C++ Templates

Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info
@Andreas__Fertig

fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed

https://AndreasFertig.Info
post@AndreasFertig.Info

What is generic programming

- Generic programming is a method to implement algorithms and data structures in the most general sensible way.
- Algorithms are written in terms of types to-be-specified-later.
- The term *generic programming* was originally coined by David Musser and Alexander Stepanov [1].
- Generic programming helps us to reduce redundancy and programming effort, while it increases reusability and flexibility.

Templates

- Templates are a kind of pattern for the compiler.
- We can *instantiate* templates with different types or values.
  - Each instantiation for a new type or value results in additional code, the fill-in template is generated with the given template argument.
- Templates reduce a lot of writers' work. We do not have to implement functions multiple times just because it's a slightly different type.
- There are different types of templates:
  - Function-templates
  - Class-templates
  - Variable-templates (since C++14).
- Templates are always initiated by the keyword `template`.

https://AndreasFertig.Info
post@AndreasFertig.Info

The different kinds of template parameters

- There are three different types of template parameters:

Type parameter whenever we use a concrete type, e.g., `int`, `char`, or even a class. Type parameters are the most common type.

none-type parameter typically values like 3. Excluded are floating-point numbers and strings (C-arrays). Since C++20, they work as well, with minor limitations.

template-template parameter is required if we pass a template as a parameter to a template.

The template parts applied

Here is a piece of code which declares and uses a template.

```
(A) The template-head
template<typename T,        (B) A type parameter
         size_t N>          (C) A NTTP
constexpr auto
Size(const T (&)[N])        (D) Expect an array of type T and size N as parameter
{
  return N;   (E) Use a NTTP
}

void Main()
{
  char buffer[16]{};

  // Prefer a range-based for-loop!
  for(int i = 0; i < Size(buffer); ++i) {
    // do something with buffer
  }
}
```


https://AndreasFertig.Info
post@AndreasFertig.Info

Function templates

- With a function template, we can implement uniform functionality for different types once and let the compiler do the filling.
- The parameters are types and values (with restrictions):
  - Types are initiated by `typename` or `class`.
  - The name of the template parameter follows.
  - The name usually has `T` for the first parameter and `U` for the second parameter.
  - Within the function, we use this name instead of a specific type.

```cpp
template<typename T>
T min(const T& a, const T& b)
{
  return (a < b) ? a : b;
}

void Main()
{
  const int a = 2;
  const int b = 1;

  printf("%d\n", min(a, b));
}
```


Function templates: Instantiation

- *instantiation* is when the compiler replaces a template, with the concrete arguments.
  - C ++ Insights helps to visualize the result.
  - The example is the output of C ++ Insights.
- The compiler instantiates the function template automatically:
  - because of the arguments
  - if he can derive these.
- If the compiler can not determine the arguments itself, they must be specified explicitly.

```cpp
template<typename T>
T min(const T& a, const T& b)
{
  return (a < b) ? a : b;
}

/* First instantiated from: insights.cpp:13 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int min<int>(const int& a, const int& b)
{
  return (a < b) ? a : b;
}
#endif

void Main()
{
  const int a = 2;
  const int b = 1;
  printf("%d\n", min(a, b));
}
```

Function templates: Specialization

- *specialization* is the procedure that provides a concrete implementation for an argument combination of a function template.

```
template<typename T>
bool equal(const T& a, const T& b)
{
  return a == b;
}

template<>
bool equal(const double& a, const double& b)
{
  return std::abs(a - b) < 0.00001;
}

void Main()
{
  int a = 2;
  int b = 1;

  printf("%d\n", equal(a, b));

  double d = 3.0;
  double f = 4.0;

  printf("%d\n", equal(d, f));
}
```


Class templates

- As with a function template, a class template is introduced by the keyword `template`.
  - Most rules of function templates also apply to class templates.
  - Methods can be implemented inside the class or outside.
  - Methods implemented outside the class require the template-head before the method as in the class.
  - Each instantiation of a class creates a new type.
- (A) represents a *type parameter*.
- (B) represents a *non-type template parameter (NTTP)*.

```
template<typename T,      (A) A type parameter
         size_t SIZE>     (B) A NTTP
struct Array {
  T*       data();
  const T* data() const
  {
    return std::addressof(mData[0]);
  }
  constexpr size_t size() const { return SIZE; }
  T*                begin() { return data(); }
  T*                end() { return data() + size(); }
  T& operator[](size_t idx) { return mData[idx]; }

  T mData[SIZE];
};

template<typename T, size_t SIZE>
T* Array<T, SIZE>::data()
{
  return std::addressof(mData[0]);
}

void Main()
{
  Array<int, 2>    ai{3, 5};
  Array<double, 2> ad{2.0};
}
```


https://AndreasFertig.Info
post@AndreasFertig.Info

Class templates: Instantiation

- Again, analogous to the function template with one important exception:
  - A class template can not automatically derive its arguments.
  - Each template argument *must* be specified explicitly.
  - Exception: C++17. Here we have *class template argument deduction (CTAD)*.

Class templates: Method templates

- Methods of a class template can themselves be a template of their own. This is called a method template.
  - A method template can be defined inside or outside a class.
  - The copy constructor and destructor can not be templates.

```cpp
template<typename T>
class Foo {
public:
  Foo(const T& x)
  : mX{x}
  {}

  template<typename U>
  Foo<T>& operator=(const U& u)
  {
    mX = static_cast<T>(u);
    return *this;
  }

private:
  T mX;
};

void Main()
{
  Foo<int> fi{3};
  fi = 2.5;
}
```

Class templates: Inheritance

- Class templates or classes can inherit from each other in any combination.
- When deriving a class template, there is a restriction:
  - In the derived class, methods and attributes of the base class are not automatically available.
- There are three possible solutions:
  - To qualify the method call by the `this` pointer.
  - Make the name known by `using Base<T>::func`.
  - Call the method of the base class directly.

```cpp
template<typename T>
class Foo {
public:
  void Func() {}
};

template<typename T>
class Bar : public Foo<T> {
public:
  void BarFunc()
  {
    // Func();
    this->Func();
    Foo<T>::Func();
  }
};

void Main()
{
  Bar<int> b{};
  b.BarFunc();
}
```


Alias templates

- Alias templates allow you to create synonyms for templates.
  - This allows a partial specialization of templates.
  - Alias templates themselves can not be further specialized.

```cpp
#include <array>

template<size_t N>
using CharArray = std::array<char, N>;

void Main()
{
  CharArray<24> ar;
}
```

Alias templates

- Alias templates allow you to create synonyms for templates.
  - This allows a partial specialization of templates.
  - Alias templates themselves can not be further specialized.
  - Can help to abstract small type differences for different products.

```
#include <array>

template<size_t N>
using CharArray =
#ifdef PRODUCT_A
  Array<char, N>;
#else
  std::array<char, N>;
#endif

void Main()
{
  CharArray<24> ar;
}
```


Guidelines for efficient use of templates

- Templates generate code for us.
  - It is as if we copy and paste our implementation and changes types or values.
  - Depending on the compiler and optimizer, this can result in a larger binary.
  - Sometimes we overlook this, and people then refer to it as *code bloat*.
- This is in our control!

https://AndreasFertig.Info
post@AndreasFertig.Info

Guidelines for efficient use of templates - An example

- The pattern of passing value and length is:
  - Typical C API.
  - Error-prone.
  - More to write & read.

```
bool Send(const char* data, size_t size)
{
  if(!data) { return false; }

  return write(data, size);
}

void Read(char* data, size_t size)
{
  if(!data) { return; }

  // fill buffer with data
}

void Main()
{
  char buffer[1’024]{};

  Read(buffer, sizeof(buffer));
  Send(buffer, sizeof(buffer));

  char buffer2[2’048]{};

  Read(buffer, sizeof(buffer2));
  Send(buffer, sizeof(buffer2));
}
```


Guidelines for efficient use of templates - An example

- The pattern of passing value and length is:
  - Typical C API.
  - Error-prone.
  - More to write & read.
- It is better with `std::array`:
  - Disadvantage here: The size must always be the same.

```
bool Send(const std::array<char, 1’024>& data)
{
  return write(data.data(), data.size());
}

void Read(std::array<char, 1’024>& data)
{
  // fill buffer with data
}

void Main()
{
  std::array<char, 1’024> buffer{};

  Read(buffer);
  Send(buffer);

  std::array<char, 2’048> buffer2{};

  //  Read(buffer2);
  //  Send(buffer2);
}
```

Guidelines for efficient use of templates - An example

- The pattern of passing value and length is:
  - Typical C API.
  - Error-prone.
  - More to write & read.
- It is better with `std::array`:
  - Disadvantage here: The size must always be the same.
  - Alternative: Make Read / Send a template with a NTTP for the size.

```cpp
template<size_t N>
bool Send(const std::array<char, N>& data)
{
  return write(data.data(), data.size());
}

template<size_t N>
void Read(std::array<char, N>& data)
{
  // fill buffer with data
}

void Main()
{
  std::array<char, 1'024> buffer{};

  Read(buffer);
  Send(buffer);

  std::array<char, 2'048> buffer2{};

  Read(buffer2);
  Send(buffer2);
}
```


Guidelines for efficient use of templates - An example

- The pattern of passing value and length is:
  - Typical C API.
  - Error-prone.
  - More to write & read.
- It is better with `std::array`:
  - Disadvantage here: The size must always be the same.
  - Alternative: Make Read / Send a template with a NTTP for the size.
  - **Disadvantage:** Code-bloat danger!

```cpp
template<size_t N>
bool Send(const std::array<char, N>& data)
{
  return write(data.data(), data.size());
}

template<size_t N>
void Read(std::array<char, N>& data)
{
  // fill buffer with data
}

void Main()
{
  std::array<char, 1'024> buffer{};

  Read(buffer);
  Send(buffer);

  std::array<char, 2'048> buffer2{};

  Read(buffer2);
  Send(buffer2);
}
```


https://AndreasFertig.Info
post@AndreasFertig.Info

Guidelines for efficient use of templates - An example

- The pattern of passing value and length is:
  - Typical C API.
  - Error-prone.
  - More to write & read.
- It is better with `std::array`:
  - Disadvantage here: The size must always be the same.
  - Alternative: Make Read / Send a template with a NTTP for the size.
  - **Disadvantage:** Code-bloat danger!
- Better: Abstract the size away.
- For example span
  - Can hold both C-Array and `std::array`.
  - Of course, range-based forready.
  - Cleaned up the code safely and with little overhead.

```cpp
bool Send(const span<char>& data)
{
  return write(data.data(), data.size());
}

void Read(span<char> data)
{
  int i = 1;
  // fill buffer with data
  for(auto& c : data) {
    c = i;
    ++i;
  }
}

void Main()
{
  std::array<char, 1’024> buffer{};

  Read(buffer);
  Send(buffer);

  char buffer2[2’048]{};

  Read(buffer2);
  Send(buffer2);
}
```


Guidelines for efficient use of templates - An example

- The pattern of passing value and length is:
  - Typical C API.
  - Error-prone.
  - More to write & read.
- It is better with `std::array`:
  - Disadvantage here: The size must always be the same.
  - Alternative: Make Read / Send a template with a NTTP for the size.
  - **Disadvantage:** Code-bloat danger!
- Better: Abstract the size away.
- For example span
  - Can hold both C-Array and `std::array`.
  - Of course, range-based forready.
  - Cleaned up the code safely and with little overhead.
  - C++20 Single Header Version of span: [2].

```cpp
template<typename T>
class span {
public:
  constexpr span() = default;
  constexpr span(T* start, const size_t len)
  : data_{start}, length{len} { }

  template<size_t N>
  constexpr span(T (&arr)[N])
  : span(arr, N) { }

  template<size_t N>
  constexpr span(const T (&arr)[N])
  : span(arr, N) { }

  template<size_t N, class AT = std::remove_const_t<T>>
  constexpr span(std::array<AT, N>& arr)
  : span(arr.data(), arr.size()) { }

  constexpr size_t size() const { return length; }
  T*               data() const { return data_; }
  bool empty() const { return nullptr != data_; }

  T* begin() const { return data_; }
  T* end() const   { return data_ + length; }

private:
  T*     data_;
  size_t length;
};
```


https://AndreasFertig.Info
post@AndreasFertig.Info

Guidelines for efficient use of templates

- Guidelines for class templates:
  - Move code, which stays the same for all instantiations into a base class.
  - Weight, if storing an additional type/value, is better than passing it as a template parameter.
- Guidelines for function templates:
  - Check if you can use them as API only but forward the actual work to a non-template function.
    - With that, you get the safety and simplicity for your users and only internally use the unsafe version.

Thinking in types

- We usually think in values as they are computed during run-time.
- Types are known at compile-time.
  - We can do checks and modifications to types at compile-time.

https://AndreasFertig.Info
post@AndreasFertig.Info

Thinking in types

- We usually think in values as they are computed during run-time.
- Types are known at compile-time.
  - We can do checks and modifications to types at compile-time.
- Let's limit `Array` not to be a pointer.
  - There is a type trait `is_pointer`, which does the job.
  - The code presented is a simplification of what is in `type_traits`.

```
(A) Helper to store a value at compile-time
template<class T, T v>
struct integral_constant {
  static constexpr T value = v;
};

(B) Aliases for clean TMP
using true_type  = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

(C) Base is_pointer template
template<class T>
struct is_pointer : false_type {};

(D) is_pointer specialization for T*
template<class T>
struct is_pointer<T*> : true_type {};

(E) Test it
static_assert(is_pointer<int*>::value);
static_assert(not is_pointer<int>::value);
```


Thinking in types

- We usually think in values as they are computed during run-time.
- Types are known at compile-time.
  - We can do checks and modifications to types at compile-time.
- Let's limit `Array` not to be a pointer.
  - There is a type trait `is_pointer`, which does the job.
  - The code presented is a simplification of what is in `type_traits`.

```
template<typename T, size_t SIZE>
struct Array {
  (A) Added a check that T is not a pointer
  static_assert(not std::is_pointer<T>::value);

  T*       data() { return std::addressof(mData[0]); }
  const T* data() const
  {
    return std::addressof(mData[0]);
  }
  constexpr size_t size() const { return SIZE; }
  T*               begin() { return data(); }
  T*               end() { return data() + size(); }
  T& operator[](size_t idx) { return mData[idx]; }

  T mData[SIZE];
};

void Main()
{
  int x{22};

  // Array<int*, 2> invalid{&x}; (B) This will not compile
  Array<int, 2> valid{x};
}
```


https://AndreasFertig.Info
post@AndreasFertig.Info

Thinking in types

- We usually think in values as they are computed during run-time.
- Types are known at compile-time.
  - We can do checks and modifications to types at compile-time.
- Let's limit `Array` not to be a pointer.
  - There is a type trait `is_pointer`, which does the job.
  - The code presented is a simplification of what is in `type_traits`.
- With C++20's Concepts.

```
template<typename T, size_t SIZE>
(A) C++20 require T to not be a pointer
requires(not std::is_pointer_v<T>) struct Array {
  T*        data() { return std::addressof(mData[0]); }
  const T* data() const
  {
    return std::addressof(mData[0]);
  }
  constexpr size_t size() const { return SIZE; }
  T*                begin() { return data(); }
  T*                end() { return data() + size(); }
  T& operator[](size_t idx) { return mData[idx]; }

  T mData[SIZE];
};

void Main()
{
  int x{22};

  // Array<int*, 2> invalid{&x};
  Array<int, 2> valid{x};
}
```


constexpr if

C++17

- An extension of `constexpr`.
  - This `if` and all branches are evaluated at compile time.
  - Only the branch which yields `true` is preserved.
- We can use it with `is_pointer` to dereference pointers and return their value and otherwise just return it.

```
template<typename T>
auto getValue(T t)
{
  (A) constexpr if for compile-time dispatching
  if constexpr(std::is_pointer_v<T>) {
    assert(nullptr != t);
    return *t;
  } else {
    return t;
  }
}

void Main()
{
  int  i  = 4;
  int* ip = &i;

  (B) All calls will result in plain int as type
  auto iv  = getValue(i);
  auto ipv = getValue(ip);
  auto itv = getValue(43);
}
```


https://AndreasFertig.Info
post@AndreasFertig.Info

C++17

constexpr if

- An extension of constexpr.
  - This if and all branches are evaluated at compile time.
  - Only the branch which yields true is preserved.
- We can use it with is_pointer to dereference pointers and return their value and otherwise just return it.
- Or with is_convertible_v to convert everything into a std::string.

```
template<typename T>
std::string str(T t)
{
  if constexpr(std::is_convertible_v<T, std::string>) {
    return t;
  } else {
    return std::to_string(t);
  }
}

void Main()
{
  auto s = str(std::string{"42"});
  auto i = str(42);
}
```


Variadic templates: Parameter pack

- Syntax:
  - (A) typename|class... Ts generates a type template parameter pack with optional name.
  - (B) Args... ts a function argument parameter pack with optional name.
  - (C) sizeof...(ts) determine the number of arguments passed.
  - (D) ts... in the body of a function to unpack the arguments.

```
template<typename T,
         typename... Ts   (A) Variadic template
        >
constexpr auto
min(const T& a,
    const T& b,
    const Ts&... ts)   (B) Parameter pack
{
  const auto m = a < b ? a : b;
  if constexpr(sizeof...(
                 ts)   (C) size of a pack
               > 0) {
    return min(m, ts...);   (D) Expand the pack
  }

  return m;
}

static_assert(min(3, 2, 3, 4, 5) == 2);
static_assert(min(3, 2) == 2);
```

Variadic templates

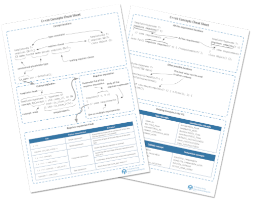
- With C++11, there are *variadic templates*:
  - Variadic templates are templates that take any number of parameters.
  - Already known by variadic macros or variadic functions.

```
A Helper functions to convert everything into a std::string
auto Normalize(const std::string& t) { return t; }
auto Normalize(const QString& t) { return t.toStdString(); }
auto Normalize(const char* t) { return std::string{t}; }

B Catch all others and apply to_string
template<class T> auto Normalize(const T& t) { return std::to_string(t); }

template<typename T, typename... Ts> auto _StrCat(std::string& ret, const T& targ, const Ts&... args)
{
  ret += Normalize(targ);
  if constexpr(sizeof...(args) > 0) {
    _StrCat(ret, args...);  C Do, as long as the pack has elements
  }
}

template<typename T, typename... Ts> auto StrCat(const T& targ, const Ts&... args)
{
  std::string ret{Normalize(targ)};

  _StrCat(ret, args...);  D Start the recursion to expand the pack

  return ret;
}
```


}

I am Fertig.

C++20 Concepts Cheat Sheet



fertig.to/subscribe

https://AndreasFertig.Info
post@AndreasFertig.Info

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++ 10.2.0
  - clang version 10.0.0 (https://github.com/llvm/llvm-project.git d32170dbd5b0d54436537b6b75beaf44324e0c28)

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

References

[1] Musser D. R. and Stepanov A. A., "Generic programming", in *Symbolic and Algebraic Computation*, Gianni P., Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 13–25. http://stepanovpapers.com/genprog.pdf

[2] Moene M., "span lite - A single-file header-only version of a C++20-like span for C++98, C++11 and later". https://github.com/martinmoene/span-lite

Images:
35: Franziska Panter

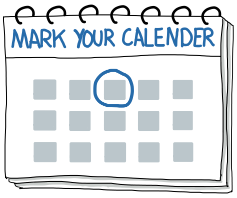
https://AndreasFertig.Info
post@AndreasFertig.Info

Upcoming Events

Training Classes

- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, September 13 - 17
- *Programmieren mit C++20*, Andreas Fertig, September 27 - 29
- *C++1x für eingebettete Systeme*, QA Systems, October 14 - 15

For my upcoming talks you can check https://andreasfertig.info/talks/.
For my courses you can check https://andreasfertig.info/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.info/newsletter/.



About Andreas Fertig


Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 20.
Andreas is involved in the C++ standardization committee, in which the new standards are developed. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus to understand constructs even better.
Before working as a trainer and consultant, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.

https://AndreasFertig.Info
post@AndreasFertig.Info