Name Resolution in C++

Elijah Shadbolt

10^{th} June, 2020

Abstract

For programmers and software engineers, giving names to different components of a system is an everyday occurrence. We use the principle of abstraction to hide the complexity of systems behind simple, robust interfaces, which represent concepts and ideas that we can understand and reason about. As with any programming language, C++ provides many ways for us to compile a program from separate, smaller sections of code. In order to reference the sections, we must abide by the syntax of the language. However, the aging language and its focus on backwards compatibility has made it hard for novice programmers to navigate safely. This article is aimed at beginner programmers who have experience with compiling simple programs involving variables, functions, and structs. The purpose of this article is to increase your understanding of the foundational language features of C++, so that you can write safer, more robust code, more efficiently. We will discuss how names and identifiers are used in C++ to declare, define, and reference different symbols in the program. We will cover a range of language features, including variables, functions, data types, scopes, function overloading, class members, and how they can be aliased.

Contents

List of Programs

List of Tables

List of Figures

1 Introduction

The programming language C++ and its surrounding environment is advanced in both age and complexity. One of the most fundamental principles that went into the design of C++ is abstraction [1]. We give names to concepts and ideas. As an example, library designers provide their users with an application programming interface (API), which hides the complexity of implementation by giving users specific ways to manipulate the system [2]. Large-scale programs and libraries are comprised of named symbols, including functions, constants, classes, and namespaces. Naming is an important part of software development.

The language provides features that allow programmers to reference sections of code by giving them user-defined names. More extreme examples of code re-use includes the Pre-Processor which can insert the contents of another file into the source code before it is compiled. The Linker can combine the compiled machine code (stored in object files) and output a single binary (executable, static library, or dynamic link library).

This article is meant to describe the most common and important ways to use names and identifiers in the C++ programming language. It is aimed at programmers who have intermediate knowledge and experience with C++. We hope to strengthen your understanding, so that you can write better code [3]. As a prerequisite, you must know how to build and run C++ programs, with an implementation of the C++ standard (compiler and linker). You should be aware of features like variables, functions, structs, pointers, and arrays.

Not even experienced C++ programmers know every little detail of the language, and we will not cover everything. The scope of this article is limited to standard version C++17 and earlier, about how names and identifiers are used in the language. This is a fitting scope for the article, because a recent survey revealed that the majority of C++ developers are allowed to use C++17 for their projects [4]. It does not cover modules, concepts, or other C++20 features. We are not fretting over naming conventions or what defines a good name. Advanced features such as virtual member functions, templates, and class lifetime semantics will be mentioned but not discussed, because they are built upon of the topics we will cover. As the language ages and new features are standardised, it is important to update our understanding of it so we can write safe and effective code efficiently, because the code we write might never go away.

2 Symbols

The core language of C++ is built upon three foundational language elements: data types, variables, and functions. A **symbol** is a specific element of the program, either a data type, variable, or function. An **identifier** (name) is a string of characters in the source code which is used to **refer** to a symbol. During the build process, the compiler will read our source code and figure out exactly what symbols should be used to build the output machine code, depending on the context in which we use their names. This process is called identifier resolution.

2.1 Identifiers

An identifier in C++ is comprised of characters from the ASCII encoding. Its characters can include letters, underscores, or decimal digits (0-9). Its first character must not be a digit, because when we try to reference the symbol the compiler would read it as a numeric literal constant. There must be no whitespace in an identifier. Unlike some other languages, C++ has very strict equality for identifiers. It is case-sensitive, so an uppercase letter does not compare equal to its lowercase counterpart, and vice versa. Two identifiers are considered equal (identical) if the strings are of the same length and each corresponding pair of characters in the strings has the same ASCII integer value. [5]

A keyword is a word which represents a core feature of the C++ language, such as `class`, `template`, `if`, `for`, and `const`. Some keywords may represent built-in symbols, such as `int` and `double`. Many keywords are reserved and cannot be used as names for new symbols. In some cases when keywords are added in new versions of the C++ standard, they can still be used as custom symbol names, for the sake of backwards compatibility.

Certain identifiers are reserved for the implementation of the C++ environment [6]. In any scope, identifiers are reserved if they contain adjacent underscores (as in `my__name`), or begin with an underscore followed by a capital letter (as in `_Bool`). Identifiers in the global namespace are reserved if they begin with an underscore (as in `_asm`). A general rule of thumb is "don't start with an underscore and don't have double underscores."

2.2 Declare, Define, Reference

In order for us programmers to add a new symbol to the program, we must **declare** it with a name and **define** what it represents [5]. The declaration describes the syntax for how the name should be referenced and used. The definition is used by the compiler to output the appropriate machine code for that little part of the program. The One Definition Rule states that symbols should have one and only one definition, because multiply defined symbols can result in linker errors or unexpected bugs. An exception to this rule is `inline` symbols, where the definition can appear multiple times but must be exactly the same. This is useful for working with header files, explained in Section 9.

In most cases, a name must be declared before it is used. This is because historically many compiler implementations were designed to compile the program in one pass or as few passes as possible, reading the source code sequentially from top-to-bottom, left-to-right (for Western text layouts). In order to use a symbol before it is defined, we can write a forward declaration, then reference it by name, and provide the definition somewhere else in the program [7]. There are some cases where no forward declaration is needed. For example, class members do not need to be forward declared if they are used within the scope of the class declaration.

2.3 Variables

Variables are declared with a type and a name. Variables can be defined and initialised in many implicit and explicit ways. They can be referenced by using their name in an expression. See Program 1 for an example. Every variable has a lifetime based on where it is stored in memory. If it is on the stack, it is a local variable and its destructor is called at the end of the scope block. Global variables and static class member variables have static storage duration, so their destructors are called (in an indeterminate order) after `main()` exits gracefully (without calling `std::terminate()` or propagating an uncaught exception). Values allocated on the heap (for example, with `new`) must be freed by the program with an appropriate deallocation function (`delete`). Heap deallocation can be made automatic with destructors and the RAII technique, providing a form of garbage collection (for example, `std::unique_ptr<T>`).

2.4 Functions

A function can be declared with a function prototype, which includes a return type and a list of formal parameters. A function can be defined (implemented) with a function body block following its prototype. A formal parameter is a variable declaration with a type and an optional name. Program 2 demonstrates how a function can be 'called' in an expression by using the function call syntax (`operator()`) on its name: writing the name followed by parentheses containing a list of arguments (values) separated by commas. The call site is the location in the code where the function is being invoked. Prototypes will be explained in more detail in Section 6.1.

2.5 Types

A data type describes the structure of a block of memory. An instance of a type is called an object. Every object has an address (location in memory). Variables and functions are objects. Data types are another way for us to represent concepts that we can understand, re-use, and build upon. New data types can be created with keywords `class`, `struct`, `union`, and `enum`.

A data type can be defined with a block of code which describes its structure in memory. The definition of the data structure includes the declarations of all its members (see Program 3). The definitions of members can be written elsewhere, or inside the type definition itself (with the `inline` keyword). Definitions of member functions written in the class definition are implicitly marked inline by the compiler. Symbols representing data types can only be referenced in a special kind of context reserved for typenames. In some cases, referring to a typename will require a template argument list.

It is also possible to forward declare a data type, but the symbol can only be used indirectly until the full definition is provided (see Program 4). Forward declared data types can only be used in pointer or reference types which are never dereferenced, or template specializations.

3 Name Spaces and Scopes

Every symbol exists within a space of names, which is used by the compiler to map names to symbols. A scope is a context in which certain name spaces are accessable in code. Scopes can be nested to form a tree. Common scopes include function body, block, class, namespace, and the global scope. Figure

1 depicts a possible layout of scopes. Note that the 'namespace' language feature keyword of C++ (see Section 3.4) is only a specific kind of name space and is similar to the global scope.

3.1 Absence of Scopes

In more constrained languages like Assembly, user-declared names are unique to the entire program. No two definitions can have identical names. All names are fully qualified names. This makes it obvious what symbol a name refers to, which makes the compilation process marginally faster. The downside is that it forces users and organisations to write more lengthy, verbose, or confusing names as their programs and libraries get larger, which detracts from readability.

When a programmer defines a function or pre-processor definition in C, it effectively introduces a new keyword into the program's global scope. It is impossible to use the same name for a different section of the program. Thus, functions in the standard library such as `tan` can only have one meaning. If programmers can only use such short, vague names as part of library interfaces which are distributed to other code bases, they will inevitably overlap and cause naming conflicts.

A workaround for this limitation is prefixing the name with something to represent the module. An example is the prefix `gl` of the function `glBindBuffer` in the OpenGL API [8]. Symbolic prefixes introduce a kind of jargon, which decreases the readability for non-programmers. New programmers will not know that "gl" stands for "Graphics Library". [9]

3.2 Identical Names

C++ provides many language features that allow names to be re-used to refer to different symbols when used in different contexts. There are three situations where we may want to have multiple symbols that share an identical name.

1. Same name in the same name space or scope.
    - Function overloading.
    - Template specialization.
2. Same name in a nested scope.

- Name hiding and shadowing.

3. Same name in a separate scope.
   - Fully qualified names.
   - Bringing names into scope (`using` keyword).

These techniques reduce the chance of naming conflicts, but the problem was shifted from a per-symbol basis to a per-scope basis. We should still give long, descriptive names to large, complex symbols (especially namespaces) [10].

3.3 Name Hiding

If a symbol in a nested scope has the same name as a symbol in a parent scope, references in the nested scope will resolve to the symbol in the nested scope instead of the parent scope (with a few exceptions). Program 5 demonstrates how a local variable name will hide a global variable name. Program 6 demonstrates how a member variable name will hide a global variable name. Program 7 demonstrates how a parameter name will hide a class member variable name.

3.4 Namespaces

The namespace language feature of C++ lets users declare nested scopes for top-level symbols, such as those found in the global namespace. In order to use a symbol from another namespace, the name can be preceded by a path of namespace names separated by the scope operator (`::`) (similar to a file system path). A scope operator without a name on the left-hand side means the path is absolute and starts at the global scope. See Program 8 for an example.

Namespaces are unique in that they are not technically symbols, and we do not need to declare all its members in one block definition. We can add members to an existing namespace in a new block of code.

4 Translation Units

Every C++ source code file (.cpp) is compiled in a separate translation unit by the compiler, which outputs an object file (.obj) for each one which con-

tains the definitions of its symbols in machine code. In order to write parts of the program in separate source files, the linker combines the object files (and static link libraries) into one complete binary (output file full of machine code), like an executable (.exe), static link library (.lib), or dynamic link library (.dll).

A symbol from another translation unit can be used by declaring it with the same signature, without providing a definition (see Program 9). Translation units which are linked together must follow the One Definition Rule, which means there must only be one definition for each symbol (function, variable, and data type).

4.1 Static Free-Standing Functions

Functions at the namespace level (not in a class) marked `static` means their definition is unique to the current translation unit. This makes it possible to have the exact same function signature refer to different implementations, depending on which translation unit is being compiled (see Program 10). The One Definition Rule appears to be violated, but it merely limits the rule to a single translation unit. This is something to be aware of, because it may bloat the binary with duplicate functions even if they have the same definition. This is why we suggest using the `inline` keyword instead of `static`. Static free-standing functions should be used in as few places as possible, and never appear in header files.

4.1.1 Static Class Members

Confusingly, the `static` keyword is used in class definitions to mean something completely different. In data types, static member functions and static member variables are unique to the type itself rather than instances. Static member functions have nothing to do with translation units, and they must follow the One Definition Rule across the whole linked binary.

4.2 Anonymous Namespaces

Anonymous namespaces are unique to their translation unit, similar to static free-standing functions. They should not be used in header files, lest they cause bloat in the binary. An anonymous namespace is declared like a namespace without an identifier. Every anonymous namespace declaration declares

a new namespace; it does not add members to an existing nameless namespace.

5 Pre-Processor

The Pre-Processor is a phase of the compilation process which takes a stream of tokens as input (sequence of text words) which is produced by the lexical analysis phase. The Pre-Processor produces pure C++ code ready for the compilation phase, by replacing and manipulating certain parts of the token stream. Pre-processor directives take up a single line of code and begin with the pound character (#). The line can be extended over multiple lines by placing a single backslash (\) at the end of the line, to tell the lexer to ignore the newline.

5.1 Pre-Processor Definitions

Pre-processor definitions introduce keywords into the source code. After declaring a pre-processor definition with a name, all places where that name is found in the code gets replaced with a block of text. The identifier cannot be re-used for different contexts in the language because the pre-processor knows of only one context: the stream of tokens. This is one of the reasons why using pre-processor definitions is considered inferior to using language features like compile-time constant variables, constexpr functions, and templates, which are subject to the syntax of the language [11]. Program 11 is an example of a pre-processor definition with a problematic name. To avoid naming conflicts between symbol names and pre-processor definitions we must revert back to the practice of verbose names and strict naming conventions. We can reduce the risk by reserving `UPPER_CASE` names purely for pre-processor definitions.

Despite its limitations, we believe pre-processor definitions provide an important language feature for experienced programming pioneers. They should remain in the ecosystem of C++ and only be used in the few situations where they are most needed.

5.2 Include Statement

The pre-processor replaces each `#include` pre-processor directive with the contents of a text file specified by a file path in quotation marks (`"filename"`) or angle brackets (`<filename>`) [12]. Program 12 demonstrates using the pre-processor `#include` statement to include a text file as source code. By convention, angle brackets indicate it is a library header file (from the standard library or a third-party library) and quotation marks indicate it is a file specific to the current project. For a table of file extensions and conventional meanings, see Table 1. Header files, which use the `#include` directive, will be explained in Section 9.

6 Functions

6.1 Function Signatures

Function prototypes have a signature, which is used to determine which function symbol the prototype is declaring. Even though two function prototypes may look different at a glance, they could have the same signature and therefore the name will be resolved to the same function symbol (see Program 13). In most cases, two function prototypes have different signatures if any of the following are true:

- Different name.
- Different number of parameters.
- Different parameter type (corresponding to position in parameter list).
- Different return type.
- Different scope (namespace, class member).
- Static vs non-static member functions.
- Const-qualifier for member functions.

Figure 2 gives an overview of the order of identifier resolution for function calls.

The names of parameters are ignored in function signatures. This is probably the reason why C++ does not allow named parameters at the call site, because of multiple function prototypes with different parameter names.

For a function parameter without a reference type, a `const`-qualified type is considered equal to the corresponding non-`const`-qualified type, because when the function is called the arguments are stored on the stack in the same way, and how they are used in the function is an implementation detail (of no consequence to the caller). Similarly, a parameter with an array type is considered the same as a parameter with a raw pointer type (again see Program 13).

6.2 Function Pointers

A specific function can be aliased by storing its address in a function pointer variable, similar to reference variables (see Program 14). In certain cases where the function name is overloaded, it may require a `static_cast`. As with other raw pointers, it would be a good idea to mark the function pointer variable `const`. The compiler can perform optimisation to remove overhead caused by the function pointer variable, especially if it is only used as an alias to a specific function.

7 Function Overloading

Functions can be overloaded when two or more function declarations with the same name in the same name space have different signatures (see Program 15). At the call site, identifier resolution determines which function symbol (implementation) to use by checking the types of the arguments (see Figure 2). This may involve implicit conversion of argument types to match the best fitting function signature [13].

7.1 Name Mangling

One role of the linker is to link function calls in one translation unit to the implementation in another translation unit. The C++ linker can only process symbols with unique names, like in assembly or C. Therefore, the names of symbols in the C++ source code must be converted to unique identifiers for linking. In certain cases C programs may need to call C++ functions, but

C names are never mangled. Name mangling is when a function is given a unique name based on its signature. The mangling process may encode information such as the parameter types and their order, possibly yielding a semi-readable name, or it may completely mangle it beyond readability for ease of linking. Name mangling can be disabled for specific function declarations by using C linkage via the `extern "C"` language feature (see Program 16).

7.2 Argument Dependent Lookup

There is a special case of identifier resolution for free standing functions which are declared in the same namespace as the definition of the type of one of its arguments. We can call the function just by its name without bringing it into scope or using a fully qualified name, and argument dependent lookup will understand which function to use because the type of its argument is defined in the same namespace as the function (see Program 17). If the function is not defined in the same namespace as the type of one of its parameters, ADL does not apply (see Program 18).

7.3 Operator Overloading

Just like functions with names, most operators can be overloaded to accept different operand types (see Program 19) [14]. The function prototype for an overloaded operator can be declared as a as a free-standing function or as a class member function (with the `this` pointer being the implicit left hand operand) (see Table 2). Some operators can only be overloaded as member functions, such as the assignment, function call, and subscript operators. Some operators cannot be overloaded, such as punctuation operators like the scope operator (`::`) and dot operator (`.`).

8 Aliases

An alias is an alternative name which refers to the same symbol, and therefore the two names are indistinguishable.

8.1 Typename Aliases

An alias for a typename can be declared with a `typedef` statement or a `using` statement. The type of an expression can be retrieved with the `decltype` keyword. See Program 30.

8.2 Reference Variables

A variable with a reference type can be considered an alias. A reference variable is a variable which stores the address of the original variable's data in memory (as if it were a const pointer to the original) and performs indirection for operations automatically. The aliasing reference variable can be optimised away by the compiler. See Program 31.

8.2.1 Class Member Variable Aliases (or lack thereof)

There is no way to declare a true alias for a class member variable, because adding a reference member variable to the class will increase the size of the structure and cause problems with lifetime semantics (see Program 32). A workaround is declaring member functions to get and set the value. Do not use pre-processor definitions, for reasons discussed in Section 5.1.

8.2.2 Unions as Class Member Variable Aliases

One situation where this is a problem is for a mathematical vector class. Many graphics libraries provide a 3D vector class (such as `vec3` of the GLM library) which has properties X, Y, and Z, which are accessed like member variables. They might also provide R, G, and B properties to let the vector represent a color with red, green, and blue components.

In order to use both sets of properties like member variables, libraries have leveraged the knowledge of how the compiler implements the layout of union types in memory. The standard states that it is undefined behaviour to access one union member variable name when the union's current value was assigned to a different member variable name. However, some implementations for certain platforms (including MSVC for Windows x86/64) store union member variables of the same type in the exact same memory layout, so the behaviour is predictable. This opens up the possibility to create class member variable aliases, at the cost of cross-platform compatibility. See Program 33 for a simplified example.

8.3 Aliasing Functions

We can bring a function name (overload set) into scope with a `using` statement (see Program 34). This can be used with Arguemnt Dependent Lookup (see Section 7.2) to call the right function depending on template type arguments. A common use case is swapping variables with `std::swap`.

Alternatively, we can write a function that forwards all its arguments to the original function. Perfect forwarding is achieved by paying attention to lifetime semantics for types such as `std::unique_ptr<T>`. Perfect forwarding makes for verbose code, but it provides better opportunities for optimisation by the compiler. See Program 35.

9 Header Files

Translation units are great, and the pre-processor makes it easier to have multiple files. If the pre-processor did not exist, we would have to forward declare everything we use from other translation units for each file (see Program 20). This process would be tedious and the code would be fragile. Since it is mostly duplicate code, the pre-processor can do it for us (see Program 21). [15]

By convention, source code files (implementation files) have the extension '.cpp', and header files have the extension '.h' or '.hpp' (see Table 1).

9.1 Header Guards

A header guard makes sure the content of the file is only compiled once for each translation unit, regardless of how many times the file has been included (see Program 22). Since it uses a pre-processor definition, the name should be verbose in order to avoid naming conflicts. One convention is to use the filename of the header.

There is a special case of the One Definition Rule for defining data structures. They cannot be defined more than once *per translation unit*. Multiple source code files can include the same header file containing a class definition. Using a header guard will ensure it is only ever defined once per translation unit.

9.2 Namespace Pollution in Header Files

The way namespaces are used in C++ is hazardous to the health of identifiers. Namespace pollution occurs when a programmer brings names into the scope of a namespace in a header file of their library's public interface. The most obvious example is '`using namespace std;`' at the top of a header. It forces all other files which `#include` the header to also have access to those names brought into that namespace scope. To use symbols in a header without polluting the namespace, we must use fully- or semi-qualified names. This approach causes the header to become more verbose (see Program 23). This in turn encourages C++ programmers to use very short names for namespaces, the most common being `std` which represents the "Standard Template Library".

C++ lacks a language feature which would allow names to be brought into scope in a restricted section of code, in such a way that they are not considered members of the scope when viewed from other sections of code in the same namespace. Such a feature would make it easier for programmers to use longer, more descriptive names for namespaces. [9]

10 Class Members

10.1 Non-Static Member Indirection

A non-static member of an instance of a class can be accessed in two different ways (see Program 24). Given a left-hand operand, if it is a reference to an object, then the dot operator (`.`) can be used. If the left-hand operand type is a pointer to an object, the member indirection "arrow" operator (`->`) can be used. Unlike other operators, the right-hand operand is the name of the member (not a value expression).

The keyword `this` can be used in the scopes of non-static member functions of a data structure. For a type T, `this` is a pointer `T*` to the current instance of T upon which the member function is being invoked. It also takes into account the const qualification of the instance at the call site [16].

10.1.1 Parameter Names Hide Member Names

Member function parameter names (in the function scope) will hide member names (in the class scope), as seen in Program 7. To avoid this, we can

follow a naming convention such as "member variables must begin with `m_` for non-static or `s_` for static," or we can use the `this` keyword.

10.2 Overloading by Const

Member functions can be overloaded by their const qualifier (which appears after the parameter list), because the signature is different depending on whether it is const qualified (see Program 25). This is used for situations like `operator[]` which would return a const reference when the object is const, or a mutable reference when the object is mutable, to allow the call site to assign a new value to the array element.

Similarly (but rarely used), r-value reference qualified member functions have a different signature to l-value reference qualified member functions.

10.3 Static Members

A static member of a class is a member of the class type itself rather than a specific instance, and can be accessed with the scope operator (`::`) (see Program 26) [16].

Static members can also be accessed from an instance with the dot operator (`instance.static_member()`), which is syntactic sugar and means (`decltype(instance)::static_member()`). This quirk of the language makes it impossible to overload by static/non-static, because we cannot declare a static member function with the same name and parameter types as a non-static member function, which is a shame.

10.4 Pointer to Member

Similar to value pointers and function pointers, a pointer to a member of a class can be used with an instance of the class to dereference different members at runtime (see Program 27).

10.5 Inheritance

A class that inherits from another class will add all the non-private properties of the base class into its own scope (name space) (see Program 28). When viewed from outside the class, the inherited properties are subject to the access permission of how it is inherited (see Program 29). [17, 1, 18]

11 Conclusion

The language of C++ and its surrounding environment provides many ways for programmers to write their own programs built from symbols with names. Variables, functions, and data types exist as symbols in scopes. Scopes can be nested to form a tree of namespaces, classes, functions, and control flow blocks. When a name is used at the call site, the compiler will perform identifier resolution to determine the specific symbol that it is referring to, based on the syntax of the language.

Different symbols can have identical names, even in the same scope. Functions can be overloaded so that for one name, different function implementations can be called depending on the types of the arguments at the call site. The function call is checked against the overload set of function signatures to determine the best fitting function to invoke.

Declaring a name in a nested scope will hide symbols with the same name in all parent scopes. In order to use those parent scope symbols, they can either be brought into scope, or referenced with a fully-qualified name, or resovled with Argument Dependent Lookup.

Continuing the theme of separating sections of code, we can write programs comprised of multiple source code files, which use forward declarations in header files to link the translation units together, to build a complete binary package (executable, static library or DLL).

We are given a plethora of choices for how we can write our programs. It is up to us to make them simple, readable, and useful. Concluding this article, we hope that you have gained a better understanding of the foundational language features of the C++ language. We encourage you to experiment with the examples and programs we have provided. The more knowledge you have of the confusing details of C++, the less likely you will be to make mistakes in important projects.

12 Related Areas

- Type inference. The auto keyword.
- Virtual member functions.
  - Unintentional shadowing of non-virtual member function instead of overriding.

  - The override and final keywords.
  - Calling the base class' implementation of a virtual member function from an overriding function in a derived class.
  - Private virtual member functions can be overridden in derived classes, but not invoked.

- Templates.
  - Structural typing.
  - Template specializations.
  - Explicit template specialization.
  - Template argument deduction.

- Typename vs Value.
  - Class member: type vs static value. The typename keyword.
  - Operator vs template parameter list. The template keyword.

- Control flow block initializers for local scope variables.

13 References

[1] B. Stroustrup, "The design of c++." Video. University Video Communications, Mar. 1994. Retrieved May. 2020. https://youtu.be/69edOm889V4.

[2] J. Bloch, "How to design a good api and why it matters." Video. Google Tech Talks, Jan. 2007. Retrieved May. 2020. https://youtu.be/aAb7hSCtvGw.

[3] B. Stroustrup, "What should we teach new software developers? why?," *Communications of the ACM*, vol. 53, pp. 40–42, Jan. 2010. Retrieved May. 2020. https://cacm.acm.org/magazines/2010/1/55760-what-should-we-teach-new-software-developers-why/fulltext.

[4] ISOCPP, "2020 annual c++ developer survey "lite"," tech. rep., Standard C++ Foundation, Jan. 2020. Retrieved May. 2020. https://isocpp.org/files/papers/CppDevSurvey-2020-04-summary.pdf.

[5] U. Kirch-Prinz and P. Prinz, *A Complete Guide to Programming in C++*. Jones and Bartlett Publishers, 2002. ch. 2. Retrieved May. 2020. http://www.lmpt.univ-tours.fr/~volkov/C++.pdf.

[6] R. Pate, "What are the rules about using an underscore in a c++ identifier?." Website, Oct. 2018. Retrieved May. 2020. https://stackoverflow.com/a/228797/8567526.

[7] S. Sunil, "What are forward declarations in c++." Website. Geeks For Geeks. Retrieved Apr. 2020. https://www.geeksforgeeks.org/what-are-forward-declarations-in-c/.

[8] "Opengl - the industry's foundation for high performance graphics." Khronos Group, 1997. Retrieved March. 2020. https://www.opengl.org/.

[9] K. Gregory, "Naming is hard: Let's do better." Video. CppCon, Sept. 2019. Retrieved June. 2020. https://youtu.be/MBRoCdtZOYg.

[10] B. Stroustrup and H. Sutter, "C++ core guidelines: Es.7." Online Living Document, 2019. Retrieved May. 2020. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[11] B. Stroustrup and H. Sutter, "C++ core guidelines." Online Living Document, 2019. Retrieved May. 2020. https://github.com/isocpp/CppCoreGuidelines.

[12] B. Kolpackov, "Practical c++ modules." Video. CppCon, Sept. 2019. Retrieved May. 2020. https://youtu.be/szHV6RdQdg8.

[13] J. Turner, "C++ code smells." Video. NDC Conferences, Sept. 2019. Retrieved May. 2020. https://youtu.be/nqfgOCU_Do4.

[14] M. Gregoire, S. Kleper, and N. Solter, *Professional C++*. Wrox, 2 ed., 2011. ch. 18. Retrieved June. 2020. http://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=452868&site=ehost-live.

[15] B. A. Lelbach, "Modules are coming." Video. Meeting C++, 2019. Retrieved May. 2020. https://youtu.be/yee9i2rUF3s.

[16] M. Gregoire, S. Kleper, and N. Solter, *Professional C++*. Wrox, 2 ed., 2011. ch. 7. Retrieved June. 2020. http://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=452868&site=ehost-live.

[17] U. Kirch-Prinz and P. Prinz, *A Complete Guide to Programming in C++*. Jones and Bartlett Publishers, 2002. ch. 23. Retrieved May. 2020. http://www.lmpt.univ-tours.fr/~volkov/C++.pdf.

[18] B. Nystrom, "Is there more to game architecture than ecs?." Video. Roguelike Celebration, Oct. 2018. Retrieved May. 2020. https://youtu.be/JxI3Eu5DPwE.

Appendices

Appendix A Programs

All these programs are console executables and can be built with Visual Studio. Most of these programs are unit tests, which do not output anything to the console unless the test failed in debug build configuration.

Program 1: Declaring, defining, and referencing variables

```
#include <cassert>
#include <string>

// Variable with static storage duration.
// Type is `int`.
// Name is `y`.
// Declaration without definition.
extern int y;

int main()
{
    // Variable with local storage duration,
    // stored on the stack.
    // Declaration and definition.
    // Initialised to the value of an empty string.
    std::string x;

    // Variable referenced by name.
    // Assignment operation to modify
    //  the value of the variable.
    x = "peach";

    // Reading the value of x and comparing it.
    assert("peach" == x);
}
```

Program 2: Declaring, defining, and referencing functions

```
#include <cstdio>

// Function prototype (forward declaration).
void my_function(int my_parameter);
```

```
int main()
{
    // Function call.
    // Resolves to a specific function symbol
    //  named `my_function` which accepts
    //  one argument of type `int`.
    // This is the call site.
    my_function(5);
}

// Function implementation (definition).
void my_function(int my_parameter)
{
    printf("the␣number␣is:␣%d\n", my_parameter);
}
```

Program 3: Declaring, defining, and referencing a class typename

```
#include <vector>
#include <cassert>

// Class declaration and definition.
class Apple
{
public:
    // Declare that `Apple` has a
    //  member variable named `x`.
    int x;

    // Declare member function with
    //  signature `int Apple::getX() const`.
    // Also define its implementation inline.
    int getX() const { return x; }

    // Declare a constructor prototype for `Apple`.
    Apple();
};

// Implementation for constructor of my_class.
// Initialises the x member variable.
Apple::Apple() : x() {}

int main()
{
    // Use the typename `Apple`.
    // Call its default constructor.
```

```
    Apple instance = Apple();
    instance.x = 3;
    assert(3 == instance.getX());

    // Use the typename 'std::vector<int>'.
    std::vector<int> vec{ 3, 4, 5 };
    assert(3 == vec[0]);
}
```

Program 4: Forward declaration of a class

```
#include <memory>
#include <cassert>

// Forward declare class Apple.
class Apple;

// Use Apple indirectly with a reference type.
int getX(Apple& apple);

// Use Apple indirectly with a pointer type.
std::unique_ptr<Apple> theApple;

// Finish defining the class.
class Apple {
public:
    int x;
};

// Now that the class is defined,
//  we can dereference the reference.
int getX(Apple& apple) { return apple.x; }

int main()
{
    // Perform oprations on instances
    //  of the Apple class.
    theApple = std::make_unique<Apple>();
    theApple->x = 3;
    assert(3 == getX(*theApple));
}
```

Program 5: Local variable names hide global variable names

```
#include <cassert>
```

```
// variable in global scope
int const apples = 0;

int main()
{
    // same name but in nested scope
    int const apples = 1;
    assert(1 == apples);
    if (true)
    {
        int const apples = 2;
        assert(2 == apples);
    }
    assert(1 == apples);
    // fully qualified name to work around name hiding
    assert(0 == ::apples);
}
```

Program 6: Member variable names hide global variable names

```
#include <cassert>

int const apples = 0;

class basket
{
    int apples = 1;

public:
    int get_apples() const
    {
        return apples;
        // same as: return this->apples;
    }

    int get_all_apples() const
    {
        // needs explicit scope operator
        // because this->apples is closer in scope.
        return ::apples;
    }
};

int main()
{
    auto b = basket();
```

```
    assert(0 == apples);
    assert(1 == b.get_apples());
    assert(0 == b.get_all_apples());
}
```

Program 7: Parameter names hide member names

```
#include <cassert>
class Basket {
private:
    int apples;
public:
    int get_apples() const {
        return apples;
    }
    // Parameter hides member variable.
    void set_apples(int apples) {
        this->apples = apples;
    }
    // Constructor initializer list
    // already knows the difference.
    explicit Basket(int apples) : apples(apples) {}
};
int main() {
    Basket basket = Basket(1);
    assert(1 == basket.get_apples());
    basket.set_apples(3);
    assert(3 == basket.get_apples());
}
```

Program 8: Nested namespaces

```
namespace alpha
{
    constexpr int apples = 1;

    namespace beta
    {
        constexpr int apples = 2;
        static_assert(2 == apples, "");

        // reach out into a parent namespace
        static_assert(1 == alpha::apples, "");
    }

    static_assert(1 == apples, "");
```

```
    namespace gamma
    {
        static_assert(1 == apples, "");

        // reach over into a sibling namespace
        static_assert(2 == beta::apples, "");
    }

    // add members to the namespace beta
    namespace beta
    {
        static_assert(2 == apples, "");
    }
}
// fully qualified name
static_assert(2 == ::alpha::beta::apples, "");
int main(){}
```

Program 9: Linking

```
/// main.cpp
#include <cassert>

// No implementation here, therefore
//  it must be in another translation unit.
double pi();

int main()
{
    assert(3.1 < pi() && pi() < 3.2);
}

/// pi.cpp
// Implementation in this translation unit.
double pi() { return 3.14; }
```

Program 10: Static Free-Standing Functions

```
/// main.cpp
#include <cassert>

static int seeds() {
        return 1;
}
int apples();
```

```
int main() {
        assert(3 == apples());
        assert(1 == seeds());
}

/// other.cpp
// Same prototype, different implementation.
static int seeds() {
        return 3;
}

int apples() {
        return seeds();
}
```

Program 11: Pre-processor definition substitution

```
#include <cassert>
#define pi (3.14)
int main()
{
    assert(pi > 3.0 && pi < 4.0);
    // The line above gets pre-processed into:
    assert( (3.14) > 3.0 && (3.14) < 4.0);

    // This will not compile:
    //int *pi = nullptr;

    // It would be pre-processed into:
    //int * (3.14) = nullptr;

    // This is why it would be a good idea to
    //  rename `pi` to `MY_LIBRARY_PI`,
    //  or better yet use constexpr.
}
```

Program 12: Include statement

```
/// file: three.txt
int get_three() { return 3; }

/// file: main.cpp

// copy all the contents of the file cassert
//  and insert it in place of this line.
```

```
#include <cassert>

// copy all the contents of three.txt
//  and insert it in place of this line.
#include "three.txt"

int main()
{
    assert(3 == get_three());
}
```

Program 13: Function signature equality

```
#include <cassert>

// Declare function with signature
// 'int my_function(int)'.
int my_function(int alpha);

// We can have multiple declarations
// of the same function signature.
int my_function(int);
// Exact same signature as above.
int my_function(int const beta);

// Forward declared with array parameter type.
int last(int data[], int length);

int main()
{
    assert(2 == my_function(0));

    int data[3]{ 0, 1, 2 };
    assert(2 == last(data, 3));
}

// Implementation (One Definition Rule).
int my_function(int)
{
    return 2;
}

// Different signature (overloaded function).
int my_function(long alpha);

// Incorrect signature (won't compile).
```

```
//long my_function(int);

// An array parameter is just a pointer parameter.
// Same signature as forward declaration.
int last(int* data, int length)
{
    return data[length - 1];
}

// `void thing()` and `void thing(void)`
// have the same signature.
void thing();
void thing(void);
```

Program 14: Function pointers

```
#include <cassert>

int originalFunction(int x) {
    return x * 2;
}

int main() {
    int (*functionPointer)(int) = originalFunction;
    assert(4 == functionPointer(2));
    assert(4 == (*functionPointer)(2));

    using fp_type = int(*)(int);
    fp_type const fp = originalFunction;
    assert(4 == fp(2));
}
```

Program 15: Overloading a function

```
#include <cassert>
#include <cstdint>

int apply(int a, int b) {
    return a + b;
}
// Different parameters, different implementation.
int apply(int a, long b) {
    return a - static_cast<int>(b);
}
// An overload can have a different return type.
char const* apply(char const* p, size_t len) {
```

```
    return p + len - 1;
}
// It can even have a different number of parameters.
int apply(int a) {
    return a;
}

// The overload set for `apply` in the global scope
// has four symbols:
//  int apply(int, int)
//  int apply(int, long)
//  char const* apply(char const*, size_t)
//  int apply(int)

int main() {
    // Resolves to `int apply(int, int)`.
    assert(8 == apply(6, 2));

    // Resolves to `int apply(int, long)`.
    assert(4 == apply(6, 2L));

    char const* const str = "hello world";
    // Resolves to `char const* apply(char const*, size_t)`.
    // The `int` argument was implicitly converted to `size_t`.
    assert('d' == *apply(str, 11));

    // Resoves to `int apply(int)`.
    assert(4 == apply(4));
}
```

Program 16: Using C linkage to avoid name mangling

```
/// main.cpp

// This function's name is mangled to
//  ?apples@@YAHHK@Z
//  in MSVC.
int apples(int, unsigned long) { return 3; }

extern "C" {
    // This function's name is not mangled.
    // Its name is `func`, and it can be
    //  declared and used in a C program.
    int func(int i) { return i + 1; }

    void run(void);
```

```
};

int main() { run(); }

/// run.c
#include <assert.h>
int func(int);
void run(void) {
        assert(2 == func(1));
}
```

Program 17: Argument Dependent Lookup

```
#include <cassert>

namespace fruit
{
    struct Apple {
        int m_seeds = 0;
    };

    int get_seeds(Apple const& apple) {
        return apple.m_seeds;
    }

    int add(int a, Apple const& apple, int b) {
        return a + apple.m_seeds + b;
    }
}

int main() {
    fruit::Apple apple = fruit::Apple();
    apple.m_seeds = 5;

    // The compiler must resolve `get_seeds`
    // with argument of type `fruit::Apple&`.
    // ADL detects it should use
    // `int fruit::get_seeds(fruit::Apple const&)`.
    assert(5 == get_seeds(apple));

    // ADL works with multiple parameters.
    assert(7 == add(1, apple, 1));
}
```

Program 18: ADL does not check different namespaces

```
#include <cassert>

namespace fruit
{
    struct Apple {
        int m_seeds = 0;
    };
}

namespace other
{
    int get_seeds(fruit::Apple const& apple) {
        return apple.m_seeds;
    }
}

int main() {
    fruit::Apple apple = fruit::Apple();
    apple.m_seeds = 5;

    // This will not compile.
    //assert(5 == get_seeds(apple));

    {
        // Instead of ADL, we can import
        // the name into this scope block.
        using other::get_seeds;
        assert(5 == get_seeds(apple));
    }
}
```

Program 19: Operator overloading

```
#include <cassert>

struct Vec2 {
    float x, y;

    // Overload addition-then-assignment operator.
    Vec2& operator+=(Vec2 const& rightHandSide) {
        x += rightHandSide.x;
        y += rightHandSide.y;
        return *this;
    }
```

```
    // Overload subtraction-then-assignment operator.
    Vec2& operator-=(Vec2 const& b) {
        x -= b.x;
        y -= b.y;
        return *this;
    }

    // Overload equal oeprator.
    bool operator==(Vec2 const& rightHandSide) const {
        return x == rightHandSide.x
            && y == rightHandSide.y;
    }
    // Overload not equal operator.
    bool operator!=(Vec2 const& b) const {
        return !(*this == b);
    }
};

// Overload addition operator.
Vec2 operator+(Vec2 leftHandSide, Vec2 const& rightHandSide) {
    leftHandSide += rightHandSide;
    return leftHandSide;
}
// Overload subtraction operator.
Vec2 operator-(Vec2 a, Vec2 const& b) {
    a -= b;
    return a;
}

int main() {
    Vec2 const alpha = Vec2{ 3, 4 };
    Vec2 const beta = Vec2{ 0, 1 };
    assert(alpha != beta);

    Vec2 const gamma = Vec2{ 3, 3 };
    assert(beta + gamma == alpha);

    Vec2 delta = { 3, 4 };
    assert(delta == alpha && delta != beta);
    delta -= gamma;
    assert(delta != alpha && delta == beta);
}
```

Program 20: Using forward declarations to link different translation units

```
/// three.cpp
```

```
// This file gets compiled in one translation unit.
#include <cassert>
int get_three() { return 3; }
void check_four(int x)
{
    assert(4 == x);
}

/// main.cpp
// This file gets compiled in another translation unit.

// Function prototype with same signature
//  as in the first translation unit.
int get_three();
void check_four(int x);

int main()
{
    check_four(get_three() + 1);
}
```

Program 21: Using header files to link different translation units

```
/// three.h
// This file is included by both source code files.
int get_three();
void check_four(int x);

/// three.cpp
// This file gets compiled in one translation unit.
#include <cassert>
#include "three.h"
// Implement the functions which were
//  declared in the header.
int get_three() { return 3; }
void check_four(int x) { assert(4 == x); }

/// main.cpp
// This file gets compiled in another translation unit.
#include "three.h"
int main()
{
    check_four(get_three() + 1);
}
```

Program 22: Header guards

```
/// basket.h
// If this file's guard is not defined...
#ifndef BASKET_H
// Define the guard and include the contents of this file.
#define BASKET_H
// (Else, do not include the contents of this file.)

class Basket {
private:
        int m_apples;
public:
        int get_apples() const;
        void set_apples(int value);
};

// Finish the header guard IF directive.
#endif


/// counter.h
#ifndef COUNTER_H
#define COUNTER_H

#include "basket.h"
int count_apples(Basket const& basket);

#endif // file


/// main.cpp
#include <cassert>
#include "basket.h"

// When this is expanded by the pre-processor,
//  it includes "basket.h" again
//  in the same translation unit.
// Without header guards,
//  this would cause a linker error.
#include "counter.h"

int main()
{
        auto basket = Basket();
        basket.set_apples(3);
```

```
        assert(3 == count_apples(basket));
}


/// basket.cpp
#include "basket.h"
int Basket::get_apples() const {
        return m_apples;
}
void Basket::set_apples(int value) {
        m_apples = value;
}


/// counter.cpp
#include "counter.h"
int count_apples(Basket const& basket) {
        return basket.get_apples();
}
```

Program 23: Avoiding namespace pollution

```
/// timers.h
#ifndef TIMERS_H
#define TIMERS_H

#include <chrono>

namespace timers
{
    // The fully qualified names are required
    // to avoid polluting this namespace
    // with symbols from other libraries.
    inline std::chrono::duration<float> time_between(
        typename std::chrono::system_clock::time_point const&
            start,
        typename std::chrono::system_clock::time_point const&
            end
    )
    {
        // Inside a scope which is not a namespace or a class,
        // we can bring symbols into scope without trouble.
        using namespace std::chrono;
        using D = duration<float>;
        return duration_cast<D>(end - start);
    }
```

```
}

#endif // file

/// main.cpp
#include <cstdio>
#include "timers.h"
int main() {
    using namespace std::chrono;
    using Clock = system_clock;
    auto const start = Clock::now();

    auto end = Clock::now();
    float total = timers::time_between(start, end).count();
    while (total < 0.2f)
    {
        end = Clock::now();
        total = timers::time_between(start, end).count();
    }

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

Program 24: Class member function indirection

```
#include <cassert>
namespace market {
    class Basket {
    private:
        int m_apples;
    public:
        // This non-static member function has signature
        //  void ::market::Basket::set_apples(int)
        void set_apples(int value) {
            this->m_apples = value;
            // same as
            //(*this).m_apples = value;
        }

        // This non-static member function has signature
        //  int ::market::Basket::get_apples() const
        int get_apples() const {
            return this->m_apples;
        }
    };
}
```

```
int main() {
    using namespace market;

    Basket basket;
    // Use dot operator on the `basket` reference
    //  and call
    //  `void ::market::Basket::set_apples(int)`
    //  on its value.
    basket.set_apples(3);

    Basket const* pBasket = &basket;
    // Use arrow operator on the `pBasket` pointer
    //  and call
    //  `int ::market::Basket::get_apples() const`
    //  on its dereferenced value.
    int apples = pBasket->get_apples();
    assert(3 == apples);
}
```

Program 25: Overloading non-static member functions by const qualifier

```
#include <cassert>
class Basket {
private:
    int m_value;
public:
    explicit Basket(int value) : m_value(value) {}

    // Calls resolve to this overload
    //  when used on mutable instances.
    int advance() { return ++m_value; }

    // Calls resolve to this overload
    //  when used on const instances.
    int advance() const { return m_value; }
};
int main() {
    Basket basket = Basket(0);
    assert(1 == basket.advance());
    assert(2 == basket.advance());

    Basket const& cref = basket;
    assert(2 == cref.advance());
    assert(2 == cref.advance());
}
```

Program 26: Static member functions

```
#include <cassert>
class Basket {
private:
    int m_apples;
public:
    explicit Basket(int apples) : m_apples(apples) {}

    // non-static member function.
    int apples() const { return m_apples; }

    // This static member function has signature
    //  int ::market::Basket::max_apples()
    static int max_apples() {
        return 4;
    }

    // This will not compile because of the
    //  non-static apples member function.
    //static int apples() { return 5; }
};
int main() {
    Basket basket = Basket(3);

    // Use scope operator on class type
    //  and call `int ::market::Basket::max_apples()`.
    assert(4 == Basket::max_apples());

    // Use dot operator on class instance
    //  and call static function
    //  `int ::market::Basket::max_apples()`.
    assert(4 == basket.max_apples());
}
```

Program 27: Pointer to member

```
#include <cassert>
struct Basket {
    int apples = 0;
    int oranges = 0;
    void set_apples(int value) { apples = value; }
    void set_oranges(int value) { oranges = value; }
};
int main() {
    using PFruit = int(Basket::*);
```

```
    PFruit pointerToFruit = &Basket::apples;

    using PSetter = void(Basket::*)(int);
    PSetter pointerToSetter = &Basket::set_apples;

    Basket basket;
    Basket* const pBasket = &basket;

    (basket.*pointerToSetter)(3);
    assert(3 == basket.apples);
    assert(0 == basket.oranges);
    assert(3 == (pBasket->*pointerToFruit));

    pointerToFruit = &Basket::oranges;
    pointerToSetter = &Basket::set_oranges;
    (basket.*pointerToSetter)(5);
    assert(3 == basket.apples);
    assert(5 == basket.oranges);
    assert(5 == (pBasket->*pointerToFruit));
}
```

Program 28: Inheritance and protected members

```
#include <cassert>

class Fruit {
private:
    // Private members are only accessible in
    // this class (and its member functions).
    int m_skinThickness;

protected:
    // Protected members are only accessible
    // in this class and classes which
    // inherit this class.
    int m_seeds;

public:
    // Public members are accessible
    // from anywhere.
    int get_fruit_seeds() const {
        return m_seeds;
    }
};

// Inherit the Fruit class and make all its
```

```
// public members publicly accessible
// from this class.
class Apple : public Fruit {
public:
    // Member functions of a
    // derived class can access
    // protected members.
    int get_apple_seeds() const {
        return m_seeds;
    }
    // The 'this' keyword includes
    // inherited properties.
    void set_apple_seeds(int value) {
        this->m_seeds = value;
    }
};

int main() {
    Apple apple = Apple();
    apple.set_apple_seeds(3);
    assert(3 == apple.get_apple_seeds());

    // We can access public properties of
    // inherited classes.
    assert(3 == apple.get_fruit_seeds());

    // Outside the class,
    // we cannot access protected properties.
    //assert(3 == apple.m_seeds);
}
```

Program 29: Changing access permission of inherited members

```
#include <cassert>

class Fruit {
private:
    int m_seeds = int();
public:
    // lifetime rule of 6
    ~Fruit() noexcept = default;
    Fruit(Fruit&&) noexcept = default;
    Fruit& operator=(Fruit&&) noexcept = default;
    Fruit(Fruit const&) = default;
    Fruit& operator=(Fruit const&) = default;
    Fruit() = default;
```

```
    explicit Fruit(int seeds) : m_seeds(seeds) {}
    int get_seeds() const { return m_seeds; }
};

class Apple : private Fruit {
public:
    using BaseType = Fruit;

    // Make all constructors of inherited class
    // publicly accessible from this class.
    using Fruit::Fruit;
    using Fruit::operator=;

    // Make all overloads of a
    // privately inherited member function
    // publicly accessible from this class.
    using Fruit::get_seeds;
};

int main() {
    Apple apple = Apple(3);
    assert(3 == apple.get_seeds());
}
```

Program 30: Aliasing typenames

```
#include <type_traits>

int main()
{
    typedef int alias_type;
    static_assert(std::is_same<int, alias_type>::value, "");

    using other = int;
    static_assert(std::is_same<int, other>::value, "");

    alias_type x = 5;
    static_assert(std::is_same<int, decltype(x)>::value, "");
}
```

Program 31: Using reference variables to alias other variables

```
#include <cassert>
int main()
{
```

```
    int originalVariable = 3;
    int& referenceVariable = originalVariable;
    referenceVariable += 4;
    assert(7 == originalVariable);
}
```

Program 32: Flawed attempt to alias a member variable

```
struct alpha
{
    int originalMemberVariable;
    int& evilVariable = originalMemberVariable;
};
//static_assert(sizeof(alpha) == sizeof(int),
//   "this fails because reference member variables "
//   "are stored as part of the struct");

static_assert(sizeof(alpha) == sizeof(int) + sizeof(int*), "");
int main() {}
```

Program 33: Aliasing class member variables with unions

```
#include <cassert>

struct vec3
{
    union { float x; float r; };
    union { float y; float g; };
    union { float z; float b; };
};
static_assert(sizeof(vec3) == 3 * sizeof(float), "");

int main()
{
    vec3 v;
    v.x = 3.0f;
    assert(3.0f == v.r);
}
```

Program 34: Bringing a function overload set into scope

```
#include <cassert>

namespace three {
    int get_apples() { return 3; }
}
```

```
int main() {
    using three::get_apples;
    assert(3 == get_apples());
    static_assert(&get_apples == &three::get_apples);
}
```

Program 35: Perfect Forwarding

```
#include <cassert>
#include <memory>
#include <iostream>

// Class which logs all lifetime semantics.
template<class ValueType>
class LifeLogger {
private:
    ValueType m_value;

public:
    ValueType& ref_value() { return m_value; }
    ValueType const& ref_value() const { return m_value; }
    ValueType copy_value() const { return m_value; }

    ~LifeLogger() noexcept
    {
        std::cout << "~(" << m_value << ")\n";
    }

    LifeLogger(LifeLogger&& b) noexcept
        : m_value(std::move(b.m_value))
    {
        std::cout << "+move(" << m_value << ")\n";
    }

    LifeLogger& operator=(LifeLogger&& b) noexcept
    {
        std::cout << "(" << m_value << ")=move(" << b.m_value <<
            ")\n", m_value;
        m_value = std::move(b.m_value);
        return *this;
    }

    LifeLogger(LifeLogger const& b)
        : m_value(b.m_value)
    {
```

```
        std::cout << "+copy(" << m_value << ")\n";
    }

    LifeLogger& operator=(LifeLogger const& b) {
        m_value = b.m_value;
        std::cout << "=copy(" << m_value << ")\n";
        return *this;
    }

    void swap(LifeLogger& b) noexcept
    {
        std::cout << "swap(" << m_value << ", " << b.m_value <<
            ")\n";
        using std::swap;
        swap(m_value, b.m_value);
    }

    LifeLogger() : m_value()
    {
        std::cout << "+()\n";
    }

    explicit LifeLogger(ValueType value)
        : m_value(std::move(value))
    {
        std::cout << "+(" << m_value << ")\n";
    }
};

template<class ValueType>
inline void swap(LifeLogger<ValueType>& a, LifeLogger<ValueType
    >& b) noexcept
{
    a.swap(b);
}



using Creature = LifeLogger<int>;

Creature originalFunction(Creature x)
{
    std::cout << "originalFunction\n";
    return Creature(x.copy_value() * 2);
}
```

```
// Basic forwarding by copying arguments (not moving).
// The extra function call can be optimised away.
inline Creature copyForwardingFunction(Creature x)
{
    return originalFunction(x);
}

// Perfect forwarding of arguments and return type.
template<class T>
decltype(auto) perfectForwardingFunction(T&& x)
{
    return originalFunction(std::forward<T>(x));
}

//  Requires C++17 for automatic type
//   deduction keyword 'decltype(auto)'.
auto const perfectForwardingLambda =
[](auto&& x) -> decltype(auto)
{
    return originalFunction(
        std::forward<decltype(x)>(x));
};

// Perfect forwarding for C++11.
template<class T>
decltype(originalFunction(std::declval<T&&>()))
cpp11PerfectForwardingFunction(T&& x)
{
    return originalFunction(std::forward<T>(x));
}



int main()
{
    std::cout << "\n[originalFunction]\n";
    {
        auto const r = originalFunction(Creature(2));
        assert(4 == r.copy_value());
    }

    std::cout << "\n[copyForwardingFunction]\n";
```

```
    {
        auto const r = copyForwardingFunction(Creature(2));
        assert(4 == r.copy_value());
    }

    std::cout << "\n[perfectForwardingFunction]\n";
    {
        auto const r = perfectForwardingFunction(Creature(2));
        assert(4 == r.copy_value());
    }

    std::cout << "\n[perfectForwardingLambda]\n";
    {
        auto const r = perfectForwardingLambda(Creature(2));
        assert(4 == r.copy_value());
    }

    std::cout << "\n[cpp11PerfectForwardingFunction]\n";
    {
        auto const r = cpp11PerfectForwardingFunction(Creature
            (2));
        assert(4 == r.copy_value());
    }
}
```

Appendix B Tables

Table 1: File extensions

Extension	Conventional Meaning
.cpp	C++ source code file (implementation file)
.c .cc	C source code file
.h	Header file for C or C++
.hpp	C++ header file

Table 2: Operators which can be overloaded (incomplete table)
R for return type, T for class type, U for other type, T:: means member function.

Operator	Description	Example Prototype
==, !=, >, <, >=, <=	Comparison	`R operator==(T const&, U const&)`
+, −, ⋆, /, %	Algebraic	`R operator+(T, U)`
&, |, <<, >>	Bitwise	`R operator&(T, U)`
&&, ||	Logical	`R operator&&(T, U)`
+, −, ˜, !	Unary	`R operator!(T)`
=	Assignment	`T& T::operator=(U)`
+=, −=, ⋆=, /=, %=, <<=, >>=	Compound Assignment	`T& operator+=(T&, U)`
++ (prefix)	Pre-Increment	`T& operator++(T&)`
++ (postfix)	Post-Increment	`T operator++(T&, int)`
−− (prefix)	Pre-Decrement	`T& operator--(T&)`
−− (postfix)	Post-Decrement	`T& operator--(T&, int)`
⋆ (prefix)	Pointer Indirection	`R& operator*(T&)`
−>	Member Indirection	`R* operator->(T&)`
& (prefix)	Address-of	`R* operator&(T&)`
()	Function Call	`R T::operator()(U1, U2, ..., UN)`
[]	Subscript	`R& T::operator[](U)`

Appendix C Figures

Figure 1: Important Scopes

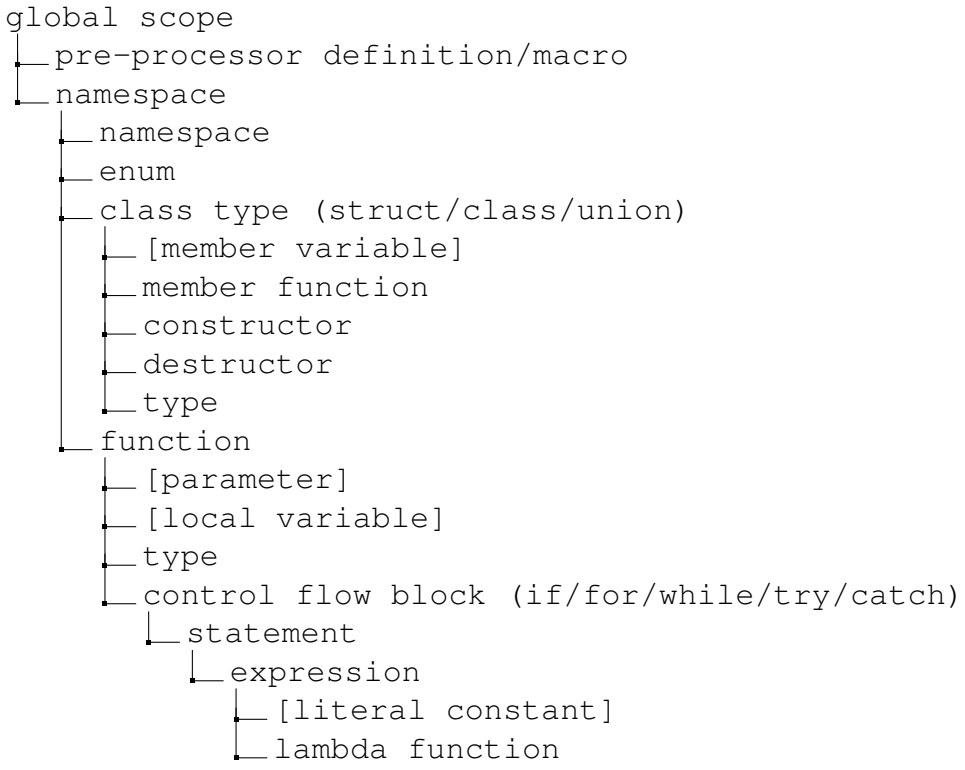

Figure 2: Function identifier resolution order

After recognising a function call from a name without additional scope operators (knowing it is not a pre-processor definition or keyword), the function symbol is resolved by checking...

- The types of the arguments:
    1. Identical parameter types.
    2. Template argument deduction.
    3. Almost identical, considering references and const qualifications.
    4. Implicit conversion to parameter types.
- Either (but not both):
    - Argument Dependent Lookup.
    - Names that are in scope. Current scope, or else parent scopes:
        1. Current scope.
        2. Block scope (callable variables).
        3. ... Block scopes ...
        4. Function scope (callable parameters).
        5. Class scope (member functions and callable member variables).
        6. Namespace scope, either (but not both):
            * Current namespace.
            * Anonymous namespace in the current namespace.
        7. ... Namespaces ...
        8. The global namespace scope.