C++

Muchang Bahng

Winter 2024

Contents

1 Basics **2**
- 1.1 Scope and Duration . . . 2
- 1.2 Internal and External Linkage . . . 4

2 Translation **4**
- 2.1 Preprocessing . . . 4
- 2.2 Compilation . . . 5
- 2.3 Linking . . . 6
  - 2.3.1 Internal vs External Linking . . . 6
- 2.4 Header Files . . . 6
- 2.5 Namespaces . . . 8

3 Types and Expressions **10**
- 3.1 Casting . . . 10
- 3.2 Enums . . . 10

4 Constants and Constant Expressions **10**
- 4.1 Compiler Optimization . . . 10
- 4.2 Constants . . . 11
- 4.3 Compile-Time Programming . . . 12

5 Control Flow and Error Handling **13**
- 5.1 If Statements . . . 13
- 5.2 Switch Statements . . . 13
- 5.3 Assert and Static Assert . . . 13

6 Functions **13**
- 6.1 Inline Functions . . . 13
  - 6.1.1 Inline Variables . . . 14
- 6.2 Overloading . . . 14
- 6.3 Deleting . . . 14
- 6.4 Default Arguments . . . 14
- 6.5 Template Functions . . . 15
- 6.6 Non-Type Template Parameters . . . 17

1 Basics

We define a bunch of terms. This may seem unnecessary, but it becomes very useful when getting into the weeds of C++.

Definition 1.1 (Variables)

1. A **literal** is a value that is directly inserted into code, e.g. `5`, `3.2`, `'a'`.[a]
2. A **variable** can be assigned to a literal, e.g. `x`.
3. The **identifier** is simply the name of the variable, and its **linkage** determines whether a declaration of that same identifier in a different scope refers to the same object.
4. An **operation** consists of an **operator** (+) and one or more **operands** (`3`, `4.3`).
5. An **expression** is simply a line of code containing variables, operations, and literals, e.g. `int x = 3 + 4;`
6. The process of executing an expression is called **evaluation**.

Variables can be **constructed** in two ways.

1. We first **declare** a variable, which tells the compiler about the existence of the variable (`int x;`). Then, we can **define** the variable, which assigns it a literal (`x = 4;`).
2. We can **initialize** a variable, which both declares it and defines it at once (`int x = 4;`).

Definition 1.2 (Functions)

1. The **declaration** of a function states the existence of the function.

```cpp
double foo(int x, double y); // one way
double foo(int, double); // another way
```

   This declaration is also called the **function prototype**, or the **function identifier**.
2. The **definition** of a function tells us the actual implementation.

```cpp
double foo(int x, double y) {
  ...
}
```

1.1 Scope and Duration

You probably know that there are two types of variables: **local variables** and **global variables**, along with their general properties. Let's specify them a bit, starting with the two most important ones: scope and duration.

Definition 1.3 (Duration)

The **duration** of an identifier governs how it will be constructed and destroyed, over its **lifetime**.

1. **Automatic duration** means that their lifetime begins at the start of the block (at `{`) and is destroyed at the end of the block `}`.
2. **Static duration** means they are created when the program starts (before `main()`) and destroyed when it ends. Variables with static duration, both local and global, are 0-intialized by default. (e.g. `static int x;` is really `static int x = 0;`). It is conventional to prefix static local variables with a `s_`.

[a] Usually, using literals by themselves such as `return 4;` should be avoided since the reader can't determine what it represents through the identifier name. We must resort to a comment or external documentation.

Definition 1.4 (Scope)

The **scope** of an identifier refers to where it is accessible by.

1. **Local variables** are variables constructed inside a **block scope** and are accessible only within that block. They have automatic duration by default, but can have static duration with the `static` keyword.

```cpp
int main() {
  int x = 2; // local variable
  static int x = 3; // static local variable
}
```

2. **Global variables** live within the **global scope** of the global or a local namespace and therefore can be accessible from anywhere. They are static variables by definition. Usually it is preferred to define global variables in a namespace. It is conventional to prefix global variables with `g_`.

It may seem like the scope and duration are related, but you can have any combination of automatic local variables (just called local variables), static local variables, and global variables. The duration talks about *when* a variables is allowed to live while the scope talks about *where* it is accessible from. Just like local variables, global variables can be const as well, and like all const variables, must be initialized.

Example 1.1 (Automatic Duration)

You can see that every block contains its own scope with its own local variables. When the block ends all variables in this block are destroyed on the stack.

```cpp
int main() {
  int x = 2;
  std::cout << x << std::endl; // 2
  {
    int y = 1;
    std::cout << x << std::endl; // 2
    std::cout << y << std::endl; // 1
  }
  return 0;
}
```

Example 1.2 (Accessing Parent Block Scope)

Local variables in a nested block have access to the parent block's scope. Both of these programs are valid. The left accesses the parent block's `x` while the right one access `x` newly created in the local scope.

```cpp
int main() {
  int x = 2;
  for (int i = 0; i < 10; i++) {
    x += 1;
  }
  std::cout << x << std::endl; // 12
  return 0;
}
.
```

```cpp
int main() {
  int x = 2;
  for (int i = 0; i < 10; i++) {
    int x = 1;
    x += 1;
  }
  std::cout << x << std::endl; // 2
  return 0;
}
```

Example 1.3 (Variable Shadowing)

You can see that the `x` is initialized in the `main()` block scope, but it gets “shadowed” by the `x` in the nested block. Once the block terminates, then it is “revealed” again.

```cpp
int main() {
  int x = 2;
  std::cout << x << std::endl; // 1
  {
    int x = 1;
    std::cout << x << std::endl; // 1
  }
  std::cout << x << std::endl; // 2
  return 0;
}
```

Static local variables are good for id generation, since they are not accessible beyond a block but still have a persistent state that does not get reset. Another good use is to use const static local variables for functions that needs to use a const value, but initializing that object is expensive. Using a local variable would instantiate it every time the function is called, but a static local variable requires us to create it once.

1.2 Internal and External Linkage

When `static` is applied to a global variable, it has a completely unrelated effect than that applied on a local variable. It means that the global variable now has internal linkage, meaning that the variable cannot be exported to other files.

2 Translation

Translating C++ code to a binary consists of multiple steps:

1. Preprocessing the code.
2. Compiling each file independently.
3. Linking all the files.

Conventionally, all of these are called *compiling*, but it really isn’t.

2.1 Preprocessing

When preprocessing, we do some boring stuff like removing comments. However, the main job is to take care of **preprocessing directives**, which are expressions with the # symbol. The most obvious is the `#include` directives, which **replaces the include directive with the contents of the included file**. That is, `#include` is really just a way to substitute code.

1. including with angle brackets, e.g. `#include <iostream>`, means that the compiler is looking for this file in the standard library files.
2. including with double quotes, e.g. `#include "tensor.h"`, means that the compiler is looking for this file locally in your project directory. It means you’ve written it.

Other directives is the `#define` directive.

1. You can define it to substitute text. It is conventionally in all upper-case.

```cpp
#define NAME "Muchang"  // all instances of NAME will be replaced with "Muchang"
```

2. Or you can define it without substitution text, where further occurrences of `NAME` will be replaced by nothing.

```
#define NAME
```

The second isn't used for substitution, but rather for **conditional compilation**, which can be useful. You just wrap C++ statements around as such.

```
#ifdef NAME
...
#endif
```

```
#ifndef NAME
...
#endif
```

To see the output after preprocessing, use the `-E` flag.

```
g++ main.cpp -E
```

2.2 Compilation

We only compile files one at a time and independently. When the compiler compiles a file, it goes through each line sequentially. Therefore, we must ensure that all functions/variables/classes are *declared* first before they are called. *Forward declaration* makes this a lot easier.

There is a difference between a declaration and a definition.

Definition 2.1 (ODR)

Remember the ODR (One Definition Rule):

1. Within a file, each function, variable, type, or template in a given scope can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined inside different namespaces) do not violate this rule.
2. Within a program, each function or variable in a given scope can only have one definition.[a]

To be honest, ODR 2 really implies ODR 1, since once the directives are preprocessed or the object files are linked, we are really left with one executable file.

Example 2.1 (ODR 1 Violation)

The following shows that in the same file, there are multiple variables defined in the function scope of `main`, and there are two definitions of `foo` in the global scope.

```
int main() {
  int x;
  int x;

  return 0;
}
.
```

```
int foo() { return 5; }
int foo() { return 5; }

int main() {
  std::cout << foo();
  return 0;
}
```

[a]This rule exists because programs can have more than one file. For example, if you have two definitions of `int add(int, int)` in two different files, the linker does not know which one to connect the declaration to.

Example 2.2 (ODR 2 Violation)

Say that `main.cpp` has the `main()` method that calls on `int add(int x, int y)`, which is forward declared. However, say that we define add in two places.

```cpp
// foo.cpp
int add(int x, int y) {
  return x + y;
}
```

```cpp
// bar.cpp
int add(int x, int y) {
  return x + y;
}
```

Then, if we run `g++ main.cpp foo.cpp bar.cpp`, the linker will complain that there is a function redefinition.

2.3 Linking

Remember, declaration is not the same thing as definition. When we do the linking, we go through all the source files in our project and match all the declarations with our definitions. The source files must all be written in the compile command.

```
g++ main.cpp add.cpp
g++ add.cpp main.cpp
```

This should not be order dependent. The source files can be

```cpp
// main.cpp
int add(int x, int y); // declaration

int main() {
  int z = add(2, 3);
  return 0;
}
```

```cpp
// add.cpp
// definition
int add(int x, int y) {
  return x + y;
}
.
.
```

2.3.1 Internal vs External Linking

2.4 Header Files

To be honest, we can just include forward declarations everywhere, but this does not scale well to large projects. If we had a set of declarations that we wanted to use over a bunch of files, we can package them nicely using a **header file**.

If we have a bunch of functions and classes written in `foo.cpp`, then it is conventional to write a `foo.h` that contains all the declarations of these expressions. Then, whenever we need to write a new file `bar.cpp` that uses functions from `foo.cpp`, we can just `#include "foo.h"`, which replaces this directive (by the preprocessor) with all the forward declarations in `foo.h`. Boom easy.

```cpp
// add.cpp
int add(int x, int y) {
  return x + y;
}
```

```cpp
// add.h
int add(int x, int y);
.
.
```

Therefore when we call `add` in `main.cpp`, we can just `#include "add.h"` to put in the declarations, making everything good. Conventionally, it is best practice for a source file to also include its paired header (e.g. `add.cpp` should also contain `#include "add.h"` at the top). This allows the compiler to discover inconsistencies between the two files, and this extra cost is negligible.[1]

[1]https://www.learncpp.com/cpp-tutorial/cpp-faq/#pairedheader

Example 2.3 (Definitions inside Header Files)

You should not add definitions (only declarations) to header files since if they are included in multiple header files, then we would have different definitions of the same function, leading to ODR 2 violation. Take a look at the following.

```cpp
// square.h
int getSquareSides() {
    return 4;
}
```

```cpp
// wave.h
#include "square.h"
```

With the following.

```cpp
#include "square.h"
#include "wave.h"
int main() {
  return 0;
}
```

This won't compile since

1. by including `square.h`, we have defined `getSquareSides()` in the global scope of `main.cpp`.
2. by including `wave.h`, we have included `square.h` which then substitutes this line with the definition of `getSquareSides()` again.

This is an ODR 1 violation.

The simple fix to the above is to just remove the `#include "wave.h"`, but what if we needed some other function from `wave.h`? Resolving this issue is not trivial if say, half of the functions in `square.h` is needed in `wave.h` and the other half is needed in `main.cpp`. We must include both of them in `main.cpp`, but then we have an inevitable redefinition. Without separating `square.h` into separate files, solving this is impossible.

Even if we didn't have definitions in header files in the first place (which is bad practice in general), repeated declarations, which are still fine, are also not really ideal either. Furthermore, custom types are typically defined in header files, so redefining them leads to an ODR violation.

Definition 2.2 (Header Guards)

Fortunately, we have **header guards**, which are conditional compilation directives that tell the compiler to include a header file at most once to the main file. You can do this in two ways.

1. Just put this to the top of the header file. The compiler will take care of redeclaration/redefinitions for you. This isn't always fail-safe.

   ```cpp
   #pragma once
   ```

2. More manually, we can use a conditional compilation directive. Put this on the top of the header.

   ```cpp
   #ifndef HEADERFILE_H
   #define HEADERFILE_H

   ...Header Contents...

   #endif
   ```

   In the beginning, `HEADERFILE_H` is not defined, so we include all of this. In a second inclusion though, `HEADERFILE_H` is defined, so the preprocessor removes this.

Note that header guards limit the number of times a header can be included in a single given file, but the header may still be repeated across separate project files. This is what we want.

2.5 Namespaces

Perhaps we want to have two functions of the same name, but we get a redefinition error. This is where namespaces come in.

Definition 2.3 (Namespace)

We can wrap each function around a **namespace**, which is written with an upper-case letter.

```
namespace Foo {
  int bar() {}
}
```

To access identifiers defined in the namespace, we must use the **scope resolution operator** `::`. If no scope resolution is given, or an empty one is given, then we look for the identifier in the global namespace.

```
int x = Foo::bar();   // Foo namespace
int y = bar();        // global namespace
int z = ::bar();      // global namespace
```

Example 2.4 (Namespace)

Say that we have two files with the same-name function in different namespaces.

```
namespace Foo {
  int doSomething(int x, int y) {
    return x + y;
  }
}
```

```
namespace Goo {
  int doSomething(int x, int y) {}
    return x - y;
  }
}
```

When we put our forward declarations, we must make sure to add the namespace using the scope resolution operator. If the namespace is not included, then the linker will look for the function in the global namespace rather than the user-defined namespace.

```
int doSomething(int x, int y); // this results in an error
int Foo::doSomething(int x, int y); // correct
int Goo::doSomething(int x, int y); // correct

int main() {
    std::cout << Foo::doSomething(4, 3) << '\n';
    std::cout << Goo::doSomething(4, 3) << '\n';
    return 0;
}
```

Let's talk about a few properties of namespaces.

Lemma 2.1 (Identifiers in Parent Namespaces)

If an identifier A in a namespace uses another identifier B without a scope resolution, then A will look for B within A's namespace. If no matching identifier for B is found, then the compiler will then check each containing namespace in sequence to see if a match is found, with the global namespace being checked last.

```cpp
#include <iostream>
void print() // this print() lives in the global namespace
{
  std::cout << " there\n";
}

namespace Foo {
  void print() // this print() lives in the Foo namespace
  {
    std::cout << "Hello";
  }

  void printHelloThere()
  {
    print();   // calls print() in Foo namespace
    ::print(); // calls print() in global namespace
  }
}

int main() {
  Foo::printHelloThere(); // prints "Hello there"
  return 0;
}
```

Lemma 2.2 (Nested Namespaces)

Namespaces can be nested as well, either of 2 ways.

```cpp
namespace Foo {
  namespace Goo{
    ...
  }
}
.
.
```

```cpp
namespace Foo {

}

namespace Foo::Goo {

}
```

Lemma 2.3 (Namespace aliases)

You can shorten namespaces using **namespace aliases**.

```cpp
namespace Active = Foo::Goo;
int x = Active::doSomething();
```

Lemma 2.4 (Using Namespace)

The **using namespace** is a directive that allows access to all members of a namespace.

3 Types and Expressions

Some types have the `_t` suffix, which just represents type. Some types have this and others don't. In C++, there is no exact size for each fundamental type (except for `char`, which is always 1 byte). There is however a lower bound, so you should always use the lower bound and for maximum portability, never assume that a type can store more bytes.

1. `sizeof(short) = 4`
2. `sizeof(int) = 4`
3. `sizeof(long) = 4`
4. `sizeof(long long) = 8`
5. `sizeof(float) = 4`
6. `sizeof(double) = 8`
7. `sizeof(long double) = 8`

Where does the `sizeof` operator come from?

3.1 Casting

We can also convert some types to different types. If the types are relatively similar, then the C++ implementation may do an **implicit typecast**. If not, then we do an **explicit typecast** in the following ways.

1. `static_cast<T>(foo)`
2. `(T)foo`
3. `dynamic_cast<T>(foo)`

3.2 Enums

4 Constants and Constant Expressions

4.1 Compiler Optimization

By default, all expressions are evaluated at runtime, but compilers have different levels of optimization. Here are some methods in which it optimizes, which follow the **as-if rule** that states that a compiler can modify a program however it likes in order to produce more optimized code, so long as those modification do not affect a program's observable behavior.

Definition 4.1 (Constant Folding)

The compiler replaces expressions that have literal operands with the result of the operation, e.g. `3 + 4` automatically gets evaluated to `7`.

Definition 4.2 (Constant Propagation)

In the code below, `x` is initialized to be `7` and will be stored in the memory allocated for `x`. On the next line, the program will go out to memory to fetch the same value to print. This is redundant. Therefore, the compiler will realize that `x` always has the constant value `7` and will replace all instances of `x` with `7`.

```cpp
#include <iostream>

int main() {
  int x { 7 };
  std::cout << x << '\n';
  return 0;
}
```

Definition 4.3 (Dead Code Elimination)

The compiler removes all code that has no noticeable effect on the program's behavior. Note that this is not a preprocessing step.

```cpp
#include <iostream>
int main() {
  int x { 7 }; // this line is removed.
  std::cout << 7 << '\n';
  return 0;
}
```

A slightly higher level optimization evaluates certain expressions during compile time.

Definition 4.4 (Compile-Time Expression)

A **compile-time expression** is an expression that must always be capable of being evaluated at compile-time.

Example 4.1 ()

Say we have the following code.

```cpp
const double x { 1.2 };
const double y { 3.4 };
const double z { x + y };
```

`z` may or may not be evaluated to `4.6` at runtime. By default it is evaluated at runtime, but it depends on the compiler and level of optimization.

Shifting some of the evaluation from runtime to compile time makes your code faster, though it may make it more difficult to debug since the compiler might rearrange the logic of your program (though in an equivalent way). Therefore, at runtime, the compiled code no longer correlates with the original source code.

4.2 Constants

Now we talk about a seemingly separate, but very related concept.

Definition 4.5 (Constant Variables)

Named constants are variables that cannot change.

1. They cannot be declared and must be initialized since they cannot change. This is called **constant expression initialization**.
2. If a variable can be made constant, it should be. It reduces bugs and gives more opportunity for compiler optimization, effectively reducing runtime and increasing compile time.
3. Function parameters that are `const` just tells the compiler that it won't be changed during the function execution. But since the variable is thrown away after the body, it doesn't really matter anyways. You can also return const types, but this is again a temporary copy and may impede compiler optimizations so it not recommended.
4. In a way, consts are just like object-like directives with substitution text, but consts follow scoping, so use consts whenever you can rather than macros.

Theorem 4.1 ()

All compiler-time expressions must be consts. However, a const variable does not guarantee that it will be evaluated in compile time. With only consts, only const *integral* variables can be a part of a constant expression. No other const variable is allowed.

Proof.

It is not surprising to see that if an expression can be evaluated at compile time, it must be a const variable. Consider the contrapositive: if it wasn't a const variable, then it may be initialized or changed during runtime and therefore the expression cannot be evaluated at compile time. However, the converse is not true.

4.3 Compile-Time Programming

Notice that when we really want a section of code to be evaluated at compile-time, the best we can do is use const variables and hope that the compiler executes it. In other words, we are dependent on the sophistication of the compiler, which is not ideal. To allow more explicit control over which parts of code we want to execute at compile-time, we can use **compile-time programming**. In C++11, compile-time programming was introduced with constant expressions, or **constexpr**s.

Definition 4.6 (Constant Expression)

A **constant expression** is an expression that must be entirely evaluatable at compile-time.[a] They generally contain the following:

1. Literals
2. Most operators with constant expression operands, e.g. `3 + 4`, `2 * sizeof(int)`
3. Constexpr variables
4. Constexpr function calls with constant expression arguments.

Any expression not a constant expression is called a **runtime expression**. The following cannot be used in a constant expression.

1. Non-const variables (e.g. `int x = 3;`)
2. Const non-integral variables, even when they have a constant expression initializer (e.g. `const double d = 1.2`). To use such variables, we need to define them with `constexpr`.
3. Function parameters.

There is a complex list of literals, operators, and variables that can and cannot be used in constant expressions.

[a]along with rules that determine how the compiler should handle these expressions.

There are still two problems. First, the limitations of constant expressions not being able to contain const non-integral variables is quite restricting. Second, even if we did have a constant expression, the compiler will by default evaluate it at runtime. Fortunately, constexpr addresses both problems.

Definition 4.7 (constexpr Keyword)

The **constexpr** variable is always a compile-time constant. As a result, a constexpr variable must be initialized with a constant expression, otherwise a compilation error will result. Here are some examples.

```cpp
constexpr double gravity = 9.8; // works for doubles now
```

Since a constexpr variable is really a constant expression, it is implicitly a const variable.

5 Control Flow and Error Handling

5.1 If Statements

Constexpr if statements can be evaluated at compile time, so we end up compiling only the block under the condition that evaluates to true.

5.2 Switch Statements

5.3 Assert and Static Assert

Assert statements can be turned off with the `#NDEBUG` directive. `static_assert` checks at compile time, so the condition must be a constant expression.

6 Functions

6.1 Inline Functions

When we make a call to a function, we add another frame to our call stack, store the address of our stack pointer, and then execute the function body in the new stack frame. This is known as the **function overhead**.

Definition 6.1 (Inline Functions)

We can avoid this by using the `inline` keyword to define **inline functions**.

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

As the name suggests, the compiler essentially replaces the function call with the function body, treating as it if it were all on the same stack frame.

We get the benefits of no function overhead while still maintaining modularity of our code. However, abusing this increases the size of our compiled executable, which may make our program slower. Most of the time, the compiler is better at optimizing this.

6.1.1 Inline Variables

6.2 Overloading

Functions can be overloaded based on their parameters, and the compiler will try to match the function call to the appropriate overload based on the arguments, called **overload resolution**. The number of parameters and types of parameters are used in differentiating, but not the return type.

6.3 Deleting

Sometimes, functions may use implicit type conversion to call. For example, look at the code.

```cpp
#include <iostream>

void printInt(int x) {
  std::cout << x << '\n';
}

int main() {
  printInt(5);    // okay: prints 5
  printInt('a');  // prints 97 -- does this make sense?
  printInt(true); // print 1 -- does this make sense?
  return 0;
}
```

Definition 6.2 (Function Deleting)

If we want to enforce that a function cannot take other parameters, we can define that function as deleted using the `= delete` specifier. A call to a deleted function will halt compilation.

```cpp
#include <iostream>

void printInt(int x) {
    std::cout << x << '\n';
}

void printInt(char) = delete; // calls to this function will halt compilation
void printInt(bool) = delete; // calls to this function will halt compilation

int main() {
  printInt(97);   // okay

  printInt('a');  // compile error: function deleted
  printInt(true); // compile error: function deleted

  printInt(5.0);  // compile error: ambiguous match

  return 0;
}
```

6.4 Default Arguments

Explicit arguments must all come before any default argument.

Default arguments can not be redeclared, and must be declared before use. Therefore, for forward declarations, the default argument can be declared in either the forward declaration or the function definition, but not both.

However, note that default arguments can lead to ambiguous matches.

6.5 Template Functions

Definition 6.3 (Template Functions)

Let's talk about the syntax. We start with the keyword `template`, which tells the compiler that we're creating a template. Next we specify all the template parameters that our template will use inside the brackets. For each type template parameter, we use the keyword `template` or `class`, followed by the name of the type template parameter (e.g. `T`).

```cpp
template <typename T>
T add(T x, T y) {
  return x + y;
}
```

Function templates are not actually functions. Their code isn't compiled or executed directly. Instead, function templates have one job: to generate functions (that are compiled and executed), called **function instantiation**. The instantiated functions are called **function instances**, and they are *implicitly inline*. When we call a function with a new template argument, it gets instantiated during translation. Therefore, if we called `add` with arguments `int` and `double`, the result of our compilation would look as if we had explicitly defined the following functions.

```cpp
template<>
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x < y) ? y : x;
}

template<>
double max<double>(double x, double y) // the generated function max<double>(double,
    double)
{
    return (x < y) ? y : x;
}
```

Here are some properties.

Lemma 6.1 (Normal Function Call Priority)

Template functions can be called in several ways. However, a normal function call syntax will prefer a non-template function over an equally viable function instantiated from a template.

```cpp
template <typename T>
T max(T x, T y)
{
    std::cout << "called max<int>(int, int)\n";
    return (x < y) ? y : x;
}

int max(int x, int y)
{
    std::cout << "called max(int, int)\n";
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // calls max<int>(int, int)
    std::cout << max<>(1, 2) << '\n';    // deduces max<int>(int, int) (non-template
     functions not considered)
    std::cout << max(1, 2) << '\n';      // calls max(int, int)

    return 0;
}
```

Lemma 6.2 (Static Local Variables)

If a static local variable is defined in a template function, every function instance will have its own copy of the static local variable.

```cpp
#include <iostream>

template <typename T>
void printIDAndValue(T value) {
  static int id{ 0 };
  std::cout << ++id << ") " << value << '\n';
}

int main() {
  printIDAndValue(12);    // 1) 12
  printIDAndValue(13);    // 2) 13
  printIDAndValue(14.5);  // 1) 14.5
  return 0;
}
```

Lemma 6.3 (No Implicit Type Conversions)

Unlike explicit functions, function instances are strict in that they will not do any implicit type conversions. In the left, the call to `max` is okay since the int will be converted to a double. On the right, however, will generate an error.

```cpp
double max(double x, double y) {
  return (x < y) ? y : x;
}

int main() {
  std::cout << max(2, 3.5) << '\n'; //
     okay
  return 0;
}
.
```

```cpp
template <typename T>
T max(T x, T y) {
  return (x < y) ? y : x;
}

int main() {
  std::cout << max<double>(2, 3.5) <<
    '\n'; // error
  return 0;
}
```

Definition 6.4 (Multiple Template Type Parameters)

Definition 6.5 (Overloading Function Templates)

Lemma 6.4 (Function Templates in Multiple Files)

When we forward declare a function template, we cannot just define the template function in another file.

```cpp
// main.cpp
template <typename T>
T addOne(T x); // template forward
    declaration

int main() {
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';
    return 0;
}
```

```cpp
// add.cpp
template <typename T>
T addOne(T x) {
  return x + 1;
}
.
.
.
.
.
```

This would get a linker error since the linker cannot see the definitions of all the *function instances*. There are two solutions to this.

1. We can use a header file that contains the function template definition and add that along with a header guard. This is recommended.

```cpp
// main.cpp
template <typename T>
T add(T x, T y);

int main() {
  std::cout << add(1, 2) << "\n";
  std::cout << add('1', 'a') << "\n";
  return 0;
}
```

```cpp
// add.cpp
template <typename T>
T add(T x, T y) {
  return x + y;
}
template int add<int>(int x, int y);
template char add<char>(char x,
    char y);
.
```

2. We can explicitly define all the necessary function instances.[a] This might be okay if we are using enum types.

```cpp
// main.cpp
#include "add.h"

int main() {
  std::cout << add(1, 2) << "\n";
  std::cout << add('a', 'b') << "\n";
  return 0;
}
```

```cpp
// add.cpp
#pragma once

template <typename T>
T add(T x, T y) {
  return x + y;
}
.
```

6.6 Non-Type Template Parameters

As of C++20, function parameters cannot be constexpr. Therefore, we cannot enforce that these parameters should be fixed at compile time. There may be times where we would like to build a constexpr from the function parameters (say, to do a `static_assert` check on some value), but function parameters cannot be

[a]Before C++20, only integral, enumeration type, or constexpr can be a template parameter.

constexpr and therefore this is impossible.

Definition 6.6 (Non-Type Template Parameters)

It turns out that non-type template parameters can indeed be constexpr, so they can indeed be used to build constexpr and therefore evaluate at compile time. Again, function instantiations are inline.

Example 6.1 (Motivation)

Say that we have this code.

```
double getSqrt(double d) {
  assert(d >= 0.0 && "getSqrt(): d must be non-negative");
  return std::sqrt(d);
}

int main() {
    std::cout << getSqrt(5.0) << '\n';
    std::cout << getSqrt(-5.0) << '\n';
    return 0;
}
```

When we run `getSqrt(-5.0)`, we will runtime assert out. While this is better than nothing, because `-5.0` is a literal (and implicitly constexpr), it would be better if we could `static_assert` so that errors such as this one would be caught at compile-time. However, `static_assert` requires a constant expression, and function parameters can't be constexpr... However, if we change the function parameter to a non-type template parameter instead, then we can do exactly as we want. The following will fail to compile.

```
template <double D>
double getSqrt() {
  static_assert(D >= 0.0, "getSqrt(): D must be non-negative");
  return std::sqrt(D);
}

int main() {
    std::cout << getSqrt<5.0>() << '\n';
    std::cout << getSqrt<-5.0>() << '\n';
    return 0;
}
```

Template parameters can't always be used over regular parameters since the parameter itself may not be a constant expression, so regular parameters are still necessary for runtime evaluation. Here are some other properties.

Lemma 6.5 ()

Non-type template parameters can be implicitly type-casted.

Lemma 6.6 ()

We can use type-deduction for non-type template parameters using `auto`.

```
template <auto N> // deduce non-type template parameter from template argument
```

```cpp
void print() {
  std::cout << N << '\n';
}

int main() {
  print<5>();   // N deduced as int `5`
  print<'c'>(); // N deduced as char `c`
  return 0;
}
```