Rose's Programming in C++

Rose Enos

2024

Adapted from the following sources:

- *C++ for Python Programmers* by Jan Pearce and Brad Miller
- *Learn C++* by www.learncpp.com
- *GoogleTest User's Guide* by Google

Contents

1 Program Structures

1.1 The Compiler

A **scripting** language is clear, intuitive, and powerful. An **industrial-strength** language is performant and formal.

C++ is large and versatile, with high and low level features. It interacts directly with hardware, increasing performance. It supports abstraction, inheritance, polymorphism, and encapsulation.

An **interpreted** language can run directly on an interpreter. A **compiled** language must be compiled into **machine code** by a compiler before it can run. Compilation causes early error detection and increases performance.

The compiler checks for grammatical errors and translates the C++ `.cpp` file to a machine code object `.o` or `.obj` file. The linker combines the object files and produces an executable `.exe` file by checking the object files, cross-file dependencies, and library dependencies. The executable file is the build.

A **preprocessor directive** is a statement begun by `#`. A **header** is a set of definitions of functions and variables.

```
#include <libraryname>
```

copies the header definitions into the current file from an installed library.

```
#include "file_name"
```

copies from a file.

Every program must have the function

```
int main() {
    ...
    return exitcode;
}
```

which is called implicitly upon execution and returns the exit code `exitcode` (`0` upon success).

A single-line comment begins with `//`. A multi-line comment begins with `/*` and ends with `*/`. By convention, documentation begins with `/**` and ends with `*/`.

1.2 Allocation

C++ supports

- Static memory allocation: static and global variables are allocated at program start and deallocated at program end.
- Automatic memory allocation: formal parameters and local variables are allocated at block start and deallocated at block end in the small **stack**.

- Dynamic memory allocation: storage is requested from the large **heap** managed by the OS.

In static and automatic allocation, size must be known at compile time and allocation and deallocation are automatic.

`new variabletype` requests the type's worth of memory in the heap, optionally initializes, and returns a pointer to the address. It follows that accessing heap-allocated objects is slower than accessing stack-allocated objects.

`delete pointer` returns heap storage to the OS. A **dangling pointer** points to deallocated memory and should not be dereferenced or deleted. A pointer should be assigned `nullptr` after deletion.

Dynamic allocation throws `bad_alloc` if the OS does not have enough memory. `new std::nothrow variabletype` suppresses the exception and returns `nullptr` on failure. Deleting a `nullptr` does nothing.

In a **memory leak**, the address of a dynamically allocated variable is lost before it is deleted. Then the variable persists inefficiently until program end.

`new variabletype[length]` dynamically allocates a C-style array with non-constant length of type `std::size_t`. `delete[] pointer` deletes a C-style array.

1.3 Literals

A **literal**, or **literal constant**, is a value directly in code. The type of a literal is determined by suffix or default. A floating point literal can use `e` scientific notation.

A C string literal is constant and lives throughout the program.

Literal	Suffix	Type
integral	`u`	`unsigned int`
integral	`L`	`long`
integral	`uL`	`unsigned long`
integral	`LL`	`long long`
integral	`uLL`	`unsigned long long`
integral	`z`	unsigned `std::size_t`
integral	`uz`	`std::size_t`
floating point	`f`	`float`
floating point	`L`	`long double`
string	`s`	`std::string`
string	`sv`	`std::string_view`

Table 1: Literal Suffixes

1.4 Variables

Statements are grouped by `{}` and ended by `;`. If there is exactly one statement in a group, curly braces are unnecessary. C++ uses **static typing** so a variable

must be declared.

Assignment assigns a value to a variable. **Copy assignment** copies a value into a variable

```
variablename = value;
```

Initialization sets the initial value of a variable at declaration. **Default initialization** defers to defined default value or does not initialize

```
variabletype variablename;
```

Copy initialization does copy assignment after allocation

```
variabletype variablename = value;
```

Direct initialization initializes complex objects more efficiently

```
variabletype variablename ( value );
```

List initialization, or **uniform initialization**, is preferred because it works for most types and raises an error on lossy conversion

```
variabletype variablename { value };
```

Value initialization, or **zero initialization** for numbers, initializes a default value for the type

```
variabletype variablename {};
```

To prevent compilation warnings for unused variables, initialize with the attribute

```
[[maybe_unused]] variabletype variablename;
```

The **as-if rule** states that the compiler can modify code if it does not change the program's observable behavior. A variable is **optimized out**, or **optimized away**, if the compiler removes it.

A **compile-time constant** is a constant whose value must be known at compile time. A **runtime constant** is any other constant.

A **constant expression** is an expression of only compile-time constants and compile-time-evaluating functions. A constant expression can be evaluated at compile time for optimization. A **runtime expression** is any other expression.

A **constant expression variable** is compile-time constant and requires a constant expression to initialize

```
constexpr variabletype variablename { constexpression };
```

A constant expression variable is preferred to a constant variable except when working with dynamic allocation.

A formal parameter is at most runtime constant and cannot be a constant expression variable.

2 Control Structures

2.1 Conditionals

`if` is a Boolean-valued function used

```
if (condition) {
    statements
} else if (condition) {
    statements
...
} else {
    statements
}
```

`switch` checks enumerated constants for equality to an integral value

```
switch(integralvalue) {
    case enumeratedconstant:
        statements
        break;
    ...
    default:
        statements
}
```

`while` repeats a block while a condition is true

```
while (condition) {
    statements
}
```

`for` repeats a block a specified number of times

```
for (declaration; condition; iteration) {
    statements
}
```

The **conditional**, or **arithmetic if**, operator evaluates conditionally

```
condition ? expriftrue : expriffalse;
```

The result operands must be of the same type.

The entire expression should be parenthesized in a compound expression. The condition should be parenthesized if it contains operators.

2.2 Functions

A function is defined

```
returntype functionName(typedparameters) {
    statements
}
```

If a function has no return value, the return type is `void`.

An argument is **passed by reference** if the parameter name is preceded by `&`. Then the parameter is the argument. Otherwise the argument is **passed by value**, and the parameter is a copy of the argument.

A parameter is an array parameter if its name is succeeded by `[]`. An array parameter does not have a length. An array argument is always passed by reference.

Function overloading is declaring multiple functions of the same name differentiated by their parameters.

1. A recursive algorithm must have a base case.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself recursively.

2.3 Function Templates

A **template** is a function or class definition with placeholder types. The compiler generates typed overloaded functions or classes from the template.

A **function template** has placeholder **type template parameters**, or **generic types**, to type formal parameters, return values, and local values. The **template parameter declaration** immediately precedes the template and declares the placeholder types

```
template <typename T>
T functionName(T params) {
    ...
}
```

In **function template instantiation**, a **specialization** is created from a **primary template**. In **implicit instantiation**, a specialization is instantiated and immediately called

```
functionName<actual_type>(args);
```

A specialization is instantiated only if a specialization of the same types has not already been instantiated.

In **template argument deduction**, the types are omitted at instantiation and deduced from the actual parameter types

```
functionName<>(args);
```

If the angular brackets are omitted, matching non-template functions take precedence. Template argument deduction without angular brackets is preferred.

Templates should be in header files. **Generic programming** is programming with templates.

2.4 Anonymous Functions

A **lambda expression**, or **closure** or **function literal**, defines an anonymous function

```
[ captures ] ( params ) -> returntype {
    ...
}
```

The return type can be omitted and defaults to `auto`. Then all possible return values must be of the same type. The parameter delimiters can be omitted where there are no parameters or return type. Lambda expressions are preferred over named functions for trivial or one-time use.

A lambda expression can initialize a variable of type `auto`. A lambda expression can be passed to a formal parameter whose type is `auto`, who is implicitly converted from a template, or whose type is `std::function<returntype>(params)>&`.

A **generic lambda** has parameters of type `auto`. A generic lambda is instantiated if it is called with new types.

A lambda expression is a constant expression if it has only constant expression captures and calls only constant expression functions.

A lambda expression can access only global, compile-time, and static identifiers. Other identifiers can be accessed by value only if they are listed in the capture clause. Captured variables persist across calls to the same lambda expression instantiation.

Captures are constant unless the lambda expression is marked mutable

```
[ captures ] (params) mutable -> returntype {
    ...
}
```

A variable can be captured by reference

```
[ &captures ] (params) -> returntype {
    ...
}
```

The **default capture** `=` or `&` captures all variables mentioned in the lambda expression by value or reference, respectively, if they are not explicitly captured. The default capture must appear first in the capture clause.

A trivial variable can be defined at instantiation of a lambda expression

```
[ newvariable { initialize } ] (params) -> returntype {
    ...
}
```

A mutable lambda expression can be passed by reference to a function using `<functional>`

```
returntype functionName(const std::function<lambdareturn(params)>& f);
```

by initializing the lambda expression as a function object

```
std::function variableName { lambdaexpression };
```

or passing it as a reference wrapper

```
functionName(std::ref(lambdaVariable));
```

2.5 Assertions

`std::abort()` is preferred over `std::exit()` for debugging.

A **precondition** must be true before, an **invariant** must be true during, and a **postcondition** must be true after a code section.

An **assertion** in `<cassert>` aborts at runtime with an error message on a false statement

```
assert(statement && message);
```

The compile option `NDEBUG` deactivates assertions.

A **static** assertion aborts at compile time

```
static_assert(constexpr, message);
```

2.6 Exceptions

A **throw**, or **raise**, statement signals an error

```
throw errorvalue;
```

The **try** block catches exceptions inside it

```
try {
    ...
}
```

At the exception jumps to a **catch** block

```
catch(errortype e) {
    ...
}
```

and resumes below the catch block.

Only catch blocks may be between a try block and a catch block. Error types are not implicitly converted.

A catch block typically

- prints an error;
- returns an error value;
- throws an exception; or

- terminates the program.

An uncaught exception unwinds the stack until a matching catch block is found. Unwinding the stack destroys the topmost function.

If an exception is not caught, `std::terminate()` is called. Then the call stack may or may not be unwound.

A **catch-all handler** is a catch block for type `...`. The catch-all handler should be last in a series of catch blocks. `main` should have a catch-all handler in production builds.

If a constructor throws an exception, the object is not constructed and initialized members are destroyed. It follows that allocations should be made by members of wrapper types.

An **exception class** is designed to be thrown as an exception. Exception classes should be caught by reference.

The catch block for a derived exception class should precede the catch block for the base exception class.

Standard library exceptions are derived from `std::exception` in `<exception>`. The member function `what()` returns the error message. The standard exception should not be thrown.

`std::runtime_error` is a popular error type to throw manually. Exceptions must support copy construction.

An **exception specification** denotes what exceptions may be thrown by a function. A **non-throwing** function cannot unwind the call stack with an exception

```
returntype functionName(params) noexcept;
```

A **potentially throwing** function may unwind the call stack with an exception.

In a function signature, `noexcept(boolvar)` returns `noexcept` on a true actual parameter. Then a function template can determine the exception specification dynamically.

Destructors are implicitly non-throwing. Implicit or defaulted constructors, assignment operators, and comparison operators are implicitly non-throwing unless they call a potentially throwing function. Non-member functions, user-defined constructors, and user-defined operators are implicitly potentially throwing.

In an expression, `noexcept(value)` returns whether a value is non-throwing, and is checked at compile time.

An **exception safety guarantee** states function or class behavior on exception.

- No guarantee.
- Basic guarantee: no memory will be leaked and the program state may be modified.
- Strong guarantee: no memory will be leaked and the program state will not be modified.

- No throw guarantee: no exception will be thrown.
- No fail guarantee: no error will occur.

Destructors, functions that deallocate memory, and functions called by no-throw functions should be no throw. Move functions, swap functions, container clearing functions, operations on `std::unique_ptr`, and functions called by no-fail functions should be no fail.

3 Classes

3.1 Member Variables

A **class invariant** is a condition that must be true while an object lives. An object is in an **invalid state** if it violates a class invariant. Then an object should not be used.

A **class** is a compound type with **member variables** and **member functions** defined

```
class ClassName {
    membervariabledeclarations
    memberfunctiondeclarations
}
```

An object of a class without a constructor is created

```
ClassName objectName { membervariablevalues };
```

A member variable value is accessed

```
objectName.memberVariableName
```

A **container** class holds objects of another type. Most container classes can

- construct an empty container;
- insert an object;
- remove an object;
- count the objects;
- remove all objects;
- access an object;
- sort the objects.

where such operations are efficient.

A **value** container stores by value. A **reference** container stores by reference or pointer. Most containers cannot hold objects of different types.

3.2 Member Functions

A function outside a class is a **non-member function**, or **free function**.

A member function on an object does not need to state the name of the object as a parameter or to access its member variables and functions because the object is passed implicitly as the **implicit object**. A member function definition is compiled after all declarations in the class.

A `class` and a `struct` are structurally identical, but `struct` should not have a constructor. A `namespace` should be used where there are no member variables.

The member variables of a constant object are non-modifiable, including by member functions. A **constant member function**

```
returntype functionName(params) const;
```

cannot modify the object or call non-constant member functions. Only constant member functions can be called on a constant object. Constant member functions are preferred over non-constant member functions. A parameter passed by constant reference is a constant object.

The **access level** of a member determines its accessibility of `public`, `private`, or `protected`. A **public** member has no access restrictions. Members of a `struct` are public by default and should not be made private or protected.

A **private** member can only be accessed by objects of its class. Members of a `class` are private by default. A class with private members cannot be initialized in aggregate. Member variables should be private or protected. Member functions should be public unless only used internally.

Private member variables should begin with `m_`. Local static variables should begin with `s_`. Global variables should begin with `g_`.

An **access specifier** sets the access level of subsequent members

```
class ClassName {
accesslevel:
    memberdeclarations
}
```

A `struct` should be used where all access can be public, aggregate initialization is sufficient, and there are no class invariants, setup, or cleanup.

Members can be accessed from their class by the prefix `ClassName::`. Nontrivial member functions should be defined outside of the class definition. A class should be defined in a header file of its name. Member functions should be defined in a source file of the same name that includes the header file.

Member functions can be defined in the header file outside of the class definition with the prefix `inline` so that they can be included into multiple files. Default arguments for member functions should be placed inside the class definition.

A **friend declaration** in a class definition declares a **friend** class or function that can access private and protected members

```
friend functionsignature;
```

A friend non-member function should take an object as a formal parameter from which to access members.

A friend non-member can be defined in the class definition. A class can be forward declared

```
class ClassName;
```

Friends should prefer using the public interface over direct member access. Non-friend functions are preferred over friend functions.

3.3 Operators

A **constructor** is a member function called immediately after non-aggregate object creation, with arguments the initializing values of the object.

```
class ClassName {
public:
    ClassName(params) {
        initialization
        setup
    }
}
```

A constructor should be non-constant. The constructor can be called on constant objects.

A **member initializer list** initializes variables based on function parameters.

```
class ClassName {
public:
    ClassName(params)
        : member { param }
        , ...
        , member { param }
    {
        setup
    }
}
```

Member variables are initialized in the order the are defined in the class, so the member initializer list should reflect the member variable definitions. A member initializer list is preferable to inline assignment because inline assignment cannot initialize constant variables and references.

A **default constructor** is a constructor that requires no formal parameters. Value initialization and default initialization call the default constructor.

A class with no constructors has an **implicit default constructor**. The **explicitly defaulted default constructor** is

```
ClassName() = default;
```

An explicit default constructor is preferred. An automatic default constructor zero initializes member variables that are default initialized. An automatic default constructor does not make the class non-aggregate.

A **copy constructor** is a constructor that requires only one formal (preferred constant) reference parameter of type its class. The **implicit copy constructor** does memberwise initialization.

A copy constructor should have no side effects besides copying. The implicit copy constructor is preferred. Pass by value and return by value call the copy constructor.

The implicit copy constructor can be written

```
ClassName(const ClassName& className) = default;
```

Copying can be prevented by

```
ClassName(const ClassName& className) = delete;
```

List initialization disallows narrowing conversions. Copy initialization only considers non-explicit constructors and conversion functions. List initialization prioritizes matching list constructors over other matching constructors.

In **copy elision**, the compiler elides a call to the copy constructor by rewriting the code even if the copy constructor has side effects.

The copy assignment operator must return `*this`. Copy assignment should handle self-assignment specially in dynamic allocation to prevent dangling pointers.

The implicit copy assignment operator does memberwise assignment. If there are constant members, the implicit copy assignment operator is deleted.

A **shallow copy** is a memberwise copy. A **deep copy** is a memberwise copy that copies dereferenced non-null pointers. Where deep copies are possible, a destructor, copy constructor, and copy assignment operator should be defined.

`<initializer_list>` gives `std::initializer_list<>`, a view for values passed by initializer list, with `size()` and indexable iterators.

List initialization has precedence over other constructors when defined. If a list constructor is defined, list assignment should be defined, copy assignment should be deep, or copy assignment should be deleted.

3.4 Move Semantics

Move semantics allow transfer of member ownership.

A move constructor or move assignment operator transfers member ownership

```
ClassName(ClassName&& className) noexcept;

ClassName& operator=(ClassName&& className) noexcept;
```

Move functions are called when they are defined and the argument for construction or assignment is an l-value returned by value or an r-value. There are implicit move functions if there are no user-declared copy functions, move functions, or destructor. Then moving is memberwise.

The moved-from object should be left in a valid state so that its destructor does not affect the moved-to object.

The **rule of five** states that the copy constructor, copy assignment operator, move constructor, move assignment operator, and destructor should be all defined or all deleted.

3.5 Inheritance

Inheritance is implemented

```
class Derived : accessmodifier Base {
    ...
}
```

A derived object is constructed in order of inheritance starting with members of the highest base class.

A derived object is constructed

```
Derived(baseParams, derivedParams)
    : Base { baseParams }
    , derivedMembers { derivedParams }
{ }
```

A derived class can only access a constructor for its immediate base class.

Destructors are called in reverse order of inheritance starting with the lowest derived class.

A **protected** member is available to the class, friend classes, and derived classes. Private members are preferred over protected members.

In **public inheritance**, inherited public members are public, inherited protected members are protected, and private members are not inherited. Public inheritance is preferred.

In **protected inheritance**, inherited public and protected members are protected and private members are not inherited.

In **private inheritance**, inherited public and protected members are private and private members are not inherited. Private inheritance is the default.

A method call on a derived object calls the most-derived matching method. A base class member can be accessed within a derived class by a scope qualifier.

A derived object can be static cast into a base object. This can help determine which method to call when the object is passed as an actual parameter.

3.6 Virtual Functions

In **upcasting**, a base class reference or base class pointer can be made from a derived object. Then the reference or pointer can only access base class members of the object.

A **virtual function** resolves to the most-derived matching method, or **override**, of the actual type of the object. The override signature must exactly match the virtual function signature. All overrides are implicitly virtual. Virtual functions should not be called in constructors or destructors.

Polymorphism is the ability to have multiple forms. **Compile-time polymorphism** is polymorphism resolved by the compiler, such as overloading and templates. **Runtime polymorphism** is polymorphism resolved at runtime, such as virtual functions.

A class with a virtual function should have a virtual destructor

```
virtual ~Class() = default;
```

A **pure virtual function**, or **abstract virtual function**, is a virtual function with no body

```
virtualfunctionsignature = 0;
```

An **abstract base class** is a class with a pure virtual function. An abstract base class cannot be instantiated. A derived class that does not implement the virtual function is an abstract base class.

A pure virtual function can be defined outside the class definition. Then a derived class may call the pure virtual function.

An **interface class** is a class with no member variables and only pure virtual functions. The name of an interface begins with `I` by convention.

In **dynamic casting**, or **downcasting**, a base class reference or pointer becomes a derived class reference or pointer

```
Derived* variableName { dynamic_cast<Derived*>(basePointer) };
```

If the object is not of the specified class or a subclass thereof, dynamic casting returns `nullptr` for a pointer and throws `std::bad_cast` for a reference. Dynamic casting only works with public inheritance without virtual functions. Dynamic casting works in only some cases involving virtual base classes.

Static downcasting does not check whether the actual type of the object is the type of the returned pointer. Virtual functions are preferred to downcasting unless the base class is not modifiable, the target member is specific to the derived class, or a pure virtual function would be required and the base class should not be an abstract base class.

The definition of ¡¡ in the base class should call a virtual print function.

3.7 Class Templates

A **class template** is defined

```
template <typename T>
class ClassName {
    ...
}
```

The name of the template is `ClassName<T>`. Within the template, the type name and angular brackets are omitted.

The members of a class template should be defined in the header file.

A **template non-type parameter** is a constant expression of specified type

```
template <typename T, variabletype variableName>
```

4 Data Structures

4.1 Atomic Types

The atomic types are integer `int`, floating point `float`, double-precision floating point `double`, Boolean `bool`, character `char`, and pointer `pointer`. All types are mutable.

The numeric types are `int`, `float`, `double`, `short int` or `short`, `long int` or `long`, and `unsigned int` or `unsigned`. The arithmetic operations are addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`. In arithmetic, `float` dominates. Parentheses `()` override order of operations. `<cmath>` provides exponentiation

```
pow(base, exponent)
```

A `bool` can be `true` (stored as `1`) or `false` (stored as `0`). A `bool` is `false` if and only if it is assigned `false` or `0`. The Boolean operators are conjunction `&&`, disjunction `||`, and negation `!`. The comparison operators are less than `<`, greater than `>`, less than or equal `<=`, greater than or equal `>=`, equal `==`, and not equal `!=`.

A `char` literal is delimited by `''`. A `string` literal is delimited by `""`.

A `pointer` stores a memory address or the falsy `nullptr`. The address-of operator `&` gives the reference of a variable

```
&variableName
```

A `pointer` is declared

```
valuetype *pointerName;
```

Then the dereferenced value is given

```
*pointerName
```

A **compound data type**, or **composite data type**, is a type constructed from the fundamental types. The standard C++ compound types are

- Function
- Array
- Pointer types
  - Pointer to object
  - Pointer to function
- Pointer to member types
  - Pointer to data member
  - Pointer to member function

- Reference types
  - L-value reference
  - R-value reference
- Enumerated types
  - Unscoped enumeration
  - Scoped enumeration
- Class types
  - Struct
  - Class
  - Union

4.2 Value Types

The **type** of an expression is its return value type. The **value category** of an expression is the category in which its resolved value lies.

An **l-value** is an identifiable object or function. An entity is identifiable if it can be differentiated from other entities. An l-value is **modifiable** if it can be modified, and is otherwise **non-modifiable**. An **r-value** is an unidentifiable value.

In assignment, the left expression must be an l-value and the right expression must be an r-value. An l-value implicitly converts to an r-value where an r-value is required.

An **l-value reference** `lvaluetype&` is a reference to a modifiable l-value. An l-value reference can be used in place of a variable for resolution and assignment. An l-value reference must be initialized.

An l-value reference is **bound** to an object **referent** in **reference binding** when it is initialized with the referent. A **dangling reference** is an l-value reference whose referent has been destroyed.

An **l-value reference to a constant value** `const lvaluetype&` is a reference to an l-value or r-value. An l-value reference to a constant value can be used for resolution but not assignment. When possible, l-value references to a constant value are preferred over l-value references.

An l-value reference to a constant value binds to a temporary object of the same type, and therefore supports implicit conversion at initialization. Then the temporary object has the same lifetime as the reference.

If a parameter is an l-value reference, then only modifiable l-values can be passed.

If a parameter is an l-value reference to a constant value, then l-values and r-values can be passed. Passing by value has higher resource complexity, while passing by reference has higher time complexity. An object is cheap to copy if its size is at most two words and it has no setup costs.

Fundamental, enumerated, and string view types should be passed by value. Class, string, array, and vector types should be passed by constant reference. A string view parameter is preferred over a string reference parameter where the function does not pass the string.

A value is **returned by reference** if the return type of the function is a reference. The value must outlive the function. The lifetime of a temporary object cannot be extended by reference between functions. A static local variable persists across function calls and should be avoided as a return value by reference.

References to non-temporary objects passed as actual parameters can be returned by reference. An r-value passed by constant reference can be returned by constant reference.

A value is **returned by address** if the return type is a pointer. Then `nullptr` can be returned. Return by reference is preferred unless it is possible that no object is returned.

4.3 C-Style Arrays

A **collection** is a set of related data.

An array is an ordered collection. Its values in memory are contiguous and equally spaced. Therefore, an out-of-bounds index references the next contiguous value in memory.

An array is **statically allocated** if its size is fixed at compile time. An array is **dynamically allocated** if pointers are used during allocation. All values must be of the same type. A **word** is a unit of data. A statically allocated array is declared

```
arraytype arrayName[arraysize];
```

or

```
arraytype arrayName[] = {values};
```

A C-style string is an array of `char` or `const char` that ends with a null terminator `0x00`.

A C-style string decays to a `const char*`. Then its length is unknown.

In **array overflow**, or **buffer overflow**, data is written past the array bounds. The `>>` operator only writes to non-decayed C-style strings to prevent array overflow. A C-style string can be inputted

```
std::cin.getline(variablename, std::size(variablename));
```

up to 254 characters, discarding the rest.

`std::size(variablename)` gets the length of the array. `<cstring>` gives

- `strlen` gets the logical length of the C-style string (characters before the terminator).
- `strcpy`, `cstrncpy`, or `strcpy_s` overwrites a C-style string with another.

- `strcat` or `strncat` concatenates two C-style strings.
- `strcmp` or `strncmp` compares two C-style strings and returns 0 if equal.

`std::string` is preferred to a non-constant C-style string.
On a C-string literal

- `auto` gives `const char*`
- `auto*` gives `const char*`
- `auto&` gives `const char&[]`

`std::cout` outputs a `const char*` or `char*` actual parameter as a C-string. A character address must be casted to be printed

```
std::cout << static_cast<const void*>(charptr) << std::endl;
```

`constexpr std::string_view` is preferred to `constexpr char*`.

`<iterator>` gives `std::begin(cont)` and `std::end(cont)` for non-decayed C-style arrays. Container libraries give them for container classes.

An iterator is **invalidated** if its element changes address or is destroyed.

4.4 Linked Lists

A **linked data structure** is a set of nodes linked by references or pointers.

A **linked list** is a linear set of nodes where each node contains an element and a pointer to the next node. The **head** is the node referenced externally to enter the list.

The **data field** of a node is the element.

The list object contains only a pointer to the head. A new node becomes the head. In **linked list traversal**, nodes are analyzed proceeding from the head.

5 The Standard Library

5.1 Containers

Containers are defined in the header of their name.

A **sequence** container is indexically ordered.

- `std::vector` is a dynamically sized array with indexing, insertion, and removal. `push_back(value)` appends a value.
- `std::deque` is a dynamically sized array with indexing, insertion, and removal. `push_back(value)` appends a value. `push_front(value)` adds a value to the front.
- `std::list` is a doubly linked list with head and tail.

An **associative** container is relationally ordered. By default elements are compared by `<`.

- `std::set` has unique elements.
- `std::multiset` has nonunique elements.
- `std::map` maps a key to exactly one value. The keys are ordered.
- `std::multimap` maps a key to at least one value.

A **container adapter** is a modified sequence container.

- `std::stack` is a deque by default with last in, first out operation.
- `std::queue` is a deque by default with first in, first out operation.
- `std::priority_queue` is a sorted queue.

5.2 Iterators

An iterator points to an element of a container. Iterators are defined per container.

- `*` gets the element.
- `++` points to the next element.
- `==` and `!=` compare elements.
- `=` points to a new position.

Containers have member functions that return iterators

- `begin()` the first element
- `end()` the element after the last element

- `cbegin()` constant first element
- `cend()` constant element after the last element

`container::iterator` can read and write. `container::const_iterator` can read.

The elements of a map are `std::pair` from `std::make_pair(val1, val2)` with `pair.first` and `pair.second`.

5.3 Algorithms

Algorithms are defined in `<algorithm>`.

- `std::min_element(start, end)` points to the minimum element.
- `std::max_element(start, end` points to the maximum element.
- `std::find(start, end, element)` points to the target element or the end iterator if not found.
- `std::sort(start, end, operator)` sorts a container according to the operator, defaulting to ascending order.
- `std::reverse(start, end)` reverses a container.

`std::iota(begin, end, initial)` fills a container with increasing consecutive integers.

5.4 Input/Output

`<iostream>` allows input and output. A **stream** is a byte sequence. An **input** stream reads data. An **output** stream writes data.

`std::istream` allows extraction `>>`. `std::ostream` allows insertion `<<`. `std::iostream` is bidirectional.

A **standard** stream interacts with the environment of the program.

- `std::cin` standard input, usually keyboard
- `std::cout` standard output, usually monitor
- `std::cerr` standard error, usually monitor, unbuffered
- `std::clog` standard error, usually monitor, buffered

An **unbuffered** output is handled immediately. A **buffered** output is stored and handled in blocks.

A **manipulator** is an object that modifies a stream through extraction or insertion

```
std::cin >> manipulator >> data;
std::cout << manipulator << data;
```

`<iomanip>` provides manipulators

- `std::endl` insert newline and flush buffered output
- `std::setw(num)` limit number of characters read at once

Extraction skips whitespace by default. `std::cin.get(char)` reads a character and does not skip whitespace. `std::cin.get(str, max)` reads a C-style string with maximum length. `get` terminates on newline and leaves the newline in the stream.

`std::cin.getline(str, max)` reads a C-style string terminates on newline and extracts and discards the newline from the stream. `std::cin.gcount()` returns the number of characters extracted by `getline`.

`<string>` provides `std::getline(std::cin, stdstr)` to read into `std::string`.

- `std::cin.ignore(count)` discards the first number of characters, default one
- `std::cin.peek()` reads but does not extract a character
- `std::unget()` puts the last extracted character into the front of the stream
- `std::putback(char)` puts a character into the front of the stream

A **flag** in `std::ios` toggles an output formatting option. `std::cout.setf(flag1 | ... | flag2)` activates flags. `unsetf` deactivates flags.

A **format group** is a set of related flags. Some flags are mutually exclusive. If mutually exclusive flags are set, one has precedence. `setf(flag, fgroup)` deactivates all other flags in the group.

A manipulator in `std` deactivates appropriate other flags by default. A member function is defined in the class `std::ios_base` or a derivative.

- `std::ios::boolalpha` make `bool` print `"true"` or `"false"` instead of `0` or `1`
  - `std::boolalpha`
  - `std::noboolalpha`
- `std::ios::showpos` prefix positive numbers with `+`
  - `std::showpos`
  - `std::noshowpos`
- `std::ios::uppercase` use uppercase characters, for example in scientific notation
  - `std::uppercase`
  - `std::nouppercase`

- `std::ios::basefield` integer display
  - `std::ios::dec` decimal base, default
    - `std::dec`
  - `std::ios::hex` hexadecimal base
    - `std::hex`
  - `std::ios::oct` octal base
    - `std::oct`
  - None: deduce base from leading characters of value
- `std::ios::floatfield` float display
  - `std::ios::fixed` decimal notation without trailing zeros
    - `std::fixed`
  - `std::ios::scientific` scientific notation
    - `std::scientific`
  - `std::ios::showpoint` decimal with trailing zeros, always shows decimal point
    - `std::showpoint`
    - `std::noshowpoint`
  - `std::setprecision(int)` if fixed or scientific, sets decimal places; otherwise sets significant digits; with rounding
    - `std::ios_base::precision()` returns precision
    - `std::ios_base::precision(int)` sets precision and returns old precision
  - None: fixed for few digits, otherwise scientific
- `std::ios::adjustfield` justification
  - `std::ios::internal` left justify sign, right justify value
    - `std::internal`
  - `std::ios::left` left justify
    - `std::left`
  - `std::ios::right` right justify
    - `std::right`
  - `std::setfill(char)` set fill character instead of space
    - `std::basic_ostream::fill()` returns fill character
    - `std::basic_ostream::fill(char)` sets fill character and returns old fill character
  - `std::setw(int)` set field width; must be done before others in the format group; only affects current statement

    - `std::ios_base::width()` returns field width
    - `std::ios_base::width(int)` sets field width and returns old field width

`<sstream>` provides string streams.

- `std::istringstream` reads normal character strings.
- `std::ostringstream` writes normal character strings.
- `std::stringstream` reads and writes normal character strings.
- `std::wistringstream` reads wide character strings.
- `std::wostringstream` writes wide character strings.
- `std::wstringstream` reads and writes wide character strings.

`<<` appends data. `streamName.str(stringvalue)` overwrites data. `>>` gets data value by value. `streamName.str()` gets all data. `streamName.clear()` resets error flags.

`<fstream>` allows file input and output with `ifstream`, `ofstream`, and `fstream`. `std::fstream { filename, mode1 | ... | moden }` opens a file. Data can be extracted and inserted. `put(char)` inputs a character. `ifstream` returns `0` on end of file.

The file closes on destruction or `close()`. The buffer is **flushed** when stored data is written to the file. The buffer is flushed on file close, `ostream::flush()`, or manipulators `std::flush` and `std::endl`. Flushing is inefficient.

Mode determines allowed file interactions

- `std::ios::app` append
- `std::ios::ate` open at end of file
- `std::ios::binary` binary data
- `std::ios::in` read, default for `ifstream`; may fail if file does not exist
- `std::ios::out` write, default for `ofstream`
- `std::ios::trunc` erase and open

A file can be reopened with `open()`.

5.5 Memory Management

A **smart pointer** is a composition class on a pointer that deallocates at destruction.

`std::unique_ptr<>` in `<memory>` is a smart pointer to an object to whom it is the sole smart pointer. Move semantics replace copy semantics

```
res2 = std::move(res1);
```

A unique pointer implements `operator*` and `operator->`, and is false if and only if it manages no object.

Container classes are preferred over smart pointers to arrays.

`std::make_unique<>(args)` is preferred over `new` in a unique pointer. A unique pointer should be returned only by value.

A unique pointer should be passed to a function as

```
func(std::move(ptr));
```

or

```
func(ptr.get());
```

`std::shared_ptr<>` in `<memory>` is a smart pointer to an object that may be pointed to by other smart pointers. The object is deallocated on destruction of the last shared pointer. Further shared pointers must use copy semantics.

`std::make_shared<>(args)` is preferred over `new` in a shared pointer. A shared pointer can be constructed from a unique pointer r-value.

A GoogleTest

GoogleTest is a testing framework whose tests are independent, repeatable, organized, portable, reusable, informative, convenient, and fast.

An **assertion** checks a condition, with result success, nonfatal failure, or fatal failure. A fatal failure ends the current function. A **test** uses assertions to verify functionality. A test succeeds if and only if no failures occur, and otherwise fails. A **test suite** is a set of structurally related tests. A **test fixture** is a class with tests and common objects and subroutines. A **test program** is a set of test suites.

An `ASSERT_*` function is an assertion that fails fatally. An `EXPECT_*` function is an assertion that fails nonfatally. A custom failure message can be set by `<<`.

A test is defined

```
TEST(TestSuiteName, TestName) {
    statements
}
```