C++: CONCEPTS AND PRACTICES

Hao Zhang
zhangh0214@gmail.com

0 Preface

This note was written when I was studying C++. This file is still under update. The main resources is as followings

- Stanley B. Lippman. Essential C++. 1st Edition. Addison-Wesley Professional. 1999.
- Stanley B. Lippman, Josee Lajoie, Barbara E. Moo. C++ Primer. 5th Edition. Addison-Wesley Professional. 2012.
- Google C++ Style Guide.[1]

1 https://google.github.io/styleguide/cppguide.html.

Contents

List of Figures

List of Tables

I BASIC C++ PROGRAMMING

1 C++ Basis

1.1 Language and Compilation Basis

DEFINITION 1.1 **C++** It can be thought of as comprising three parts:

- The low-level language, much of which is inherited from C.
- More advanced language features that allow us to define our own types and to organize large-scale programs and systems.
- The standard library, which uses these advanced features to provide useful data structures and algorithms.

1.1.1 The `main` Function

DEFINITION 1.2 **`main`** Function called by the operating system to execute a C++ program. Each program must have one and only one function named `main`.

On most systems, the value returned from `main` is a status indicator. A return value of `0` indicates success. A nonzero return has a meaning that is defined by the system. Ordinarily a nonzero return indicates what kind of error occurred.

1.1.1.1 `main` with Empty Paremeter List

```
int main() {...}
```

1.1.1.2 Handling Command-Line Options

```
int main(int argc, char *argv[]) {...}
// Equivalent form.
int main(int argc, char **argv) {...}
```

- `argc`: passes the number of strings in `argv` array.
- `argv`: an array of pointers to C-style character strings. The first element in `argv` (i.e., `argv[0]`) points either to the name of the program or to the empty string. Subsequent elements pass the arguments provided on the command line.

1.1.2 Compiling and Executing a Program

1.1.2.1 Compilation and C++ Files

DEFINITION 1.3 **Source file** Term used to describe a file that contains a C++ program.

DEFINITION 1.4 **Object file** File holding object code generated by the compiler from a given source file. An executable file is generated from one or more object files after the files are linked together.

DEFINITION 1.5 **Link** Compilation step in which multiple object files are put together to form an executable program.

DEFINITION 1.6 **Executable file** File, which the operating system executes, that contains code corresponding to our program.

DEFINITION 1.7 **Temporary** Unnamed object created by the compiler while evaluating an expression. A temporary exists until the end of the largest expression that encloses the expression for which it was created.

1.1.2.2 Three kinds of Compilation Errors

DEFINITION 1.8 **Syntax errors** A grammatical error in the C++ language, such as used colon, not a semicolon, after `endl`.

DEFINITION 1.9 **Type errors** Such as passing a string literal to a function that expects an `int` argument.

DEFINITION 1.10 **Declaration errors** Failure to declare a name, such as forgetting to use `std::` for a name from the library and misspelling the name of an identifier.

1.1.2.3 Preprocessor

DEFINITION 1.11 **Preprocessor** A program that runs as part of compilation of a C++ program, and changes the source text of our programs.

DEFINITION 1.12 **Preprocessor variable** Variable managed by the preprocessor. The preprocessor replaces each preprocessor variable by its value before our program is compiled.

DEFINITION 1.13 **`#define`** Preprocessor directive that defines a preprocessor variable.

DEFINITION 1.14 **`#ifdef`** Preprocessor directive that determines whether a given variable is defined.

DEFINITION 1.15 **`#ifndef`** Preprocessor directive that determines whether a given variable is not defined.

DEFINITION 1.16 **`#endif`** Preprocessor directive that ends an `#ifdef` or `#ifndef` region.

1.1.2.4 Comments

DEFINITION 1.17 **Comments** Program text that is ignored by the compiler. C++ has two kinds of comments: single-line and paired. Single-line comments start with a `//`. Everything from the `//` to the end of the line is a comment. Paired comments begin with a `/*` and include all text up to the next `*/`.

We often need to comment out a block of code during debugging. Because that code might contain nested comment pairs (`/*` and `*/`) and comment pairs do net nest, the best way to comment a block of code is to insert single-line comments (`//`) at the beginning of each line in the section we want to ignore, since everything inside a single-line comments are ignored including nested commnet pairs.

1.1.2.5 Best Practice

DEFINITION 1.18 **Edit-Compile-Debug** The process of getting a program to execute properly.

DEFINITION 1.19 **Compiler extension** Feature that is added to the language by a particular compiler. Programs that rely on compiler extensions cannot be moved easily to other compilers.

1.2 Aids for Debugging

1.2.1 The `assert` Preprocessor Macro

DEFINITION 1.20 **Preprocessor macro** Preprocessor facility that behaves like an inline function. Aside from `assert`, modern C++ programs make very little use of preprocessor macros.

DEFINITION 1.21 **`assert`** Preprocessor macro that takes a single expression, which it uses as a condition. When the preprocessor variable `NDEBUG` is not defined, `assert` evaluates the condition and, if the condition is false, writes a message and terminates the program; if the condition is true, `assert` does nothing. The `assert` macro is defined in the `<cassert>` header.

Preprocessor names are managed by the preprocessor not the compiler. As a result, we refer to `assert`, not `std::assert`.

The `assert` macro is often used to check for conditions that "cannot happen".

```
assert(word.size() > threshold);
```

It is a good idea to avoid using the name `assert` for our own purposes even if we don't include `<cassert>`. Many headers include the `<cassert>` header, which means that even if you don't directly include that file, your programs are likely to have it included anyway.

1.2.2 The `NDEBUG` Preprocessor Variable

If `NDEBUG` is defined, `assert` does nothing. By default, `NDEBUG` is not defined, so, by default, `assert` performs a run-time check. `assert` can be useful as an aid in getting a program debugged but should not be used to substitute for runtime logic checks or error checking that the program should do.

```
#ifndef NDEBUG
...
#endif
```

the preprocessor defines five other names that can be useful in debugging.
`__FILE__` string literal containing the name of the file.
`__func__` string literal containing the name of the function.
`__LINE__` integer literal containing the current line number.
`__TIME__` string literal containing the time the file was compiled.
`__DATE__` string literal containing the date the file was compiled.

1.3 Declaration, Definition, and Header File

1.3.1 Declaration and Definition

DEFINITION 1.22 **Separate compilation** Ability to split a program into multiple separate source files.

To support separate compilation, C++ distinguishes between declarations and definitions.

DEFINITION 1.23 **Declaration** Asserts the existence of a variable, function, or type defined elsewhere. A variable declaration specifies the type and name of a variable. Names may not be used until they are defined or declared.

DEFINITION 1.24 **Definition** Allocates storage for a variable of a specified type and optionally initializes the variable. A variable definition is a declaration, and it also allocates storage and may provide the variable with an initial value. Names may not be used until they are defined or declared.

A file that wants to use a name defined elsewhere includes a declaration for that name. To obtain a declaration that is not also a definition, we add the `extern` keyword and may not provide an explicit initializer. An `extern` that has an initializer is a definition.

```
extern int i;  // Declares but does not define.
int j;  // Declares and defines.
extern int k = 0;  // Declares and defines.
```

DEFINITION 1.25 **Name lookup** Process by which the use of a name is matched to its declaration.

Variables must be defined exactly once but can be declared many times. If it is invoked in multiple programs, there must be multiple declarations available.

1.3.2 Header File

Rather than seperately declare the function in each file, we maintain only a single declaration in a header file. The header file is then included in each program text file that wishes to use the funcion. If the declaration changes, only one modification in the header file is needed. All users of the function automatically include the updated function declaration.

DEFINITION 1.26 **Header** Mechanism whereby the declarations of a class or other names are made available to multiple programs. Every program that uses a library facility must include its associated header.

Headers (usually) contain entities (such as class definitions and `const` and `constexpr` variables that can be defined only once in any given file. Objects defined at global scope are also declared in a header file.

We will not put definitions in a header file because the header file is included in multiple text files in within a program. The only exception is `inline` function, which shall be expanded by the compiler. To make definition avaiable to the compiler at each invocation point, we must place the `inline` function definitions insider a header file.

If you are familiar with Java, header is where the so called “interface” should appear, the detailed implementation should be in the source file. The implementation is compiled once and it linked to our program whenever we wish to use it. Header has two advantages.

- User of the library can clearly see what function they are use, and do not care about the implementation details.
- Sperating out interface and implementation can protect the source code.

Code inside headers ordinarily should not use `using` declarations. This is because if a header has a using declaration, then every program that includes that header gets that same

using declaration. As a result, a program that didn't intend to use the specified library name might encounter unexpected name conflicts.

DEFINITION 1.27 **# include** Directive that makes code in a header available to a program.

DEFINITION 1.28 **Header guard** Preprocessor variable used to prevent a header from being included more than once in a single file.

1.3.3 Best Practices

In general, every `.cpp` file should have an associated `.h` file. There are some common exceptions, such as unit-tests and small `.cpp` files containing just a `main` function.

All header files should have header guards to prevent multiple inclusion. The format of the symbol name should be:

```
PROJECT_PATH_FILE_H_
```

For example, the file `foo/include/bar/baz.h` in project `foo` should have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif  // FOO_BAR_BAZ_H_
```

Prefer placing the definitions for template and inline functions in the same file as their declarations.a

Avoid using forward declarations where possible. Just `#include` the headers you need.

1.3.3.1 Names and Order of Includes

In `dir/A.cpp` or `dir/A_test.cpp`, whose main purpose is to implement or test the stuff in `dir/A.h`, order your includes as follows:

- `dir/A.h`.
- C system files.
- C++ system files.
- Other libraries' header files.
- Your project's header files.

With the preferred ordering, if `dir/A.h` omits any necessary includes, the build of `dir/A.cpp` or `dir/A_test.cpp` will break.

1.4 Variable, Expression, and Statement

DEFINITION 1.29 **Object** A region of memory that has a type.

DEFINITION 1.30 **Varable** A named object or reference. In C++, variables must be declared before they are used.

DEFINITION 1.31 **Expression** The smallest unit of computation in a C++ program. An expression is composed of one or more operands and usually one or more operator. An expression yields a result when it is evaluated. Expressions can be used as operands, so we can write compound expressions requiring the evaluation of multiple operators.

The simplest form of an expression is a single literal or variable. The result of such an expression is the value of the variable or literal.

DEFINITION 1.32 **Result** Value or object obtained by evaluating an expression.

DEFINITION 1.33 **Condition** An expression that yields a result that is either `true` or `false`. A value of zero/`nullptr` is `false`; any other value yields `true`.

DEFINITION 1.34 **Statement** A part of a program that specifies an action to take place when the program is executed. An expression followed by a semicolon is a statement; other kinds of statements include blocks and `if`, `for`, and `while` statements, all of which contain other statements within themselves.

A statement is analogous to a sentence in a natural language.

DEFINITION 1.35 **Expression statement** An expression followed by a semicolon. An expression statement causes the expression to be evaluated.

1.5 Namespaces

DEFINITION 1.36 **Namespace** Mechanism for gathering all the names defined by a library or other collection of programs into a single scope. Unlike other scopes in C++, a namespace scope may be defined in several parts. The namepsace may be opened and closed and reopened again in disparate parts of the program. Namespaces help avoid inadvertent name clashes. The names defined by the C++ library are in the namespace `std`.

DEFINITION 1.37 **Name Clash** Occurs when multiple entities that have the same name in an application so that the program cannot distinguish between the two.

DEFINITION 1.38 **Namespace pollution** Occurs when all the names of classes and functions are placed in the global namespace. Large programs that use code written by multiple independent parties often encounter collisions among names if these names are global.

DEFINITION 1.39 **Global namespace** The (implicit) namespace in each program that holds all global definitions.

```
// A.h
#include <...>
namespace CPP_PRIMER {
class A { ... };
}

// A.cpp
#include <...>
#include "A.h"
namespace CPP_PRIMER {
...
}

// main.cpp
#include "A.h"
using CPP_PRIMER::A;

int main() {
  A a;
}
```

Names defined in a namespace may be accessed directly by other members of the namespace, including scopes nested within those members. Code outside the namespace must indicate the namespace in which the name is defined.

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Prefer grouping functions with a namespace instead of using a class as if it were a namespace. Static methods of a class should generally be closely related to instances of the class or the class's static data.

DEFINITION 1.40 **Unnamed namespace** Namespace that is defined without a name. Names defined in an unnamed namespace may be accessed directly without use of the scope operator. Each file has its own unique unnamed namespace. Names in an unnamed namespace are not visible outside that file.

When definitions in a `cpp` file do not need to be referenced outside that file, place them in an unnamed namespace or declare them `static`. Do not use either of these constructs in `.h` files.

DEFINITION 1.41 **Scope operator (`::` operator)** Access names in a namespace.

DEFINITION 1.42 **`using` declaration** Mechanism to inject a single name from a namespace into the current scope such that it is accessible directly.

```
usin enamespace::name;
```

1.6 IO Library

1.6.1 Stream

DEFINITION 1.43 **Stream** A sequence of characters read from or written to an IO device. The term stream is intended to suggest that the characters are generated, or consumed, sequentially over time.

DEFINITION 1.44 **`iostream`** Header that provides the library types for stream-oriented input and output.

DEFINITION 1.45 **`istream`** Library type providing stream-oriented input.

DEFINITION 1.46 **`ostream`** Library type providing stream-oriented output.

1.6.2 Standard Input and Output Objects

DEFINITION 1.47 **Standard input** Input stream usually associated with the window in which the program executes.

DEFINITION 1.48 **`cin`** `istream` object used to read from the standard input.

DEFINITION 1.49 **Standard output** Output stream usually associated with the window in which the program executes.

DEFINITION 1.50 **`cout`** `ostream` object used to write to the standard output. Ordinarily used to write the output of a program.

DEFINITION 1.51 **`cerr`** `ostream` object tied to the standard error, which often writes to the same device as the standard output. Usually used for error messages or other output that is not part of the normal logic of the program.

DEFINITION 1.52 **`clog`** `ostream` object tied to the standard error. By default, writes to `clog` are buffered. Usually used to report information about program execution to a log file.

1.6.3 Manipulator

DEFINITION 1.53 **Buffer** A region of storage used to hold data. IO facilities often store input (or output) in a buffer and read or write the buffer independently from actions in the program. Output buffers can be explicitly flushed to force the buffer to be written.

DEFINITION 1.54 **`endl` manipulator** Writing `endl` has the effect of ending the current line and flushing the buffer associated with that device. Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written.

By default
- Reading `cin` flushes `cout`.
- `cout` is flushed when the program ends normally.
- Writes to `cerr` are not buffered.

DEFINITION 1.55 **Manipulator** A function-like object that "manipulates" a stream. Manipulators can be used as the right-hand operand to the overloaded IO operators, << and >>. Most manipulators change the internal state of the object. Such manipulators often come in pairs — one to change the state and the other to return the stream to its default state.

The fact that a manipulator makes a persistent change to the format state can be useful when we have a set of IO operations that want to use the same formatting. Manipulators that change the format state of the stream usually leave the format state changed for all subsequent IO.

1.6.4 Reading an Unkown Number of Inputs

```
while (cin >> val) {...}
```

`cin >> val` reads the value from the standard input and stores that in `val`. The input operator returns its left operand, which in this case is `cin`.

When we use an istream as a condition, the effect is to test the state of the stream. The stream is valid (the condition yields `true`) if the stream hasn't encountered an error. An istream becomes invalid (the condition yield `false`) when
- Hit end-of-file (Ctrl-D).
- Or encounter an invalid input, such as reading a `int` value that is not an integer.

Thus, our while executes until we encounter end-of-file (or an input error).

DEFINITION 1.56 **End-of-file** System-specific marker that indicates that there is no more input in a file.

1.6.5 Formatted and Unformatted IO

DEFINITION 1.57 **Formatted IO** IO operations that use the types of the objects being read or written to define the actions of the operations. Formatted input operations perform whatever transformations are appropriate to the type being read, such as converting ASCII numeric strings to the indicated arithmetic type and (by default) ignoring whitespace. Formatted output routines convert types to printable character representations, pad the output, and may perform other, type-specific transformations.

DEFINITION 1.58 **Unformatted IO** Operations that treat the stream as an undifferentiated byte stream. Unformatted operations place more of the burden for managing the IO on the user.

2 Variables and Basic Types

2.1 Variables

DEFINITION 2.1 **Identifier** Sequence of characters that make up a name. Identifiers are case-sensitive.

DEFINITION 2.2 **Keyword** Predefined names given special meaning within the language.

2.1.1 Scope of a Name

DEFINITION 2.3 **Scope** The scope of a name is the part of the program's text in which that name is visible. C++ has several levels of scope:

- Global: names defined outside any other scope.
- Class: names defined inside a class.
- Namespace: names defined inside a namespace.
- Block/Local: names defined inside a block.

Scopes nest. Once a name is declared, it is accessible until the end of the scope in which it was declared.

DEFINITION 2.4 **Class scope** Each class defines a scope. Class scopes are more complicated than other scopes—member functions defined within the class body may use names that appear even after the definition.

DEFINITION 2.5 **In scope** Name that is visible from the current scope.

DEFINITION 2.6 **Inner scope** Scope that is nested inside another scope.

DEFINITION 2.7 **Outer scope** Scope that encloses another scope.

Once a name has been declared in a scope, that name can be used by scopes nested inside that scope. Names declared in the outer scope can also be redefined in an inner scope.

DEFINITION 2.8 **Lifetime** The lifetime of an object is the time during the program's execution that the object exists.

2.2 Variable Initialization

DEFINITION 2.9 **Initialization** Give an object a value at the same time that it is created.

DEFINITION 2.10 **Initialized** A variable given an initial value when it is defined. Variables usually should be initialized.

Initialization is not assignment. Initialization happens when a variable is given a value when it is created.

DEFINITION 2.11 **Assignment** Obliterates an objects current value and replaces that value by a new one.

2.2.1 Four Forms of Initialization

```
int i = 0;  // Copy initialization.
int i(0);  // Direct initialization.
int i = {0};  // List initialization.
int i{0};  // List initialization.
```

2.2.1.1 Copy Initialization

DEFINITION 2.12 **Copy initialization** Form of initialization that uses an = to supply an initializer for a newly created object. The newly created object is a copy of the given initializer. Copy initialization is also used when we pass or return an object by value and when we initialize an array or an aggregate class. Copy initialization uses the copy constructor or the move constructor, depending on whether the initializer is an lvalue or an rvalue.

2.2.1.2 Direct Initialization

DEFINITION 2.13 **Direct initialization** Form of initialization that does not include an =.

When we use direct initialization, we are asking the compiler to use ordinary function matching to select the constructor that best matches the arguments we provide. When we use copy initialization, we are asking the compiler to copy the right-hand operand into the object being created, converting that operand if necessary. Whether we use copy or direct initialization matters if we use an initializer that requires conversion by an `explicit` constructor.

```
vector<int> v1(10);  // ok: direct initialization.
// error: constructor that takes a size is explicit.
vector<int> v2 = 10;
```

Copy initialization is inherited from C language. It works well with the data objects of the built-in types and for class objects that can be initialized with a single value, but it does not work well with class objects that require multiple initial values. Direct initialization was introduced to handle multiple value initialization.

2.2.1.3 List Initialization

DEFINITION 2.14 **List initialization** Form of initialization that uses curly braces to enclose one or more initializers. The compiler will not let us list initialize variables of built-in type if the initializer might lead to the loss of information.

2.2.2 Default and Value Initialization

DEFINITION 2.15 **Default initialization** When we define a variable without an explicit initializer, the variable is default initialized. Objects of class type that we do not explicitly initialize have a value that is defined by the class. Objects of built-in type defined at global scope are initialized to 0; those defined at local scope are uninitialized and have undefined values. Therefore, we recommend initializing every object of built-in type.

Default initialization happens when

- We define non-`static` variables or arrays at block scope without initializers.
- A class that itself has members of class type uses the synthesized default constructor.
- Members of class type are not explicitly initialized in a constructor initializer list.

DEFINITION 2.16 **Uninitialized variable** Variable that is not given an initial value. Variables of class type for which no initial value is specified are initialized as specified by the class definition. Variables of built-in type defined inside a function are uninitialized unless explicitly initialized. It is an error to try to use the value of an uninitialized variable. Uninitialized variables are a rich source of bugs.

DEFINITION 2.17 **Undefined** Usage for which the language does not specify a meaning. Knowingly or unknowingly relying on undefined behavior is a great source of hard-to-track runtime errors, security problems, and portability problems.

DEFINITION 2.18 **Value initialization** Initialization in which built-in types are initialized to zero and class types are initialized by the class's default constructor. Objects of a class type can be value initialized only if the class has a default constructor. Elements are initialized as a copy of this compiler-generated value.

Value initialization happens

- During array initialization when we provide fewer initializers than the size of the array.
- When we define a local `static` object without an initializer.
- When we explicitly request value initialization by writing an expressions of the form `T()` where `T` is the name of a type (The `vector` constructor that takes a single argument to specify the `vector`'s size uses an argument of this kind to value initialize its element initializer.)

2.3 Primitive Built-in Types

2.3.1 Machine-Level Representation of the Built-in Types

DEFINITION 2.19 **Address** Number by which a byte in memory can be found.

DEFINITION 2.20 **Byte** The smallest chunk of bits of addressable memory. In C++ a byte has at least as many bits as are needed to hold a character in the machines basic character set. On most machines a byte contains 8 bits.

DEFINITION 2.21 **Word** The natural unit of integer computation on a given machine. Usually a word is large enough to hold an address. On most machines a word is either 4 or 8 bytes.

To give meaning to memory at a given address, we must know the type of the value stored there. The type determines how many bits are used and how to interpret those bits. In other words, in C++, a type defines both the concepts of a data element and the operations that are possible on those data.

2.3.2 Type

DEFINITION 2.22 **Type checking** The process by which the compiler verifies that the way objects of a given type are used is consistent with the definition of that type.

DEFINITION 2.23 **Statically typed** C++ is a statically typed language, which means that types are checked at compile time.

DEFINITION 2.24 **Type specifier** The name of a type.

DEFINITION 2.25 **Built-in type** Type defined by the language, including arithmetic types and `void`.

DEFINITION 2.26 **Arithmetic types** Built-in types representing boolean values, characters, integers, and floating-point numbers.

DEFINITION 2.27 **`void` type** Special-purpose type that has no operations and no value. It is not possible to define a variable of type `void`.

2.3.3 Arithmetic Types

Arithmetic types include integral types and floating-point types.

DEFINITION 2.28 **Signed** Integer type that holds negative or positive values, including zero.

DEFINITION 2.29 **Unsigned** Integer type that holds only values greater than or equal to zero.

2.3.4 Deciding which Type to Use

- Do not use plain `char` or `bool` in arithmetic expressions. Use them only to hold characters or truth values.
- Use an unsigned type when you know that the values cannot be negative.
- Use `int` for integer arithmetic. If your data values are larger than the minimum guaranteed size of an `int`, then use `long long`.
- Use `double` for floating-point computations; `float` usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible.

2.3.5 Literals

DEFINITION 2.30 **Literal** A value such as a number, a character, or a string of characters. The value cannot be changed. Literal characters are enclosed in single quotes, literal strings in double quotes.

2.3.5.1 Integer and Floating-Point Literals

Be default, decimal literals are signed, and it has the smallest type of `int`, `long`, or `long long` in which the literal's value fits.

Integer literals that begin with `0` are interpreted as octal. Those that begin with either `0x` or `0X` are interpreted as hexadecimal. Octal and hexadecimal literals have the smallest type of `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` in which the literal's value fits.

Floating-point literals include either a decimal point or an exponent specified using scientific notation such as `3.14, 3.14E0, 0., 0e0, .001`. By default, floating-point literals have type `double`.

2.3.5.2 Character and Character String Literals

A character enclosed within single quotes is a literal of type `char`.

DEFINITION 2.31 **String literal/Character string literal** A sequence of zero of more characters enclosed in double quotation marks. The type of a string literal is array of constant `char`s. The compiler appends a null character (`'\0'`) to every string literal.

Two string literals that appear adjacent to one another and that are separated only by spaces, tabs, or newlines are concatenated into a single literal. We use this form of literal when we need to write a literal that would otherwise be too large to fit comfortably on a single line:

```
cout << "A really long string literal "
  "that spans two lines." << endl;
```

2.3.5.3 Escape sequence

DEFINITION 2.32 **Escape sequence** Alternative mechanism for representing characters, particularly for those without printable representations. An escape sequence is a backslash followed by a character, three or fewer octal digits, or an `x` followed by a hexadecimal number.

DEFINITION 2.33 **Nonprintable character** A character with no visible representation, such as a control character, a backspace, newline, and so on.

2.4 Compound Types

DEFINITION 2.34 **Compound Types** A compound type is a type that is defined in terms of another type, such as references and pointers.

DEFINITION 2.35 **Base type** Type specifier, possibly qualified by `const`, that precedes the declarators in a declaration. The base type provides the common type on which the declarators in a declaration can build.

DEFINITION 2.36 **Declarator** The part of a declaration that includes the name being defined and an optional type modifier.

A declaration is a base type followed by a list of declarators. If the declarations are nothing more than variable names, the type of such variables is the base type of the declaration. More complicated declarators specify variables with compound types that are built from the base type of the declaration.

2.4.1 Reference

DEFINITION 2.37 **Reference** A reference defines an alternative name for an object. A reference is not an object. Instead, a reference is just another name for an already existing object.

DEFINITION 2.38 **Bind** Associating a name with a given entity so that uses of the name are uses of the underlying entity. For example, a reference is a name that is bound to an object.

When we define a reference, we bind the reference to its initializer. References must be initialized, and there is no way to rebind a reference. A reference may be bound only to an object, not to a literal or to the result of a more general expression.

After a reference has been defined, all operations on that reference are actually operations on the object to which the reference is bound.

2.4.2 Pointer

DEFINITION 2.39 **Pointer** An object that can hold the address of an object, the address one past the end of an object, or zero. A pointer is an object in its own right.

2.4.2.1 Null Pointers

DEFINITION 2.40 **Null pointer** Pointer whose value is `0`. A null pointer is valid but does not point to any object.

DEFINITION 2.41 **`nullptr`** Literal constant that denotes the null pointer. `nullptr` is a literal that has a special type that can be converted to any other pointer type.

Define a pointer only after the object to which it should point has been defined. If there is no object to bind to a pointer, then initialize the pointer to `nullptr`.

A important difference between a pointer and a reference paramter is that a pointer may or may not actually address an object. Before we dereference a pointer, we must always make sure that it is not set to `nullptr`. A reference, however, always refers to some object, so the check is unnecessary.

2.4.2.2 Condition on a Pointer

```
if (p) {...}
```

If the pointer is `nullptr`, the condition is `false`. Any nonzero pointer evaluates as `true`.

2.4.2.3 `void *` Pointers

DEFINITION 2.42 **`void *` pointer** Pointer type that can point to any non-`const` type, but the type of the object at that address is unknown. Such pointers may not be dereferenced.

We can only do a limited number of things with a `void *` pointer:

- Compare (using == or !=) it to another pointer.
- Assign it to another `void *` pointer.
- Pass it to or return it from a function.

2.5 `const` qualifier

DEFINITION 2.43 **`const`** Type qualifier used to define objects that may not be changed. `const` objects must be initialized, because there is no way to give them a value after they are defined.

DEFINITION 2.44 **`const` object** Object cannot be modified from its initial value. Any attempt to assign a value to a `const` object results in a compile-time error.

2.5.1 By Default, `const` Objects are Local to a File

When a `const` object is initialized from a compile-time constant, the compiler will usually replace uses of the variable with its corresponding value during compilation.

In order to see the variable's initializer to substitute the value for the variable, `const` objects are defined local to a file. When we define a `const` with the same name in multiple files, it is as if we had written definitions for separate variables in each file.

To share a `const` object among multiple files, we must use the keyword `extern` on both its definition and declaration(s).

2.5.2 References to `const`

DEFINITION 2.45 **References to `const`/`const` reference** A reference that may not change the value of the object to which it refers. A reference to const may be bound to a const object, a nonconst object, or the result of an expression.

We can bind a reference to `const` to

- A non-`const` object.
- A literal.
- A more general expression.

2.5.3 Pointers and `const`

DEFINITION 2.46 **Pointer to `const`** Pointer that can hold the address of a const object. A pointer to const may not be used to change the value of the object to which it points.

We can use a pointer to `const` to point to

- A non-`const` object.

DEFINITION 2.47 **`const` pointers** A `const` pointer must be initialized, and once initialized, its value (i.e., the address that it holds) may not be changed. We indicate that the pointer is `const` by putting the `const` after the `*`. This placement indicates that it is the pointer, not the pointed-to type, that is `const`.

The fact that a pointer is itself `const` says nothing about whether we can use the pointer to change the underlying object.

DEFINITION 2.48 **Top-level `const`** The const that specifies that an object itself may not be changed. Top-level `const` can appear in any object type.

DEFINITION 2.49 **Low-level `const`** A `const` that is not top-level. Such consts are integral to the type and are never ignored. Low-level `const` appears in the base type of compound types such as pointers or references.

Pointers can have both top-level and low-level `const` independently. `const` in reference types is always low-level.

When we copy an object, top-level `const` is ignored since copying an object doesnt change the copied object. On the other hand, low-level `const` is never ignored. In general, we can convert a non-`const` to `const` but not the other way round.

2.5.4 `constexpr` and Constant Expressions

DEFINITION 2.50 **Constant expressions** A constant expression is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. A `const` object that is initialized from a constant expression is also a constant expression.

DEFINITION 2.51 **`constexpr`** Variable that represents a constant expression.

We can ask the compiler to verify that a variable is a constant expression by declaring the variable in a `constexpr` declaration. Variables declared as `constexpr` are implicitly `const` and must be initialized by constant expressions.

We can initialize a `constexpr` reference from

- An object that remains at a fixed address.

We can initialize a `constexpr` pointer from

- `nullptr`.
- An object that remains at a fixed address.

Variables defined inside a function ordinarily are not stored at a fixed address. Hence, we cannot use a `constexpr` pointer to point to such variables. On the other hand, the

address of an object defined outside of any function is a constant expression, and so may be used to initialize a `constexpr` pointer.

When we define a pointer in a `constexpr` declaration, the `constexpr` specifier applies to the pointer, not the type to which the pointer points, i.e., `constexpr` imposes a top-level `const` on the objects it defines. Like any other `constant` pointer, a `constexpr` pointer may point to a `const` or a non-`const` type.

```
// constexpr pointer to the const int i.
constexpr const int *cp = &i;
// constexpr pointer to the int j.
constexpr int *p  = &j;
```

2.6 Rvalue Reference

In some circumstances, an object is immediately destroyed after it is copied. In those cases, moving, rather than copying, the object can provide a significant performance boost.

The library containers, `string`, and `shared_ptr` classes support move as well as copy. The IO and `unique_ptr` classes can be moved but not copied.

2.6.1 Rvalue Reference

2.6.1.1 Lvalue Reference and Rvalue Reference

DEFINITION 2.52 **Lvalue reference** Reference that can bind to an lvalue.

Functions that return lvalue references, along with the assignment, subscript, dereference, and prefix increment/decrement operators, are all examples of expressions that return lvalues. We can bind an lvalue reference to the result of any of these expressions.

DEFINITION 2.53 **Rvalue reference** Reference to an object that is about to be destroyed. As a result, we know that there can be no other users of that object, and we are free to "move" resources from an rvalue reference to another object.

Functions that return a nonreference type, along with the arithmetic, relational, bitwise, and postfix increment/decrement operators, all yield rvalues. We cannot bind an lvalue reference to these expressions, but we can bind either an lvalue reference to `const` or an rvalue reference to such expressions.

```
int i = 42;
int &rri1 = i;  // Error. i is a lvalue.
int &&rri = i * 42;  // OK.
```

2.6.1.2 Lvalues Persist; Rvalues are Ephemeral

Lvalues and rvalues differ from each other in an important manner: Lvalues have persistent state, whereas rvalues are either literals or temporary objects created in the course of evaluating expressions.

Because rvalue references can only be bound to temporaries, code that uses an rvalue reference is free to take over resources from the object to which the reference refers.

2.6.1.3 Variables are Lvalues

A variable is an lvalue; we cannot directly bind an rvalue reference to a variable even if that variable was defined as an rvalue reference type. After all, a variable persists until it goes out of scope.

```
int &&rri1 = 42;  // OK.
int &&rri2 = rri1;  // Error. rri1 is an lvalue.
```

2.6.1.4 The Library `move` Function

DEFINITION 2.54 **`move`** Library function used to bind an rvalue reference to an lvalue. Calling `move` implicitly promises that we will not use the moved-from object except to destroy it or assign a new value to it.

We can obtain an rvalue reference bound to an lvalue by calling the library function `move`, which is defined in the `<utility>` header. Calling `move` tells the compiler that we have an lvalue that we want to treat as if it were an rvalue.

```
int &&rri3 = std::move(rri1);  // OK.
```

After a call to `move`, we can destroy a moved-from object and can assign a new value to it, but we cannot use the value of a moved-from object.

Code that uses `move` should use `std::move`, not `move`. Doing so avoids potential name collisions.

Outside of class implementation code such as move constructors or move-assignment operators, use `std::move` only when you are certain that you need to do a move and that the move is guaranteed to be safe.

2.6.1.5 `static_cast` from an Lvalue to an Rvalue Reference Is Permitted

Even though we cannot implicitly convert an lvalue to an rvalue reference, we can explicitly cast an lvalue to an rvalue reference using `static_cast`. Although we can write such casts directly, it is much easier to use the library `move` function.

2.6.2 Overloading on Moving and Copying

Overloaded functions that distinguish between moving and copying a parameter typically have one version that takes a `const T &` and one that takes a `T &&`.

```
// Copy: binds to any kind of type T.
void push_back(const T &val) {
  // Ensure that there is room for another element.
  check_and_alloc();
  // The T's copy constructor will be used.
  alloc.construct(end++, val);
}

// Move: binds to only to modifiable rvalues of type T.
void push_back(T &&val) {
  check_and_alloc();
  // The T's move constructor will be used.
  alloc.construct(end++, std::move(val));
}
```

We can pass any object that can be converted to type `T` to the first version. This version copies data from its parameter. We can pass only an rvalue that is not `const` to the second version. This version is an exact match (and a better match) for non-`const` rvalues and will be run when we pass a modifiable rvalue. This version is free to steal resources from its parameter.

Ordinarily, there is no need to define versions of the operation that take a `const T &&` or a `T &`. Usually, we pass an rvalue reference when we want to "steal" from the argument. In order to do so, the argument must not be `const`. Similarly, copying from an object should not change the object being copied. As a result, there is usually no need to define a version that take a `T &` parameter.

```
string s = "some string";
// Lvalue; Calls push_back(const string &);
vec.push_back(s);
// Rvalue; Calls push_back(string &&);
vec.push_back("done");
```

2.7 Dealing with Types

2.7.1 Type Aliases

DEFINITION 2.55 **Type Aliases** A type alias is a name that is a synonym for another type. Defined through either a `typedef` or an alias declaration.

There are two ways to define a type alias

```
typedef double wages;
using wages = double;  // Alias declaration.
```

DEFINITION 2.56 **`typedef`** Defines an alias for another type. When `typedef` appears in the base type of a declaration, the names defined in the declaration are type names.

DEFINITION 2.57 **Alias declaration** Defines a synonym for another type: `using name = type` declares name as a synonym for the type type .

Declarations that use type aliases that represent compound types and `const` can yield surprising results.

```
typedef char *pstring;
// cstr is a constant pointer to char.
const pstring cstr = 0;
```

When we use `pstring` in a declaration, the base type of the declaration is a pointer type.

2.7.2 The `auto` Type Specifier

DEFINITION 2.58 **`auto`** Type specifier that deduces the type of a variable from its initializer.

When we use a reference, the compiler uses that object's type for `auto`'s type deduction. `auto` ordinarily ignores top-level `const`.

2.7.3 The `decltype` Type Specifier

DEFINITION 2.59 **`decltype`** Type specifier that deduces the type of a variable or an expression. The compiler analyzes the expression to deduce its type but does not evaluate the expression.

When the expression to which we apply `decltype` is a variable, `decltype` returns the type of that variable, including top-level `const` and references. When we apply decltype to an expression that is not a variable, `decltype` returns a reference type for expressions if the expression yields an lvalue (that can stand on the left-hand side of the assignment).

`decltype((variable))` is always a reference type, but `decltype(variable)` is a reference type only if variable is a reference. This is because if we wrap the variable's name in parentheses, the compiler will evaluate the operand as an expression. A variable is an expression that can be the left-hand side of an assignment.

3 Arrays

3.1 Arrays

DEFINITION 3.1 **Array** Data structure that holds a collection of unnamed objects that are accessed by an index.

3.1.1 Defining and Initializing Built-in Arrays

The dimension must be known at compile time, which means that the dimension must be a constant expression. As with variables of built-in type, the elements in an array are default initialized.

We can list initialize the elements in an array. When we do so, we can omit the dimension. If we specify a dimension, the number of initializers must not exceed the specified size. Any remaining elements are value initialized.

We cannot initialize an array as a copy of another array, nor is it legal to assign one array to another.

```
// Array of ten pointers to int.
int *ptrs[10];
// Pointer to an array of ten ints.
int (*p_array)[10] = &arr;
// Reference to an array of ten ints.
int (&ref_array)[10] = arr;
// Reference to an array of ten pointers.
int *(&ref_ptrs)[10] = ptrs;
```

3.1.2 Accessing the Elements of an Array

DEFINITION 3.2 **Subscript operator (`[]`)** `p[n]` is a synonym for `*(p+n)`.

DEFINITION 3.3 **Index** Value used in the subscript operator to denote the element to retrieve from a `string`, `vector`, or array.

DEFINITION 3.4 **Buffer overflow** Serious programming bug that results when we use an index that is out-of-range for a container or an array.

DEFINITION 3.5 **`size_t`** Machine-dependent unsigned integral type defined in the `<cstddef>` header that is large enough to hold the size of the largest possible array.

We can use a range `for` or the subscript operator to access elements of an array. When we use a variable to subscript an array, we normally should define that variable to have type `size_t`.

Unlike subscripts for `vector` and `string`, the index of the built-in subscript operator is not an unsigned type. The index used with the built-in subscript operator can be a negative value. Of course, the resulting address must point to an element in (or one past the end of) the array to which the original pointer points.

3.1.3 Pointers and Arrays

In most expressions, when we use an object of array type, the compiler ordinarily converts the array to a pointer to the first element in that array.

One such implication is that when we use an array as an initializer for a variable defined using `auto`, the deduced type is a pointer, not an array. It is worth noting that this conversion does not happen when we use `decltype`. The type returned by `decltype` is array.

```
#include <iterator>
int *beg = std::begin(ia);
int *end = std::end(ia);
```

These functions act like the similarly named container members. They take an argument that is an array.

we can use an array to initialize a `vector`.

```
vector<int> ivec(std::begin(int_arr), std::end(int_arr));
```

3.1.4 Pointer Arithmetic

DEFINITION 3.6 **Pointer arithmetic** The arithmetic operations that can be applied to pointers. Pointers to arrays support the same operations as iterator arithmetic.

DEFINITION 3.7 **`ptrdiff_t`** Machine-dependent signed integral type defined in the `<cstddef>` header that is large enough to hold the difference between two pointers into the largest possible array.

3.1.5 C-style Character Strings

DEFINITION 3.8 **C-style strings** Null-terminated character array. String literals are C-style strings. C-style strings are inherently error-prone.

DEFINITION 3.9 **Null-terminated string** String whose last character is followed by the null character (`'\0'`).

3.2 Multidimensional Arrays

What are commonly referred to as multidimensional arrays are actually arrays of arrays. We define an array whose elements are arrays by providing two dimensions: the dimension of the array itself and the dimension of its elements.

```
// array of size 3; each element is an array of ints of size 4.
int ia[3][4];
```

3.2.1 Initializing the Elements of a Multidimensional Array

Multidimensional arrays may be initialized by specifying bracketed values for each row:

```
int ia[3][4] = {
    {0, 1, 2, 3},
    {4, 5, 6, 7},
    {8, 9, 10, 11}
};
```

The nested braces are optional.

3.2.2 Subscripting a Multidimensional Array

If an expression provides as many subscripts as there are dimensions, we get an element with the specified type. If we supply fewer subscripts than there are dimensions, then the result is the inner-array element at the specified index.

Use a pair of nested for loops to process the elements in a multidimensional array.

```
constexpr size_t m = 3, n = 4;
int ia[m][n];
for (size_t i = 0; i != m; ++i) {
  for (size_t j = 0; j != n; ++j) {
    ia[i][j] = i * n + j;
  }
}
```

3.2.3 Using a Range `for` with Multidimensional Arrays

To use a multidimensional array in a range `for`, the loop control variable for all but the innermost array must be references. We do so in order to avoid the normal array to pointer conversion.

If we want to change the element.

```
size_t count = 0;
for (auto &row: ia) {
  for (auto &val: row) {
    val = count++;
  }
}
```

If we do not write to the elements.

```
for (auto &row: ia) {
  for (auto val: row) {
    cout << val << endl;
  }
}
```

If `row` is not a reference, when the compiler initializes `row` it will convert each array element (like any other object of array type) to a pointer to that array's first element. As a result, in this loop the type of `row` is `int *`. The inner for loop is illegal since that loop attempts to iterate over an `int *`.

3.2.4 Pointers and Multidimensional Arrays

When we use the name of a multidimensional array, it is automatically converted to a pointer to the first element in the array. That is, a pointer to the first inner array.

```
int ia[3][4];
int (*p)[4] = ia; // p points to an array of four ints.
// p now points to the last element in ia.
p = &ia[2];
p = ia + 2;
```

4 Dynamic Memory

4.1 Concepts

DEFINITION 4.1 **Stack** Used for non-`static` objects defined inside functions. Stack objects exist only while the block in which they are defined is executing.

DEFINITION 4.2 **Static memory** Used for local `static` objects, for class `static` data members, and for variables defined outside any function. `static` objects are allocated before they are used, and they are destroyed when the program ends.

Objects allocated in stack or `static` memory are automatically created and destroyed by the compiler.

DEFINITION 4.3 **Heap/Free store** Memory pool available to a program to hold dynamically allocated objects.

DEFINITION 4.4 **Dynamically allocated** Object that is allocated on the free store. Objects allocated on the free store exist until they are explicitly deleted.

Programs tend to use dynamic memory for one of three purposes

- They don't know how many objects they'll need, e.g., container classes.
- They don't know the precise type of the objects they need, e.g., in OOP.
- They want to share data between several objects.

4.2 Managing Memory Directly

Classes that do manage their own memory — unlike those that use smart pointers — cannot rely on the default definitions for the members that copy, assign, and destroy class objects.

It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed.

4.2.1 `new`

DEFINITION 4.5 **`new`** Allocates memory from the heap. `new T` allocates and constructs an object of type `T` and returns a pointer to that object; if `T` is an array type, new returns a pointer to the first element in the array. Similarly, `new T[n]` allocates `n` objects of type `T` and returns a pointer to the first element in the array. By default, the allocated object is default initialized. We may also provide optional initializers. `n` must have integral type but need not be a constant.

We can initialize a dynamically allocated object using direct initialization. We can use traditional construction (using parentheses), we can also use list initialization (with curly braces):

```
int *pi1 = new int;  // Default initialized.
int *pi2 = new int();  // Value initialized.


int *pi = new int(1024);
string *ps = new string(10, 'c');
vector<int> *pv = new vector<int>{0, 1, 2, 3};
int *pia = new int[3]{0, 1, 2, 3};
```

It is legal to use `new` to allocate `const` object.

```
const int *pci = new const int(1024);
```

DEFINITION 4.6 **Placement `new`** Form of `new` that takes additional arguments passed in parentheses following the keyword `new`; for example, `new (nothrow) int` tells `new` that it should not throw an exception. `nothrow` is defined in the `<new>` header.

4.2.2 `delete`

DEFINITION 4.7 **`delete`** Frees memory allocated by `new`. `delete p` frees the object and `delete [] p` frees the array to which `p` points. `p` may be null or point to memory allocated by `new`.

DEFINITION 4.8 **Dangling pointer** A pointer that refers to memory that once had an object but no longer does. After the `delete`, the pointer becomes a dangling pointer. Program errors due to dangling pointers are notoriously difficult to debug.

Resetting the value of a pointer after a `delete` to `nullptr` can make it clear that the pointer points to no object. However, when several pointers point to the same memory. Resetting the pointer we use to `delete` that memory has no effect on any of the other pointers that still point at the (freed) memory.

4.3 Smart Pointers

4.3.1 Concepts

DEFINITION 4.9 **Smart pointer** Library type that acts like a pointer but can be checked to see whether it is safe to use. The type takes care of deleting memory when appropriate.

DEFINITION 4.10 **Reference count** Counter that tracks how many users share a common object. Used by smart pointers to know when it is safe to delete memory to which the pointers point.

The library define three kinds of smart pointers in `<memory>`

DEFINITION 4.11 **`shared_ptr`** Smart pointer that provides shared ownership: The object is deleted when the last `shared_ptr` pointing to that object is destroyed.

DEFINITION 4.12 **`unique_ptr`** Smart pointer that provides single ownership: The object is deleted when the `unique_ptr` pointing to that object is destroyed. `unique_ptr` cannot be directly copied or assigned.

DEFINITION 4.13 **`weak_ptr`** Smart pointer that points to an object managed by a `shared_ptr`. The `shared_ptr` does not count `weak_ptr` when deciding whether to delete its object.

4.3.2 Smart Pointers and Exceptions

When a function is exited, whether through normal processing or due to an exception, all the local objects are destroyed. In this case, the smart pointer class ensures that memory is freed when it is no longer needed even if the block is exited prematurely. In contrast, memory that we manage directly is not automatically freed when an exception occurs. If an exception happens between the `new` and the `delete`, then this memory can never be freed.

Classes that allocate resources, and that do not define destructors to free those resources, can be subject to the same kind of errors that arise when we use dynamic memory. If an exception happens between when the resource is allocated and when it is freed, the program will leak that resource.

DEFINITION 4.14 **Deleter** Function passed to a smart pointer to use in place of `delete` when destroying the object to which the pointer is bound.

4.3.3 Smart Pointer Pitfalls

- Don't use the same built-in pointer value to initialize (or reset) more than one smart pointer.
- Don't `delete` the pointer returned from `get()`.
- Don't use `get()` to initialize or reset another smart pointer.
- If you use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.
- If you use a smart pointer to manage a resource other than memory allocated by `new`, remember to pass a deleter.

4.4 Dynamic Arrays

Most applications should use a library container such as `vector` rather than dynamically allocated arrays. Using a container is easier, less likely to contain memory-management bugs, and is likely to give better performance.

DEFINITION 4.15 **`allocator`** Library class that allocates unconstructed memory.

5 Expressions

5.1 Lvalue and Rvalue

DEFINITION 5.1 **Lvalue** An expression that yields an object or function. A non-`const` lvalue that denotes an object may be the left-hand operand of assignment.

DEFINITION 5.2 **Rvalue** Expression that yields a value but not the associated location, if any, of that value.

Roughly speaking, when we use an object as an rvalue, we use the object's value (its contents). When we use an object as an lvalue, we use the object's identity (its location in memory).

We can use an lvalue when an rvalue is required, but we cannot use an rvalue when an lvalue (i.e., a location) is required.

5.2 Operators

DEFINITION 5.3 **Operands** Values on which an expression operates. Each operator has one or more operands associated with it.

DEFINITION 5.4 **Operator** Symbol that determines what action an expression performs. The language defines a set of operators and what those operators mean when applied to values of built-in type. The language also defines the precedence and associativity of each operator and specifies how many operands each operator takes. Operators may be overloaded and applied to values of class type.

DEFINITION 5.5 **Compound expression** An expression involving more than one operator.

DEFINITION 5.6 **Call operator (`()`)** A pair of parentheses "`()`" following a function name. The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

DEFINITION 5.7 **Dot operator (`.`)** Left-hand operand must be an object of class type and the right-hand operand must be the name of a member of that object. The operator yields the named member of the given object.

DEFINITION 5.8 **Scope operator (`::`)** Among other uses, the scope operator is used to access names in a namespace. For example, `std::cout` denotes the name `cout` from the namespace `std`.

DEFINITION 5.9 **= operator** Assigns the value of the right-hand operand to the object denoted by the left-hand operand.

DEFINITION 5.10 **Comma operator (`,`)** operator. Binary operator that is evaluated left to right. The result of a comma expression is the value of the right-hand operand. The result is an lvalue if and only if that operand is an lvalue.

DEFINITION 5.11 **Conditional operator (`?:`)** Provides an if-then-else expression of the form

```
cond ? expr1 : expr2;
```

If the condition cond is true, then expr1 is evaluated. Otherwise, expr2 is evaluated. The type expr1 and expr2 must be the same type or be convertible to a common type. Only one of expr1 or expr2 is evaluated.

DEFINITION 5.12 **Address-of operator (`&`)** Yields the address of the object to which it is applied.

DEFINITION 5.13 **Dereference operator (`*`)** Dereferencing a pointer returns the object to which the pointer points. Assigning to the result of a dereference assigns a new value to the underlying object.

DEFINITION 5.14 **Subscript operator (`[]`)** obj[i] yields the element at position i from the container object obj. Indices count from zerothe first element is element 0 and the last is the element indexed by obj.size() - 1. Subscript returns an object. If p is a pointer and n an integer, p[n] is a synonym for *(p+n).

DEFINITION 5.15 **Arrow operator (`->`)** Combines the operations of dereference and dot operators: a-¿b is a synonym for (*a).b.

DEFINITION 5.16 **Output operator (`<<`)** The << operator takes two operands: The left-hand operand must be an ostream object; the right-hand operand is a value to print. The operator writes the given value on the given ostream. The result of the output operator is its left-hand operand. That is, the result is the ostream on which we wrote the given value. Therefore, output operations can be chained together.

DEFINITION 5.17 **Input operator (`>>`)** It takes an istream as its left-hand operand and an object as its right-hand operand. It reads data from the given istream and stores what was read in the given object. Like the output operator, the input operator returns its left-hand operand as its result. Therefore, input operations can be chained together.

DEFINITION 5.18 **sizeof** Operator that returns the size, in bytes, to store an object of a given type name or of the type of a given expression.

5.3 Precedence and Associativity

DEFINITION 5.19 **Precedence** Defines the order in which different operators in a compound expression are grouped. Operators with higher precedence are grouped more tightly than operators with lower precedence.

DEFINITION 5.20 **Associativity** Determines how operators with the same precedence are grouped. Operators can be either right associative or left associative.

5.4 Order of Evaluation

DEFINITION 5.21 **Order of evaluation** Order in which the operands to an operator are evaluated. Order of operand evaluation is independent of precedence and associativity. In most cases, the compiler is free to evaluate operands in any order. However, the operands are always evaluated before the operator itself is evaluated.

In an expression such as `f() + g() * h() + j()`, there are no guarantees as to the order in which these functions are called.

There are four operators that do guarantee the order in which operands are evaluated

- Logical AND operator (`&&`).
- Logical OR (`||`).
- Conditional operator (`? :`).
- Comma operator (`,`).

DEFINITION 5.22 **Short-circuit evaluation** Term used to describe how the logical AND and logical OR operators execute. If the first operand to these operators is sufficient to determine the overall result, evaluation stops. We are guaranteed that the second operand is not evaluated.

5.5 Implicit Type Conversions

5.5.1 Conversion and Implicit Conversion

DEFINITION 5.23 **Conversion** Process whereby a value of one type is transformed into a value of another type. The language defines conversions among the built-in types. Conversions to and from class types are also possible.

DEFINITION 5.24 **Implicit conversion** A conversion that is automatically generated by the compiler. Given an expression that needs a particular type but has an operand of a

differing type, the compiler will automatically convert the operand to the desired type if an appropriate conversion exists.

5.5.2 When Implicit Conversions Occur

- In conditions, non-`bool` expressions are converted to `bool`.
- In initializations, the initializer is converted to the type of the variable; in assignments, the right-hand operand is converted to the type of the left-hand.
- In arithmetic and relational expressions with operands of mixed types, the types are converted to a common type.
- Conversions also happen during function calls.

5.5.3 The Arithmetic Conversion

DEFINITION 5.25 **Arithmetic conversion** A conversion from one arithmetic type to another. In the context of the binary arithmetic operators, arithmetic conversions usually attempt to preserve precision by converting a smaller type to a larger type (e.g., integral types are converted to floating point).

The implicit conversions among the arithmetic types are defined to preserve precision, if possible.

5.5.3.1 Conversions in Initialization and Assignment

- non-`bool` arithmetic types → `bool`: `false` if the value is `0`; `true` otherwise.
- `bool` → non-`bool` arithmetic types: `1` if the `bool` is `true`; `0` if the `bool` is `false`.
- Floating-point type → integral type: the value is truncated. The value that is stored is the part before the decimal point.
- Integral type → floating-point type: the fractional part is zero. Precision may be lost if the integer has more bits than the floating-point object can accommodate.
- Out-of-range value → unsigned type: the remainder of the value modulo the number of values the target type can hold.
- Out-of-range value → signed type: the result is undefined.

5.5.3.2 Integral Promotions

DEFINITION 5.26 **Integral promotions/Promoted** Conversions that take a smaller integral type to its most closely related larger integral type. Operands of small integral types (e.g., `short`, `char`, etc) are always promoted, even in contexts where such conversions might not seem to be required.

The types `bool`, `char`, `signed char`, `unsigned char`, `short`, and `unsigned short` are promoted to `int` if all possible values of that type fit in an `int`. Otherwise, the value is promoted to `unsigned int`.

5.5.3.3 Operands of Unsigned Type

As usual, integral promotions happen first. If the resulting type(s) match, no further conversion is needed. If both (possibly promoted) operands have the same signedness, then the operand with the smaller type is converted to the larger type.

When the signedness differs:

- If the type of the unsigned operand is the same as or larger than that of the signed operand, the signed operand is converted to unsigned.
- If the signed operand has a larger type than the unsigned operand. In this case, the result is machine dependent.
  - If all values in the unsigned type fit in the larger type, then the unsigned operand is converted to the signed type.
  - If the values don't fit, then the signed operand is converted to the unsigned type.

For example, given an `unsigned` and an `int`, the `int` is converted to `unsigned`.

If the operands are `long` and `unsigned int`, and `int` and `long` have the same size, the `long` will be converted to `unsigned int`. If the `long` type has more bits, then the `unsigned int` will be converted to `long`.

5.5.3.4 Best Practice

Don't mix signed and unsigned types. If we use both `unsigned` and `int` values in an arithmetic expression, the `int` value ordinarily is converted to `unsigned`. Likewise, if we subtract a value from an `unsigned`, we must be sure that the result cannot be negative.

5.5.4 Other Implicit Conversions

5.5.4.1 Array to Pointer Conversions

In most expressions, when we use an array, the array is automatically converted to a pointer to the first element in that array.

This conversion is not performed when an array is used with `decltype` or as the operand of the address-of (`&`), `sizeof`, or `typeid`. The conversion is also omitted when we initialize a reference to an array.

5.5.4.2 Function to Pointer Conversions

5.5.4.3 Pointer Conversions

A constant integral value of `0` and the literal `nullptr` can be converted to any pointer type; a pointer to any non-`const` type can be converted to `void *`, and a pointer to any type can be converted to a `const void *`.

There is an additional pointer conversion that applies to types related by inheritance.

5.5.4.4 Conversions to `bool`

There is an automatic conversion from arithmetic or pointer types to `bool`. If the pointer or arithmetic value is zero, the conversion yields `false`; any other value yields `true`.

5.5.4.5 Conversion to `const`

We can convert a pointer to a non-`const` type to a pointer to the corresponding `const` type, and similarly for references.

The reverse conversion — removing a low-level `const` — does not exist.

5.5.4.6 Conversions Defined by Class Types

Class types can define conversions that the compiler will apply automatically. The compiler will apply only one class-type conversion at a time.

5.6 Explicit Type Conversions

DEFINITION 5.27 **Cast** An explicit conversion.

5.6.1 `static_cast`

DEFINITION 5.28 **`static_cast`** An explicit request for a well-defined type conversion. Often used to override an implicit conversion that the compiler would otherwise perform.

For example,

```
int i, j;
double ret = static_cast<double>(i) / j;
```

A `static_cast` is often useful when a larger arithmetic type is assigned to a smaller type to turn off the warning generated by compilers.

A `static_cast` is also useful to perform a conversion that the compiler will not generate automatically. For example, we can use a `static_cast` to retrieve a pointer value that was stored in a `void *` pointer.

```
void *p = &d;
double *dp = static_cast<double *>(p);
```

5.6.2 `const_cast`

DEFINITION 5.29 **`const_cast`** A cast that converts a low-level `const` object to the corresponding non-`const` type or vice versa.

```
const char *cp;
// Ok, but writing through p is undefined.
char *p = const_cast<char *>(cp);
```

Once we have cast away the `const` of an object, the compiler will no longer prevent us from writing to that object. If the object was originally not a `const`, using a cast to obtain write access is legal. However, using a `const_cast` in order to write to a `const` object is undefined.

`const_casts` are most useful in the context of overloaded functions. For a function

```
const string &shorter(const string &s1, const string &s2) {
  return s1.size() <= s2.size() ? s1 : s2;
}
```

We can call the function on a pair of non-`const string` arguments, but we'll get a reference to a const string as the result. We might want to have a version of `shorter` that, when given non-`const` arguments, would yield a plain reference. We can write this version of our function using a `const_casts`.

```
string &shorter(string &s1, string &s2) {
  auto &shorter_str = shorter(
      const_cast<const string &>(s1),
      const_cast<const string &>(s2));
  return const_cast<string &>(shorter_str);
}
```

Becase we know the return value is bound to one of our original, non-`const` arguments. It is safe to cast that `string` back to a plain `string &` in the return.

5.6.3 `reinterpret_cast`

DEFINITION 5.30 **`reinterpret_cast`** Interprets the contents of the operand as a different type. Inherently machine dependent and dangerous.

```
int *pi;
char *pc = reinterpret_cast<char *>(pi);
```

5.6.4 `dynamic_cast`

DEFINITION 5.31 **`dynamic_cast`** Used in combination with inheritance and run-time type identification.

6 Statements

6.1 Basic Concepts

DEFINITION 6.1 **Expression statement** An expression followed by a semicolon. An expression statement causes the expression to be evaluated.

DEFINITION 6.2 **null statement** An empty statement. Indicated by a single semicolon.

DEFINITION 6.3 **Flow of control** Execution path through a program.

DEFINITION 6.4 **Block/Compound statement** Sequence of zero or more statements enclosed in curly braces. A block is a statement, so it can appear anywhere a statement is expected.

DEFINITION 6.5 **Curly brace** Curly braces delimit blocks. An open curly ({) starts a block; a close curly (}) ends one.

6.2 Statements

DEFINITION 6.6 **`goto` statement** Statement that causes an unconditional transfer of control to a specified labeled statement elsewhere in the same function. `goto` obfuscate the flow of control within a program and should be avoided.

DEFINITION 6.7 **labeled statement** Statement preceded by a label. A label is an identifier followed by a colon. Label identifiers are independent of other uses of the same identifier.

DEFINITION 6.8 **`if` statement** Conditional execution based on the value of a specified condition. If the condition is true, the `if` body is executed. If not, control flows to the statement following the `if`.

DEFINITION 6.9 **`if else` statement** Conditional execution of code following the `if` or the `else`, depending on the truth value of the condition.

DEFINITION 6.10 **Dangling `else`** Colloquial term used to refer to the problem of how to process nested `if` statements in which there are more `if` than `else`. In C++, an `else` is always paired with the closest preceding unmatched `if`. Note that curly braces can be used to effectively hide an inner `if` so that the programmer can control which `if` a given `else` should match.

DEFINITION 6.11 **`for` statement** Iteration statement that provides iterative execution. Ordinarily used to step through a container or to repeat a calculation a given number of times.

DEFINITION 6.12 **Range `for`** Statement that iterates through a sequence.

DEFINITION 6.13 **`while` statement** Iteration statement that provides iterative execution so long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

DEFINITION 6.14 **`do while` statement** Like a while, except that the condition is tested at the end of the loop, not the beginning. The statement inside the do is executed at least once.

```
do {
  ...
} while (...);
```

Variables used in condition must be defined outside the body of the `do while` statement.

DEFINITION 6.15 **`break` statement** Terminates the nearest enclosing loop or switch statement. Execution transfers to the first statement following the terminated loop or switch.

DEFINITION 6.16 **`continue` statement** Terminates the current iteration of the nearest enclosing loop. Execution transfers to the loop condition in a while or do, to the next iteration in a range for, or to the expression in the header of a traditional for loop.

DEFINITION 6.17 **`switch` statement** A conditional statement that starts by evaluating the expression that follows the switch keyword. Control passes to the labeled statement with a case label that matches the value of the expression. If there is no matching label, execution either continues at the default label, if there is one, or falls out of the switch if there is no default label.

```
switch (c) {
case 'a':
  ++cnt_a;
  break;
case 'b':
case 'c':
  ++cnt_bc;
  break;
default:
  break;
}
```

DEFINITION 6.18 **`case` label** Constant expression that follows the keyword `case` in a `switch` statement. No two `case` labels in the same `switch` statement may have the same value.

DEFINITION 6.19 **`default` label** `case` label that matches any otherwise unmatched value computed in the `switch` expression.

7 Exception Handling

7.1 Exception Handling Overview

DEFINITION 7.1 **Exception** Run-time anomalies that exist outside the normal functioning of a program, such as losing a database connection or encountering unexpected input.

DEFINITION 7.2 **Exception handling** Language-level support for managing run-time anomalies. One independently developed section of code can detect and "raise" an exception that another independently developed part of the program can "handle". The error-detecting part of the program `throw` an exception; the error-handling part handles the exception in a `catch` clause of a `try` block.

Exception handling is generally used when one part of a program detects a problem that it cannot resolve. The designer can recognize the problem, but do not know how serious the problem is to the overall program. Having signaled what happened, the detecting part stops processing.

```
void f() {
  ...
  throw runtime_error("Error");
}

try {
  f();
} catch (runtime_error e) {  // Exception declaration.
  cerr << e.what() << endl;  // Exception handler.
}
```

7.2 A `throw` Expression

DEFINITION 7.3 **`throw e`** Expression that interrupts the current execution path. Each `throw` transfers control to the nearest enclosing `catch` clause that can handle the type of exception that is thrown. The expression `e` is copied into the exception object.

DEFINITION 7.4 **Exception object** Object used to communicate between the `throw` and `catch` sides of an exception to inform the handling part about what went wrong. The object is created at the point of the `throw` and is a copy of the thrown expression. The exception object exists until the last handler for the exception completes. The type of the object is the `static` type of the thrown expression.

DEFINITION 7.5 **Exception class** Set of classes defined by the standard library to be used to represent errors.

DEFINITION 7.6 **Raise/Throw** Often used as a synonym for throw. C++ programmers speak of "throwing" or "raising" an exception interchangably.

7.3 The `try` Block

DEFINITION 7.7 **`try` block** Block of statements enclosed by the keyword `try` and one or more `catch` clauses. If the code inside the `try` block raises an exception and one of the `catch` clauses matches the type of the exception, then the exception is handled by that `catch`. Otherwise, the exception is passed out of the `try` to a `catch` further up the call chain.

DEFINITION 7.8 **`catch` clause/Exception handler** Part of the program that handles an exception. A `catch` clause consists of the keyword `catch` followed by an exception declaration and a block of statements. The selected handler is the one nearest in the call chain that matches the type of the thrown object. The code inside a `catch` does whatever is necessary to handle an exception of the type defined in its exception declaration.

Multiple `catch` clauses with types related by inheritance must be ordered from most derived type to least derived.

DEFINITION 7.9 **Exception declaration** `catch` clause declaration that specifies the type of exception that the `catch` can handle. The declaration acts like a parameter list, whose single parameter is initialized by the exception object. If the exception specifier is a nonreference type, then the exception object is copied to the catch.

Variables declared inside a `try` block are inaccessible outside the block — in particular, they are not accessible to the `catch` clauses.

DEFINITION 7.10 **Stack unwinding** The process whereby the functions are exited in the search for a `catch`. It continues up the chain of nested function calls until a `catch` clause for the exception is found, or the `main` function itself is exited without having found a matching `catch`. Local objects constructed before the exception are destroyed before entering the corresponding `catch`.

During stack unwinding, destructors are run on local objects of class type. Because destructors are run automatically, they should not `throw`. If, during stack unwinding, a destructor throws an exception that it does not also `catch`, the program will be terminated. In practice, because destructors free resources, it is unlikely that they will throw exceptions. All of the standard library types guarantee that their destructors will not raise an exception.

DEFINITION 7.11 **`terminate`** Library function that is called if an exception is not caught or if an exception occurs while a handler is in process. `terminate` ends the program.

DEFINITION 7.12 **Rethrow** A `throw` that does not specify an expression. A rethrow is valid only from inside a `catch` clause, or in a function called directly or indirectly from a `catch`. Its effect is to rethrow the exception object that it received.

DEFINITION 7.13 **Exception safe** Term used to describe programs that behave correctly when exceptions are thrown.

DEFINITION 7.14 **Catch-all** A `catch` clause in which the exception declaration is `(...)`. A catch-all `clause` catches an exception of any type. It is typically used to catch an exception that is detected locally in order to do local cleanup. The exception is then rethrown to another part of the program to deal with the underlying cause of the problem.

The only way for a constructor to handle an exception from a constructor initializer is to write the constructor as a function `try` block.

DEFINITION 7.15 **Function `try` block** Used to catch exceptions from a constructor initializer. The keyword `try` appears before the colon that starts the constructor initializer list (or before the open curly of the constructor body if the initizlier list is empty) and closes with one or more `catch` clauses that appear after the close curly of the constructor body.

```
A(initializer_list<T> il) try:
    _data(il) {
  ...
} catch (const bad_alloc &e) { ... }
```

DEFINITION 7.16 **Nonthrowing specification** An exception specification that promises that a function won't throw. If a nonthrowing functions does throw, `terminate` is called. Nonthrowing specifiers are `noexcept` without an argument or with an argument that evaluates as `true` and `throw()`.

DEFINITION 7.17 **`noexcept` specification** Keyword used to indicate whether a function throws. When `noexcept` follows a function's parameter list, it may be optionally followed by a parenthesized constant expression that must be convertible to `bool`. If the expression is omitted, or if it is `true`, the function throws no exceptions. An expression that is `false` or a function that has no exception specification may throw any exception.

II PROCEDURE PROGRAMMING

8 Functions

8.1 Function Basics

DEFINITION 8.1 **Procedural Programming** A programming paradigm based upon the concept of the procedure call. Procedures, also known as functions, simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

Procedural programming has three benefits

- Programs are simpler to understand because they are a sequence of function calls rather than the code sequence to accomplish each operation.
- Functions can be reused.
- It is easier to distribute the work across multiple programmers or groups within a project.

8.1.1 Writing a Function

DEFINITION 8.2 **Function** Independent name unit that performs some computation. It consists of four parts: the return type, the function name, the parameter list, and the function body.

DEFINITION 8.3 **Return type** Type of the value returned by a function.

DEFINITION 8.4 **Function name** Name by which a function is known and can be called.

DEFINITION 8.5 **Function body** Block that defines the actions performed by a function.

DEFINITION 8.6 **Parameter list** Part of the definition of a function. Possibly empty list that specifies what arguments can be used to call the function.

DEFINITION 8.7 **Function prototype/Interface** Function declaration, consisting of the name, return type, and parameter types of a function. To call a function, its prototype must have been declared before the point of call.

8.1.2 Parameters and Arguments

When we invoke a function, a specail area of memory is set up on what is called the program stack. Within this special area of memory, there is space to hold the value of each function parameter and local objects defined within the function. When the function completes, this area of memory is discarded (We say that it is popped from the program stack).

DEFINITION 8.8 **Parameters** Local variables declared inside the function parameter list. Parameters are initialized by the arguments provided in each function call. Parameter initialization works the same way as variable initialization.

DEFINITION 8.9 **Arguments** Values supplied in a function call that are used to initialize the function's parameters. We have no guarantees about the order in which arguments are evaluated.

A good function can handle all kinds of bad inputs. One way is to check the legality of parameters, and throw an exception when an invalid parameter is found.

It is better to communicate between functions using parameters rather than use objects defined at global scope. One reason is that a function that is dependent on an object defined at global scope is harder to reuse in a different context.

DEFINITION 8.10 **Local `static` objects** Local objects whose value persists across calls to the function. Local `static` objects that are created and initialized before the first time execution passes through the object's definition, and are destroyed when the program ends. If a local static has no explicit initializer, it is value initialized.

In multi-core programming, local `static` objects are very hard to deal with.

8.2 Argument Passing

8.2.1 Passing Arguments by Value

DEFINITION 8.11 **Pass by value** How arguments are passed to parameters of a non-reference type. A non-reference parameter is a copy of the value of its corresponding argument. Changes made to the parameter have no effect on the argument.

Pointers behave like any other nonreference type. When we copy a pointer, the value of the pointer is copied. After the copy, the two pointers are distinct. However, a pointer also gives us indirect access to the object to which that pointer points. We can change the value of that object by assigning through the pointer.

8.2.2 Passing Arguments by Reference

DEFINITION 8.12 **Pass by reference** Description of how arguments are passed to parameters of reference type. Reference parameters work the same way as any other use of references; the parameter is bound to its corresponding argument. Reference parameters often used to allow a function to change the value of one or more of its arguments.

Passing arguments by reference has two advantages

- Allow us to modify directly the actual object being passed to the function.
- Eliminate the overhead of copying a large object. Reference parameters that are not changed inside a function should be references to `const`.

A function can return only a single value. However, sometimes a function has more than one value to return. Reference parameters let us effectively return multiple results.

It is not recommend to pass built-in types by reference. The reference mechanism is primarily intended to support the passing of class objects as parameter to functions.

8.2.3 `const` Parameters and Arguments

We can pass either a `const` or a non-`const` object to a parameter that has a top-level `const`. A parameter that has a top-level `const` is indistinguishable from one without a top-level `const`.

```
void fcn(const int i) ...
void fcn(int i)  // Error: redefines fcn(int).
```

On the other hand, we can overload based on whether the parameter is a reference (or pointer) to the `const` or non-`const` version of a given type; such `const` are low-level.

```
void fcn(const int &ri) ...
void fcn(int &ri)  // New function.
```

We can pass a `const` object (or a pointer to `const`) only to the version with a `const` parameter. Because there is a conversion to `const`, we can call either function on a non-`const` object or a pointer to non-`const`. However, the compiler will prefer the non-`const` versions when we pass a non-`const` object or pointer to non-`const`.

8.2.4 Array Parameters

Because we cannot copy an array, we cannot pass an array by value. Because arrays are converted to pointers, when we pass an array to a function, we are actually passing a pointer to the array's first element. Even though we cannot pass an array by value, we can write a parameter that looks like an array. Despite appearances, these three declarations of print are equivalent.

```
void print(const int *);
void print(const int[]);
void print(const int[10]);

int i = 0;
int arr[2] = {0, 1};
```

```
print(&i);
print(arr);
```

When a function does not need write access to the array elements, the array parameter should be a pointer to `const`. A parameter should be a plain pointer to a non-`const` type only if the function needs to change element values.

A multidimensional array is passed as a pointer to its first element. That is, a pointer to an array. The size of the second (and any subsequent) dimension is part of the element type and must be specified.

```
// A is a pointer to an array of ten ints.
void print(int (*A)[10]);
void print(int A[][10]);  // Equivalent definition.
```

Because arrays are passed as pointers, functions ordinarily don't know the size of the array they are given. There are three common techniques used to manage pointer parameters.

8.2.4.1 Using a Marker to Specify the Extent of an Array

Requires the array itself to contain an end marker. For example, C-style strings are stored in character arrays in which the last character of the string is followed by a null character.

```
void print(const char *cp) {
  if (!cp) {  // If cp is nullptr.
    return;
  }
  for (auto it = cp; *it; ++it) {
    // The character it points to is not a null character.
    cout << *it;
  }
}
```

8.2.4.2 Using the Standard Library Conventions

Pass pointers to the first and one past the last element in the array.

```
void print(const int *begin, const int *end) {
  for (auto it = begin, it != end; ++it) {
    cout << *it << endl;
  }
```

```
}
```

8.2.4.3 Explicitly Passing a Size Parameter

```
void print(const int *ia, size_t n) {
  for (size_t i = 0; i != n; ++i) {
    cout << ia[i] << endl;
  }
}
```

8.2.5 Functions with Varying Parameters

DEFINITION 8.13 **`initializer_list`** Library class that represents a comma-separated list of objects of a single type enclosed inside curly braces. The elements in an `initializer_list` are always `const` values.

```
void error_msg(initializer_list<string> il) {
  for (auto it = il.begin(); it != il.end(); ++it) {
    cout << *it << endl;
  }
}
```

We can also use a range `for` to process the elements.

When we pass a sequence of values to an `initializer_list` parameter, we must enclose the sequence in curly braces.

```
error_msg({"function", str1, str2});
```

A function with an `initializer_list` parameter can have other parameters as well.

8.3 Return Types and the `return` Statement

8.3.1 Functions That Return a Value

Values are returned in exactly the same way as variables and parameters are initialized: The return value is used to initialize a temporary at the call site, and that temporary is the result of the function call.

Never return a reference or pointer to a local object. When a function completes, its storage is freed, references to local objects refer to memory that is no longer valid.

Calls to functions that return references are lvalues; other return types yield rvalues.

Functions can return a braced list of values. As in any other return, the list is used to initialize the temporary that represents the function's return.

8.3.2 Returning a Pointer to an Array

Because we cannot copy an array, a function cannot return an array. However, a function can return a pointer or a reference to an array. The most straightforward way is to use a type alias.

```
using arr = int[10];
// Returns a pointer to an array of ten ints.
arr *fun(int i);
int (*fun(int i))[10];  // Equivalent form.
// Equivalent form using a trailing return type.
auto func(int i) -> int (*)[10];
```

8.4 Features of Specialized Uses

8.4.1 Default Arguments

DEFINITION 8.14 **Default argument** Value specified to be used when an argument is omitted in a call to the function.

If a parameter has a default argument, all the parameters that follow it must also have default arguments. We order the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last.

It is legal to redeclare a function multiple times. However, each parameter can have its default specified only once. Thus, any subsequent declaration can add a default only for a parameter that has not previously had a default specified. Default arguments ordinarily should be specified with the function declaration in an appropriate header.

Local variables may not be used as a default argument. Names used as default arguments are resolved in the scope of the function declaration. The value that those names represent is evaluated at the time of the call.

8.4.2 `inline` and `constexpr` Functions

DEFINITION 8.15 **`inline` function** Request to the compiler to expand a function at the point of call, if possible. Inline functions avoid the normal function-calling overhead.

The `inline` specification is only a request to the compiler. The compiler may choose to ignore this request.

DEFINITION 8.16 **`constexpr` Function** Function that may return a constant expression. A `constexpr` function is implicitly `inline`. The `return` type and the type of each parameter in a must be a literal type. A `constexpr` function is not required to return a constant expression.

`inline` and `constexpr` functions normally are defined in headers.

A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!

Another useful rule of thumb: it's typically not cost effective to inline functions with loops or switch statements.

Usually recursive functions should not be inline.

8.4.3 Overloaded Functions

DEFINITION 8.17 **Overloaded function** Function that has the same name as at least one other function. Overloaded functions must differ in the number or type of their parameters.

8.5 Function Matching

8.5.1 Function Matching Process

DEFINITION 8.18 **Function matching/Overload resolution** Compiler process by which a call to an overloaded function is resolved. Arguments used in the call are compared to the parameter list of each overloaded function.

There are three steps in function matching.

1. Identifies the candidate functions.
2. Identifies the visible functions from the set of candidate functions.
3. Looks at each argument in the call and selects the viable function for which the corresponding parameter best matches the argument.

DEFINITION 8.19 **Candidate function** Set of functions that are considered when resolving a function call. The candidate functions are all the functions with the name used in the call for which a declaration is in scope at the time of the call.

DEFINITION 8.20 **Viable functions** Subset of the candidate functions that could match a given call. Viable functions have the same number of parameters as arguments to the call, and each argument type can be converted to the corresponding parameter type.

8.5.2 Three Possible Outcomes

DEFINITION 8.21 **No match** Compile-time error that results during function matching when there is no visible function with parameters that match the arguments in a given call.

DEFINITION 8.22 **Best match** Function selected from a set of overloaded functions for a call. If a best match exists, the selected function is a better match than all the other viable candidates for at least one argument in the call and is no worse on the rest of the arguments.

DEFINITION 8.23 **Ambiguous call** Compile-time error that results during function matching when two or more functions provide an equally good match for a call.

Casts should not be needed to call an overloaded function. The need for a cast suggests that the parameter sets are designed poorly.

8.5.3 Argument Type Conversions

Conversions are ranked as follows.

1. An exact match, including array or function type to the corresponding pointer type, and a top-level `const` is added to or discarded from the argument.
2. Match through a low-level `const` conversion.
3. Match through a promotion.
4. Match through an arithmetic or pointer conversion.
5. Match through a class-type conversion.

8.6 Pointers to Functions

A function's type is determined by its return type and the types of its parameters. The function's name is not part of its type. The type of the pointer must match one of the overloaded functions exactly. The most straightforward way is to use a type alias.

```
// pf points to a function returning bool that takes two const
// string references.
using fun = bool (const string &, const string &);
fun *pf = nullptr;
bool (*pf)(const string &, const string &) = nullptr;
```

When we use the name of a function as a value, the function is automatically converted to a pointer.

```
pf = shorter;
pf = &shorter;  // Equivalent assignment.
```

We can use a pointer to a function to call the function to which the pointer points.

```
pf("a", "b");
(*pf)("a", "b");  // Equivalent assignment.
fun("a", "b");  // Equivalent assignment.
```

Just as with arrays, we cannot define parameters of function type but can have a parameter that is a pointer to function. As with arrays, we can write a parameter that looks like a function type, but it will be treated as a pointer:

```
void bigger(const string &str1, const string &str2,
    bool pf(const string &, const string &));
// Equivalent form.
void bigger(const string &str1, const string &str2,
    bool (*pf)(const string &, const string &));
// Equivalent form.
void bigger(const string &str1, const string &str2, fun pf);
// Equivalent form.
void bigger(const string &str1, const string &str2, fun *pf);
```

When we pass a function as an argument, we can do so directly. It will be automatically converted to a pointer

```
g(str1, str2, shorter);
```

As with arrays, we can't return a function type but can return a pointer to a function type. Similarly, we must write the return type as a pointer type. Unlike what happens to parameters that have function type, the return type is not automatically converted to a pointer type. We must explicitly specify that the return type is a pointer type

```
fun *g(int);
// Equivalent form.
bool (*g(int))(const string &, const string &);
// Equivalent form using a trailing return type.
auto g(int) -> bool (*)(const string &, const string &);
```

9 Other Callable Objects

DEFINITION 9.1 **Callable object** Object that can appear as the left-hand operand of the call operator. Pointers to functions, lambdas, and objects of a class that defines an overloaded function call operator are all callable objects.

C++ has several kinds of callable objects

- Functions.
- Pointers to functions.
- Lambdas.
- Objects created by `bind`.
- Function object.

DEFINITION 9.2 **Call signature** Represents the interface of a callable object. A call signature includes the return type and a comma-separated list of argument types enclosed in parentheses.

A call signature corresponds to a function type. For example

```
int (int, int)
```

DEFINITION 9.3 **Function table** Container, often a `map` or a `vector`, that holds values that can be called.

When using a `map`, we'll use a `string` corresponding to an operator symbol as the key; the value will be the function that implements that operator. When we want to evaluate a given operator, we'll index the `map` with that operator and call the resulting element.

DEFINITION 9.4 **Function object** Object of a class that defines an overloaded call operator. Function objects can be used where functions are normally expected.

9.1 Lambda Expressions

9.1.1 Lambda Expressions

DEFINITION 9.5 **Lambda expression** Callable unit of code. A lambda is somewhat like an unnamed, inline function.
`[`capture list`]` `(`parameter list`)` `->` return type `{`function body`}`
A lambda can omit the return type. If the function body is a single return statement, the return type is inferred from the type of the object that is returned. Otherwise, an omitted return type defaults to void. A lambda may not have default arguments.

DEFINITION 9.6 **Capture list** A comma-separated list of names defined in the surrounding function that specifies which variables from the surrounding context the lambda expression may access. A lambda may use a variable local to its surrounding function only if the lambda captures that variable in its capture list. The capture list is used for local non-`static` variables only; lambdas can use local `static` and variables declared outside the function (like `cout`) directly.

If we want to be able to change the value of a captured variable, we must follow the parameter list with the keyword `mutable`.

9.1.2 Classes Representing Lambdas

When we write a lambda, the compiler translates that expression into an unnamed object of an unnamed class. The classes generated from a lambda contain an overloaded function-call operator. For example,

```
stable_sort(v.begin(), v.end,(),
    [] (const string &str1, const string &str2)
    { return str1.size() < str2.size(); });
```

acts like an unnamed object of a class that would look like

```
class lt {
public:
  bool operator()(const string &str1, const string &str2) const {
    return str1.size() < str2.size();
  }
};
stable_sort(v.begin(), v.end,(), lt());
```

The third argument is a newly constructed `lt` object. The code in `stable_sort` will "call" this object each time it compares two `string`s. When the object is called, it will execute the body of its call operator.

By default, the function-call operator in a class generated from a lambda is a `const` member function. If the lambda is declared as `mutable`, then the call operator is not `const`.

When a lambda captures a variable by reference, the compiler is permitted to use the reference directly without storing that reference as a data member in the generated class.

In contrast, variables that are captured by value are copied into the lambda. As a result, classes generated from lambdas that capture variables by value have data members corresponding to each such variable. For example,

```
find_if(v.begin(), v.end(),
    [size] (const string &s) { return s.size() > size; });
```

would generate a class that looks like

```
class p {
private:
  size_t _size;

public:
  p(size_t size): _size(size) {}
  bool operator()(const string &s) const {
    return s.size() > _size;
  }
}
find_if(v.begin(), v.end(), p(size));
```

9.2 Library-Defined Function Objects

The standard library defines a set of classes that represent the arithmetic, relational, and logical operators. Each class defines a call operator that applies the named operation.

III GENERIC PROGRAMMING I

10 Generic Programming and Function Template

10.1 Basic Concepts

DEFINITION 10.1 **Generic Programming** The containers, iterators, and generic algorithms are all examples of generic programming. When we write a generic program, we write the code in a way that is independent of any particular type. When we use a generic program, we supply the type(s) or value(s) on which that instance of the program will operate. Templates are the foundation of generic programming.

DEFINITION 10.2 **Template** A blueprint or formula from which specific class types or functions can be created. To use a class template, we must specify additional information such as the element type to transform that blueprint into a specific class or function. That transformation happens during compilation.

DEFINITION 10.3 **Function template** Definition from which specific functions can be instantiated. A function template is defined using the `template` keyword followed by a comma-separated list of one or more template parameters enclosed in <> brackets, followed by a function definition.

DEFINITION 10.4 **Class template** Definition from which specific classes can be instantiated. Class templates are defined using the `template` keyword followed by a comma-separated list of one or more template parameters enclosed in <> brackets, followed by a class definition.

Definitions of function templates and member functions of class templates are ordinarily put into header files.

Template programs should try to minimize the number of requirements placed on the argument types.

DEFINITION 10.5 **Instantiate** Compiler process whereby the actual template argument(s) are used to generate a specific instance of the template in which the parameter(s) are replaced by the corresponding argument(s). Functions are instantiated automatically based on the arguments used in a call. We must supply explicit template arguments whenever we use a class template.

Class templates differ from function templates in that the compiler cannot deduce the template parameter type(s) for a class template. Instead, to use a class template we must supply additional information inside angle brackets following the template's name. That extra information is the list of template arguments to use in place of the template parameters.

DEFINITION 10.6 **Instantiation** Class or function generated by the compiler from a template. Each instantiation of a class template constitutes an independent class.

10.2 Template Parameters

DEFINITION 10.7 **Template parameter list** List of parameters, separated by commas, to be used in the definition or declaration of a template. Each parameter may be a type or non-type parameter.

In a template definition, the template parameter list cannot be empty.

DEFINITION 10.8 **Template parameter** Name specifed in the template parameter list that may be used inside the definition of a template. Template parameters can be type or non-type parameters. To use a class template, we must supply explicit arguments for each template parameter. The compiler uses those types or values to instantiate a version of the class in which uses of the parameter(s) are replaced by the actual argument(s). When a function template is used, the compiler deduces the template arguments from the arguments in the call and instantiates a specific function using the deduced template arguments.

DEFINITION 10.9 **Template argument** Type or value used to instantiate a template parameter.

10.2.1 Type Parameters and Non-type Parameters

10.2.1.1 Type Parameters

DEFINITION 10.10 **Type parameter** Name used in a template parameter list to represent a type. Each type parameter must be preceded by the keyword `typename` or `class`.

```
template<typename T>
int compare1(const T &v1, const T &v2) {...}
// Instantiation.
cout << compare1(0, 1) << endl;
```

10.2.1.2 Non-type Parameters

DEFINITION 10.11 **Non-type parameter** A template parameter that represents a value rather than a type. Nontype parameters are specified by using a specific type name instead of the `typename` or `class` keyword. When the template is instantiated, non-type parameters are replaced with a value supplied by the user or deduced by the compiler. These values must be constant expressions.

```
template<size_t N1, size_t N2>
int compare2(const char (&p1)[N1], const char (&p2)[N2]) {
  return strcmp(p1, p2);
}
// Instantiation.
// The compiler will use the size of the literals to
// instantiate a version of the template with the sizes
// substituted for N1 and N2.
cout << compare2("hi", "mom") << endl;
```

10.2.2 Template Parameters and Scope

The name of a template parameter can be used after it has been declared and until the end of the template declaration or definition.

A template parameter hides any declaration of that name in an outer scope. Unlike most other contexts, however, a name used as a template parameter may not be reused within the template.

10.2.3 Using Class Members That Are Types

Assuming `T` is a template type parameter, When the compiler sees code such as `T::mem` it won't know until instantiation time whether `mem` is a type or a `static` data member. However, in order to process the template, the compiler must know whether a name represents a type.

By default, the language assumes that a name accessed through the scope operator is not a type. As a result, if we want to use a type member of a template type parameter, we must explicitly tell the compiler that the name is a type. We do so by using the keyword `typename`, not `class`.

```
typename T::mem A = a;
```

10.2.4 Default Template Arguments

DEFINITION 10.12 **Default template arguments** A type or a value that a template uses if the user does not supply a corresponding template argument.

```
// F represents the type of a callable object.
// f will be a default-initialized object of type F.
template <typename T, typename F = less<T>>
```

```
int compare(const T &v1, const T &v2, F f = F()) {
  ...
}
```

10.3 Declaration and Definition of a Template

10.3.1 Template Declarations

A template declaration must include the template parameters.

```
template <typename T>
int compare(const T &, const T &);
```

Declarations for all the templates needed by a given file usually should appear together at the beginning of a file before any code that uses those names.

10.3.2 `inline` and `constexpr` Function Templates

A function template can be declared `inline` or `constexpr` in the same ways as non-template functions. The `inline` or `constexpr` specifier follows the template parameter list and precedes the return type.

10.3.3 Controlling Instantiations

The fact that instantiations are generated when a template is used means that the same instantiation may appear in multiple object files. When two or more separately compiled source files use the same template with the same template arguments, there is an instantiation of that template in each of those files.

In large systems, the overhead of instantiating the same template in multiple files can become significant. Under the new standard, we can avoid this overhead through an explicit instantiation.

DEFINITION 10.13 **Explicit instantiation** A declaration that supplies explicit arguments for all the template parameters. Used to guide the instantiation process. If the declaration is extern, the template will not be instantiated; otherwise, the template is instantiated with the specified arguments. There must be a non-`extern` explicit instantiation somewhere in the program for every `extern` template declaration.

When the compiler sees an extern template declaration, it will not generate code for that instantiation in that file.

```
// Instantiation declaration.
extern template int compare(const int &, const int &);
// Instantiation definition.
template int compare(const int &, const int &);
```

Because the compiler automatically instantiates a template when we use it, the `extern` declaration must appear before any code that uses that instantiation.

10.4 Template Argument Deduction

DEFINITION 10.14 **Template argument deduction** Process by which the compiler determines which function template to instantiate. The compiler examines the types of the arguments that were specified using a template parameter. It automatically instantiates a version of the function with those types or values bound to the template parameters.

10.4.1 Conversions and Template Type Parameters

- Top-level `const` in either the parameter or the argument are ignored.
- `const` conversions: A function parameter that is a reference (or pointer) to a `const` can be passed a reference (or pointer) to a non-`const` object.
- Array- or function-to-pointer conversions: If the function parameter is not a reference type, then the normal pointer conversion will be applied to arguments of array or function type. An array argument will be converted to a pointer to its first element. Similarly, a function argument will be converted to a pointer to the function's type.

`const` conversions and array or function to pointer are the only automatic conversions for arguments to parameters with template types. Other conversions, such as the arithmetic conversions, derived-to-base, and user-defined conversions, are not performed.

10.4.2 Function-Template Explicit Arguments

DEFINITION 10.15 **Explicit template argument** Template argument supplied by the user in a call to a function or when defining a template class type. Explicit template arguments are supplied inside angle brackets immediately following the template's name.

Explicit template argument(s) are matched to corresponding template parameter(s) from left to right. An explicit template argument may be omitted only for the trailing (rightmost) parameters, and then only if these can be deduced from the function parameters. For example

```
// T1 cannot be deduced: it does not appear in the
```

```
// function parameter list.
template <typename T1, typename T2, typename T3>
T1 sum(T2 a, T3 b) {...}

// T1 is explicitly specified;
// T2 and T3 are inferred from the argument types.
auto val = sum<long long>(i1, i2);
```

Normal conversions also apply for arguments whose template type parameter is explicitly specified.

10.4.3 Trailing Return Types and Type Transformation

For example, when we don't know the exact type we want to return, but we do know that we want that type to be a reference to the element type of the sequence we're processing. To define this function, we must use a trailing return type. Because a trailing return appears after the parameter list, it can use the function's parameters.

```
template <typename T>
auto f1(T begin, T end) -> decltype(*begin) {
  return *begin;
}
```

The dereference operator returns an lvalue, so the type deduced by `decltype` is a reference to the type of the element that `begin` denotes.

DEFINITION 10.16 **Type transformation** Class templates defined by the library that transform their given template type parameter to a related type.

```
// Return a copy of an element from the range.
template <typename T>
auto f2(T begin, T end)
    -> typename remove_reference<decltype(*begin)>::type {
  return *begin;
}
```

Note that `type` is member of a class that depends on a template parameter. As a result, we must use typename in the declaration of the return type to tell the compiler that `type` represents a type.

10.4.4 Function Pointers and Argument Deduction

When we initialize or assign a function pointer from a function template, the compiler uses the type of the pointer to deduce the template argument(s).

When the address of a function-template instantiation is taken, the context must be such that it allows a unique type or value to be determined for each template parameter.

10.4.5 Template Argument Deduction and References

10.4.5.1 Reference Collapsing and Rvalue Reference Parameters

For example,

```
template <typename T>
void f(T &&);
```

Assuming `i` is an `int` object, we might think that a call such as `f(i)` would be illegal. After all, `i` is an lvalue, and normally we cannot bind an rvalue reference to an lvalue. However, the language defines two exceptions to normal binding rules that allow this kind of usage.

The first exception is that, if we pass an lvalue to a function parameter that is an rvalue reference to a template type parameter, the compiler deduces the template type parameter as the argument's lvalue reference type. So, when we call `f(i)`, the compiler deduces the type of `T` as `int &`, not `int`.

Deducing `T` as `int &` would seem to mean that `f`'s function parameter would be an rvalue reference to the type `int &`. Ordinarily, we cannot (directly) define a reference to a reference. However, it is possible to do so indirectly through a type alias or through a template type parameter.

The second exception is that, if we indirectly create a reference to a reference, then those references "collapse". The references collapse to form an ordinary lvalue reference type. References collapse to form an rvalue reference only in the specific case of an rvalue reference to an rvalue reference.

That is, for a given type `T`:

- `T & &`, `T & &&`, and `T && &` all collapse to type `T &`.
- The type `T && &&` collapses to `T &&`.

The combination of the reference collapsing rule and the special rule for type deduction for rvalue reference parameters means that an argument of any type can be passed to a function parameter that is an rvalue reference to a template parameter type (i.e., `T &&`). When an lvalue is passed to such a parameter, the function parameter is instantiated as an ordinary, lvalue reference (`T &`). For example, we can call `f` on an lvalue. The resulting instantiation for `f(i)` would be something like

```
void f<int &>(int &);
```

10.4.5.2 Writing Template Functions with Rvalue Reference Parameters

For example

```
template <typename T>
void f(T &&val) {
  T i = val;
  ...
}
```

When we call `f` on an rvalue, such as the literal `42`, `T` is `int`. In this case, the local variable `i` has type `int` and is initialized by copying the value of the parameter `val`. On the other hand, when we call `f` on the lvalue, then `T` is `int &`. The initialization of `i` binds `i` to `val`. It is surprisingly hard to write code that is correct when the types involved might be plain (nonreference) types or reference types.

In practice, rvalue reference parameters are used in one of two contexts: Either the template is forwarding its arguments, or the template is overloaded.

```
template <typename T>
void f(T &&);  // Binds to non-const rvalues.

template <typename T>
void f(const T &);  // Lvalues and const rvalues.
```

10.4.5.3 Understanding `std::move`

We can use move to obtain an rvalue reference bound to an lvalue.

```
template <typename T>
typename remove_reference<T>::type &&move(T &&val) {
  return static_cast<typename remove_reference<T>::type &&>(val);
}
```

`move`'s function parameter, `T &&`, is an rvalue reference to a template parameter type. Through reference collapsing, this parameter can match arguments of any type.

When we pass an rvalue, e.g., `string &&`. The deduced type of `T` is `string`. The body of this function returns `static_cast<string &&>(val)`, so the cast does nothing. Therefore, the result of this call is the rvalue reference it was given.

```
string&& move(string &&t);
```

When we pass an lvalue, the deduced type of `T` is `string &`. In this case, `val` converts to `string &&`.

```
string&& move(string &t);
```

10.4.6 Forwarding

Some functions need to forward one or more of their arguments with their types unchanged to another, forwarded-to, function. In such cases, we need to preserve everything about the forwarded arguments, including whether or not the argument type is `const`, and whether the argument is an lvalue or an rvalue.

10.4.6.1 Defining Function Parameters That Retain Type Information

Through reference collapsing, if we define the function parameters as `T &&`, we can preserve the lvalue/rvalue property of a typelate parameter. Using a reference parameter (either lvalue or rvalue) lets us preserve constness, because the `const` in a reference type is low-level.

```
template <typname F, typename T1, typename T2>
void flip(F f, T1 &&val1, T2 &&val2) {
  f(val2, val1);
}
```

If we call `flip(f, i, 42)`, the type deduced for `T1` is `int &`, which means that the type of `val1` collapses to `int &`.

This works fine for functions that take lvalue references but cannot be used to call a function that has an rvalue reference parameter. A function parameter, like any other variable, is an lvalue expression. As a result, we cannot passes an lvalue to a rvalue reference parameter.

10.4.6.2 Using `std::forward` to Preserve Type Information in a Call

`forward` is defined in the `<utility>` header. `forward` must be called with an explicit template argument. `forward` returns an rvalue reference to that explicit argument type. That is, the return type of `forward<T>` is `T &&`.

Ordinarily, we use `forward` to pass a function parameter that is defined as an rvalue reference to a template type parameter. Through reference collapsing on its return type, `forward` preserves the lvalue/rvalue nature of its given argument.

If that argument was an rvalue, then `T` is an ordinary (non-reference) type and `forward<T>` will return `T &&`. If the argument was an lvalue, then — through reference collapsing — `T` itself is an lvalue reference type. In this case, the return type is an rvalue reference to an lvalue reference type. Again through reference collapsing — this time on the return type — `forward<T>` will return an lvalue reference type.

```
template <typname F, typename T1, typename T2>
void flip(F f, T1 &&val1, T2 &&val2) {
  f(std::forward<T2>(val2), std::forward<T1>(val1));
}
```

As with `std::move`, it's a good idea not to provide a using declaration for `std::forward`.

10.5 Overloading and Templates

Function matching is affected by the presence of function templates in the following ways:

- The candidate functions for a call include any function-template instantiation for which template argument deduction succeeds.
- The candidate function templates are always viable, because template argument deduction will have eliminated any templates that are not viable.
- As usual, the viable functions (template and non-template) are ranked by the conversions, if any, needed to make the call.
- Also as usual, if exactly one function provides a better match than any of the others, that function is selected. However, if there are several functions that provide an equally good match, then:
  - If there is only one non-template function in the set of equally good matches, the non-template function is called.
  - If there are no non-template functions in the set, but there are multiple function templates, and one of these templates is more specialized than any of the others, the more specialized function template is called.
  - Otherwise, the call is ambiguous.

Declare every function in an overload set before you define any of the functions. That way you dont have to worry whether the compiler will instantiate a call before it sees the function you intended to call.

10.6 Variadic Template

DEFINITION 10.17 **Variadic template** Template that takes a varying number of template arguments. A template parameter pack is specified using an elipsis (e.g., `...`).

DEFINITION 10.18 **Parameter pack** Template or function parameter that represents zero or more parameters.

DEFINITION 10.19 **Template parameter pack** Parameter pack that represents zero or more template parameters.

DEFINITION 10.20 **Function parameter pack** Parameter pack that represents zero or more function parameters.

```
// T2 is a template parameter pack;
// rest is a function parameter pack.
// T2 represents zero or more template type parameters.
// rest represents zero or more function parameters.
template <typename T1, typename... T2>
void f(const T1 &val1, const T2 &... rest);

// The compiler also deduces the number of parameters in the pack.
// The following one will be instantiated as:
// foo(const int &, const string &, const int &, const double &).
foo(ival, str, 42, dval);
foo(str, 42, "hi");
foo(dval, str);
foo("hi");
```

10.6.1 Operations on Variadic Templates

When we need to know how many elements there are in a pack, we can use the `sizeof...` operator. Like `sizeof`, `sizeof...` returns a constant expression and does not evaluate its argument.

```
template<typename ... Ts>
void f(Ts... vals) {
  // Number of type parameters.
  cout << sizeof...(Ts) << endl;
  // Number of function parameters.
```

```
    cout << sizeof...(vals) << endl;
  }
```

Aside from taking its size, the only other thing we can do with a parameter pack is to expand it. When we expand a pack, we also provide a pattern to be used on each expanded element.

DEFINITION 10.21 **Pack expansion** Process by which a parameter pack is replaced by the corresponding list of its elements.

DEFINITION 10.22 **Pattern** Defines the form of each element in an expanded parameter pack.

10.6.2 Writing a Variadic Function Template

We can use an `initializer_list` to define a function that can take a varying number of arguments. However, the arguments must have the same type (or types that are convertible to a common type). Variadic functions are used when we know neither the number nor the types of the arguments we want to process.

Variadic functions are often recursive. The first call processes the first argument in the pack and calls itself on the remaining arguments. To stop the recursion, we'll also need to define a corresponding non-variadic function.

```
// Function to end the recursion and print the last element
// this function must be declared before the variadic version
// of print is defined.
template<typename T>
ostream &print(ostream &out, const T &val) {
  return out << val;
}

// This version of print will be called for all but the
// last element in the pack.
template <typename T, typename... Ts>
ostream &print(ostream &out,
    const T &val, const Ts&... vals) {  // Expand Ts.
  out << val << " ";  // Print the first argument.
  // Recursive call; print the other arguments.
  return print(out, vals...);  // Expand vals.
}
```

10.6.3 Forwarding Parameter Packs

We can use variadic templates together with `forward` to write functions that pass their arguments unchanged to some other function. For example, the `emplace_back` member of the library containers is a variadic member template that uses its arguments to construct an element directly in space managed by the container.

```
template <typename... Ts>
// && means that each function parameter will be an rvalue
// reference to its corresponding argument.
void emplace_back(Ts &&...vals) {
  check_and_alloc();
  // std::forward<Ts>(vals)... generates elements with the form
  // std::forward<T_i>(val_i);
  _alloc.construct(end++, std::forward<Ts>(vals)...);
}
```

10.7 Template Specializations

DEFINITION 10.23 **Template specialization** Redefinition of a class template, a member of a class template, or a function template, in which some (or all) of the template parameters are specified. A template specialization may not appear until after the base template that it specializes has been declared. A template specialization must appear before any use of the template with the specialized arguments. Each template parameter in a function template must be completely specialized.

In order to specialize a template, a declaration for the original template must be in scope. Moreover, a declaration for a specialization must be in scope before any code uses that instantiation of the template. Therefore, templates and their specializations should be declared in the same header file. Declarations for all the templates with a given name should appear first, followed by any specializations of those templates.

10.7.1 Defining a Function Template Specialization

For example, by using the following template, we can compare any two types.

```
template <typename T>
int compare(const T &val1, const T &val2) {...}
```

However, the general definition is not appropriate for a particular type, namely, character pointers. We'd like compare to compare character pointers by calling `strcmp` rather than by comparing the pointer values.

To handle character pointers (as opposed to arrays), we can define a template specialization. The empty brackets indicate that arguments will be supplied for all the template parameters of the original template.

```
template <typename T>
int compare(const char *const &p1, const char *const &p2) {
  return strcmp(p1, p2);
}
```

When we define a specialization, the function parameter type(s) must match the corresponding types in a previously declared template. In the original template aree references to a `const` type. As with type aliases, the interaction between template parameter types, pointers, the `const` version of a pointer type is a constant pointer.

10.7.2 Function Overloading versus Template Specializations

Specializations instantiate a template; they do not overload it. As a result, specializations do not affect function matching.

11 STL Preview

DEFINITION 11.1 **Standard template library** Collection of types and functions that every C++ compiler must support. The library provides the types that support IO. C++ programmers tend to talk about "the library", meaning the entire standard library. They also tend to refer to particular parts of the library by referring to a library type, such as the "iostream library", meaning the part of the standard library that defines the IO classes.

DEFINITION 11.2 **Library type** Type, such as `istream`, defined by the standard library.

DEFINITION 11.3 **`std`** Name of the namespace used by the standard library.

The Standard Template Library (STL) consists of two primary components: a set of container classes, and a set of generic algorithms to operate over these containers. The library containers are class templatesand the library algorithms are function templates.

DEFINITION 11.4 **Container** A type whose objects hold a collection of objects of a given type. Each library container type is a template type. To define a container, we must specify the type of the elements stored in the container. With the exception of `array`, the library containers are variable-size.

DEFINITION 11.5 **Sequential container** Type that holds an ordered collection of objects of a single type. Elements in a sequential container are accessed by position.

DEFINITION 11.6 **Associative container** Type that holds a collection of objects that supports efficient lookup by key.

DEFINITION 11.7 **Generic algorithms** Type-independent algorithms. They operate on elements of differing type and across multiple container types.

The generic algorithms achieve type independence by being implemented as function templates. The achieve container independence by not operating directly on the container. Rather, they are passed an iterator pair, marking the range of on the elements over which to iterate.

The generic algorithms do not themselves execute container operations. Algorithms never change the size of the underlying container. Algorithms may change the values of the elements stored in the container, and they may move elements around within the container. They do not, however, ever add or remove elements directly.

12 Sequential Containters

12.1 Sequential Containters Types

12.1.1 Sequential Containters Types

DEFINITION 12.1 **`string`** Library type representing variable-length sequences of characters.

DEFINITION 12.2 **`vector`** Sequential container. Elements in a vector can be accessed by their positional index. Supports fast random access to elements. We can efficiently add or remove vector elements only at the back. Adding elements to a vector might cause it to be reallocated, invalidating all iterators into the vector. Adding (or removing) an element in the middle of a vector invalidates all iterators to elements after the insertion (or deletion) point.

DEFINITION 12.3 **`deque`** Sequential container. Elements in a deque can be accessed by their positional index. Supports fast random access to elements. Like a vector in all respects except that it supports fast insertion and deletion at the front of the container as well as at the back and does not relocate elements as a result of insertions or deletions at either end.

DEFINITION 12.4 **`list`** Sequential container representing a doubly linked list. Elements in a list may be accessed only sequentially; starting from a given element, we can get to another element only by traversing each element between them. Iterators on list support both increment (++) and decrement (--). Supports fast insertion (or deletion) anywhere in the list. Iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

DEFINITION 12.5 **`forward_list`** Sequential container that represents a singly linked list. Elements in a `forward_list` may be accessed only sequentially; starting from a given element, we can get to another element only by traversing each element between them. Iterators on `forward_list` do not support decrement (--). Supports fast insertion (or deletion) anywhere in the `forward_list`. Unlike other containers, insertions and deletions occur after a given iterator position. As a consequence, `forward_list` has a "before-the-beginning" iterator to go along with the usual off-the-end iterator. Iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

DEFINITION 12.6 **`array`** Fixed-size sequential container. To define an array, we must give the size in addition to specifying the element type. Elements in an array can be accessed by their positional index. Supports fast random access to elements.

12.1.2 Deciding Which Sequential Container to Use

It is best to use `vector` unless there is a good reason to prefer another container.

If your program has lots of small elements and space overhead matters, don't use `list` or `forward_list`.

If the program needs to insert elements in the middle of the container only while reading input, and subsequently needs random access to the elements.

- First, decide whether you actually need to add elements in the middle of a container. It is often easier to append to a `vector` and then call the library `sort` function to reorder the container when you're done with input.
- If you must insert into the middle, consider using a `list` for the input phase. Once the input is complete, copy the `list` into a `vector`.

If you're not sure which container to use, write your code so that it uses only operations common to both `vector` and `list`: Use iterators, not subscripts, and avoid random access to elements. That way it will be easy to use either a `vector` or a `list` as necessary.

12.2 Sequential Container Members

DEFINITION 12.7 **`size_type`** Name of types defined by the string and vector classes that are capable of containing the size of any string or vector, respectively. Library classes that define size_type define it as an unsigned type.

DEFINITION 12.8 **`difference_type`** A `signed` integral type defined by `vector` and `string` that can hold the distance between any two iterators.

DEFINITION 12.9 **`getline`** Function defined in the string header that takes an istream and a string. The function reads the stream up to the next newline, storing what it read into the string, and returns the istream. The newline is read and discarded.

12.3 Specialized `forward_list` Operations

When we add or remove an element, the element before the one we added or removed has a different successor. To add or remove an element, we need access to its predecessor in order to update that element's links. However, `forward_list` is a singly linked list. In a singly linked list there is no easy way to get to an element's predecessor. For this reason, the operations to add or remove elements in a `forward_list` operate by changing the element after the given element. That way, we always have access to the elements that are affected by the change.

```
// Remove odd elements.
```

```
for (auto prev = flst.before_begin(), curr = flst.begin();
  curr != flst.end(); /* empty */) {
  if (*curr % 2 == 1) {
    curr = flst.earse_after(prev);
  }
  else {
    prev = curr;
    ++curr;
  }
}
```

12.4 `string` Search Operations

```
// Find all occurrences.
for (string::size_type i = 0;
    (i = s.find_first_of(substr, i)) != string::npos;
    ++i) {
  cout << s[i] << endl;
}
```

12.5 Container Operations May Invalidate Iterators

```
// Insert a duplicate of odd-valued elements and remove
// even-valued elements.
for (auto it = vec.begin(); it != vec.end(); ++it) {
  if (*it % 2 == 1) {
    it = vec.insert(it, *it);
    it += 2;
  }
  else {
    it = vec.erase(it);
  }
}
```

12.6 Container Adaptors

DEFINITION 12.10 **Adaptor** Library type, function, or iterator that, given a type, function, or iterator, makes it act like another. There are three sequential container adaptors: `stack`, `queue`, and `priority_queue`. Each adaptor defines a new interface on top of an underlying sequential container type.

DEFINITION 12.11 **`stack`** Adaptor for the sequential containers that yields a type that lets us add and remove elements from one end only.

DEFINITION 12.12 **`queue`** Adaptor for the sequential containers that yields a type that lets us add elements to the back and remove elements from the front.

DEFINITION 12.13 **`priority_queue`** Adaptor for the sequential containers that yields a queue in which elements are inserted, not at the end but according to a specified priority level. By default, priority is determined by using the less-than operator for the element type.

By default both `stack` and `queue` are implemented in terms of `deque`, and a `priority_queue` is implemented on a `vector`. We can override the default container type by naming a sequential container as a second type argument when we create the adaptor.

There are constraints on which containers can be used for a given adaptor.

- `stack` requires only `push_back`, `pop_back`, and `back` operations.
- `queue` requires `back`, `push_back`, `front`, and `push_front`.
- `priority_queue` requires random access in addition to the `front`, `push_back`, and `pop_back` operations.

```
stack<int> stk;
for (size_t i = 0; i != 10; ++i) {
  stk.push(i);
}
while(!stk.empty()) {
  cout << stk.top() << endl;
  stk.pop();
}
```

13 Iterators

DEFINITION 13.1 **Iterator** Object used to access and navigate among the elements of a container. It supports the same set of operators as the built-in pointer (`++, *, ==, !=`), but allow us to provide a unqiue implementation of those operators.

13.1 Basic Concepts

DEFINITION 13.2 **`begin`** Container operation that returns an iterator referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty. Whether the returned iterator is const depends on the type of the container. Also, free-standing library function that takes an array and returns a pointer to the first element in the array.

DEFINITION 13.3 **`end`** Container operation that returns an iterator referring to the (nonexistent) element one past the end of the container. Whether the returned iterator is const depends on the type of the container. Also, freestanding library function that takes an array and returns a pointer one past the last element in the array.

DEFINITION 13.4 **cbegin** Container operation that returns a const_iterator referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty.

DEFINITION 13.5 **cend** Container operation that returns a const_iterator referring to the (nonexistent) element one past the end of the container.

DEFINITION 13.6 **off-the-beginning iterator** Iterator denoting the (nonexistent) element just before the beginning of a forward_list. Returned from the forward_list member before_begin. Like the end() iterator, it may not be dereferenced.

DEFINITION 13.7 **Off-the-end iterator** Iterator that denotes one past the last element in the range. Commonly referred to as the end iterator.

DEFINITION 13.8 **iterator range** Range of elements denoted by a pair of iterators. The first iterator denotes the first element in the sequence, and the second iterator denotes one past the last element. If the range is empty, then the iterators are equal (and vice versaif the iterators are unequal, they denote a nonempty range). If the range is not empty, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator. By incrementing the iterator, each element in the sequence can be processed.

DEFINITION 13.9 **left-inclusive interval** A range of values that includes its first element but not its last. Typically denoted as `[i, j)`, meaning the sequence starting at and including i up to but excluding j.

DEFINITION 13.10 **Iterator arithmetic** Operations on vector or string iterators: Adding or subtracting an integral value and an iterator yields an iterator that many elements ahead of or behind the original iterator. Subtracting one iterator from another yields the distance between them. Iterators must refer to elements in, or off-the- end of the same container.

13.2 Several Additional Iterators

The library defines several additional kinds of iterators in the `<iterator>` header.

13.2.1 Insert iterator

DEFINITION 13.11 **Insert iterator** Iterator adaptor that generates an iterator that uses a container operation to add elements to a given container. When we assign through an insert iterator, a new element equal to the right-hand value is added to the container.

There are three kinds of inserters.

DEFINITION 13.12 **`back_inserter`** Iterator adaptor that takes a reference to a container and generates an insert iterator that uses `push_back` to add elements to the specified container.

DEFINITION 13.13 **`front_inserter`** Iterator adaptor that, given a container, generates an insert iterator that uses `push_front` to add elements to the beginning of that container.

DEFINITION 13.14 **`inserter`** Iterator adaptor that takes an iterator and a reference to a container and generates an insert iterator that uses `insert` to add elements just ahead of the element referred to by the given iterator.

```
it = inserter(vec, p);
*it = val;

// Equivalent form.
it = vec.insert(p, val);
// increment it such that it denotes the same element as before.
++it;
```

13.2.2 Stream iterator

DEFINITION 13.15 **Stream iterator** Iterator that can be bound to input or output streams and can be used to iterate through the associated IO stream. These iterators treat their

corresponding stream as a sequence of elements of a specified type. Using a stream iterator, we can use the generic algorithms to read data from or write data to stream objects.

DEFINITION 13.16 **istream_iterator** Stream iterator that reads an input stream.

DEFINITION 13.17 **ostream_iterator** Iterator that writes to an output stream.

```
vector<int> vec;
for (istream_iterator<int> it(cin), end; it != end; ++it) {
  vec.push_back(*it);
}

// Equivalent form.
istream_iterator<int> it(cin), end;
vector<int> vec(it, end);


ostream_iterator<int> it(cout, " ");
for (const auto &val: vec) {
  *it++ = val;
}
cout << endl;

// Equivalent form.
ostream_iterator<int> it(cout, " ");
copy(vec.begin(), vec.end(), it);
cout << endl;
```

13.2.3 Reverse iterator

DEFINITION 13.18 **Reverse iterator** Iterator that moves backward, rather than forward, through a sequence. These iterators exchange the meaning of ++ and --.

We can get a forward iterator by calling the `reverse_iterator`'s `base` member. [`vec.crbegin(), rit`) and [`rit.base(), vec.end()`) refer to the same elements in `vec`.

13.2.4 Move iterator

DEFINITION 13.19 **Move iterator** Iterator adaptor that generates an iterator that, when dereferenced, yields an rvalue reference.

We transform an ordinary iterator to a move iterator by calling the library `make_move_iterator` function.

14 Generic Algorithms

14.1 Basic Concepts

DEFINITION 14.1 **Predicate** Function that returns a type that can be converted to `bool`. Often used by the generic algorithms to test elements. Predicates used by the library are either unary (taking one argument) or binary (taking two).

DEFINITION 14.2 **Unary predicate** Predicate that has one parameter.

DEFINITION 14.3 **Binary predicate** Predicate that has two parameters.

DEFINITION 14.4 **`bind`** Library function that binds one or more arguments to a callable expression. It takes a callable object and generates a new callable that "adapts" the parameter list of the original object. `bind` is defined in the `<functional>` header.
`auto f_bind = bind(f, arg_list)`

When we call `f_bind`, `f_bind` calls `f`, passing the arguments in arg_list. The arguments in arg_list may include names of the form _n, where n is an integer. These arguments are "placeholders" representing the parameters of `f_bind`. They stand "in place of" the arguments that will be passed to `f_bind`. The number n is the position of the parameter in the generated callable: `std::placeholders::_1` is the first parameter in `f_bind`, `std::placeholders::_2` is the second, and so forth.

By default, the arguments to `bind` that are not placeholders are copied into the callable object that `bind` returns. If we want to pass an object to `bind` without copying it, we must use the library `ref` and `cref` functions, which functions are defined in the `<functional>` header.

DEFINITION 14.5 **`ref`** Library function that generates a copyable object from a reference to an object of a type that cannot be copied.

DEFINITION 14.6 **`cref`** Library function that returns a copyable object that holds a reference to a `const` object of a type that cannot be copied.

```
auto f_bind = bind(f, ref(cout), _1);
```

14.2 Generic Algorithms

Most of the generic algorithms are defined in the `<algorithm>` header.

14.2.1 Read-Only Algorithms

```
find(begin, end, sought)
find_if(begin, end, p)
```

`find` returns an iterator to the first element that is equal to that value. `find_if` returns an iterator to the first element that the uniary predicate returns `true`. If there is no match, find returns its second iterator to indicate failure.

```
count_if(begin, end, p)
```

It returns a count of how often the predicate is true.

```
accumulate(begin, end, init_val)
```

It is in the `<numeric>` header. The type of the third argument to accumulate determines which addition operator is used and is the type that accumulate returns. For example,

```
string sum = accumulate(svec.cbegin(), svec.cend(), string(""));
```

The explicit `string` is necessary, because there is no + operator for type `const char *`.

```
equal(begin1, end1, beg2)
```

It determines whether two sequences hold the same values. It compares each element from the first sequence to the corresponding element in the second. It returns `true` if the corresponding elements are equal, `false` otherwise. The first two denote the range of elements in the first sequence; the third denotes the first element in the second sequence. Algorithms that take a single iterator denoting a second sequence assume that the second sequence is at least as large at the first.

```
for_each(begin, end, f)
```

It takes a callable object and calls that object on each element in the input range.

14.2.2 Algorithms That Write Container Elements

```
fill(begin, end, val)
```

It assigns the given value to each element in the input sequence.

```
fill_n(begin, n, val)
```

It assigns the given value to the specified number of elements starting at the element denoted to by the iterator. Algorithms that write to a destination iterator assume the destination is large enough to hold the number of elements being written. One way to ensure that an algorithm has enough elements to hold the output is to use an insert iterator.

```
vector<int> vec;
fill_n(back_inserter(vec), 10, 0);
```

```
copy(begin1, end1, beg2)
```

It copies elements from its input range into elements in the destination. The value returned by copy is the (incremented) value of its destination iterator. That is, it points just past the last element copied into.

```
replace(begin, end, val1, val2)
```

It reads a sequence and replaces every instance of a given value with another value.

```
replace_copy(begin1, end1, begin2, val1, val2)
```

This algorithm takes a third iterator argument denoting a destination in which to write the adjusted sequence.

14.2.3 Algorithms That Reorder Container Elements

```
sort(begin, end)
sort(begin, end, lt)
stable_sort(begin, end)
stable_sort(begin, end, lt)
```

It arranges the elements in the input range into sorted order using the element type's < operator. `lt` is a predicate. A stable sort maintains the original order among equal elements.

```
reverse(begin, end)
reverse_copy(begin1, end1, begin2)
```

```
unique(begin, end)
unique(begin, end, eq)
```

It reorders the elements so that the each element appears once in the front portion of the range and returns an iterator one past the unique range.

```
partition(begin, end, p)
stable_partition(begin, end, p)
```

It takes a predicate and partitions the container so that values for which the predicate is `true` appear in the first part and those for which the predicate is `false` appear in the second part. The algorithm returns an iterator just past the last element for which the predicate returned `true`.

```
transform(begin1, end1, begin2, f)
```

The algorithm calls the given callable on each element in the input sequence and writes the result to the destination. When the input iterator and the destination iterator are the same, `transform` replaces each element in the input range with the result of calling the callable on that element.

DEFINITION 14.7 **iterator categories** Conceptual organization of iterators based on the operations that an iterator supports. Iterator categories form a hierarchy, in which the more powerful categories offer the same operations as the lesser categories. The algorithms use iterator categories to specify what operations the iterator arguments must support. As long as the iterator provides at least that level of operation, it can be used. For example, some algorithms require only input iterators. Such algorithms can be called on any iterator other than one that meets only the output iterator requirements. Algorithms that require random-access iterators can be used only on iterators that support random-access operations.

DEFINITION 14.8 **forward iterator** Iterator that can read and write elements but is not required to support –.

DEFINITION 14.9 **bidirectional iterator** Same operations as forward iterators plus the ability to use –to move backward through the sequence.

DEFINITION 14.10 **reverse iterator** Iterator that moves backward through a sequence. These iterators exchange the meaning of ++ and –.

DEFINITION 14.11 **random-access iterator** Same operations as bidirectional iterators plus the relational operators to compare iterator values, and the subscript operator and arithmetic operations on iterators, thus supporting random access to elements.

DEFINITION 14.12 **input iterator** Iterator that can read, but not write, elements of a sequence.

DEFINITION 14.13 **output iterator** Iterator that can write, but not necessarily read, elements.

15 Associative Containers

15.1 Associative Container Types

DEFINITION 15.1 **map** Associative container type that defines an associative array. Like `vector`, `map` is a class template. A `vector`, however, is defined with two types: the type of the key and the type of the associated value. In a map, a given key may appear only once. Each key is associated with a particular value. Dereferencing a map iterator yields a `pair` that holds a `const` key and its associated value.

DEFINITION 15.2 **Associative array** Array whose elements are indexed by key rather than positionally. We say that the array maps a key to its associated value. The `map` type if often referred to as an associative array.

DEFINITION 15.3 **multimap** Associative container similar to `map` except that in a `multimap`, a given key may appear more than once. `multimap` does not support subscripting.

DEFINITION 15.4 **set** Associative container that holds keys. In a `set`, a given key may appear only once.

DEFINITION 15.5 **multiset** Associative container type that holds keys. In a `multiset`, a given key may appear more than once.

15.2 Associative Container Members and Operations

DEFINITION 15.6 **Strict weak ordering** Relationship among the keys used in an associative container. In a strict weak ordering, it is possible to compare any two values and determine which of the two is less than the other. If neither value is less than the other, then the two values are considered equal.

DEFINITION 15.7 **key_type** Type defined by the associative containers that is the type for the keys used to store and retrieve values. For a map, key_type is the type used to index the map. For set, key_type and value_type are the same.

DEFINITION 15.8 **mapped_type** Type defined by map types that is the type of the values associated with the keys in the map.

DEFINITION 15.9 **value_type** Type of the element stored in a container. For set and multiset, value_type and key_type are the same. For map and multimap, this type is a pair whose first member has type const key_type and whose second member has type mapped_type.

DEFINITION 15.10 **`pair`** Type that holds two `public` data members named `first` and `second`. The pair type is a template type that takes two type parameters that are used as the types of these members.

DEFINITION 15.11 **Dereference operator (`*`)** When applied to a map, set, multimap, or multiset iterator * yields a value_type. Note, that for map and multimap, the value_type is a pair.

DEFINITION 15.12 **Subscript operator (`[]`)** Defined only for nonconst obejcts of type map and unordered_map. For the map types, [] takes an index that must be a key_type (or type that can be converted to key_type). Yields a mapped_type value.

15.3 Finding Elements in a `multimap` or `multiset`

15.3.1 Use `find` and `count`

```
auto count = my_set.count(sought);
for (auto it = my_set.find(sought); count != 0; ++it, --count) {
  cout << it->second << endl;
}
```

15.3.2 Iterator-Oriented Solution

```
auto begin = my_set.lower_bound(sought);
auto end = my_set.upper_bound(sought);
for (auto it = begin; it != end; ++it) {
  cout << it->second << endl;
}
```

15.3.3 The `equal_range` Function

```
auto range = my_set.equal_range(sought);
for (auto it = range.first; it != range.second ++it) {
  cout << it->second << endl;
}
```

15.4 The Unordered Containers

DEFINITION 15.13 **Unordered associative container** Associative containers that use hashing rather than a comparison operation on keys to store and access elements. The performance of these containers depends on the quality of the hash function.

DEFINITION 15.14 **Hash function** Function that maps values of a given type to integral (`size_t`) values. Equal values must map to equal integers; unequal values should map to unequal integers where possible.

Unordered associative containers are most useful when we have a key type for which there is no obvious ordering relationship among the elements. These containers are also useful for applications in which the cost of maintaining the elements in order is prohibitive. We can usually use an unordered container in place of the corresponding ordered container, and vice versa.

DEFINITION 15.15 **`hash`** Special library template that the unordered containers use to manage the position of their elements.

15.4.1 Unordered Containers Types

DEFINITION 15.16 **`unordered_map`** Container with elements that are keyvalue pairs, permits only one element per key.

DEFINITION 15.17 **`unordered_multimap`** Container with elements that are key-value pairs, allows multiple elements per key.

DEFINITION 15.18 **`unordered_multiset`** Container that stores keys, allows multiple elements per key.

DEFINITION 15.19 **`unordered_set`** Container that stores keys, permits only one element per key.

15.4.2 Use an Unordered Container

We can directly define unordered containers whose key is one of the built-in types (including pointer types), or a `string`, or a smart pointer. However, we cannot directly define an unordered container that uses a our own class types for its key type. We must supply our own version of the `hash` template.

```
size_t hash_function(const Sales_data &data) {
  return hash<string>()(data.isbn());
```

```
}

// Arguments are bucket size, hash function, and equality function.
unordered_multiset<Sales_data,
    decltype(hash_function) *, decltype(eq) *>
    bookstore(42, hash_function, eq);
```

IV OBJECT-BASED PROGRAMMING

16 Classes

16.1 Data Abstraction

DEFINITION 16.1 **Data structure** A logical grouping of data and operations on that data. In C++ we define our own data structures by defining a class.

DEFINITION 16.2 **Abstract data type** Data structure that encapsulates (hides) its implementation.

DEFINITION 16.3 **Data abstraction** Programming technique that focuses on the interface to a type. Data abstraction lets programmers ignore the details of how a type is represented and think instead about the operations that the type can perform. Data abstraction is fundamental to both object-oriented and generic programming.

DEFINITION 16.4 **Encapsulation** Separation of implementation from interface; encapsulation hides the implementation details of a type. In C++, encapsulation is enforced by putting the implementation in the `private` part of a class.

DEFINITION 16.5 **access specifier** Keywords public and private. Used to define whether members are accessible to users of the class or only to friends and members of the class. Specifiers may appear multiple times within a class. Each specifier sets the access of the following members up to the next specifier.

DEFINITION 16.6 **private members** Members defined after a private access specifier; accessible only to the friends and other class members. Data members and utility functions used by the class that are not part of the types interface are usually declared private.

DEFINITION 16.7 **public members** Members defined after a public access specifier; accessible to any user of the class. Ordinarily, only the functions that define the interface to the class should be defined in the public sections.

DEFINITION 16.8 **Implementation** The (usually `private`) members of a class that define the data and any operations that are not intended for use by code that uses the type.

DEFINITION 16.9 **Interface** The (`public`) operations supported by a type. Ordinarily, the interface does not include data members.

16.2 Defining Abstract Data Types

DEFINITION 16.10 **Class** C++ mechanism for defining our own abstract data types. Classes may have data, function, or type members. A class defines a new type and a new scope.

DEFINITION 16.11 **Class type** A type defined by a class. The name of the type is the class name.

16.2.1 Class Declaration

DEFINITION 16.12 **Class declaration** The keyword `class` (or `struct`) followed by the class name followed by a semicolon. If a class is declared but not defined, it is an incomplete type.

DEFINITION 16.13 **Forward declaration** Declaration of an as yet undefined name. Most often used to refer to the declaration of a class that appears prior to the definition of that class.

DEFINITION 16.14 **Incomplete type** Type that is declared but not defined. It is not possible to use an incomplete type to create a object of that type. It is legal to define references or pointers to incomplete types, and we can declare (but not define) functions that use an incomplete type as a parameter or return type. Similarly, the class must be defined before a reference or pointer is used to access a member of the type.

16.2.2 Member

16.2.2.1 Data Member

DEFINITION 16.15 **Data member** Data elements that constitute an object. Every object of a given class has its own copies of the class' data members. Data members may be initialized when declared inside the class.

DEFINITION 16.16 **`mutable` data member** Data member that is never `const`, even when it is a member of a `const` object. A `mutable` member can be changed inside a `const` function. We indicate such members by including the `mutable` keyword in their declaration.

16.2.2.2 Member Function

DEFINITION 16.17 **Member function/Method** Class member that is a function. Ordinary member functions are bound to an object of the class type through the implicit `this` pointer. `static` member functions are not bound to an object and have no this pointer. Member functions may be overloaded; when they are, the implicit `this` pointer participates in the function matching. Functions defined in the class are implicitly `inline`.

DEFINITION 16.18 **`this` pointer** Implicit value passed as an extra argument to every nonstatic member function. The `this` pointer points to the object on which the function

is invoked. `this` is a `const` pointer (`A *const this`). We cannot change the address that `this` holds.

DEFINITION 16.19 **`const` member function** A member function that may not change an object's ordinary (i.e., neither `static` nor mutable) data members. A `const` following the parameter list indicates that `this` is a pointer to `const` (`const A *const this`). A member function may be overloaded based on whether the function is `const`. Objects that are `const`, and references or pointers to `const` objects, may call only `const` member functions. A `const` member function that returns `*this` as a reference should have a return type that is a reference to `const`.

16.2.2.3 Rvalue and Lvalue Reference Member Functions

Ordinarily, we can call a member function on an object, regardless of whether that object is an lvalue or an rvalue.

```
string s1 = "a value";
string s2 = "another";
// Call find on an rvalue.
auto n = (s1 + s2).find('a');
s1 + s2 = "wow!";  // Assign to rvalue.
```

When we want to prevent such usage in our own classes, we would like to indicate the lvalue/rvalue property of this in the same way that we define `const` member functions; we place a reference qualifier after the parameter list.

DEFINITION 16.20 **Reference qualifier** Symbol used to indicate that a non-`static` member function can be called on an lvalue or an rvalue. The qualifier, `&` or `&&`, follows the parameter list or the `const` qualifier if there is one. A function qualified by `&` may be called only on lvalues; a function qualified by `&&` may be called only on rvalues.

We can also overload a function based on its reference qualifier. If a member function has a reference qualifier, all the versions of that member with the same parameter list must have reference qualifiers.

16.3 Other Class Features

16.3.1 Friendship

DEFINITION 16.21 **Friend** Mechanism by which a class grants access to its non-`public` members. Friends have the same access rights as members. Both classes and

functions may be named as friends. Friends are not members of the class and are not affected by the access control of the section in which they are declared.

A friend declaration only specifies access. It is not a general declaration of the function. If we want users of the class to be able to call a friend function, then we must also declare the function separately from the friend declaration. We usually declare each friend (outside the class) in the same header as the class itself.

Friendship is not transitive. Each class controls which classes or functions are its friends.

Friendship is generally required for performance reasons. For a simple read or write of a data member, an `inline public` access function is usually an adequate alternative to friendship.

Although overloaded functions share a common name, they are still different functions. Therefore, a class must declare as a friend each function in a set of overloaded functions that it wishes to make a friend:

Constructors and destructors are the best place to manage object resources. Constructors are automatically called when they are created, and destructors are called automatically when they are destroyed, thus it is ensured that resources are properly obtained and released. This is called RAII (Resource Acquisition is Initialization). The concept of resources is larger than memory, device, file handles, etc. are all resources.

16.3.2 `static` Class Members

We say a member is associated with the class by adding the keyword `static` to its declaration.

The `static` members of a class exist outside any object, `static` data members are not initialized by the class' constructors. `static` data members are defined outside any function. The best way to ensure that the object is defined exactly once is to put the definition of `static` data members in the same file that contains the definitions of the class non-`inline` member functions.

We can provide in-class initializers for `static` members that have `const` integral type and must do so for `static` members that are `constexpr` of literal type. The initializers must be constant expressions. Such members are themselves constant expressions. Even if a `const static` data member is initialized in the class body, that member ordinarily should be defined outside the class definition.

`static` member functions are not bound to any object; they do not have a `this` pointer. As a result, `static` member functions may not be declared as `const`, and we may not refer to `this` in the body of a `static` member. When we define a `static` member outside the class, we do not repeat the `static` keyword. The keyword appears only with the declaration inside the class body.

We can access a `static` member directly through the scope operator. Even though `static` members are not part of the objects of its class, we can use an object, reference, or

pointer of the class type to access a `static` member. Member functions can use `static` members directly, without the scope operator.

16.3.3 Aggregate and Literal Class

DEFINITION 16.22 **Aggregate class** Class with only `public` data members that has no in-class initializers or constructors, and it has no base classes or `virtual` functions. Members of an aggregate can be initialized by a brace-enclosed list of initializers in declaration order.

If the list of initializers has fewer elements than the class has members, the trailing members are value initialized.

DEFINITION 16.23 **Literal class** An aggregate class whose data members are all of literal type is a literal class. A nonaggregate class, that meets the following restrictions, is also a literal class.

- The data members all must have literal type.
- The class must have at least one `constexpr` constructor.
- If a data member has an in-class initializer, the initializer for a member of built-in type must be a constant expression, or if the member has class type, the initializer must use the member's own `constexpr` constructor.
- The class must use default definition for its destructor.

16.4 Constructor

DEFINITION 16.24 **Constructor** A special member function used to initialize objects. Each constructor should give each data member a well-defined initial value. Constructors may not be declared as `const`. When we create a `const` object of a class type, the object does not assume its "constness" until after the constructor completes the object's initialization. Thus, constructors can write to `const` objects during their construction.

16.4.1 Default constructor

DEFINITION 16.25 **Default constructor** Constructor that is used if no initializer is supplied. The default constructor is one that takes no arguments. A constructor that supplies default arguments for all its parameters also defines the default constructor.

DEFINITION 16.26 **Synthesized default constructor** The default constructor created (synthesized) by the compiler for (and only for) classes that do not explicitly define any constructors. This constructor initializes the data members from their in-class initializers, if present; otherwise it default initializes the data members.

If a class has a member that has a class type, and that class doesn't have a default constructor, then the compiler can't initialize that member. For such classes, we must define our own version of the default constructor.

DEFINITION 16.27 **`= default`** Syntax used after the parameter list of the declaration of the default constructor inside a class to signal to the compiler that it should generate the constructor, even if the class has other constructors.

16.4.2 Initialization of Data Members

DEFINITION 16.28 **In-class initializer** Initializer provided as part of the declaration of a class data member. When we create objects, the in-class initializers will be used to initialize the data members. In-class initializers must either be enclosed inside curly braces or follow an = sign (copy initialization). We may not specify an in-class initializer inside parentheses. Members without an initializer are default initialized.

Classes that have members of built-in or compound type usually should rely on the synthesized default constructor only if all such members have in-class initializers.

DEFINITION 16.29 **Constructor initializer list** Specifies initial values of the data members of a class. The members are initialized to the values specified in the initializer list before the body of the constructor executes. Members that do not appear in the constructor initializer list are initialized by the corresponding in-class initializer (if there is one) or are default initialized. Constructors should not override in-class initializers except to use a different initial value.

Members that are `const` or references must be initialized. Similarly, members that are of a class type that does not define a default constructor also must be initialized. Therefore, we must use the constructor initializer list to provide values for members that are `const`, reference, or of a class type that does not have a default constructor.

Members are initialized in the order in which they appear in the class definition. It is a good idea to write constructor initializers in the same order as the members are declared. Moreover, when possible, avoid using members to initialize other members.

DEFINITION 16.30 **Delegating constructor** Constructor with a constructor-initializer list that has one entry that designates another constructor of the same class to do the initialization.

16.4.3 `explicit` Constructor

DEFINITION 16.31 **`explicit` constructor** Constructor that can be called with a single argument but cannot be used in an implicit conversion. A constructor is made explicit by

prepending the keyword `explicit` to its declaration, not repeated on a definition. The `explicit` keyword is meaningful only on constructors that can be called with a single argument.

When a constructor is declared `explicit`, it can be used only with the direct form of initialization. Moroever, the compiler will not use this constructor in an automatic conversion.

DEFINITION 16.32 **Converting constructor** A non-`explicit` constructor that can be called with a single argument. Such constructors implicitly convert from the argument's type to the class type. The compiler will automatically apply only one class-type conversion.

Type conversion operators, and constructors that are callable with a single argument, must be marked explicit in the class definition. As an exception, copy and move constructors should not be explicit, since they do not perform type conversion.

Constructors that cannot be called with a single argument should usually omit explicit. Constructors that take a single `initializer_list` parameter should also omit explicit, in order to support copy-initialization.

17 Copy Control

17.1 Concepts

DEFINITION 17.1 **Copy control** Special members that control what happens when objects of class type are copied, moved, assigned, and destroyed. The compiler synthesizes appropriate definitions for these operations if the class does not otherwise declare them.

The copy and move constructors define what happens when an object is initialized from another object of the same type. The copy- and move-assignment operators define what happens when we assign an object of a class type to another object of that same class type. The destructor defines what happens when an object of the type ceases to exist.

17.2 Five Copy Control Members

17.2.1 Copy Constructor

DEFINITION 17.2 **Copy constructor** Constructor that initializes a new object as a copy of another object of the same type. A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values. The copy constructor is applied implicitly to pass objects to or from a function by value. If we do not provide the copy constructor, the compiler synthesizes one for us.

```
A();  // default constructor.
A(const A &a);  // Copy constructor.
```

DEFINITION 17.3 **Synthesized copy/move constructor** A version of the copy or move constructor that is generated by the compiler for classes that do not explicitly define the corresponding constructor. Unless it is defined as deleted, a synthesized copy or move constructor memberwise initializes the new object by copying or moving members from the given object, respectively.

DEFINITION 17.4 **Memberwise copy/assign** How the synthesized copy and move constructors and the copy- and move-assignment operators work. Taking each non-`static` data member in turn, the synthesized copy or move constructor initializes each member by copying or moving the corresponding member from the given object; the copy- or move-assignment operators copy-assign or move-assign each member from the right-hand object to the left. Members of built-in or compound type are initialized or assigned directly. Members of class type are initialized or assigned by using the member's corresponding copy/move constructor or copy-/move-assignment operator.

17.2.2 Copy-assignment Operator

DEFINITION 17.5 **Copy-assignment operator** Version of the assignment operator that takes an object of the same type as its type. Ordinarily, the copy-assignment operator has a parameter that is a reference to `const` and returns a reference to its object. The compiler synthesizes the copy-assignment operator if the class does not explicitly provide one.

```
A &operator=(const A &a);  // Assignment operator.
```

There are two points to keep in mind when you write an assignment operator.

- Assignment operators must work correctly if an object is assigned to itself.
- Most assignment operators share work with the destructor and copy constructor.

The copy-assignment operator often does the same work as is needed in the copy constructor and destructor. In such cases, the common work should be put in `private` utility functions.

DEFINITION 17.6 **Synthesized copy-/move-assignment operator** A version of the copy- or move-assignment operator created (synthesized) by the compiler for classes that do not explicitly define assignment operators. Unless it is defined as `deleted`, a synthesized assignment operator memberwise assigns (moves) the right-hand operand to the left.

17.2.3 Move constructor

17.2.3.1 Move Constructor

DEFINITION 17.7 **Move constructor** Constructor that takes an rvalue reference to its type. Typically, a move constructor moves data from its parameter into the newly created object. After the move, it must be safe to run the destructor on the given argument.

```
A(A &&a) noexcept;  // Move constructor.
```

Move constructors and move assignment operators that cannot throw exceptions should be marked as `noexcept` between the parameter list and the `:` that begins the constructor initializer list.

The compiler will synthesize a move constructor or a move-assignment operator only if the class doesn't define any of its own copy-control members and if every non-`static` data member of the class can be moved. When a class doesn't have a move operation, the corresponding copy operation is used in place of move through normal function matching.

Classes that define a move constructor or move-assignment operator must also define their own copy operations. Otherwise, those members are deleted by default.

When a class has both a move constructor and a copy constructor, the compiler uses ordinary function matching to determine which constructor to use. Similarly for assignment.

```
class A {
  A(const A &);
  A(A &&);
  A &operator=(const A &);
  A &operator=(A &&);
};

A a1, a2;
a1 = a2;  // a2 is an lvalue; copy assignment.
A f();
a2 = f();  // f() is an rvalue; move assignment.
```

In the first assignment, `v2` is an lvalue. The move version of assignment is not viable, because we cannot implicitly bind an rvalue reference to an lvalue. Hence, this assignment uses the copy-assignment operator. In the second assignment, `f()` expression is an rvalue. In this case, both assignment operators are viable. Calling the copy-assignment operator requires a conversion to `const`, whereas `A &&` is an exact match. Hence, the second assignment uses the move-assignment operator.

17.2.4 Move-assignment operator

DEFINITION 17.8 **Move-assignment operator** Version of the assignment operator that takes an rvalue reference to its type. Typically, a move-assignment operator moves data from the right-hand operand to the left. After the assignment, it must be safe to run the destructor on the right-hand operand.

```
A &operator=(A &&a) noexcept;
// Move-assignment operator.
```

We can define assignment operator for both the move- and copy-assignment operator.

```
A &operator=(A a) {
  swap(*this, a);
  return *this;
}
```

That operator has a nonreference parameter, which means the parameter is copy initialized. Depending on the type of the argument, copy initialization uses either the copy constructor

or the move constructor; lvalues are copied and rvalues are moved. As a result, this single assignment operator acts as both the copy-assignment and move- assignment operator.

```
a = a2;  // a2 is an lvalue; copy constructor.
a = std::move(a2); //  move constructor.
```

17.2.5 Destructor

DEFINITION 17.9 **Destructor** Special member function that cleans up an object when the object goes out of scope or is deleted. The compiler automatically destroys each data member. Members of class type are destroyed by invoking their destructor; no work is done when destroying members of built-in or compound type. In particular, the object pointed to by a pointer member is not deleted by the destructor.

```
~A();  // Destructor.
```

In a constructor, members are initialized before the function body is executed, and members are initialized in the same order as they appear in the class. In a destructor, the function body is executed first and then the members are destroyed. Members are destroyed in reverse order from the order in which they were initialized.

DEFINITION 17.10 **Synthesized destructor** Version of the destructor created (synthesized) by the compiler for classes that do not explicitly define one. The synthesized destructor has an empty function body.

17.2.6 Swap

In addition to defining the copy-control members, classes that manage resources often also define a function named `swap`. Defining `swap` is particularly important for classes that we plan to use with algorithms that reorder elements.

If a class does not define `swap`, then the algorithm uses the library `swap` function. It involves a copy and two assignments.

```
T temp = a;  // Copy initialization.
a = b;  // Copy assignment.
b = temp;  // Copy assignment.
```

DEFINITION 17.11 **Copy and swap** Technique for writing assignment operators by copying the right-hand operand followed by a call to swap to exchange the copy with the left-hand operand.

```
// a is passed by value, by A's copy constructor.
A &operator=(A a) {
  swap(*this, a);
  return *this;
}  // a is destroyed.
```

Assignment operators that use copy and swap are automatically exception safe and correctly handle self-assignment. By copying `a` before changing `this`, it handles self assignment in the same was as we did in our original assignment operator. It manages exception safety in the same way as the original definition as well. The only code that might throw is the new expression inside the copy constructor. If an exception occurs, it will happen before we have changed the left-hand operand.

17.3 The Rule of Three/Five

DEFINITION 17.12 **The Rule of Three/Five** If a class needs a destructor, it almost surely also needs the copy-assignment operator and a copy constructor. If a class needs a copy constructor, it almost surely needs a copy-assignment operator, and vice versa. Nevertheless, needing either the copy constructor or the copy-assignment operator does not (necessarily) indicate the need for a destructor.

All five copy-control members should be thought of as a unit: Ordinarily, if a class defines any of these operations, it usually should define them all. Some classes must define the copy constructor, copy-assignment operator, and destructor to work correctly. Such classes typically have a resource that the copy members must copy. Ordinarily, copying a resource entails some amount of overhead. Classes that define the move constructor and move-assignment operator can avoid this overhead in those circumstances where a copy isn't necessary.

17.4 `= default` and `= delete`

DEFINITION 17.13 **`= default`** Syntax used after the parameter list of the declaration of the copy control members to signal to the compiler that it should generate the synthesized versions of the copy control members.

DEFINITION 17.14 **Deleted function** Function that may not be used. We delete a function by specifying `= delete` on its declaration. A common use of deleted functions is to tell the compiler not to synthesize the copy and/or move operations for a class. The destructor should not be a deleted member.

If a class has a data member that cannot be default constructed, copied, assigned, or destroyed, then the corresponding member will be a deleted function. A member that has a deleted or inaccessible destructor causes the synthesized default and copy constructors to be defined as deleted since otherwise we could create objects that we could not destroy. The compiler will not synthesize a default constructor for a class with a reference member or a `const` member that cannot be default constructed. A class with a `const` member cannot use the synthesized copy-assignment operator: After all, that operator attempts to assign to every member.

17.5 Copy Control and Resource Management

Ordinarily, classes that manage resources that do not reside in the class must define the copy-control members. Some classes also need copy control in order to do some bookkeeping.

In order to define these members, we first have to decide what copying an object of our type will mean. In general, we have two choices: We can define the copy operations to make the class behave like a value or like a pointer.

17.5.1 Value-like Class

Classes that behave like values have their own state, e.g., library containers. When we copy a value-like object, the copy and the original are independent of each other. Changes made to the copy have no effect on the original, and vice versa.

A good pattern to use when you write an assignment operator is to first copy the right-hand operand into a local temporary. After the copy is done, it is safe to destroy the existing members of the left-hand operand. Once the left-hand operand is destroyed, copy the data from the temporary into the members of the left-hand operand.

17.5.2 Pointer-like Class

Classes that act like pointers share state, e.g., `shared_ptr`. When we copy objects of such classes, the copy and the original use the same underlying data. Changes made to the copy also change the original, and vice versa. The easiest way to make a class act like a pointer is to use `shared_ptr` to manage the resources in the class. However, sometimes we want to manage a resource directly. In such cases, it can be useful to use a reference count.

DEFINITION 17.15 **Reference count** Programming technique often used in copy-control members. A reference count keeps track of how many objects share state. Constructors (other than copy/move constructors) set the reference count to 1. Each time a new copy is

made the count is incremented. When an object is destroyed, the count is decremented. The assignment operator and the destructor check whether the decremented reference count has gone to zero and, if so, they destroy the object. The counter cannot be a direct member of a class object, we store the counter in dynamic memory instead.

When you write an assignment operator, it must handle self-assignment. We do so by incrementing the reference count in right-hand operand before decrementing the count in the left-hand operand. That way if both objects are the same, the counter will have been incremented before we check to see if should be deleted:

18 Overloaded Operations and Conversions

18.1 Basic Concepts

DEFINITION 18.1 **Overloaded operator** Function that redefines the meaning of one of the built-in operators. Overloaded operator functions have the name operator followed by the symbol being defined. Overloaded operators must have at least one operand of class type. Overloaded operators have the same precedence, associativity and number of operands as their built-in counterparts.

Overloaded operators are functions with special names: the keyword `operator` followed by the symbol for the operator being defined. An operator function must either be a member of a class or have at least one parameter of class type. When an overloaded operator is a member function, `this` is bound to the left-hand operand. Member operator functions have one less (explicit) parameter than the number of operands.

Four symbols (`+`, `-`, `*`, `&`) serve as both unary and binary operators. Either or both of these operators can be overloaded. The number of parameters determines which operator is being defined.

Ordinarily, the comma, address-of, logical AND, and logical OR operators should not be overloaded. This is because that the overloaded versions of these operators do not preserve order of evaluation and/or short-circuit evaluation.

Choosing member or non-member implementation.

- The assignment (=), subscript (`[]`), call (`()`), and member access arrow (`->`) operators must be defined as members.
- Operators that change the state of their object or that are closely tied to their given type — such as increment, decrement, and dereference — usually should be members.
- The compound-assignment operators ordinarily ought to be members. However, unlike assignment, they are not required to be members.
- Symmetric operators — those that might convert either operand, such as the arithmetic, equality, relational, and bitwise operators — usually should be defined as ordinary nonmember functions.

18.2 Overloading Operators

18.2.1 Input and Output Operators

Input and output operators that conform to the conventions of the `iostream` library must be ordinary nonmember functions. These operators cannot be members of our own class. If they were, then the left-hand operand would have to be an object of our class type. Therefore, IO operators usually must be declared as `friend`.

Generally, output operators should print the contents of the object, with minimal formatting. They should not print a newline.

```
ostream &operator<<(ostream &out, const A &a) {
  out << ...
  return out;
}
```

Input operators must deal with the possibility that the input might fail; output operators generally don't bother.

Rather than checking each read, we check once after reading all the data and before using those data.

```
istream &operator>>(istream &in, A &a) {
  in >> ...
  if (in) ... // Check that the input succeeded.
  else {
    // Input failed: give the object the default state.
    a = A();
  }
  return out;
}
```

If there was an error, we do not worry about which input failed. Instead, we reset the entire object to the empty `A` by assigning a new, default-initialized `A` object to `a`.

Some input operators need to do additional data verification. The input operator might need to set the stream's condition state to indicate failure even though technically speaking the actual IO was successful. Usually an input operator should set only the `failbit`.

18.2.2 Arithmetic and Relational Operators

Ordinarily, we define the arithmetic and relational operators as nonmember functions in order to allow conversions for either the left- or right-hand operand. Classes that define an arithmetic operator generally define the corresponding compound assignment operator as well.

Classes that define both an arithmetic operator and the related compound assignment ordinarily ought to implement the arithmetic operator by using the compound assignment.

```
A operator+(const A &a1, const A &a2) {
  A ret = a1;  // Copy data members.
  ret += a2;  // use compound assignment.
  return ret;
```

```
}
```

Equality operators usually compare every data member and treat two objects as equal if and only if all the corresponding members are equal.

```
bool operator==(const A &a1, const A &a2) {
  return a1._data1 == a2._data1 && a1._data2 == a2._data2;
}

bool operator!=(const A &a1, const A &a2) {
  return !(a1 == a2);
}
```

Classes for which the equality operator is defined also often (but not always) have relational operators. In particular, because the associative containers and some of the algorithms use the less-than operator, it can be useful to define an `operator<`. Ordinarily the relational operators should

- Define an ordering relation that is consistent with the requirements for use as a key to an associative container;
- Define a relation that is consistent with == if the class has both operators. In particular, if two objects are `!=`, then one object should be < the other.

Therefore, if a single logical definition for < exists, classes usually should define the < operator. However, if the class also has ==, define < only if the definitions of < and == yield consistent results.

18.2.3 Assignment Operators

Assignment operators can be overloaded. Assignment operators, regardless of parameter type, must be defined as member functions.

```
A &operator(const A &a);  // Copy assignment.
A &operator(A &&a);  // Move assignment.
// For list initialization:
// A a;
// a = {...};
A &operator(initializer_list<T> il);
```

Unlike the copy- and move-assignment operators, the operator for list initialization does not need to check for self-assignment. The parameter is an `initializer_list<T>`, which means that `il` cannot be the same object as the one denoted by `this`.

Compound assignment operators are not required to be members. However, we prefer to define all assignments, including compound assignments, in the class.

```
A &operator+=(const A &a);
```

18.2.4 Subscript Operator

Classes that represent containers from which elements can be retrieved by position often define the subscript operator. The subscript operator must be a member function. If a class has a subscript operator, it usually should define two versions: one that returns a plain reference and the other that is a `const` member and returns a reference to `const`.

```
A &operator[](size_t i);
const A &operator[](size_t i) const;
```

18.2.5 Increment and Decrement Operators

The increment (++) and decrement (–) operators are most often implemented for iterator classes. Classes that define increment or decrement operators should define both the prefix and postfix versions. These operators usually should be defined as members.

```
A &operator++() {  // Prefix operators.
  ...
  return *this;
}

A &operator--() {
  ...
  return *this;
}

A operator++(int) {  // Postfix operators.
  A a = *this;
  ++*this;
  return a;
}

A operator--(int) {
  A a = *this;
  --*this;
```

```
  return a;
}
```

18.2.6 Member Access Operators

The dereference (`*`) and arrow (`->`) operators are often used in classes that represent iterators and in smart pointer classes. Operator arrow must be a member. The dereference operator is not required to be a member but usually should be a member as well.

The overloaded arrow operator must return either a pointer to a class type or an object of a class type that defines its own operator arrow.

```
T &operator*() const;

T *operator->() const {
  return &this->operator*();
}
```

18.2.7 Function-Call Operator

Classes that overload the call operator allow objects of its type to be used as if they were a function. The function-call operator must be a member function. A class may define multiple versions of the call operator, each of which must differ as to the number or types of their parameters.

18.3 Overloading, Conversions, and Operators

DEFINITION 18.2 **Class-type conversion/User-definded conversion** Conversions to or from class types are defined by constructors and conversion operators, respectively. Non-`explicit` constructors that take a single argument define a conversion from the argument type to the class type. Conversion operators define conversions from the class type to the specified type.

DEFINITION 18.3 **Conversion operator** A member function that defines a conversions from the class type to another type. A conversion operator must be a member of the class from which it converts and is usually a `const` member. These operators have no return type and take no parameters. They return a value convertible to the type of the conversion operator.

```
operator int() const {
  return _val;
}
```

If there is no single one-to-one mapping between an object of class type and a value of built-in type. In such cases, it is better not to define the conversion operator. Instead, the class ought to define one or more ordinary members to extract the information in these various forms.

DEFINITION 18.4 **`explicit` conversion operator** Conversion operator preceeded by the `explicit` keyword. Such operators are used for implicit conversions only in conditions, or explicitly through a `static_cast`.

Conversion to `bool` is usually intended for use in conditions. As a result, operator `bool` ordinarily should be defined as `explicit`.

Don't define mutually converting classes — if class `A` has a constructor that takes an object of class `B`, do not give `B` a conversion operator to type `A`.

Avoid conversions to the built-in arithmetic types. In particular, if you do define a conversion to an arithmetic type, then

- Do not define overloaded versions of the operators that take arithmetic types. If users need to use these operators, the conversion operation will convert objects of your type, and then the built-in operators can be used.
- Do not define a conversion to more than one arithmetic type. Let the standard conversions provide conversions to the other arithmetic types.

In a call to an overloaded function, the rank of an additional standard conversion (if any) matters only if the viable functions require the same user- defined conversion. If different user-defined conversions are needed, then the call is ambiguous.

Providing both conversion functions to an arithmetic type and overloaded operators for the same class type may lead to ambiguities between the overloaded operators and the built-in operators.

V OBJECT ORIENTED PROGRAMMING

19 Object-Oriented Programming: Preview

Both object-oriented programming (OOP) and generic programming deal with types that are not known at the time the program is written. The distinction between the two is that in generic programming types become known during compilation, whereas in OOP types are not known until run time.

DEFINITION 19.1 **Object-oriented programming** Method of writing programs using *data abstraction*, *inheritance*, and *dynamic binding*.

- Using *data abstraction*, we can define classes that separate interface from implementation.
- Through *inheritance*, we can define classes that model the relationships among similar types.
- Through *dynamic binding*, we can use objects of these types while ignoring the details of how they differ.

OOP provides an ISA relationship between classes.

20 Inheritance

20.1 Basic Concepts

DEFINITION 20.1 **Inheritance** Programming technique for defining a new class (known as a derived class) in terms of an existing class (known as the base class). The derived class inherits the members of the base class.

DEFINITION 20.2 **Base class** Class from which other classes inherit. The members of the base class become members of the derived class.

A class must be defined, not just declared, before we can use it as a base class.

DEFINITION 20.3 **Derived class** Class that inherits from another class. A derived class can override the virtuals of its base and can define new members. A derived-class scope is nested in the scope of its base class(es); members of the derived class can use members of the base class directly.

```
class D: public B {  // Class derivation list.
  ...
};
```

DEFINITION 20.4 **Class derivation list** List of base classes, each of which may have an optional access level, from which a derived class inherits. If no access specifier is provided, the inheritance is `public` if the derived class is defined with the `struct` keyword, and is `private` if the class is defined with the `class` keyword.

A derived object contains multiple parts

- Subobjects corresponding to each base class from which the derived class inherits.
- A subobject containing the (non-`static`) members defined in the derived class itself.

Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type as if it were an object of its base type(s).

DEFINITION 20.5 **Direct base class** Base class from which a derived class inherits directly. Direct base classes are specified in the derivation list of the derived class. A direct base class may itself be a derived class.

DEFINITION 20.6 **Indirect base class** Base class that does not appear in the derivation list of a derived class. A class from which the direct base class inherits, directly or indirectly, is an indirect base class to the derived class.

Sometimes we define a class that we don't want others to inherit from. We can prevent a class from being used as a base by following the class name with `final`:

```
class A final {
  ...
}
```

20.2 Access Control and Inheritance

DEFINITION 20.7 **Accessible** Base class member that can be used through a derived object. Accessibility depends on the access specifier used in derivation list of the derived class and the access level of the member in the base class.

20.2.1 `protected` Members

DEFINITION 20.8 **`protected` access specifier** Members defined after the `protected` keyword may be accessed by the members and friends of a derived class. However, these members are only accessible through derived objects. `protected` members are not accessible to ordinary users of the class.

20.2.2 `public`, `protected`, and `private` Inheritance

DEFINITION 20.9 **`public` inheritance** The `public` interface of the base class is part of the `public` interface of the derived class.

DEFINITION 20.10 **`protected` inheritance** The `public` and `protected` members of the base class are `protected` members of the derived class.

DEFINITION 20.11 **`private` inheritance** The `public` and `protected` members of the base class are `private` members of the derived class.

20.2.3 Friendship and Inheritance

Just as friendship is not transitive, friendship is also not inherited.

20.2.4 Exempting Individual Members

Sometimes we need to change the access level of a name that a derived class inherits. We can do so by providing a `using` declaration. A derived class may provide a `using` declaration only for names it is permitted to access.

```
class D: private B {
public:
  using B::member;
}
```

20.3 Defining a Derived Classes

20.3.1 Derived-Class Constructors

Each class controls how its members are initialized. Although a derived object contains members that it inherits from its base, it cannot directly initialize those members. Like any other code that creates an object of the base-class type, a derived class must use a base-class constructor to initialize its base-class part.

```
D(...): B(...), ... {
  ...
}
```

The base class is initialized first, and then the members of the derived class are initialized in the order in which they are declared in the class.

20.3.2 Inheritance and `static` Members

If a base class defines a `static` member, there is only one such member defined for the entire hierarchy. Regardless of the number of classes derived from a base class, there exists a single instance of each `static` member.

`static` members obey normal access control. If the member is `private` in the base class, then derived classes have no access to it. Assuming the member is accessible, we can use a `static` member through either the base or derived.

```
B::static_member();
D::static_member();
d.static_member();
static_member();  // Accessed through this.
```

20.3.3 Declarations of Derived Classes

A derived class is declared like any other class. The declaration contains the class name but does not include its derivation list.

```
class D;
```

20.4 Multiple and Virtual Inheritance

DEFINITION 20.12 **Multiple inheritance** Class with more than one direct base class. The derived class inherits the members of all its base classes. A separate access specifier may be provided for each base class.

DEFINITION 20.13 **Virtual inheritance** Form of multiple inheritance in which derived classes share a single copy of a base that is included in the hierarchy more than once.

DEFINITION 20.14 **Virtual base class** Base class that specifies `virtual` in its own derivation list. A virtual base part occurs only once in a derived object even if the same class appears as a virtual base more than once in the hierarchy. In nonvirtual inheritance a constructor may initialize only its direct base class(es). When a class is inherited virtually, that class is initialized by the most derived class, which therefore should include an initializer for all of its virtual parent(s).

21 Dynamic Binding

21.1 Conversions and Inheritance

21.1.1 Derived-to-Base Conversion

DEFINITION 21.1 **Derived-to-base conversion** Implicit conversion of a derived object to a reference to a base class, or of a pointer to a derived object to a pointer (built-in pointer or smart pointer) to the base type.

```
B b;  // Object of base type.
D d;  // Object of derived type.
B *pb = &b;  // pb points to a B object.
pb = &d;  // pb points to the B part of d.
B &rb = d;  // rb bounds to the B part of d.
```

The conversion from derived to base exists because every derived object contains a base-class part to which a pointer or reference of the base-class type can be bound. This fact has a crucially important implication: When we use a reference (or pointer) to a base-class type, we don't know the actual type of the object to which the pointer or reference is bound.

21.1.2 Static Type and Dynamic Type

DEFINITION 21.2 **Static type** Type with which a variable is defined or that an expression yields. Static type is known at compile time.

DEFINITION 21.3 **Dynamic type** Type of an object at run time.

- The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression's static type.
- The dynamic type of an object to which a reference refers or to which a pointer points may differ from the static type of the reference or pointer.

A pointer or reference to a base-class type can refer to an to object of derived type. In such cases the static type is reference (or pointer) to base, but the dynamic type is reference (or pointer) to derived.

DEFINITION 21.4 **Polymorphism** As used in object-oriented programming, refers to the ability to obtain type-specific behavior based on the dynamic type of a reference or pointer.

21.1.3 Accessibility of Derived-to-Base Conversion

- User code may use the derived-to-base conversion only if `D` inherits `public` from `B`.

- Member functions and friends of `D` can use the conversion to `B` regardless of how `D` inherits from `B`.
- Member functions and friends of classes derived from `D` may use the derived-to-base conversion if `D` inherits from `B` using either `public` or `protected`.

In short, for any given point in your code, if a `public` member of the base class would be accessible, then the derived-to-base conversion is also accessible, and not otherwise.

21.1.4 No Implicit Conversion From Base to Derived

There is no similar guarantee for base-class objects. A base object has only the members defined by the base class; it doesn't have the members defined by the derived class.

We cannot convert from base to derived even when a base pointer or reference is bound to a derived object, since the compiler has no way to know (at compile time) that a specific conversion will be safe at run time. In those cases when we know that the conversion from base to derived is safe, we can use a `static_cast` to override the compiler.

For any given point in your code, if a `public` member of the base class would be accessible, then the derived-to-base conversion is also accessible, and not otherwise.

Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type as if it were an object of its base type(s).

21.1.5 Conversions in Copy-Control

Although the automatic conversion applies only to pointers and references, most classes in an inheritance hierarchy (implicitly or explicitly) define the copy-control members. As a result, we can often copy, move, or assign an object of derived type to a base-type object. However, copying, moving, or assigning a derived-type object to a base-type object copies, moves, or assigns only the members in the base-class part of the object.

DEFINITION 21.5 **Sliced down** What happens when an object of derived type is used to copy, move, or assign an object of the base type. The derived portion of the object is "sliced down", leaving only the base portion, which is assigned to the base. The derived part of the object is ignored.

```
// This is actually calling B's constructor B(const B &).
B b = d;
// This is actually calling B's assignment operator:
// operator=(const B &).
b = d;
```

When we pass a derived object to a base-class constructor/assignment operator, the constructor that is run is defined in the base class. That constructor knows only about the members of the base class itself.

21.2 Virtual Functions and Dynamic Binding

21.2.1 Virtual Functions

DEFINITION 21.6 **Virtual function** Member function that defines type-specific behavior. The base class defines functions as `virtual` when it expects its derived classes to define for themselves. Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound.

A function that is `virtual` in a base class is implicitly `virtual` in its derived classes. A derived class must include in its own class body a declaration of all the virtual functions it intends to define for itself. The `virtual` keyword appears only on the declaration inside the class and may not be used on a function definition that appears outside the class body.

When a derived class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. When a derived class overrides a virtual, the parameters in the base and derived classes must match exactly.

21.2.2 Dynamic Binding

DEFINITION 21.7 **Dynamic binding/Run-time binding** Delaying until run time the selection of which function to run. In C++, dynamic binding refers to the runtime choice of which virtual function to run based on the underlying type of the object to which a reference or pointer is bound.

Dynamic binding happens only when a `virtual` function is called through a pointer or a reference. When we call a `virtual` function on an expression that has a plain — nonreference and nonpointer — type, that call is bound at compile time.

Member functions that are not declared as `virtual` are resolved at compile time, not run time.

21.2.3 Virtual Function and Default Argument

If a call uses a default argument, the value that is used is the one defined by the static type through which the function is called. Virtual functions that have default arguments should use the same argument values in the base and derived classes.

21.2.4 Circumventing the Virtual Mechanism

In some cases, we want to prevent dynamic binding of a call to a virtual function; we want to force the call to use a particular version of that virtual. We can use the scope operator to do so.

```
// Calls the version from the base class regardless of the
// dynamic type of bp
pb->B::f();
```

Ordinarily, only code inside member functions (or friends) should need to use the scope operator to circumvent the virtual mechanism. The most common usage is when a derived-class virtual function calls the version from the base class. The versions defined in the derived classes would do whatever additional work is particular to their own type.

The static type of an object, reference, or pointer determines which members of that object are visible. Even when the static and dynamic types might differ (as can happen when a reference or pointer to a base class is used), the static type determines what members can be used.

A derived-class member with the same name as a member of the base class hides direct use of the base-class member. Therefore, aside from overriding inherited virtual functions, a derived class usually should not reuse names defined in its base class.

21.3 Class Scope under Inheritance

Each class defines its own scope within which its members are defined. Under inheritance, the scope of a derived class is nested inside the scope of its base classes.

21.3.1 Name Lookup Happens at Compile Time

The static type of an object, reference, or pointer determines which members of that object are visible, i.e., the static type determines what members can be used.

21.3.2 Name Collisions and Inheritance

A derived class can reuse a name defined in one of its direct or indirect base classes. A derived-class member with the same name as a member of the base class hides direct use of the base-class member.

As a result, functions defined in a derived class do not overload members defined in its base class(es). The base member is hidden even if the functions have different parameter lists.

We can use a hidden base-class member by using the scope operator.

Aside from overriding inherited virtual functions, a derived class usually should not reuse names defined in its base class.

21.3.3 The `override` and `final` Specifiers

We can specify `override` on a virtual function in a derived class. Then the compiler will reject a program if a function marked `override` does not override an existing virtual function.

DEFINITION 21.8 **`override`** Virtual function defined in a derived class that has the same parameter list as a virtual in a base class overrides the base-class definition.

```
void f() override {
  ...
}
```

We can also designate a function as `final`. Any attempt to override a function that has been defined as `final` will be flagged as an error.

`final` and `override` specifiers appear after the parameter list (including any `const` or reference qualifiers) and after a trailing return.

21.3.4 Overriding Overloaded Functions

As with any other function, a member function (virtual or otherwise) can be overloaded. A derived class can override zero or more instances of the overloaded functions it inherits. If a derived class wants to make all the overloaded versions available through its type, then it must override all of them or none of them.

Sometimes a class needs to override some, but not all, of the functions in an overloaded set. Instead of overriding every base-class version that it inherits, a derived class can provide a `using` declaration for the overloaded member. A `using` declaration specifies only a name; it may not specify a parameter list. Thus, a `using` declaration for a base-class member function adds all the overloaded instances of that function to the scope of the derived class. Having brought all the names into its scope, the derived class needs to define only those functions that truly depend on its type. It can use the inherited definitions for the others.

21.3.5 Name Lookup and Inheritance

Given the call `p->f()` (or `a.f()`), the following four steps happen.

1. First determine the static type of `p` (or `a`). Because we're calling a member, that type must be a class type.
2. Look for `f` in the class that corresponds to the static type of `p` (or `a`). If `f` is not found, look in the direct base class and continue up the chain of classes until `f` is found or the last class is searched. If `f` is not found in the class or its enclosing base classes, then the call will not compile.
3. Once `f` is found, do normal type checking to see if this call is legal given the definition that was found.
4. Assuming the call is legal, the compiler generates code, which varies depending on whether the call is virtual or not.
   - If `f` is virtual and the call is made through a reference or pointer, then the compiler generates code to determine at run time which version to run based on the dynamic type of the object.
   - Otherwise, if the function is nonvirtual, or if the call is on an object (not a reference or pointer), the compiler generates a normal function call.

21.4 Abstract Base Classes

DEFINITION 21.9 **Pure virtual function** Virtual function declared in the class header using `= 0` just before the semicolon. A pure virtual function need not be (but may be) defined. Classes with pure virtuals are abstract classes. If a derived class does not define its own version of an inherited pure virtual, then the derived class is abstract as well.

DEFINITION 21.10 **Abstract (base) class** Class that has one or more pure virtual functions. We cannot create objects of an abstract base-class type.

DEFINITION 21.11 **Refactoring** Redesigning programs to collect related parts into a single abstraction, replacing the original code with uses of the new abstraction. Typically, classes are refactored to move data or function members to the highest common point in the hierarchy to avoid code duplication.

21.5 Constructors and Copy Control

21.5.1 Virtual Destructors

21.5.1.1 Virtual Destructors

Base classes ordinarily should define a virtual destructor. Virtual destructors are needed even if they do no work.

If that pointer points to a type in an inheritance hierarchy, it is possible that the static type of the pointer might differ from the dynamic type of the object being destroyed. We arrange to run the proper destructor by defining the destructor as virtual in the base class.

Executing `delete` on a pointer to base that points to a derived object has undefined behavior if the base's destructor is not virtual.

21.5.1.2 Virtual Destructors Turn Off Synthesized Move

If a class defines a destructor — even if it uses `= default` to use the synthesized version — the compiler will not synthesize a move operation for that class.

21.5.2 Synthesized Copy Control and Inheritance

21.5.2.1 Synthesized Copy Control

The synthesized copy-control members in a base or a derived class execute like any other synthesized constructor, assignment operator, or destructor: They memberwise initialize, assign, or destroy the members of the class itself. In addition, these synthesized members initialize, assign, or destroy the direct base part of an object by using the corresponding operation from the base class.

For example, the synthesized `D` default constructor runs the `B` default constructor; the synthesized `D` copy constructor uses the (synthesized) `B` copy constructor.

21.5.2.2 Base Classes and Deleted Copy Control in the Derived

- If the default constructor, copy constructor, copy-assignment operator, or destructor in the base class is deleted or inaccessible, then the corresponding member in the derived class is defined as deleted.
- If the base class has an inaccessible or deleted destructor, then the synthesized default and copy constructors in the derived classes are defined as deleted,
- As usual, the compiler will not synthesize a deleted move operation. If we use `= default` to request a move operation, it will be a deleted function in the derived if the corresponding operation in the base is deleted or inaccessible. The move constructor will also be deleted if the base class destructor is deleted or inaccessible.

21.5.2.3 Move Operations and Inheritance

Most base classes define a virtual destructor. As a result, by default, base classes generally do not get synthesized move operations. Moreover, by default, classes derived from a base class that doesn't have move operations don't get synthesized move operations either.

Because lack of a move operation in a base class suppresses synthesized move for its derived classes, base classes ordinarily should define the move operations if it is sensible to do so.

21.5.3 Derived-Class Copy-Control Members

21.5.3.1 Derived-Class Copy-Control Members

When a derived class defines a copy or move operation, that operation is responsible for copying or moving the entire object, including base-class members.

Unlike the constructors and assignment operators, the destructor is responsible only for destroying the resources allocated by the derived class. Recall that the members of an object are implicitly destroyed. Similarly, the base-class part of a derived object is destroyed automatically.

21.5.3.2 Defining a Derived Copy or Move Constructor

When we define a copy or move constructor for a derived class, we ordinarily use the corresponding base-class constructor to initialize the base part of the object.

By default, the base-class default constructor initializes the base-class part of a derived object. If we want copy (or move) the base-class part, we must explicitly use the copy (or move) constructor for the base class in the deriveds constructor initializer list.

```
D(const D &d): B(d) {  // Copy the base member.
  ...
}
D(D &&d): B(std::move(d)) {  // Move the base member.
  ...
}
```

21.5.3.3 Derived-Class Assignment Operator

Like the copy and move constructors, a derived-class assignment operator, must assign its base part explicitly.

```
D &operator=(const D &d)
{
  B::operator=(d);  // Assigns the base part.
  ...
  return *this;
}
```

21.5.3.4 Derived-Class Destructor

A derived destructor is responsible only for destroying the resources allocated by the derived class.

Objects are destroyed in the opposite order from which they are constructed: The derived destructor is run first, and then the base-class destructors are invoked, back up through the inheritance hierarchy.

```
// B::~B invoked automatically.
~D() {
  ...  // Do what it takes to clean up derived members.
}
```

21.5.3.5 Calls to Virtuals in Constructors and Destructors

While an object is being constructed it is treated as if it has the same class as the constructor; calls to virtual functions will be bound as if the object has the same type as the constructor itself. Similarly, for destructors. This binding applies to virtuals called directly or that are called indirectly from a function that the constructor (or destructor) calls.

If a constructor or destructor calls a virtual, the version that is run is the one corresponding to the type of the constructor or destructor itself.

21.5.4 Inherited Constructors

21.5.4.1 Inherited Constructors

A derived class can reuse the constructors defined by its direct base class. A class may inherit constructors only from its direct base. A class cannot inherit the default, copy, and move constructors. If the derived class does not directly define these constructors, the compiler synthesizes them as usual.

A derived class inherits its base-class constructors by providing a `using` declaration that names its (direct) base class.

```
class D: public B {
public:
  using B::B;  // Inherit B's constructors.
}
```

The compiler generates a derived constructor corresponding to each constructor in the base. That is, for each constructor in the base class, the compiler generates a constructor in the derived class that has the same parameter list.

```
D(params): B(args) {}
```

`parms` is the parameter list of the constructor, and `args` pass the parameters from the derived constructor to the base constructor.

If the derived class has any data members of its own, those members are default initialized.

21.5.4.2 Characteristics of an Inherited Constructor

Unlike using declarations for ordinary members, a constructor `using` declaration does not change the access level of the inherited constructor(s).

A `using` declaration can't specify `explicit` or `constexpr`. If a constructor in the base is `explicit` or `constexpr`, the inherited constructor has the same property.

If a base-class constructor has default arguments, those arguments are not inherited. Instead, the derived class gets multiple inherited constructors in which each parameter with a default argument is successively omitted.

If a base class has several constructors, then with two exceptions, the derived class inherits each of the constructors from its base class.

The first exception is that a derived class can inherit some constructors and define its own versions of other constructors. If the derived class defines a constructor with the same parameters as a constructor in the base, then that constructor is not inherited. The one defined in the derived class is used in place of the inherited constructor.

The second exception is that the default, copy, and move constructors are not inherited. These constructors are synthesized using the normal rules. An inherited constructor is not treated as a user-defined constructor. Therefore, a class that contains only inherited constructors will have a synthesized default constructor.

21.6 Containers and Inheritance

When we use a container to store objects from an inheritance hierarchy, we generally must store those objects indirectly.

When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers) to the base class. As usual, the dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base.

VI GENERIC PROGRAMMING II

22 Class Templates

22.1 Defining a Class Template

```
template <typename T>
class A {
  ...
};
```

Inside the scope of a class template, we may refer to the template without specifying template argument(s).

22.1.1 Instantiation Definitions Instantiate All Members

An instantiation definition for a class template instantiates all the members of that template including inline member functions. When the compiler sees an instantiation definition it cannot know which member functions the program uses. Hence, unlike the way it handles ordinary class template instantiations, the compiler instantiates all the members of that class. Even if we do not use a member, that member will be instantiated. Consequently, we can use explicit instantiation only for types that can be used with all the members of that template.

An instantiation definition can be used only for types that can be used with every member function of a class template.

22.1.2 Member Functions of Class Templates

A class template member function is itself an ordinary function. However, each instantiation of the class template has its own version of each member. As a result, a member function of a class template has the same template parameters as the class itself.

Therefore, a member function defined outside the class template body must say to which class the member belongs. For example,

```
template <typename T>
void A<T>::f();
```

Inside the scope of a class template, we may refer to the template without specifying template argument(s).

```
A& operator++();
```

By default, a member of an instantiated class template is instantiated only if the member is used.

22.2 Class Templates and Friends

When a class contains a friend declaration, the class and the friend can independently be templates or not.

- A class template that has a nontemplate friend grants that friend access to all the instantiations of the template.
- When the friend is itself a template, the class granting friendship controls whether friendship includes all instantiations of the template or only specific instantiation(s).

22.2.1 One-to-One Friendship

In order to refer to a specific instantiation of a template (class or function) we must first declare the template itself. A template declaration includes the template's template parameter list.

```
template <typename T>
class B;

template <typename T>
class A {
  friend class B<T>;
};
```

22.2.2 General and Specific Template Friendship

No forward declaration required when we befriend all instantiations

```
template <typename T>
class A {
  // All instances of B are friends of each instance of A.
  template <typename D>
  friend class B;
};
```

22.2.3 Befriending the Template's Own Type Parameter

We can make a template type parameter a friend.

```
template <typename T> class A {
  // Grants access to the type used to instantiate Bar.
  friend T;
};
```

22.3 Other Class Template Features

22.3.1 Template Type Aliases

```
typedef A<T> AT;
```

Because a template is not a type, we cannot define a `typedef` that refers to a template.

However, we can define a type alias for a class template

```
template<typename T>
using twin = pair<T, T>
twin<string> authors;  // pair<string, string>
```

22.3.2 `static` Members of Class Templates

Each instantiation of the template class has its own instance of the `static` members. To use a `static` member through the class, we must refer to a specific instantiation.

However, there must be exactly one definition of each `static` data member of a template class. We define a `static` data member as a template similarly to how we define the member functions of that template.

```
template <typename T>
size_t A<T>::static_member = 0;
```

22.3.3 Template Default Arguments and Class Templates

If a class template provides default arguments for all of its template parameters, and we want to use those defaults, we must put an empty bracket pair following the template's name.

```
template <typename T = int>
class A {
  ...
};

A<> a;
```

22.4 Member Templates

DEFINITION 22.1 **Member template** Member function that is a template. A member template may not be virtual.

A class — either an ordinary class or a class template — may have a member function that is itself a template.

22.4.1 Member Templates of Ordianary (Non-template) Classes

```
class A {
  template <typename T>
  void f(T a) { ... }
};
```

22.4.2 Member Templates of Class Templates

In this case, both the class and the member have their own, independent, template parameters.

When we define a member template outside the body of a class template, we must provide the template parameter list for the class template and for the function template.

```
template <typename T>
template <typename It>
A<T>::A(It begin, It end) {
  ...
}
```

To instantiate a member template of a class template, the compiler typically deduces template argument(s) for the member template's own parameter(s) from the arguments passed in the call.

22.5 Partial Specialization

DEFINITION 22.2 **Partial specialization** Version of a class template in which some some but not all of the template parameters are specified or in which one or more parameters are not completely specified.

VII ADVANCED TOPICS

23 Specialized Library Facilities

23.1 The `tuple` Type

DEFINITION 23.1 **`tuple`** Template that generates types that hold unnamed members of specified types. There is no fixed limit on the number of members a tuple can be defined to have.

A `tuple` is most useful when we want to combine some data into a single object but do not want to bother to define a data structure to represent those data.

To be continued.

23.2 The `bitset` Type

DEFINITION 23.2 **`bitset`** Standard library class that holds a collection of bits of a size that is known at compile time, and provides operations to test and set the bits in the collection.

To be continued.

23.3 Regular Expressions

DEFINITION 23.3 **Regular expression** A way of describing a sequence of characters.

To be continued.

23.4 Random Numbers

DEFINITION 23.4 **Random-number generator** Combination of a random-number engine type and a distribution type. They are defined in the `<random>` header.

23.4.1 Random-Number Engines and Distribution

DEFINITION 23.5 **Random-number engine** Library type that generates random unsigned numbers. The random-number engines are function-object classes that define a call operator that takes no arguments and returns a random unsigned number. Engines are intended to be used only as inputs to random-number distributions.

DEFINITION 23.6 **default_random_engine** The library defines several random-number engines that differ in terms of their performance and quality of randomness. Each compiler designates one of these engines as the `default_random_engine` type. This type is intended to be the engine with the most generally useful properties.

DEFINITION 23.7 **Random-number distribution** Standard library type that transforms the output of a random-number engine according to its named distribution. For example, `uniform_int_distribution<T>` generates uniformly distributed integers of type `T`, `normal_distribution<T>` generates normally distributed numbers, and so on.

C++ programs should not use the library `rand` function. Instead, they should use the `default_random_engine` along with an appropriate distribution object.

```
// Generate unsigned random integers.
default_random_engine e;
// Uniformly distributed in range [0, 9].
uniform_int_distribution<> d(0,9);
for (size_t i = 0; i != 10; ++i) {
  // unif uses e as a source of numbers.
  // Each call returns a uniformly distributed value
  //  in the specified range.
  cout << d(e) << " ";
}
cout << endl;
```

A given random-number generator always produces the same sequence of numbers. A function with a local random-number generator should make that generator (both the engine and distribution objects) `static`. Otherwise, the function will generate the identical sequence on each call.

DEFINITION 23.8 **Seed** Value supplied to a random-number engine that causes it to move to a new point in the sequence of number that it generates.

Perhaps the most common approach to pick a seed is to call the system `time` function. This function, defined in the `<ctime>` header, returns the number of seconds since a given epoch. The `time` function takes a single parameter that is a pointer to a structure into which to write the time. If that pointer is null, the function just returns the time.

```
default_random_engine e(time(0));
```

Because `time` returns time as the number of seconds, this seed is useful only for applications that generate the seed at second-level, or longer, intervals. It might wind up with the same seed several times if the program is run repeatedly as part of an automated process;

23.4.2 Other Kinds of Distributions

```
uniform_int_distribution<> d(lower_bound, upper_bound);
uniform_real_distribution<> d(lower_bound, upper_bound);
normal_distribution<> d(mean, std);
// It returns true with probability p, default p = 0.5.
bernoulli_distribution d(p);
```

24 Specialized Tools and Techniques

24.1 Controlling Memory Allocation

Some applications need to take control of how memory is allocated. They can do so by defining their own versions — either class specific or global — of the library `operator new` and `operator delete` functions.

To be continued.

24.2 Run-Time Type Identification

Some programs need to directly interrogate the dynamic type of an object at run time. Run-time type identification (RTTI) provides language-level support for this kind of programming. RTTI applies only to classes that define virtual functions; type information for types that do not define virtual functions is available but reflects the `static` type.

To be continued.

24.3 Pointer to Class Member

When we define a pointer to a class member, the pointer type also encapsulates the type of the class containing the member to which the pointer points. A pointer to member may be bound to any member of the class that has the appropriate type. When we dereference a pointer to member, we must supply an object from which to fetch the member.

To be continued.

24.4 Enumerations

Type that groups a set of named integral constants.

To be continued.

24.5 Nested Classes

Classes defined in the scope of another class. Such classes are often defined as implementation classes of their enclosing class.

To be continued.

24.6 `union`: A Space-Saving Class

Special kind of class that may define several data members, but at any point in time, only one member may have a value. `union` are most often nested inside another class type.

To be continued.

24.7 Local Classes

Classes defined inside a function. All members of a local class must be defined in the class body. There are no `static` data members of a local class.

To be continued.

24.8 Inherently Nonportable Features

Including bit-fields and `volatile`, which make it easier to interface to hardware, and linkage directives, which make it easier to interface to programs written in other languages.

To be continued.