tell me, please, which way I ought to go from here?"

```
template<auto pmember>
struct Index {
  using Key = ???;
  using Model = ???;
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};
```

Declaring our template: V1

```
template<auto pmember, typename Key, typename Model>
struct Index {
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};

using AgeIndex = Index<&Person::age, int, Person>;
```

Declaring our template: V1

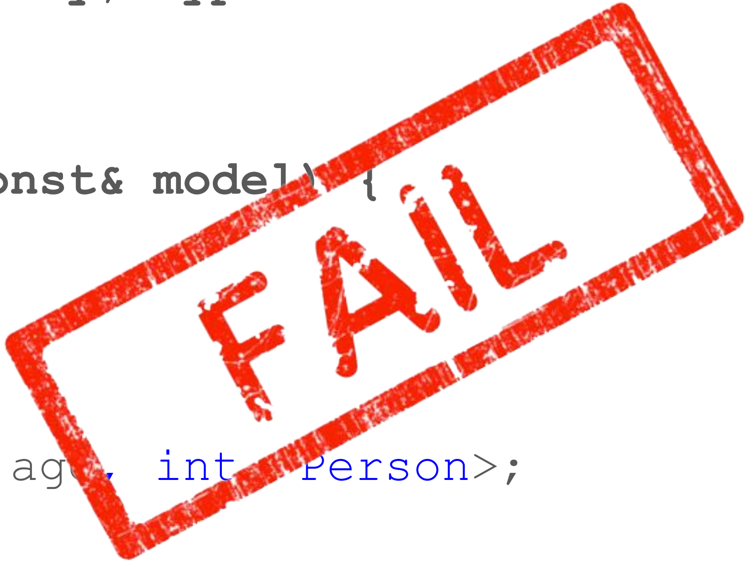
```
template<auto pmember, typename Key, typename Model>
struct Index {
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};

using AgeIndex = Index<&Person::age, int, Person>;
```

Declaring our template: V1

```
template<auto pmember, typename Key, typename Model>
struct Index {
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};

using AgeIndex = Index<&Person::age, int, Person>;
```



Declaring our template: V2

```
template<Key Model::*pmember, typename Key,
  typename Model>
struct Index {
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};
```

Declaring our template: V2

```
template<Key Model::*pmember, typename Key,
  typename Model>
struct Index {
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};
```

Invalid syntax. Doesn't compile :-(

`Key` and `Model` are only declared after they are used

Declaring our template: V2

```
template<Key Model::*pmember, typename Key,
  typename Model>
struct Index {
  std::map<Key, int> data;
  static Key extractKey(Model const& model) {
    return model.*pmember;
  }
};
```

Invalid syntax. Doesn't compile :-(

`Key` and `Model` are only declared after they are used

Declaring our template: V3

Our template must take a single non-type parameter

We want the compiler to deduce 2 types to our pointer to member.

How?

With a templated function!

Why?

Because the compiler does template argument deduction!

Declaring our template: V3

```
template<typename Key, typename Model>
Key determineKeyType(Key Model::* pMember);
```

Compiler argument deduction

```
template<typename Key, typename Model>
Key determineKeyType(Key Model::* pMember);


determineKeyType(&Person::age);
```

Compiler deduces `Person` as `Model`

and `int` as `Key`

Determining the type of an expression

```
template<typename Key, typename Model>
Key determineKeyType(Key Model::* pMember);
```

```
template<auto pMember>
using Key_t = decltype(determineKeyType(pMember));
```

decltype gives the type of the expression

Example: same_v`<Key_t<&Person::age>, int>`

Why no implementation?

```
template<typename Key, typename Model>
Key determineKeyType(Key Model::* pMember);


template<auto pMember>
using Key_t = decltype(determineKeyType(pMember));
```

`decltype` pretends to evaluate its argument, but doesn't!

It returns the type the result would be, if it was evaluated.

Ugly syntax

```
template<typename Key, typename Model>
Key determineKeyType(Key Model::* pMember);
```

Alias template to avoid ugly syntax

```
template<typename Key, typename Model>
using P2Mem = Key Model::*;

template<typename Key, typename Model>
Key determineKeyType(P2Mem<Key, Model> pMember);
```

Aliases help to give better names to types

Extracting the Model type similar to Key

```
template<typename Key, typename Model>
Model determineModelType(P2Mem<Key, Model> pMember);

template<auto pMember>
using Model_t = decltype(determineModelType(pMember));
```

Almost there

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;
};
```

How to define extractKey function

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return model ??? pMember;
  }
};
```

How to define extractKey function

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return model.*pMember;
  }
};
```

Indexes on data members

```
using AgeIndex = Index<&Person::age>;

using FirstNameIndex = Index<&Person::firstName>;
```

Indexes on pointers to member functions?

```
using FullNameIndex = Index<&Person::fullName>;
```

What does the compiler say?

```
using FullNameIndex = Index<&Person::fullName>;
```

```
alice.cpp: In instantiation of ‘struct
Index<&Person::fullName>’:
alice.cpp:160:62:   required from here
alice.cpp:132:14: error: function returning a function
   static Key extractKey(Model const& model) {
              ^~~~~~~~~~
alice.cpp: In function ‘int main()’:
alice.cpp:160:64: error: ‘extractKey’ is not a member of
‘FullNameIndex {aka Index<&Person::fullName>}’
```

Indexes on pointers to member functions?

```
using FullNameIndex = Index<&Person::fullName>;
```

Doesn't work because

```
return model.*pMember;
```

does not invoke the member function.

Indexes on pointers to function members?

```
using FullNameIndex = Index<&Person::fullName>;
```

Doesn't work because

```
return model.*pMember;
```

does not invoke the member function.

Indexes on pointers to function members?

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return model.*pMember; // Compiler complains here
  }
};
```

extractKey for pointer to function members

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return model.*pMember();
  }
};
```

extractKey for pointer to function members

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return model.*pMember();
  }
};
```

But now we break pointers to data members

std::invoke(F&& f, Args…&& args)

Works on:

Ordinary functions

string foo(int, string const&);

```
string s = std::invoke(foo, 1, "hi")
string s = foo(1, "hi")
```

std::invoke(F&& f, Args…&& args)

Works on:

Ordinary functions

Function objects

```
std::invoke(f, 1, "hello")
 f(1, "hello")
```

std::invoke(F&& f, Args…&& args)

Works on:

Ordinary functions

Function objects

Pointers to function members

std::`invoke(&Person::setAge, alice, 8)`

`alice.setAge(8)`

std::invoke(F&& f, Args…&& args)

Works on:

Ordinary functions

Function objects

Pointers to function members

Pointers to data members

std::`invoke(&Person::age, alice)`

`alice.age`

extractKey using std::invoke

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return std::invoke(pMember, model);
  }
};
```

extractKey using std::invoke

```
template<auto pMember>
struct Index {
  using Key = Key_t<pMember>;
  using Model = Model_t<pMember>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    return std::invoke(pMember, model);
  }
};
```

Extracting types for pointers to function members

```
template<typename Key, typename Model>
using P2MemFn = Key (Model::*)();

template<typename Key, typename Model>
Model determineModelType(P2Mem<Key, Model> pMember);

template<typename Key, typename Model>
Model determineModelType(P2MemFn<Key, Model> pMember);
```

Extracting types for pointers to function members

```
template<typename Key, typename Model>
using P2MemFn = Key Model::*();

template<typename Key, typename Model>
Model determineModelType(P2Mem<Key, Model> pMember);

template<typename Key, typename Model>
Model determineModelType(P2MemFn<Key, Model> pMember);
```

Fails, as C++ does not allow overloaded template functions.

https://stackoverflow.com/questions/47060448

Barry advised me to do the following:

```
template<typename>
struct TypeExtractor;
```

Forward declare a templated class

Why?

Future partial specializations

https://stackoverflow.com/questions/47060448

Specialize the template so we can match on Key and Model.

```
template<typename Key, typename Model>
struct TypeExtractor<Key Model::*> {
  using Model_t = Model;
  using Key_t = ???;
};
```

https://stackoverflow.com/questions/47060448

Specialize the template so we can match on Key and Model.

```
template<typename Key, typename Model>
struct TypeExtractor<Key Model::*> {
  using Model_t = Model;
  using Key_t = Key;
};
```

https://stackoverflow.com/questions/47060448

Specialize the template so we can match on Key and Model.

```
template<typename Key, typename Model>
struct TypeExtractor<Key Model::*> {
  using Model_t = Model;
  using Key_t = Key;
};
```

Works fine for pointers to data members

Fails with pointers to member functions

https://stackoverflow.com/questions/47060448

Specialize the template so we can match on Key and Model.

```
template<typename Key, typename Model>
struct TypeExtractor<Key Model::*> {
  using Model_t = Model;
  using Key_t = std::conditional_t<
    std::is_function_v<Key>,
    std::invoke_result_t<Key Model::*, Model>,
    Key>;
};
```

https://stackoverflow.com/questions/47060448

Specialize the template so we can match on Key and Model.

```
template<typename Key, typename Model>
struct TypeExtractor<Key Model::*> {
  using Model_t = Model;
  using Key_t = std::conditional_t<
    std::is_function_v<Key>,
      std::invoke_result_t<Key Model::*, Model>,
      Key>;
};
```

https://stackoverflow.com/questions/47060448

Daniel Frey suggested to use decay_t to simplify all of this.

```
template<typename Key, typename Model>
struct TypeExtractor<Key Model::*> {
  using Model_t = Model;
  using Key_t = std::decay_t<
    std::invoke_result_t<Key Model::*, Model>
  >;
};
```

Alias templates

```
template<auto pMember>
using Key_t = typename
  TypeExtractor<decltype(pMember)>::Key_t;
```

Alias templates

```
template<auto pMember>
using Key_t = typename
  TypeExtractor<decltype(pMember)>::Key_t;

template<auto pMember>
using Model_t = typename
  TypeExtractor<decltype(pMember)>::Model_t;
```

Curiouser and curiouser

Why decltype(pMember)?

```
template<auto pMember>
using Key_t = typename
  TypeExtractor<decltype(pMember)>::Key_t;
```

But in the template specialization we have a non-type parameter,

```
struct TypeExtractor<Key Model::*> {
```

Using ordinary functions

```
int yearsToRetirement(Person const& person) {
  return 65 - person.age;
}
```

Can we write `Index<yearsToRetirement>`?

Using ordinary functions

```
int yearsToRetirement(Person const& person) {
  return 65 - person.age;
}
```

Can we write `Index<yearsToRetirement>`?

Yes, by adding another specialization of TypeExtractor

Using ordinary functions

```
template <typename Key, typename Model>
struct TypeExtractor<Key (*)(Model const&)> {
  using Model_t = Model;
  using Key_t = Key;
};
```

Using ordinary functions

```
int yearsToRetirement(Person const& person) {
  return 65 - person.age;
}

Index<yearsToRetirement> myPensionIndex;
```

Summary

We can create Indexes using

- pointers to data members: `Index<&Person::age>`
- pointers to member functions: `Index<&Person::birthYear>`
- standalone functions: `Index<yearsToRetirement>`

Summary

We can create Indexes using

- pointers to data members: `Index<&Person::age>`
- pointers to member functions: `Index<&Person::birthYear>`
- standalone functions: `Index<yearsToRetirement>`

We can go further! But maybe not today :-(

Tools

`godbolt`

`cppinsights.io`

`Metashell`

Questions

Jonathan O'Connor

The Cluster Company GmbH

Email: jonathan.oconnor@theclustercompany.de
or: ninkibah@eircom.net

Twitter: @ninkibah

Mobile: +353 86 824 9736

Code: https://github.com/ninkibah/talks

Down the rabbit hole

Indexes with multiple fields

```
Index<&Person::age, &Person::fullname>
```

Indexes with multiple fields

```
Index<&Person::age, &Person::fullname>
```

```
struct AgeAndFullNameIndex :
public Index<std::tuple<int, std::string>, Person>
{
  static auto extractKey(Person const& p) {
    return std::make_tuple(p.age, p.fullName());
  }
};
```

Variadic templates

Making sure we have a common POD type

```
// forward declare our variadic template
template<auto...>
struct model_type;
```

The implementation pattern used gcc's type_traits std::__and and std::__or

Specialize for no arguments

```
template<> // Specific specialization
struct model_type<> {
  static constexpr bool value = true;
  using type = void;
};
```

value is true if all arguments have the same Model type.

Specialize for one argument

```
template<auto V1>
struct model_type<V1> {
  static constexpr bool value = true;
  using type = Model_t<V1>;
};
```

Specialize for two arguments

```
template<auto V1, auto V2>
struct model_type<V1, V2> {
  static constexpr bool value =
    std::is_same_v<Model_t<V1>, Model_t<V2>>;
  using type = Model_t<V1>;
};
```

Specialize for three or more arguments

```
template<auto V1, auto V2, auto V3, auto...Vn>
struct pod_type<V1, V2, V3, Vn...> {
  static constexpr bool value =
    std::is_same_v<Model_t<V1>, Model_t<V2>> &&
    model_type<V2, V3, Vn...>::value;

  using type = Model_t<V1>;
};
```

Generating the Key type

```
template<auto...>
struct key_type;
```

This uses the same pattern used by model_type

Specialize for one argument

```
template<auto V1>
struct key_type<V1> {
  using type = Key_t<V1>;
};
```

Specialize for many arguments

```
template<auto V1, auto...Vn>
struct key_type<V1, Vn...> {
  using type =
    std::tuple<Key_t<V1>, Key_t<Vn>...>;
};
```

Figuring out where to put the ...

Instead of Vn… imagine V2, V3, V4

Write the code you need for them:

```
std::tuple<Key_t<V1>, Key_t<V2>, Key_t<V3>, Key_t<V4>>;
```

Now look for the common part:

Figuring out where to put the ...

Instead of Vn… imagine V2, V3, V4

Write the code you need for them:

```
std::tuple<Key_t<V1>, Key_t<V2>, Key_t<V3>, Key_t<V4>>;
```

Now look for the common part:

```
std::tuple<Key_t<V1>, Key_t<V2>, Key_t<V3>, Key_t<V4>>;
```

Figuring out where to put the ...

Instead of Vn… imagine V2, V3, V4

Write the code you need for them:

```
std::tuple<Key_t<V1>, Key_t<V2>, Key_t<V3>, Key_t<V4>>;
```

Now look for the common part:

```
std::tuple<Key_t<V1>, Key_t<V2>, Key_t<V3>, Key_t<V4>>;
```

Replace the first common part with an expression using Vn, and append ...

```
std::tuple<Key_t<V1>, Key_t<Vn>...>;
```

alias templates for model_type_t and key_type_t

```
template<auto… Values>
using model_type_t =
    typename model_type<Values...>::type;

template<auto… Values>
using key_type_t = typename key_type<Values...>::type;
```

I'm late, I'm late

```
template<auto… Extractors>
struct Index {
  using Model = model_type_t<Extractors…>;
  using Key = key_type_t<Extractors…>;
  std::map<Key, int> data;

  static Key extractKey(Model const& model) {
    // ???
  }
};
```

I'm late, I'm late

sizeof…

fold expressions

```
static auto extractKey(Model const& model) {
  if constexpr (sizeof...(Extractors) == 1)
    return std::invoke(Extractors..., model);

  else

    return std::make_tuple(
      std::invoke(Extractors, model)...);
}
```

Where next?

Support for parent classes of a model class

```
struct Person;

struct Employee : public Person {
  int employeeNr;
};

Index<&Person::fullName, &Employee::employeeNr>

Index for table of Employees.
```

In Memoriam: John Carolan

The Cheshire Cat's smile pattern

AKA PIMPL

First used by John circa 1987