Back to Basics: Templates – Part 1

Bob Steagall

CppCon 2021

Overview

- Rationale
- Template fundamentals
- Template categories in detail

Goals and References

- Goals
  - Cover major features
  - Explain some important terminology and concepts
  - Point to next steps

- Recommended references
  - *C++ Templates The Complete Guide,* Second Edition

    David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor – Addison-Wesley 2018

  - *Effective Modern C++*

    Scott Meyers – O'Reilly 2015

  - *The C++ Programming Language*, Fourth Edition

    Bjarne Stroustrup – Addison-Wesley 2013

  - cppreference.com

Rationale

The Bad Old Days – Reuse With Cut-N-Paste

```
int
min(int a, int b)
{
    return (a < b) ? a : b;
}


double
min(double a, double b)
{
    return (a < b) ? a : b;
}


string
min(string a, string b)
{
    return (a < b) ? a : b;
}
```

The Bad Old Days – Reuse With Cut-N-Paste

```
struct int_node
{
    int_node*  next;
    int        value;
};

struct int_list
{
    int_node*  front;
    int_node*  back;
};


void
int_list_append(int_list* l, int val);

void
int_list_prepend(int_list* l, int val);

void
int_list_clear(int_list* l);
```

```
struct double_node
{
    double_node*  next;
    double        value;
};

struct double_list
{
    double_node*  front;
    double_node*  back;
};


double
dbl_list_append(int_list* l, double val);

void
dbl_list_prepend(int_list* l, double val);

void
dbl_list_clear(int_list* l);
```

The Bad Old Days – Reuse With Type Erasure

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void *, const void *));


int cmp_dbl(const void* va, const void* vb)
{
    double a = *((double const*) va);
    double b = *((double const*) vb);

    if (a < b)         return -1;
    else if (a == b) return 0;
    else               return 1;
}

void f()
{
    double  dbl_data[4] = { 3.14159, 1.41421, 2.71828, 1.61803 };

    qsort(&dbl_data[0], 4u, sizeof(double), &cmp_dbl);
}
```

The Bad Old Days – Reuse With Type Erasure + Cut-N-Paste

```
#include <stdlib.h>
#include <string.h>

...

int cmp_str(const void* va, const void* vb)
{
    char const* pa = (char const*) va;
    char const* pb = (char const*) vb;

    return strcmp(pa, pb)
}

void g()
{
    string  str_data[5] = { "these", "are", "not", "the", "droids" };

    qsort(&str_data[0], 5u, sizeof(string), &cmp_dbl);
}
```

The Bad Old Days – Reuse With Type Erasure + Cut-N-Paste

```
#include <stdlib.h>
#include <string.h>

...

int cmp_str(const void* va, const void* vb)
{
    char const* pa = (char const*) va;
    char const* pb = (char const*) vb;

    return strcmp(pa, pb)
}

void g()
{
    string  str_data[5] = { "these", "are", "not", "the", "droids" };

    qsort(&str_data[0], 5u, sizeof(string), &cmp_dbl);  //- Error!
}
```

The Bad Old Days – Code Reuse With Macros

```
...
#define BUILD_COMPARE(TYPE)                                 \
    int cmp_ ## TYPE(const void* va, const void* vb)    \
    {                                                   \
        TYPE const*  pa = static_cast<TYPE const*>(va); \
        TYPE const*  pb = static_cast<TYPE const*>(vb); \
                                                        \
        if (*pa < *pb)        return -1;                \
        else if (*pa == *pb) return 0;                  \
        else                  return 1;                 \
    }

BUILD_COMPARE(float)
BUILD_COMPARE(double)

void h()
{
    float    data[4] = { 4.0, 3.0, 2.0, 1.0 };
    qsort(&data[0], 4u, sizeof(float), &cmp_float);  //- OK
    qsort(&data[0], 4u, sizeof(float), &cmp_dbl);    //- Error
}
```

Code Reuse

- These problems have been around a *long* time
- In the 1970's, some languages began allowing algorithms to be written in terms of types to-be-specified-later
- Algorithms were then instantiated on demand using type arguments
- This approach is now known as **generic programming**

What is Generic Programming?

> Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.
>
> — David Musser, Alexander Stepanov
> *Generic Programming* (1988)

What is Generic Programming?

> Following Stepanov, we can define generic programming without mentioning language features: **Lift algorithms and data structures from concrete examples to their most general and abstract form**.

— Bjarne Stroustrup
Evolving a language in and for the real world: C++ 1991-2006 (2007)
[emphasis mine]

Code Reuse with Generic Programming

- These problems have been around a *long* time
- In the 1970's, some languages began allowing algorithms to be written in terms of types to-be-specified-later
- Algorithms were then instantiated on demand using type parameters
- This approach came to be known as **generic programming**

C++ supports generic programming with **templates**

Template Categories

Function Templates (C++98/03)

- Recipes for making *functions*

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

template<class T>
void  swap(T& a, T& b);

template<class RandomIt, class Compare>
void  sort(RandomIt first, RandomIt last, Compare comp);
```

Class Templates (C++98/03)

- Recipes for making *classes*

```
template<class T, size_t N>
struct array
{...};


template<class T, class Alloc = allocator<T>>
class vector
{...};


template<class Key, class Val,
         class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class map
{...};
```

Member Function Templates (C++98/03)

- Recipes for making *member functions*

```
template<class T, class Alloc = allocator<T>>
class vector
{
  public:
    ...

    using iterator       = ...;
    using const_iterator = ...;
    ...

    template<class InputIter>
    iterator  insert(const_iterator pos, InputIter first, InputIter last) {...}
    ...
};
```

Alias Templates (C++11)

- Recipes for making *type aliases*

```
template<class T>
using sa_vector = vector<T, my_special_allocator<T>>;

sa_vector<float>   fv;



template<class Key, class Val>
using my_map = map<Key, Val, greater<Key>>;

my_map<string, int> msi;



template<class T, ptrdiff_t C, class A = std::allocator<T>, class CT = void>
using general_row_vector =
  basic_matrix<matrix_storage_engine<T, extents<1,C>, A, matrix_layout::row_major>, CT>;

general_row_vector<double, 20>  rv;
```

Variable Templates (C++14)

- Recipes for making *variables* or *static data members*

```
template<class T>
inline constexpr T   pi = T(3.14159265358979323851L);

template<class T>
T  circular_area(T r)  { return pi<T> * r * r; }


template< class T >
inline constexpr bool   is_arithmetic_v = is_arithmetic<T>::value;

void init(T* p, size_t N)
{
    if constexpr (is_arithmetic<T>)
        memcpy(p, 0, sizeof(T)*N);
    else
        uninitialized_fill_n(p, N, T());
}
```

Lambda Templates (C++20)

- Recipes for making *lambdas*

```cpp
auto multiply = []<class T>(T a, T b) { return a * b; };

auto d0 = multiply(1.0, 2.0);
```

Template Fundamentals

Template Terminology

- Discussing templates with clarity means using terminology with precision

- How do we refer to templates used to "generate" classes?
  - Classes, structs, and unions are referred to generally as *class types*
  - *Class template* indicates a *parametrized description* of a family of classes

- C++ also provides parametrized descriptions of
  - Functions
  - Member functions
  - Type aliases
  - Variables
  - Lambdas

Template Terminology

- The standard treats terms `thing template` consistently
  - `template` is the noun, indicating a parametrized description
  - `thing` is an adjective, specifying the family of things being parametrized

So, we have:

This kind of template...	... is a parametrized description of a family of...
class template	classes
function template	functions
member function template	member functions
alias template	type aliases
variable template	variables
lambda template	lambda functions

- Also, the associated verb is **parametrize** or **parameterize** – not templatize!

Translation Units

- Compilation
  - The process of converting human-readable source code into binary object files
  - From a high-level perspective, there are four stages of compilation:
    - Lexical analysis
    - Syntax analysis
    - Semantic analysis
    - Code generation
  - In C++, we typically generate one object file for each source file

- Linking
  - The process of combining object files and binary libraries to make a working program

- The standard calls the compilation process **translation**

Translation Units

- In C++, translation is performed in nine well-defined stages

- Phases 1 through 6 perform lexical analysis
  - These are what we usually refer to as *pre-processing*
  - The output of Phase 6 is a **translation unit**

- A translation unit is defined [5.1] as
  - A source file
  - Plus all the headers and source files included via `#include` directives
  - Minus any source lines skipped by conditional inclusion preprocessing directives (#ifdef)
  - And all macros expanded

Translation Units

- Phases 7 and 8 perform syntax analysis, semantic analysis, and codegen
  - These are what we usually refer to as *compilation*
  - Templates are parsed in Phase 7
  - Templates are instantiated in Phase 8
  - The output is called a *translated translation unit* (e.g., object code)
- Phase 9 performs program image creation
  - This is what we usually think of as *linking*
  - The output is an executable image suitable for the intended execution environment

Phases of Translation (6) – a Sample TU For Fun

```
//  hello.h
//=========
#ifndef HELLO_H_INC
#define HELLO_H_INC

#include <iostream>

void print_hello();

#endif
```

```
//  hello.cpp
//===========
#include "hello.h"

void print_hello()
{
    std::cout << "Hello!" << std::endl;
}
```

```
//  main.cpp
//==========
#include "hello.h"

int main()
{
    print_hello();
    return 0;
}
```

- Compilers provide a way to inspect TU contents (or something close to it)
  - With GCC, you can use the `–E` flag:

```
$ g++ -std=c++20 -E main.cpp  | egrep -v '#' | tee main.i
$ g++ -std=c++20 -E hello.cpp | egrep -v '#' | tee hello.i
```

- How many lines in `main.i`? `hello.i`?
  - 41,625 / 41,624 {with GCC 10.2 on Ubuntu 18.04}

Declarations and Definitions

- An **entity** is one of these things:
  - value
  - object
  - reference
  - structured binding
  - function
  - enumerator
  - type
  - class member
  - bit-field
  - template
  - template specialization
  - namespace
  - pack

Declarations and Definitions

- A **name** is the use of an identifier that denotes an entity (or label)
  - Every name that denotes an entity is introduced by a **declaration**

- A **declaration** introduces one or more **names** into a translation unit
  - A declaration may also *re-introduce* a name into a translation unit

- A **definition** is a declaration that fully defines the entity being introduced

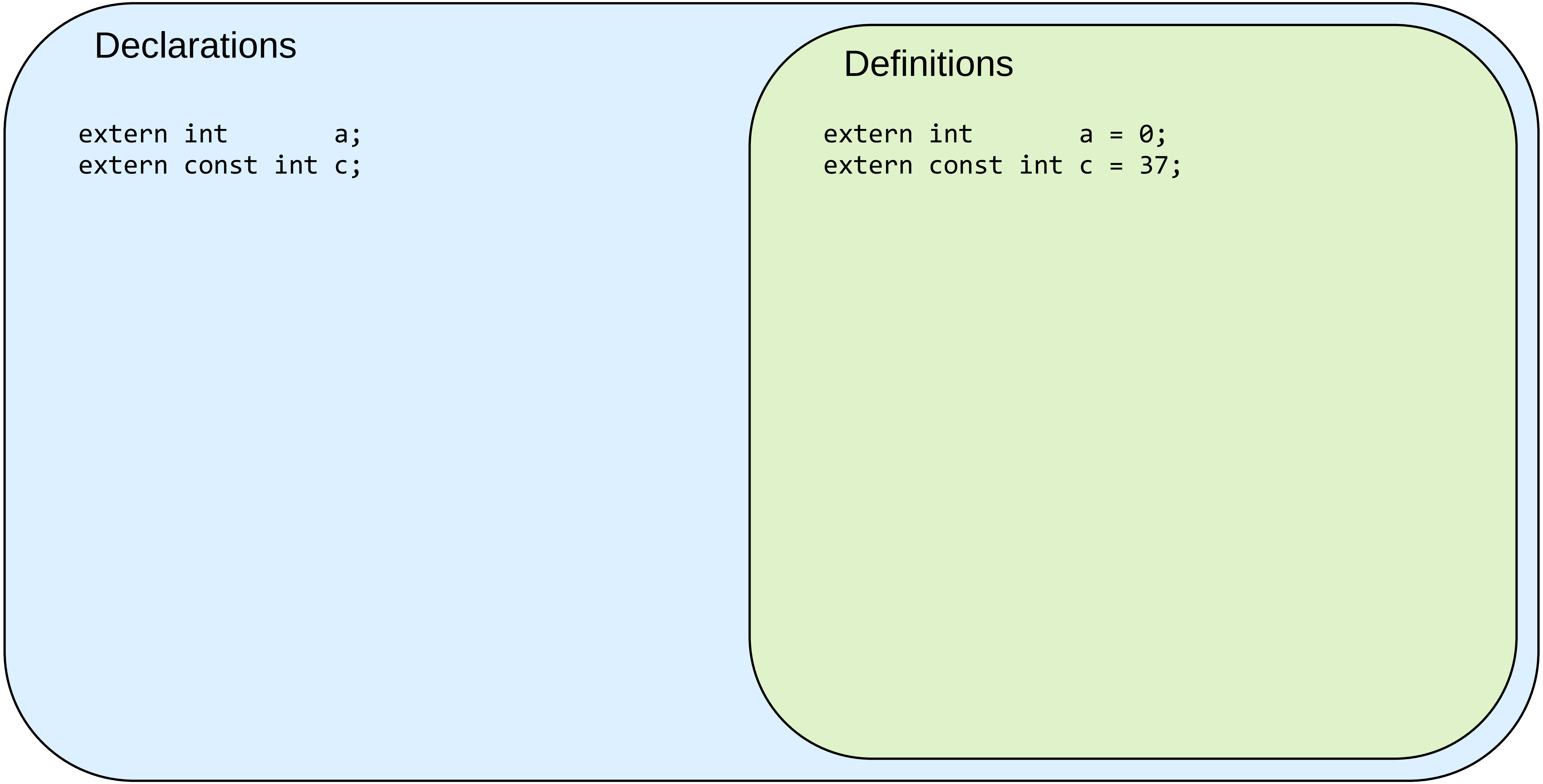
- A **variable** is an entity introduced by the declaration of an object
  - Or of a reference other than a non-static data member

Declarations and Definitions

- Every declaration *is also a definition*, unless:
    - It is a function declaration without a corresponding definition of the body
    - It is a parameter declaration in a function declaration that is not a definition
    - It is a declaration of a class name without a corresponding definition
    - It is a template parameter
    - It is a `typedef` declaration
    - It is a `using` declaration
    - It contains the `extern` specifier
    - And a few other cases…

- The set of definitions is a proper subset of the set of declarations

Declarations and Definitions



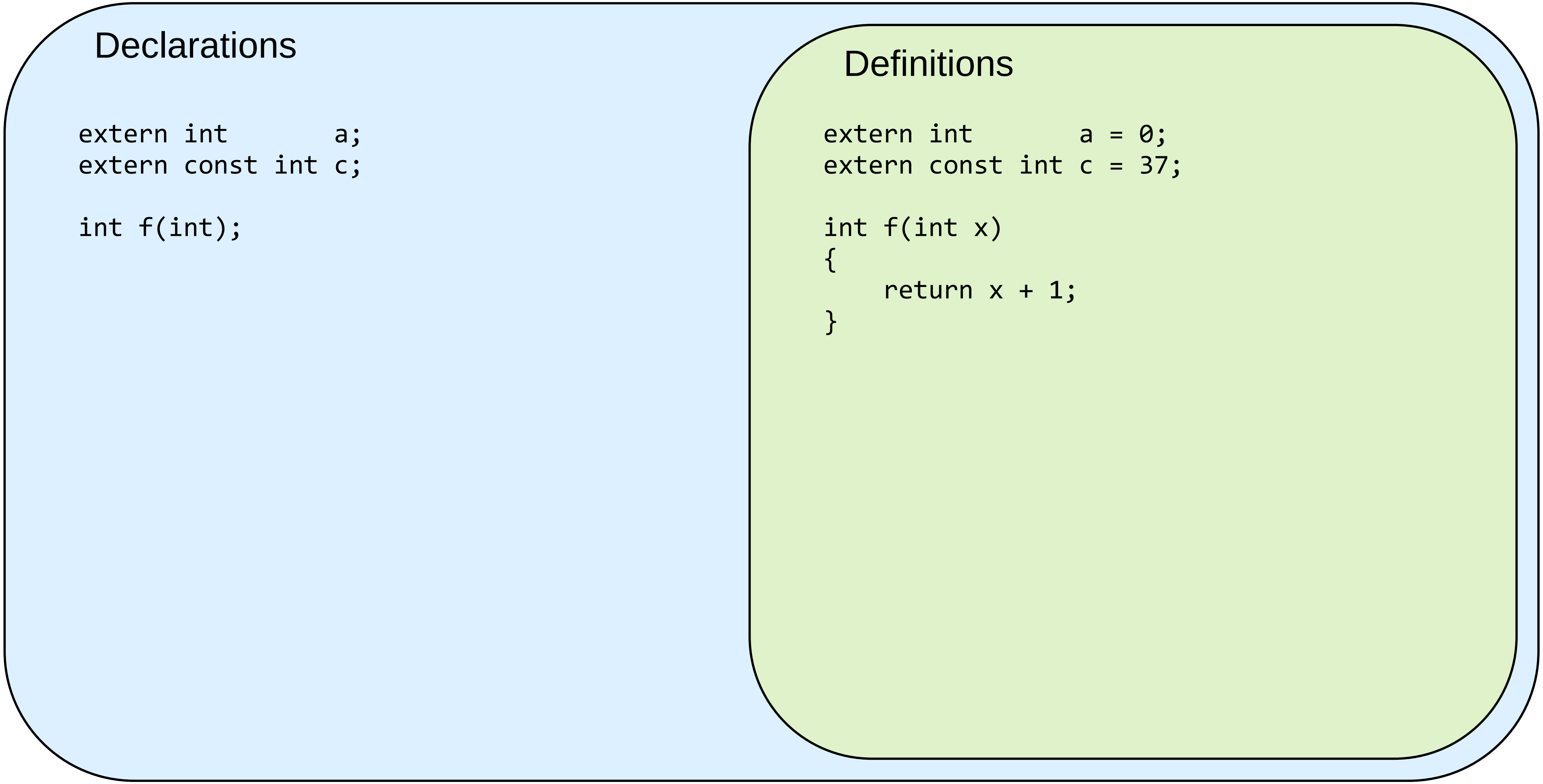
Declarations

```
extern int       a;
extern const int c;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;
```

Declarations and Definitions



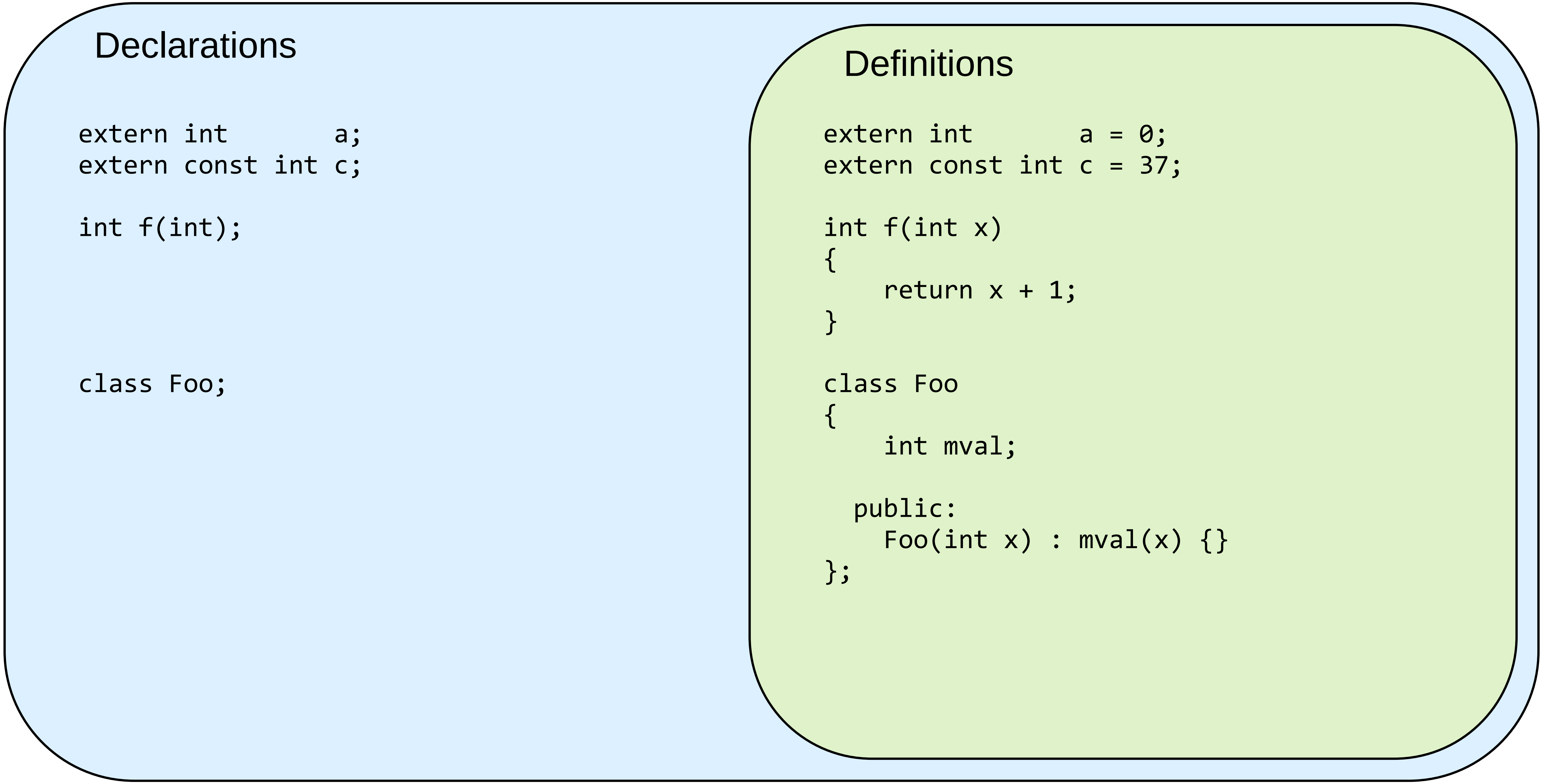
Declarations

```
extern int       a;
extern const int c;

int f(int);
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}
```

Declarations and Definitions



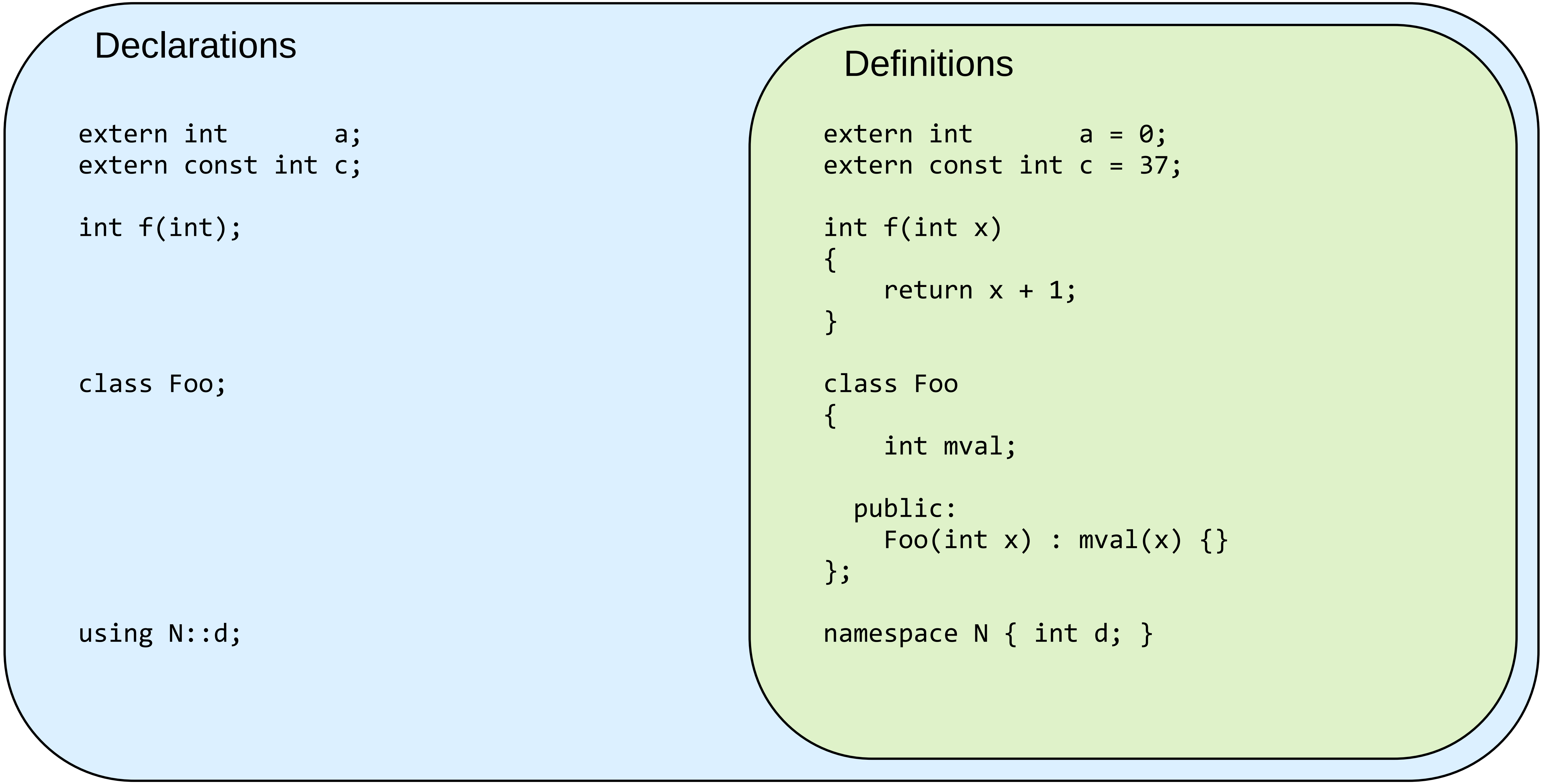
Declarations

```
extern int       a;
extern const int c;

int f(int);




class Foo;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}

class Foo
{
    int mval;

  public:
    Foo(int x) : mval(x) {}
};
```

Declarations and Definitions



Declarations

```
extern int       a;
extern const int c;

int f(int);
```

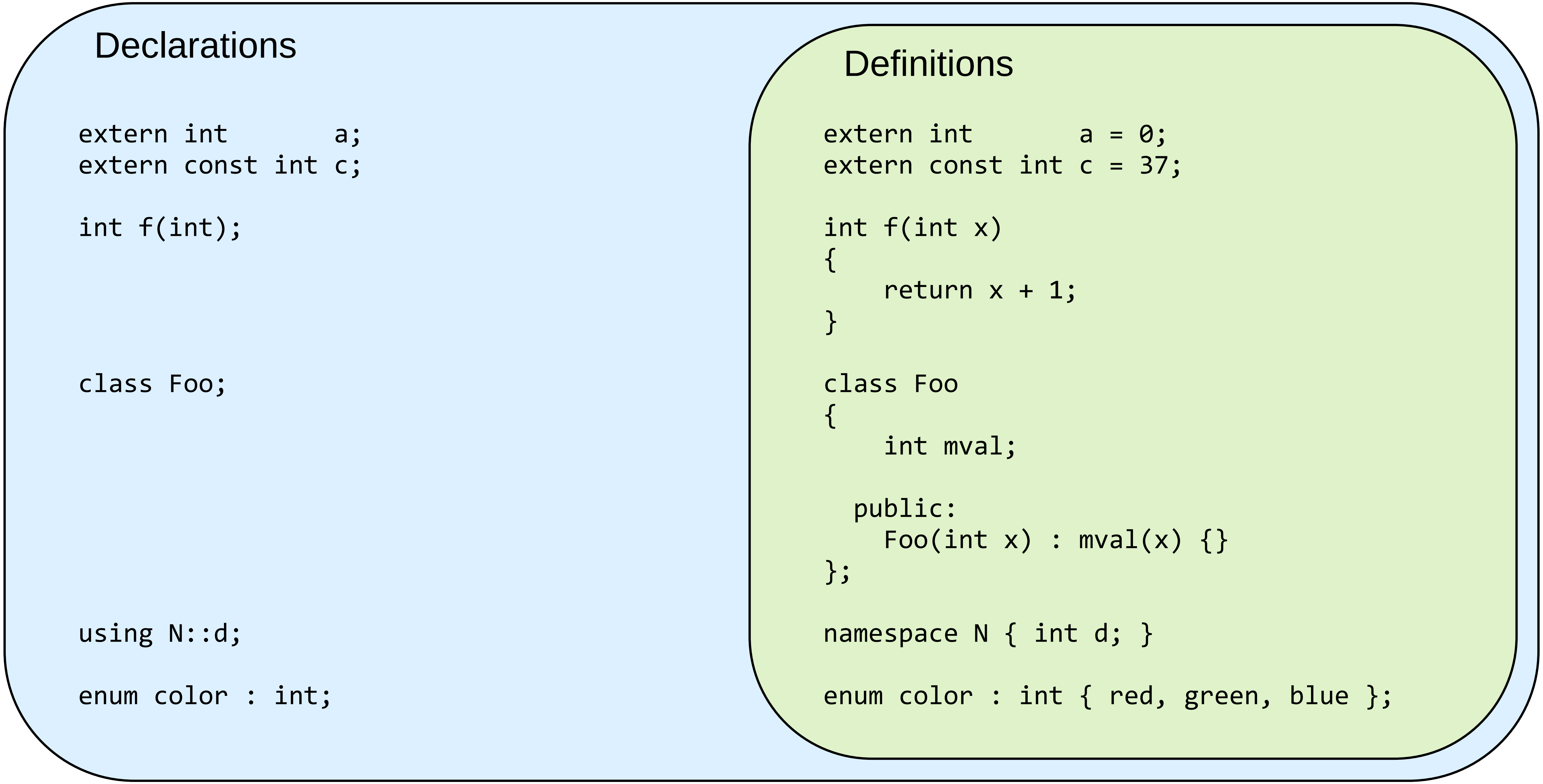
```
class Foo;
```

```
using N::d;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}

class Foo
{
    int mval;

  public:
    Foo(int x) : mval(x) {}
};

namespace N { int d; }
```

Declarations and Definitions



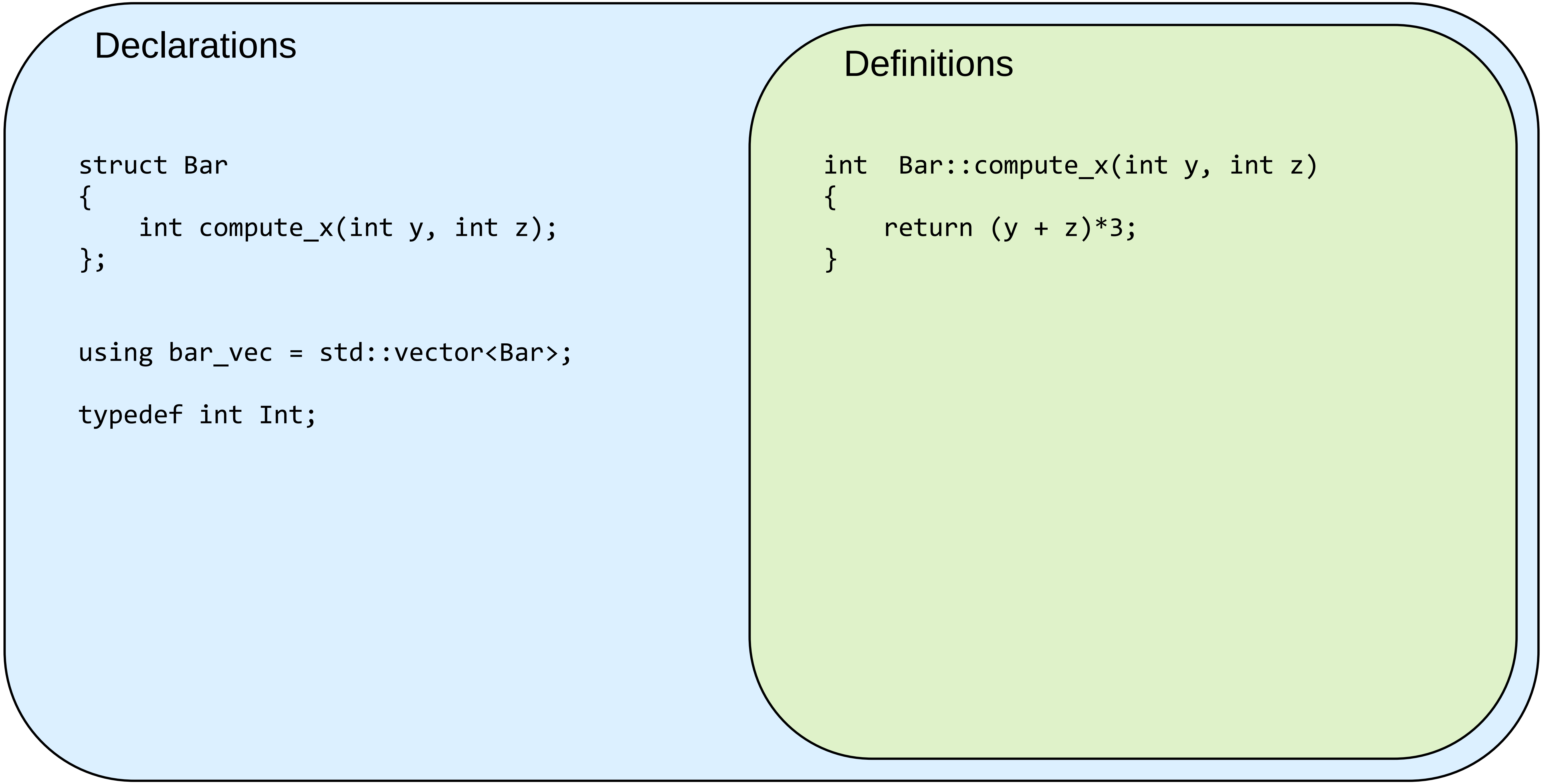
Declarations

```
extern int       a;
extern const int c;

int f(int);




class Foo;








using N::d;

enum color : int;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}

class Foo
{
    int mval;

  public:
    Foo(int x) : mval(x) {}
};

namespace N { int d; }

enum color : int { red, green, blue };
```

Declarations and Definitions



Declarations

```
struct Bar
{
    int compute_x(int y, int z);
};


using bar_vec = std::vector<Bar>;

typedef int Int;
```

Definitions

```
int  Bar::compute_x(int y, int z)
{
    return (y + z)*3;
}
```

Template Declarations and Definitions

```
template<class T>
T const&  max(T const& a, T const& b); //- Declaration of function template max

template<class T>
T const&  max(T const& a, T const& b)  //- Definition of function template max
{
    return (a > b) ? a : b;
}


template<class T1, class T2>
struct pair;                           //- Declaration of class template pair

template<class T1, class T2>
struct pair                            //- Definition of class template pair
{
    T1  first;
    T2  second;
    ...
};
```

Template Declarations and Definitions

```
template<class T, class Alloc = allocator<T>>
class vector
{
  public:
    ...

    using iterator       = ...;
    using const_iterator = ...;
    ...

    template<class InputIter>      //- Declaration of member function template insert
    iterator  insert(const_iterator pos, InputIter first, InputIter last);
    ...
};

template<class T, class Alloc>     //- Definition of member function template insert
template<class InputIter> auto
vector<T,Alloc>::insert(const_iterator pos, InputIter first, InputIter last) -> iterator
{ ... }
```

Template Declarations and Definitions

```
template<class T, class Alloc = allocator<T>>
class vector
{
  public:
    ...

    using iterator       = ...;
    using const_iterator = ...;
    ...

    template<class InputIter>     //- Definition of member function template insert
    iterator  insert(const_iterator pos, InputIter first, InputIter last)
    { ... }
};

template<class Key, class Val>
using my_map = map<Key, Val, greater<Key>>;      //- Declaration of alias template my_map

template<class T>
constexpr T pi = T(3.1415926535897932385L);      //- Declaration of variable template pi
```

The One-Definition Rule (ODR)

- A given translation unit can contain at most one definition of any:
  - variable
  - function
  - class type
  - enumeration type
  - template
  - default argument for a parameter for a function in a given scope
  - default template argument

- There may be multiple declarations, but **there can only be one definition**

The One-Definition Rule (ODR)

- A program must contain exactly one definition of every non-`inline` variable or function that is used in the program
  - Multiple declarations are OK, but only one definition

- For an inline variable or an inline function, a definition is required in every translation unit that uses it
  - `inline` was originally a suggested optimization made to the compiler
  - It has now evolved to mean "multiple definitions are permitted"

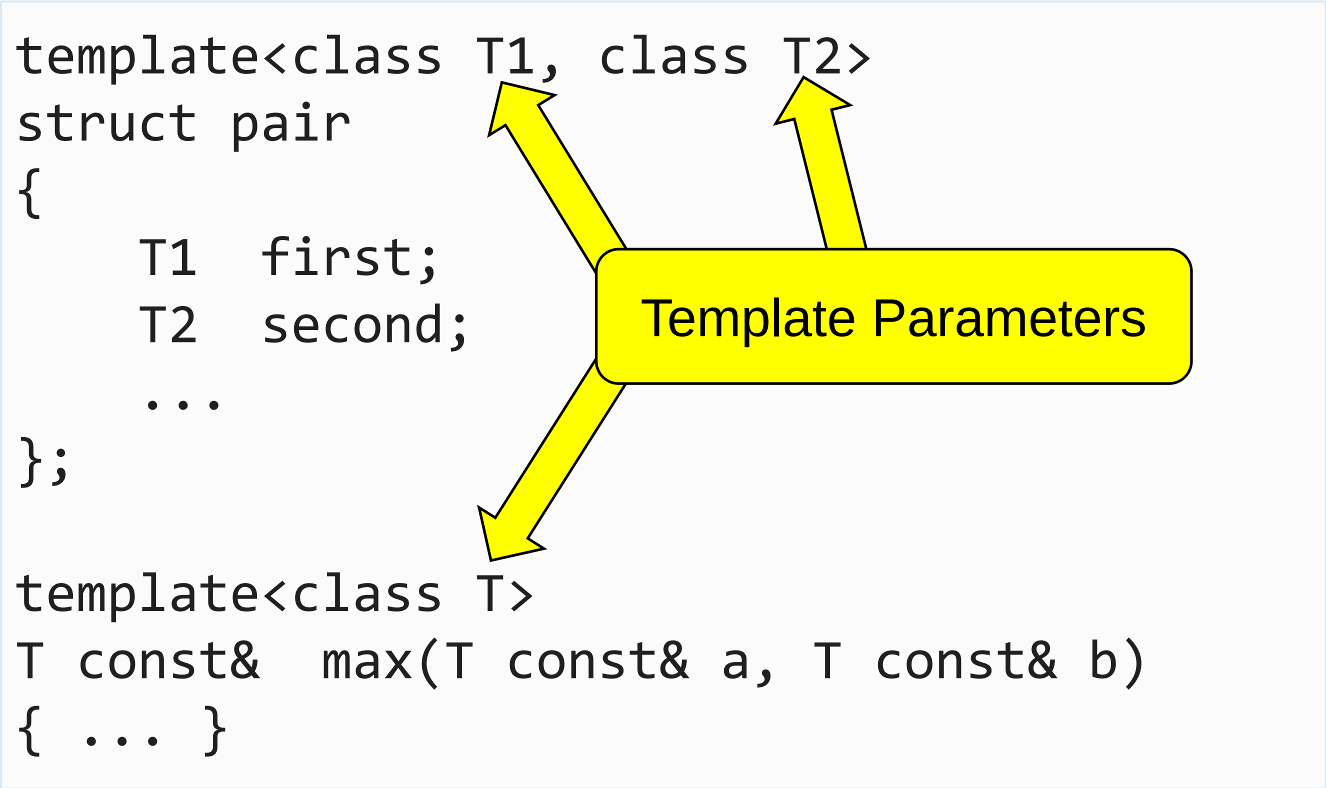
- Exactly one definition of a class must appear in any translation unit that uses it in such a way that the class must be complete

- The same rules for inline variables and functions **also apply to templates**

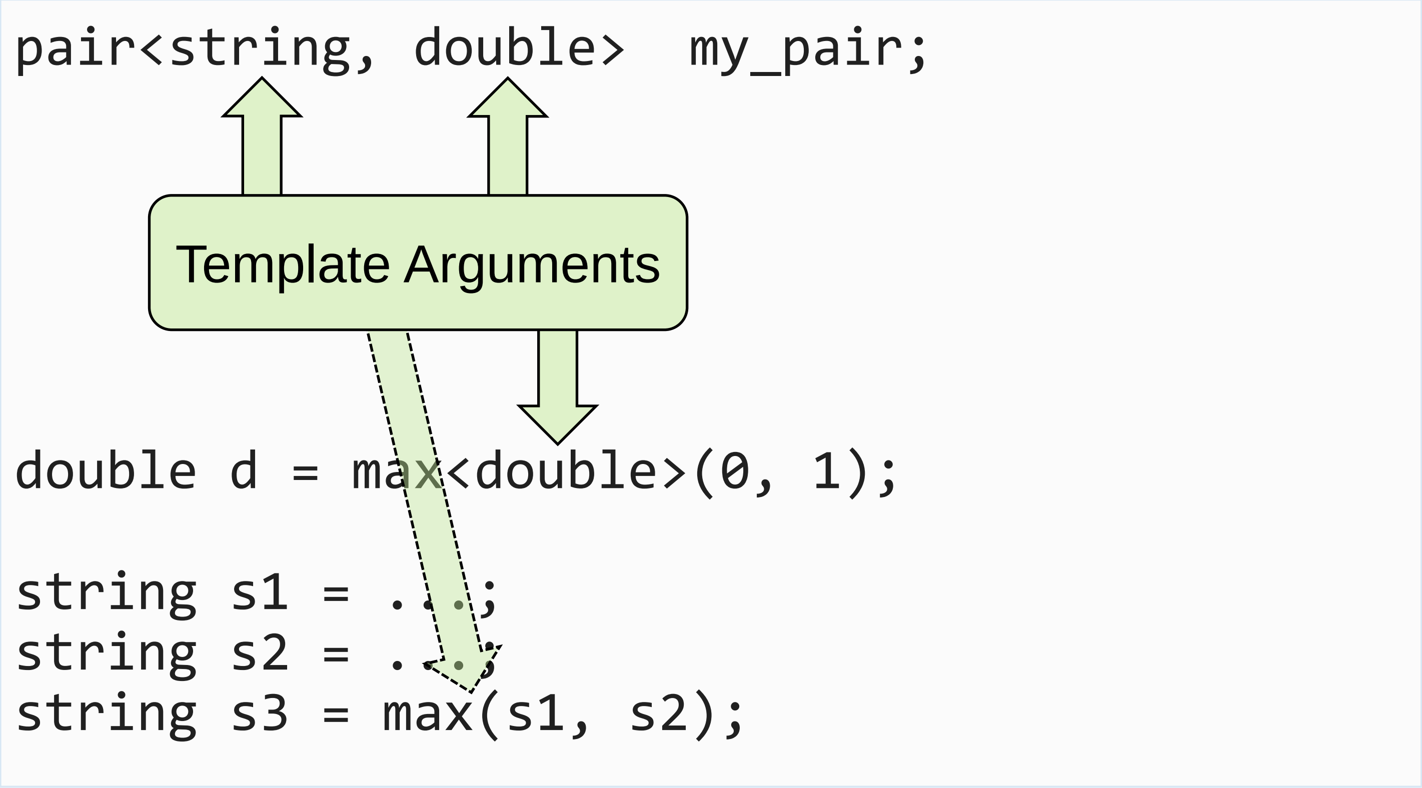
The One-Definition Rule (ODR)

- My simple guidelines for observing ODR:
- For an `inline` thing (variable or function) that get used in a translation unit, make sure it is defined at least once somewhere in that translation unit
- For a non-`inline`, non-`template` thing that gets used, make sure it is defined exactly once in across all translation units
- For a `template` thing, define it in a header file, include the header where the thing is needed, and let the toolchain decide where it is defined
  - Except in rare circumstances where finer control is required

Template Parameters and Template Arguments



```
template<class T1, class T2>
struct pair
{
    T1  first;
    T2  second;
    ...
};

template<class T>
T const&  max(T const& a, T const& b)
{ ... }
```



```
pair<string, double>  my_pair;



double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

- **Template parameters** are the names that come after the `template` keyword in a template declaration

- **Template arguments** are the concrete items **substituted** for template parameters to create a template **specialization**

Template Parameters

- Template parameters come in three flavors
  - Type parameters
  - Non-type template parameters (NTTPs)
  - Template-template parameters

- Type parameters
  - Most common
  - Declared using the `class` or `typename` keywords

```
template<class T1, class T2>       struct pair;
template<typename T1, typename T2> struct pair;


template<class T>     T max(T const& a, T const& b);
template<typename T>  T max(T const& a, T const& b);
```

Non-Type Template Parameters (NTTPs)

- Template parameters don't have to be types:

```
template<class T, size_t N>
class Array
{
    T   m_data[N]
    ...
};

Array<foobar, 10>  some_foobars;
```

```
template<int Incr>
int IncrementBy(int val)
{
    return val + Incr;
}

int x = ...;
int y = IncrementBy<42>(x);
```

- NTTPs denote constant values that can be determined at compile or link time, and their type must be
  - An integer or enumeration type (most common)
  - A pointer or pointer-to-member type
  - std::nullptr_t
  - And a couple of other things…

Template-Template Parameters

- Template parameters can themselves be templates
  - Placeholders for class or alias templates
  - Declared like class templates, but only the `class` and `typename` keywords can be used

```
#include <vector>
#include <list>

template<class T, template<class U, class A =std::allocator<U>> class C>
struct Adaptor
{
    C<T>  my_data;
    void  push_back(T const& t) { my_data.push_back(t); }
};

Adaptor<int, std::vector>    a1;
Adaptor<long, std::list>     a2;

a1.push_back(0);
a2.push_back(1);
```

Default Template Arguments

- Template parameters can have default arguments

```
template<class T, class Alloc = allocator<T>>
class vector {...};

template<class T, size_t N = 32>
class Array {...}

template<class T, template<class U, class A = allocator<U>> class C = vector>
struct Adaptor {...};


vector<double> vec;  //- std::vector<double, std::allocator<double>>
Array<long>    arr;  //- Array<long, 32>
Adaptor<int>   adp;  //- Adaptor<int, std::vector<int, std::allocator<int>>>
```

Default Template Arguments

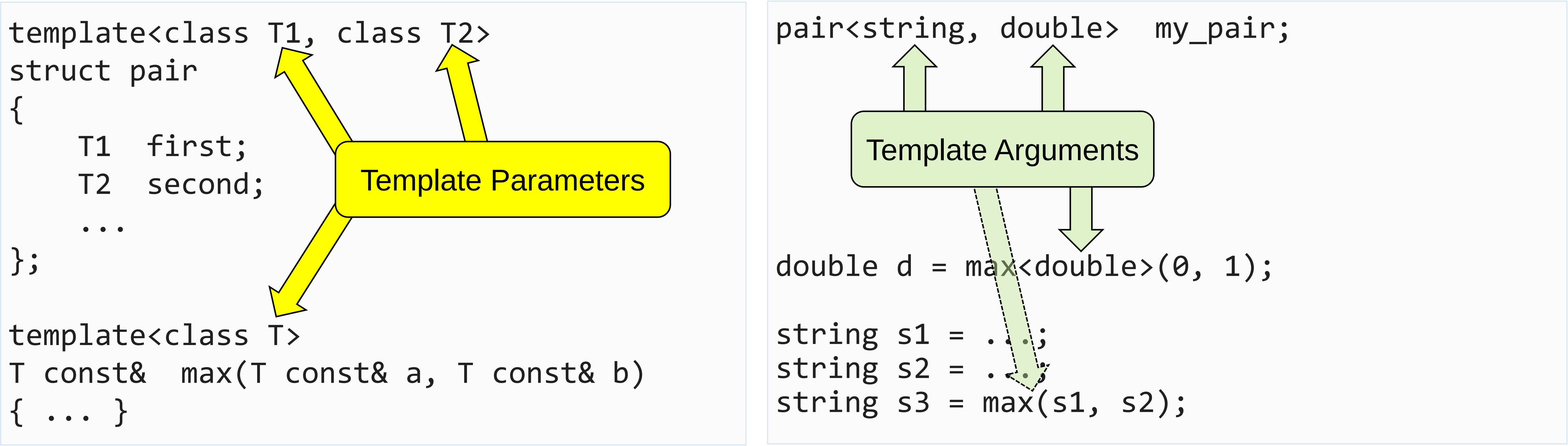
- Default arguments must occur at the end of the list for class, alias, and variable templates

```
template<class T0, class T1=int, class T2=int, class T3=int>
class quad;     //- OK

template<class T0, class T1=int, class T2=int, class T3=int, class T4>
class quint;    //- Error
```

- Function templates don't have this requirement
  - Template type deduction can determine the template parameters

```
template<class RT=void, class T>
RT*  address_of(T& value)
{
    return static_cast<RT*>(&value);
};
```

Template Parameters and Template Arguments



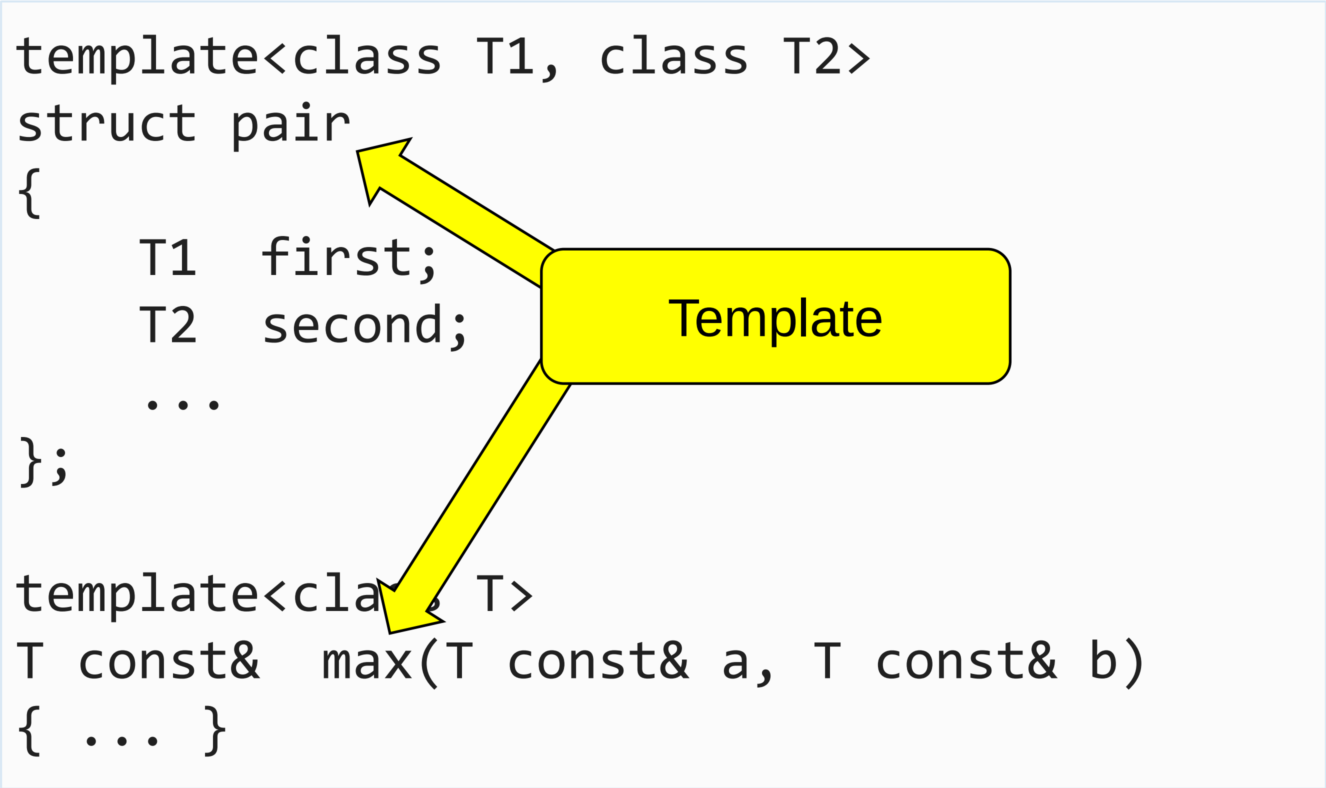
```
template<class T1, class T2>
struct pair
{
    T1  first;
    T2  second;
    ...
};

template<class T>
T const&  max(T const& a, T const& b)
{ ... }
```

```
pair<string, double>  my_pair;


double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

- **Template parameters** are the names that come after the `template` keyword in a template declaration

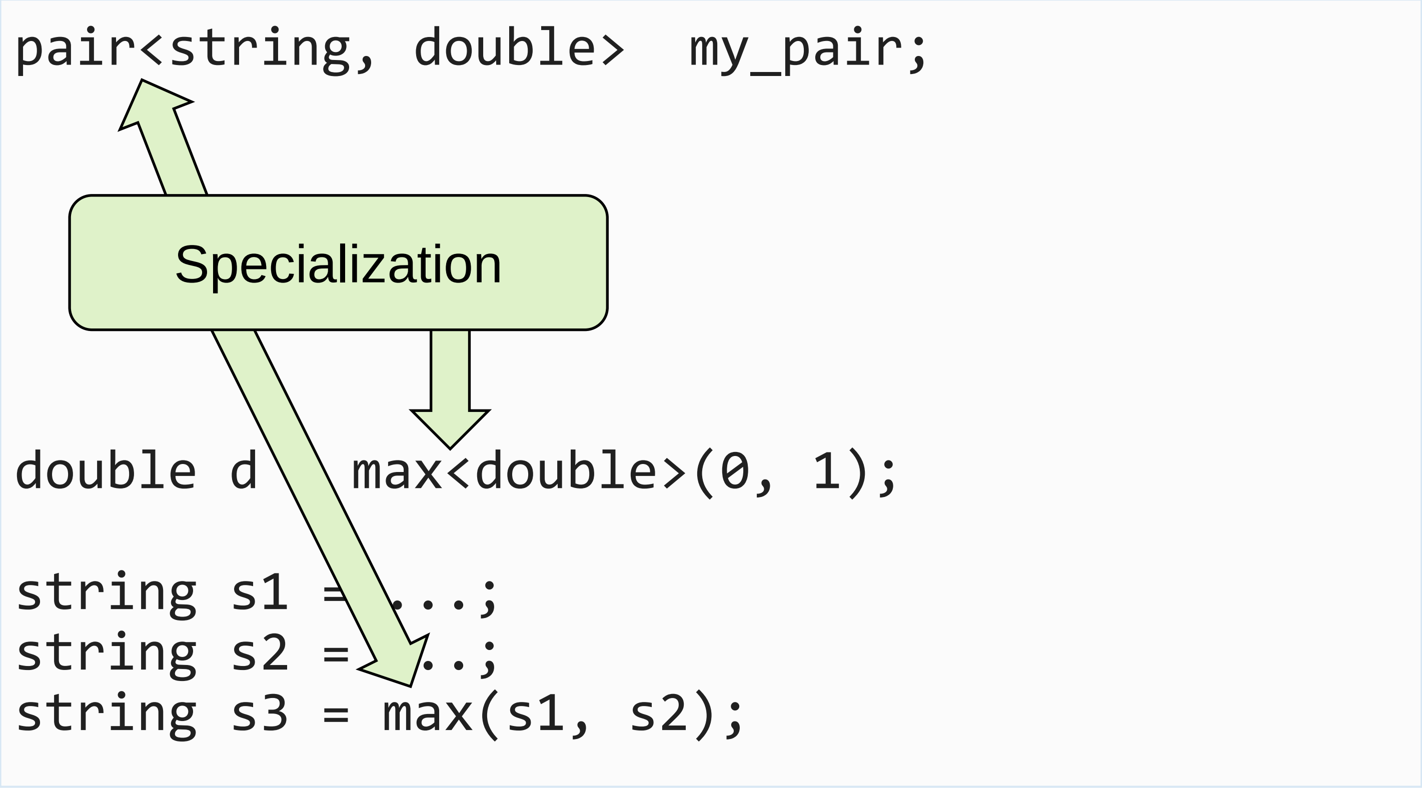
- **Template arguments** are the concrete items **substituted** for template parameters to create a template **specialization**

Substituting Template Arguments for Template Parameters



```
template<class T1, class T2>
struct pair
{
    T1  first;
    T2  second;
    ...
};

template<class T>
T const&  max(T const& a, T const& b)
{ ... }
```

Template



```
pair<string, double>  my_pair;



double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

Specialization

- **Template parameters** are the names that come after the `template` keyword in a template declaration

- **Template arguments** are the concrete items **substituted** for template parameters to create a template **specialization**

Specialization

- The concrete entity resulting from *substituting* template arguments for template parameters is a **specialization**

- These entities are named, and the name has the syntactic form

  template-name<argument-list>

  - This name is formally called a **template-id**

- From the earlier example
  - `pair` is a class template
  - `max` is a function template

```
template<class T1, class T2>
struct pair
{
    T1  first;
    T2  second;
    ...
};

template<class T>
T const&  max(T const& a, T const& b)
{ ... }
```

Specialization

- The concrete entity resulting from *substituting* template arguments for template parameters is a **specialization**

- These entities are named, and the name has the syntactic form

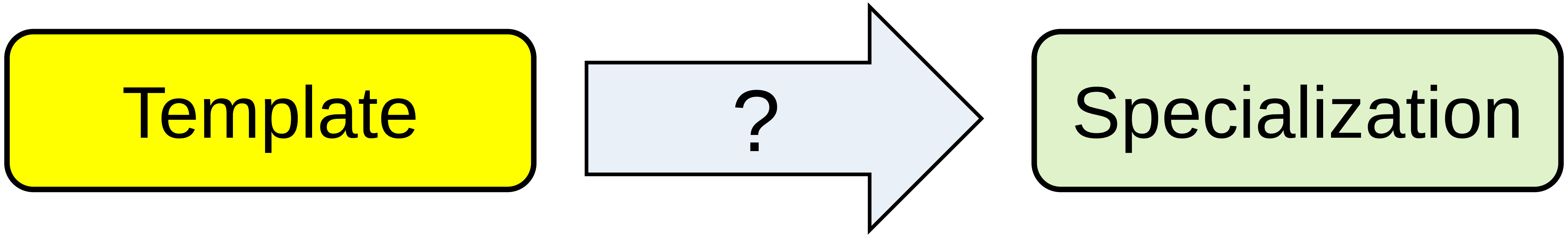
  `template-name<argument-list>`

  - This name is formally called a **template-id**

- From the earlier example
  - `pair<string, double>`
    `max<double>`
    `max<string>`
    are the names of specializations

```
pair<string, double>  my_pair;

double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

From Template to Specialization



- The template is a recipe that tells how to generate something useful
- A specialization is the useful thing built from that recipe
- Q: How do we get from template to specialization?
- A1: *Instantiation*
  A2: *Explicit specialization*

Instantiation

- At some point we'll want to use the recipe and make a *thing*
  - Most of the time the compiler knows how to cook the recipe for us
- At various times, the compiler will ***substitute*** concrete (actual) template arguments for the template parameters used by a template
- Sometimes this substitution is tentative
  - The compiler checks to see if a possible substitution could be valid
- Sometimes the result of this substitution is used to create a specialization ...

Instantiation

- **Template instantiation** occurs when *the **compiler** substitutes template arguments for template parameters* in order to define an entity
  - I.e., generate a specialization of some template

- The specialization from instantiating a class template is sometimes called (informally) an *instantiated class*
  - Likewise for the other template categories (*instantiated function*, etc.)
  - These are also informally called *instantiations*

- Template instantiation can occur in two possible ways
  - Implicitly
  - Explicitly

Instantiation and Specialization

- What are the relationships between instantiation and specialization?

  NB: arrow means *Is-A*

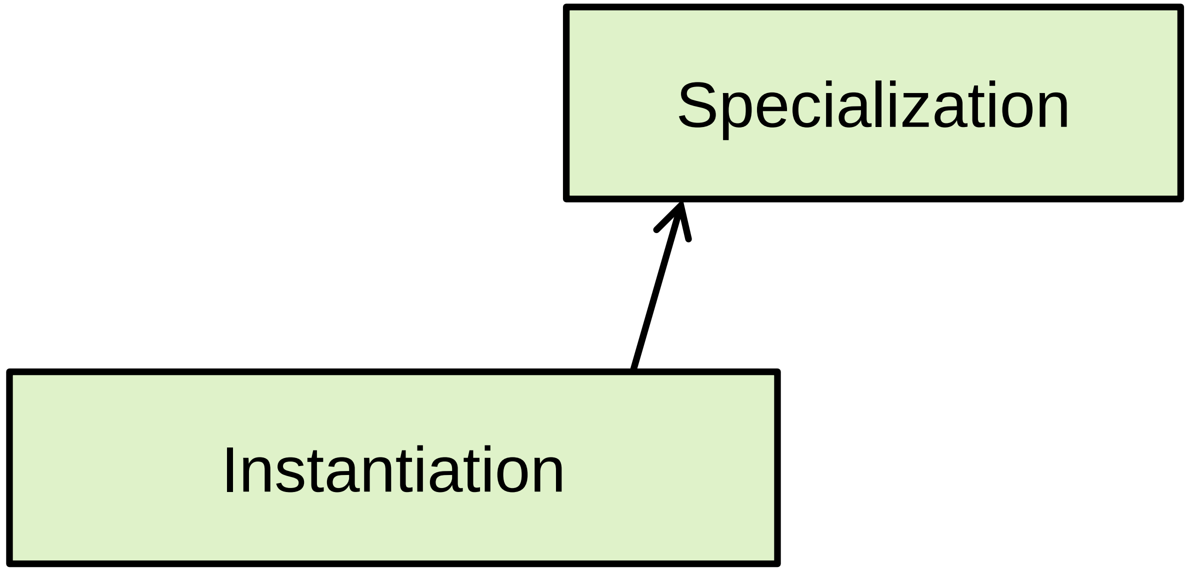


Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Implicit Instantiation

- In general, when the compiler sees the use of a specialization in code, it will create the specialization by substituting template arguments for parameters
    - This totally automatic, requiring no guidance from the code
- This is called **implicit**, **on-demand**, or **automatic** instantiation
    - The compiler decides where, when, and how much of the specialization to create

```
pair<string, double>  my_pair;

double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

Implicit Instantiation

- In general, when the compiler sees the use of a specialization in code, it will create the specialization by substituting template arguments for parameters
  - This totally automatic, requiring no guidance from the code

- This is called **implicit**, **on-demand**, or **automatic** instantiation
  - The compiler decides where, when, and how much of the specialization to create

- For class templates, implicit instantiation doesn't necessarily instantiate all the members of the class
  - The compiler might not generate non-virtual member functions or static data members
  - Consider:

```
void f()
{
    vector<int>  v{1, 2};
}
```

Instantiation and Specialization

- What are the relationships between instantiation and specialization?
  NB: arrow means *Is-A*

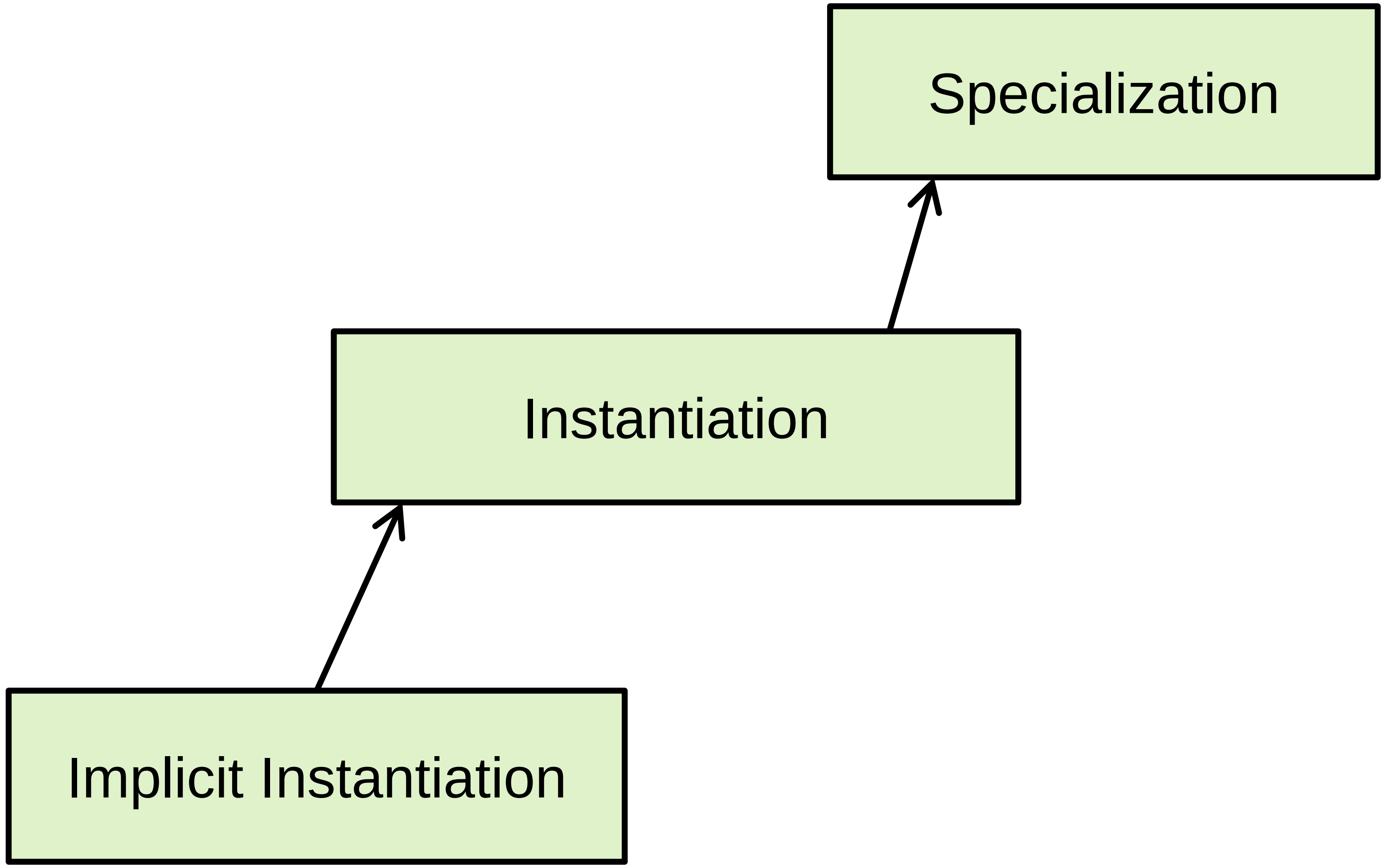


Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Explicit Instantiation

- Sometimes we want to control the where and when of instantiation

- This can be accomplished with **explicit instantiation**

```
//- Source file my_foo.cpp
template class vector<foo>;                     //- Definition
template class vector<foo, my_allocator<foo>>;  //- Definition

template void  swap<foo>(foo&, foo&);           //- Definition
template void  swap(bar&, bar&);                //- Definition
```

- Explicit instantiation of a class template instantiates *all* members

- However, individual member functions can be explicitly instantiated

```
template void vector<foo, my_allocator<foo>>::push_back(foo const&);    //- Definition
```

Explicit Instantiation

- For each template instantiated in a program, there must be exactly one definition of the corresponding specialization
  - If you explicitly instantiate a template in one translation unit, you must not explicitly instantiate in another translation unit

```
//- Source file my_foo.cpp
template class vector<foo>;                         //- Definition
template class vector<foo, my_allocator<foo>>;      //- Definition

template void  swap<foo>(foo&, foo&);               //- Definition
template void  swap(bar&, bar&);                    //- Definition
```

```
//- Header file my_foo.h
extern template class vector<foo>;                         //- Declared, not defined
extern template class vector<foo, my_allocator<foo>>;      //- Declared, not defined

extern template void  swap<foo>(foo&, foo&);               //- Declared, not defined
extern template void  swap(bar&, bar&);                    //- Declared, not defined
```

Instantiation and Specialization

- What are the relationships between instantiation and specialization?

  NB: arrow means *Is-A*

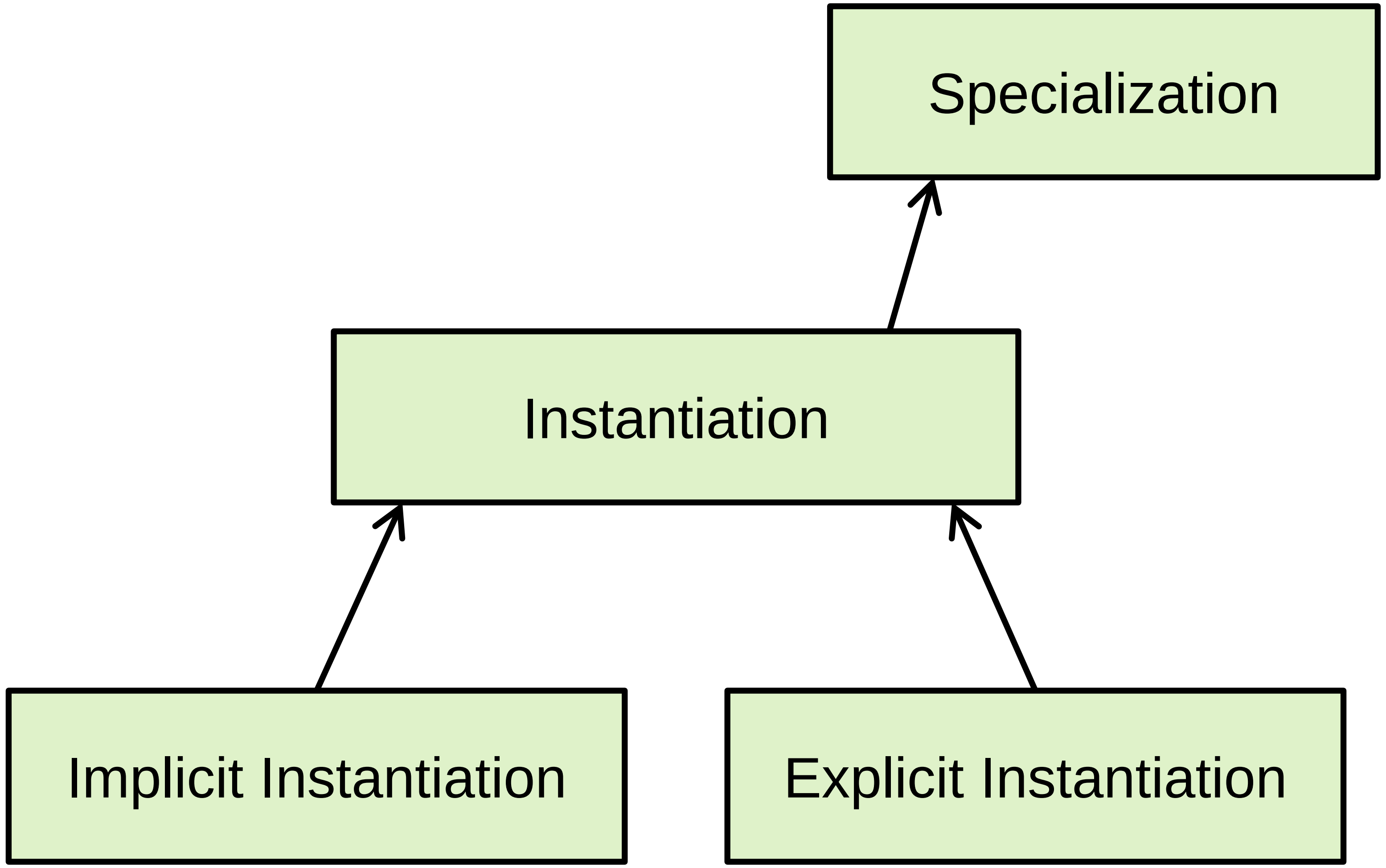


Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Explicit Specialization

- What if you want to customize the behavior of a template for a special situation?

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

char const*  p0 = "hello";
char const*  p1 = "world";
char const*  pr = min(p0, p1);     //- What's the answer?  There is no answer
```

- Sometimes situations arise where the template won't work properly

Explicit Specialization

- We can use explicit specialization – a user-provided implementation of a template with all template parameters fully substituted

```
template<class T>
T const&  min(T const& a, T const& b)                   //- Primary template
{
    return (a < b) ? a : b;
}

template<>
char const*  min(char const* pa, char const* pb)        //- Full specialization; this is only
{                                                       //  valid if a function template min
    return (strcmp(pa, pb) < 0) ? pa : pb;              //  has already been declared
}

char const*  p0 = "hello";
char const*  p1 = "world";
char const*  pr = min(p0, p1);   //- What's the answer?
```

Explicit Specialization

- It's probably more common to use explicit specialization with class templates

```
template<class T>
struct my_less                                              //- Primary template
{
    bool operator()(T const& a, T const& b) const
    {
         return (a < b) ? a : b;
    }
};

template<>
struct my_less<char const*>                                 //- Full specialization
{
    bool operator()(char const* pa, char const* pb) const
    {
         return strcmp(pa, pb) < 0;
    }
}

map<char const*, int, my_less<char const*>>  m1;
```

- An explicit specialization is valid only if a primary template has been declared

Instantiation and Specialization

- What are the relationships between instantiation and specialization?

  NB: arrow means *Is-A*

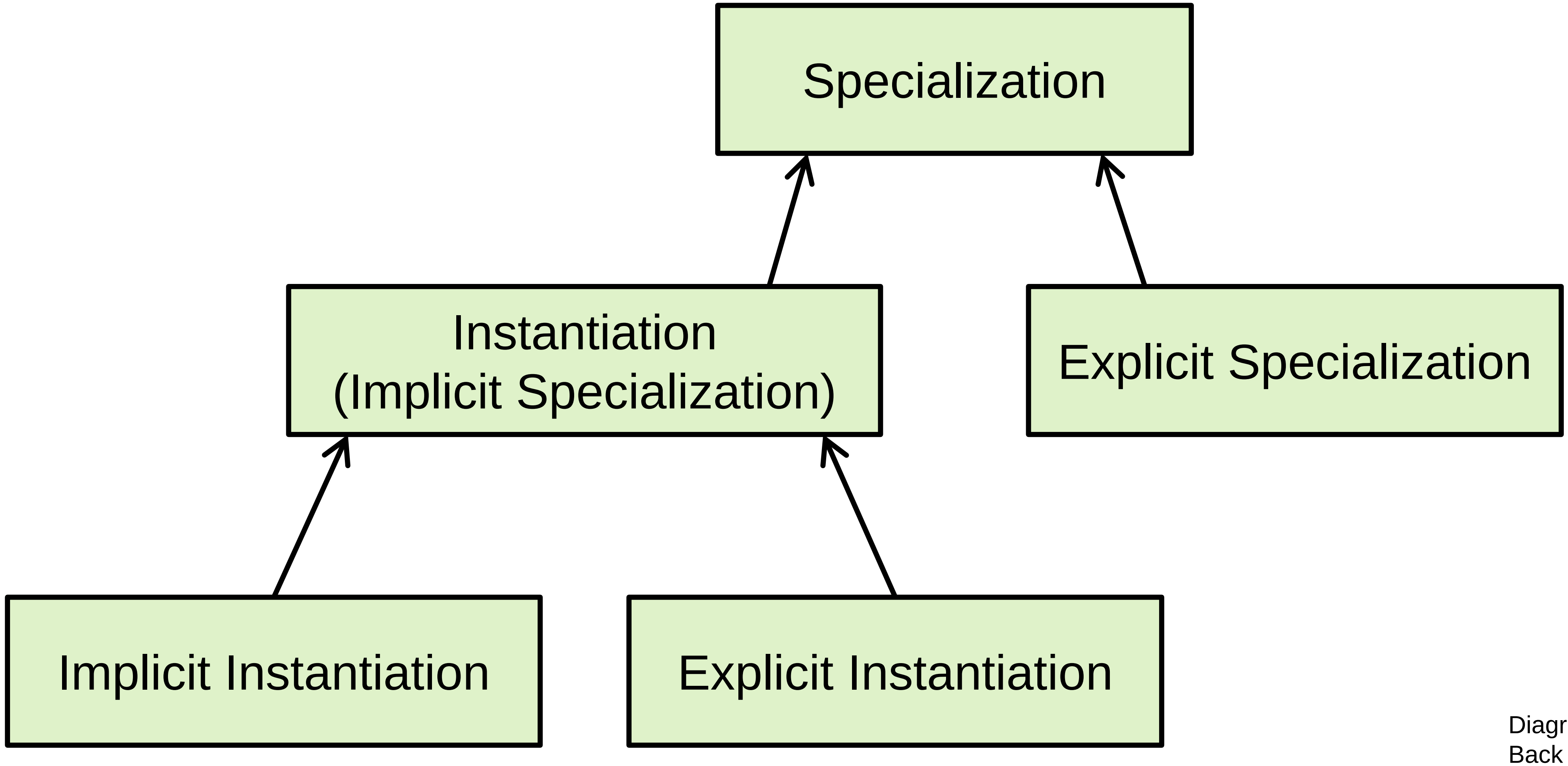


Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Thank You for Attending! (Join us Friday for Part 2)

Talk: `github.com/BobSteagall/CppCon2021`

Blog: `bobsteagall.com`