C++

Muchang Bahng

Winter 2024

Contents

1 Objects

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 (Statements)

A **statement** is an instruction that causes the program to perform some action. Statements are the smallest independent unit of computation in the C++ language, and they are ended with a semicolon.

```
int x; //  declaration statement
int y = 2; // initialization statements
int z = 2 + 3;
```

1.1 Types

Definition 1.2 (Type)

A **type** is a protocol that defines the set of possible values and a set of operations that can be performed on those values.[a] There are several categorizations for types.

1. At the very basic level, we have **primitive types**, which are always built-in types that come with C++ or the standard library (e.g. int, double, boolean).
2. **Compound types** are types that are built in by primitive types. In here, we have
   (a) **Built-in types** which come with C++ or the standard library (e.g. functions)
   (b) **User-defined types** which the user defines (e.g. enums[b], classes, structs)

Note that the protocols for each type is dependent on the version of C++, the computer architecture, and the compiler. Therefore we have to be careful to accommodate them.

Great, now let's see what primitive types are supported in C++.

Definition 1.3 (Integral Types)

Integral types represent a proper subset of $\mathbb{Z}$.

1. The signed numbers include `int`, `long`, `long long` and are stored in two's complement representation.

[a] This is similar to a mathematical set endowed with some operations. This definition is quite abstract, but it suffices.
[b] Since they are implemented with integral types.

```
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x0000000100003fa0 a.out`main + 28
a.out`main:
->  0x100003fa0 <+28>: add    sp, sp, #0x10
    0x100003fa4 <+32>: ret
    0x100003fa8:       udf    #0x1
    0x100003fac:       udf    #0x1c
Target 0: (a.out) stopped.
(lldb) x/4xb $sp+4
0x16fdfec44: 0xff 0xff 0xff 0xff
(lldb) x/4xb $sp+8
0x16fdfec48: 0x01 0x00 0x00 0x00
```

Figure 1: Two's complement representation of ± 1 on my machine from inspecting memory in `lldb`. Note that this is little endian, with the least significant hex coming first.

2. The unsigned numbers include `unsigned int`, `unsigned long`, `unsigned long long` and are stored regularly.

```
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x0000000100003fa0 a.out`main + 28
a.out`main:
->  0x100003fa0 <+28>: add    sp, sp, #0x10
    0x100003fa4 <+32>: ret
    0x100003fa8:       udf    #0x1
    0x100003fac:       udf    #0x1c
Target 0: (a.out) stopped.
(lldb) x/4xb $sp+4
0x16fdfec44: 0x00 0x01 0x00 0x00
(lldb) x/4xb $sp+8
0x16fdfec48: 0x01 0x00 0x00 0x00
```

Figure 2: Regular representation of unsigned integers `256 = 0x1000` and `1 = 0x01`.

Definition 1.4 (Floating Point Types)

Floating point types represent a proper subset of $\mathbb{R}$.

Definition 1.5 (Char)

`Character types` represent the extended ASCII character set and is always 1 Byte. Some nice facts to know.

1. The numbers 0-9 take characters `48` to `57`.
2. The uppercase letters A-Z take `65` to `90`.
3. The lowercase letters a-z take `97` to `122`.

Definition 1.6 (Boolean)

A **boolean** stores 1 bit of memory, but in practice it takes up 1 Byte since a Byte is the smallest addressable unit of memory in most computer architectures.

Definition 1.7 (Void)

The **void** type is the analogous to the null or none type in other languages. It is a type that does not represent a type, stating that an object has no type.

Example 1.1 (Types with _t Suffix?)

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.

1.1.1 Casting

Definition 1.8 (Typecasting)

The action of converting one type to another type is called **typecasting**.

1. The programmer can **explicitly typecast** by calling an operator to change an objects type.
2. If two objects are relatively similar,[a] then the C++ implementation may do an **implicit typecast** to convert it automatically.

Here we list some situations when there is implicit typecasting.

Lemma 1.1 (Variable Initialization)

When initializing (or assigning a value to) a variable with a value of a different type.

```
double d = 3; // int value 3 implicitly converted to type double
d = 6;
```

Lemma 1.2 (Function and Operators)

Function calls and operators will implicitly typecast between char and signed int. This can both be convenient and a pain to work with.

```
int main() {
  char x = 'a';         // 97 in ASCII
  std::cout << x + 4;   // 101
  std::cout << x - 200; // -103
  return 0;
}
```

Lemma 1.3 (Return Types)

When the type of a return value is different from the function's declared return type.

```
float func() {
  return 3.0; // double value 3.0 implicitly converted to float
}
```

[a] defined very loosely here

Lemma 1.4 (Truthy and Falsy Values)

When we use a non-Boolean value in an if-statement.

```
if (5) { // int value 5 implicitly converted to true
  ...
}
```

How do we explicitly typecast? C++ defines a few operators (not functions! explained later) that does this.

Definition 1.9 (C Style Cast)

Similar to typecasting in C, we can do the following.

```
(T)foo
```

Definition 1.10 (Static Cast)

We can conduct a static typecast, which happens during compile time.

```
static_cast<T>(foo)
```

Definition 1.11 (Dynamic Cast)

```
dynamic\_cast<T>(foo)
```

1.2 Variables

Definition 1.12 (Value)

The definition of a **value** is quite abstract. It is simply some data with a **type**. In computers, the value gets *encoded* into some sequence of bits. The **identity** of a value is purely determined by the abstract concept it represents.

Definition 1.13 (Objects and Variables)

An **object** represents a memory of storage (typically RAM or CPU cache) that can hold a value. It has 4 properties.

1. Like a value, it has a type representing the type of value it holds.
2. It has an **address** representing where the value is stored.
3. The identity is determined not just by the value that it stores, but also its address.
4. It *may* have a **name**.[a] Objects with a name are called **variables**, and those without a name are called **anonymous**. The **linkage** of the variable determines which address the variable refers to.

Note that in C++, the definition of an object slightly differs than in more general contexts.

[a] Typically some alphanumeric string like `x`.

Definition 1.14 (Literal)

A **literal** is a value that is directly inserted into code, e.g. `5`, `3.2`, `'a'`. It is not a variable since it does not have an name, and it is not an object either since it doesn't have an address.

Theorem 1.1 (Types are not Objects!)

If a type defines the protocol, then wouldn't this be stored in memory? No, this is for the compiler.

1.2.1 Scope and Duration

Now that we've established variables, we can talk about their scope.

Definition 1.15 (Block)

A **block** is a portion of code that is within curly braces `{ ...}`. Note that C++ is not line sensitive.

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 1.16 (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_`.

Definition 1.17 (Scope)

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

1. **Block scope** refers to a variable being accessible within a certain block.
2. **Global scope** refers to a variable being accessible from everywhere.

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.

Definition 1.18 (Local Variables)

Local variables are variables constructed inside a **block scope** and are accessible only within that block. They have automatic duration by default.

Definition 1.19 (Static Local Variables)

If we talk about **static local variables**, they are variables with block scope but having static duration.
This may be a bit counterintuitive, but say that there is a variable that you want to persist for the

entire program, but you only want to modify that value within a function.

```cpp
void increment() {
  static int val = 1;  // static duration. Initializer only executed once.
  ++val;
  std::cout << val << "\n";
} // val is not destroyed here, but becomes inaccessible

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

Definition 1.20 (Global Variables)

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_`.

Example 1.2 (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 1.3 (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 1.4 (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.

```
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.

1.2.2 Internal and External Linkage

Remember that given a name, its *linkage* determines whether other declarations of that name refer to the same object or not.

Definition 1.21 (Static Global Variables)

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.

1.3 Operators and Functions

In Python, there is no difference between functions and operators since every operator (e.g. `+`) gets mapped to a dunder method (e.g. `__add()__`). In C++, there are differences. Operators and functions are similar in behavior, but we should know that operators are more like *keywords* while function are *compound types*.

Definition 1.22 (Function)

A **function** is a set of statements enclosed in a block, which uses a sequence of **parameters** and a **return type**. It has the following properties.

1. It *may* have a name (e.g. `foo()`). Those without a name are called **anonymous functions**.
2. By default, a call to a function requires us to jump to a separate piece of code.

Definition 1.23 (Operations)

An **operator** is a keyword with a fixed syntax which also does some operation. An **operation** consists of an operator (`+`) and one or more **operands** (`3`, `4.3`).

1. Operators come as a part of C++ (`sizeof`, `+`) or the C++ standard library (`std::cout «`).
2. Unlike a function, an operator does not jump to another sequence in the code and is compiled to a sequence of instructions by the compiler.
3. Operators usually have a fixed number of parameters, while functions can use different sets of

operands (overloading).
4. Operators have built-in precedence rules (e.g. multiplication before addition)

Definition 1.24 (`sizeof` Operator)

The `sizeof` operator returns the size (in bytes) of its operand.
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`

1.4 Declaration vs Definition

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;`).

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

```
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.

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

1.5 Expressions

Definition 1.25 (Expression)

An **expression** is simply a line of code containing variables, operations, literals, and function names/-calls. The process of executing an expression is called **evaluation**. The **type category** is simply the type of the value, object, or function that results from the evaluated expressions. The **value category** indicates whether the expression resolves to a value, an object, a function, or nothing (this list is exhaustive).

2 Translation

Now that we've gotten the basics, we should learn more about the translation process so that we can avoid definition conflicts and know how to work with multi-file programs.

Definition 2.1 (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.

```
#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 2.2 (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.

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

  return 0;
}
.
```

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

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

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

[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.

```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;
}
.
.
```

2.4 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]

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.

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

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 2.3 (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.

```
#pragma once
```

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

```
#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 2.4 (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.

```
#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.

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

```
namespace Foo {

}

namespace Foo::Goo {

}
```

Lemma 2.3 (Namespace aliases)

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

```
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 Constants and Constant Expressions

One of the greatest advantages of C++ is that it is compiled, which allows us to reduce the runtime by offloading computations into compile time. This is particularly important for speed-sensitive programs such as algorithmic trading. Therefore, we should be familiar with consts and constexprs.

3.1 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 3.1 (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 3.2 (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`.

```
#include <iostream>

int main() {
```

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

Definition 3.3 (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.

```
#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 3.4 (Compile-Time Expression)

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

Example 3.1 ()

Say we have the following code.

```
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.

3.2 Constants

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

Definition 3.5 (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 3.1 ()

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.

3.3 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 3.6 (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.

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.

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

Definition 3.7 (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.

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

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

4 Lvalue and Rvalues

4.1 Lvalues and Rvalues

There are two types of a value expressions prior to C++11.

1. An **lvalue** expression evaluates to a named object (variable) or function. A **modifiable lvalue** can be modified, while a **non-modifiable lvalue** cannot be modified (because it is const or constexpr).
2. An **rvalue** expression evaluates to everything else, such as unnamed objects (values), literals, or unnamed functions (anonymous functions). They are not identifiable (meaning they have to be used immediately) and only exist within the scope of the expression in which they are used.

Example 4.1 (Assignment Statement)

An **assignment statement** requires the use of the **assignment operator** and two subexpressions, which are the operands. Note that the whole statement is also an expression.

```
int x = 2;
```

1. It requires the left operand to be a modifiable lvalue expression, and
2. the right operand to be an rvalue expression.

Lemma 4.1 (Implicit Conversion of lvalue to rvalue)

It turns out that in assignment statements, lvalues can also be on the right side since they are implicitly converted to rvalues.

This gets very important when learning about references later on.

5 Control Flow and Error Handling

We are probably familiar with for loops and if statements, but C++ gives us a much wider suite of keywords and operators to choose from. In here, we revisit three things:

1. *Conditional statements.* We visit them by comparing if and switch statements, along with seeing how they may be evaluated at compile time when using constexprs.
2. *Loops.* We can approach them more formally now that we know about scope and duration.
3. *Assert and exit statements.*

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.

5.4 Halt Statements

6 Named Functions

Now we revisit named functions (as opposed to anonymous functions, which we need to know about structs for), and explore it a bit more. This is our first compound type that we will delve into.

Definition 6.1 (Named Functions)

Note that when a function is called, it creates a new stack and *copies* the arguments into the new stack frame. It does the evaluation and returns whatever expression by again *copying*, and all the variables in the stack are destroyed. It is a *compound type* of form

```
T funcName(T arg1, T arg2, ...)
```

This picture of a function is especially important when dealing with its nuances.

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 6.2 (Inline Functions)

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

```
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.

6.2 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 6.3 (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 6.4 (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`).

```
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.

```
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.

```
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.

```
#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.

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

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

```
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 6.5 (Multiple Template Type Parameters)

Definition 6.6 (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 constexpr and therefore this is impossible.

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

Definition 6.7 (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
void print() {
  std::cout << N << '\n';
```

```
}

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

7 References

References and pointers are the next compound types that we will look at. While the language we have explained so far is pretty good, there is a problem. We've said that a variable is simply an object with a name. Since it's an object, it has an address where it stores some value at that address. Say that we want to create two separate variables that has the same address, so that we can have two paths to modify the value. We cannot do this since the new initialized variable stores a copy of the value at a different address.

```
int x = 2; // stores 2 at address A
int y = y; // stores 2 at address B
```

Therefore modifying `y` will not modify `x`. This problem of not being able to create two names that bind to the same object is problematic, and this is a generalization of two more specific problems.

Example 7.1 (Copying During Functions Calls May Be Expensive)

We have explained that when calling a function, it copies all of the arguments in the stack to the new stack frame. This may be good for isolation, but this is a double-edged sword. If we have a large object to copy to do some read operations on, this may be inefficient.

Example 7.2 (In-Place Modification)

If we want a function to modify the value of one of its arguments, this is impossible since it just copies the argument in a new variable, modifies this, and then gets deleted. We could have it return the modified object to override the old one in the parent frame, but this copying of the return value is again slow.

This is where references and pointers come into the rescue. They are similar in that they mainly serve the same purpose, but their behaviors can differ. Generally, references are considered safe while pointers are considered dangerous. There are things that pointers can do that references cannot, and vice versa.

7.1 Lvalue References

Definition 7.1 (Lvalue Reference)

A **reference** is an *alias* for an existing variable (we say it is **bound** to the variable), not a variable (and therefore not an object) itself.[a] However, whatever we do to the reference will persist in the original variable. There are two types of references.

1. `lvalue references` are references that refer to an lvalue. 99% of the time we work with lvalue references.

```
int x = 2;
int& y = x;
```

2. `rvalue references` are references that refer to an rvalue.

A reference evaluates to the variable when used in an expression.

Here we list a few important properties.

Lemma 7.1 (Typechecking)

Lvalue references will (usually) only bind to an object matching its referenced type.

Lemma 7.2 (Initialization)

Lvalue references must be initialized. They cannot be declared.

Lemma 7.3 (No Reseating)

Lvalue references can't be reseated (changed to refer to another object).

Lemma 7.4 (Scope and Duration)

Lvalue references follow the same scoping and duration rules that normal variables do.

Lemma 7.5 (References of References)

You cannot have references of references, since the right-expression in the assignment statement will evaluate to the variable.

```
int x = 5;
int &y = x;
int &z = y;   // y evaluates to x, so z is still a reference to int
```

Lemma 7.6 ()

Lvalue references and referents have independent lifetimes. An lvalue reference should always be initialized after the referent. However, one can by destroyed before the other.

1. If the reference is destroyed before the referent, this is fine.
2. If the referent is destroyed before the reference, this results in a **dangling reference**.

7.2 Rvalue References

Rvalue references are useful in that they can extend the lifespan of the object they are initialized with to the lifespan of the rvalue reference.

Definition 7.2 (Rvalue Reference)

[a] If possible, the reference may be replaced with the variable name by the compiler. This isn't always possible, so perhaps references may require storage.

7.3 Pass and Return by Reference

Theorem 7.1 (Object must outlive Function)

The programmer must be sure that the object being referenced outlives the function returning the reference. Otherwise, the reference will be let dangling.

8 Pointers

Definition 8.1 (Pointer)

A **pointer** is an object that holds a memory address as its value. Given an address, we can **dereference** them with `*` and get their address using `&`.

```
int x = 4;
int* y = &x;
std::cout << y << '\n';
std::cout << *x << '\n';
```

We can modify what the value that the pointer points to by dereferencing the pointer. If we have a const variable, then we must use a const pointer, which must be initialized.

```
const int x = 4;
const int *p = &x;  // good
int *p = &x;        // compilation error
```

Immediately this seems extremely similar to lvalue references. Here we list a few properties which help differentiate them.

Lemma 8.1 (Typechecking)

Pointers will (usually) only bind to an object matching its pointed type.

Lemma 8.2 (Declaration is Okay)

Pointers can be declared rather than initialized. If it is, then it is known as a **wild pointer**. Rather, we should initialize it to null to make it a **null pointer**. We can use the `nullptr` literal.

```
int* p;
int* q = nullptr // null pointer since it's not holding address
```

Dereferencing both a wild and null pointer leads to undefined behavior. Null pointers are falsy, so we can use them to evaluate whether we have a null pointer.

Lemma 8.3 (Reseating Allowed)

Pointers can be reseated, meaning that we can change the address that the pointer is pointing to.

```
int x = 3;
int y = 4;
int *p = &x;
*p = &y;      // points now to &y from &x
```

We can see that through both references and pointers, we can indirectly access an object. References may be more convenient since the dereferencing happens implicitly while for pointers, we must explicitly use the * operator.

It's worth noting that the address of operator doesn't return a literal, but rather a pointer variable that stores the address.

```
int x = 5;
std::cout << &x // returns pointer, not address literal
```

Again, if you destroy the object that the pointer is pointing to, then we get a **dangling pointer**, which leads to undefined behavior. It is easy to test whether a pointer is null or not, but if it isn't, there is no easy way to determine if it's dangling.

8.1 Pass and Return by Address

In addition to pass by value and reference, we can pass in the address of an object as an argument into a function.

Definition 8.2 (Pass by Address)

Given a function that takes in a pointer p, we can interpret it as
1. a pass by address of the object type p is pointing to.
2. a pass by value of the pointer p

Therefore, the address will be copied, but the actual object will not be.

```
int doSomething(int* p);
```

8.2 Smart, Shared, and Unique Pointers

8.3 Function Pointers

9 Enumerations

If we wanted to define a new type that takes values in some discrete space, then we can use an enum.

Definition 9.1 (Unscoped Enumerations)

An **enumeration** is a compound data type whose values are restricted to a set of symbolic constants, called **enumerators**. These enumerators will implicitly convert to integral values as such.

```
enum Color {
  red,    // 0
  green,  // 1
  blue    // 2
};

int main() {
  Color shirt = red;
  std::cout << shirt; // prints 0
}
```

They must be fully defined before we can use it. A forward declaration is not sufficient. Enumerations are implicitly constexpr. Unscoped enumerations have the same scope as where they are defined in.

If they are defined in the global namespaces, then they have global scope.

Theorem 9.1 (Integral Labels can be Explicitly Assigned)

We can actually explicitly label.

```
enum Animal {
  cat = -3,    // values can be negative
  dog,         // -2
  pig,         // -1
  horse = 5,
  giraffe = 5, // shares same value as horse
  chicken,     // 6
};
```

10 Structs

Definition 10.1 (Structs)

Structs are compound types that allow you to store multiple values of different types. We can initialize them using curly-braces, known as **aggregate initialization**.

```
struct Employee {
    int id {};
    int age {};
    double wage {};
};

int main() {
    Employee frank = { 1, 32, 60000.0 }; // copy-list initialization using braced list
    Employee joe { 2, 28, 45000.0 };     // list initialization using braced list
    Employee bob {2, 28}                 // bob.wage value-initialized to 0.0
    return 0;
}
```

11 Classes

11.1 Basics

Definition 11.1 (Class)

A **class** is a keyword that is used to make a user-defined compound type.

11.2 Functors

Definition 11.2 (Functors)

Functors are callable objects.

This is similar to Python's `__call__()` dunder method.

11.2.1 Anonymous Functions and Captures

It's a bit weird that we talk about anonymous functions in the class, but this is exactly because lambda functions are implemented as classes under the hood. j

Definition 11.3 (Anonymous Functions)

11.3 Inheritance

12 Virtual Functions

13 Standard Library

Now that we've built up the basics, we can go into the implementation of the data structures and algorithms in the standard library.

13.1 String

13.2 Array

13.3 Vector

14 Dynamic Memory Allocation

So far, we've worked only in the stack, where our variables were limited to its scope and were destroyed after the block ends. If we want to keep objects in a more persistent memory location, we use the heap.

15 Operator Overloading