C++ Programming Language

The code in this guide was tested against GCC 8.2.1 and should work with any C++ compiler that fully implements the C++17 standard.

Source Code Comments

- There are two types of source code comments:
  - **Line comments** start with two forward slashes // and continue until the end of the line of text.
  - **Block comments** can contain multiple lines of text, start with /*, and end with */.
- Prior to compilation, comments are replaced with a single space character.

Example:

```
std::cout << "Hello, world!\n";  // Say hello!
/*
This next line of code prints
a hello world message to standard out.
*/
std::cout << "Hello, world!\n";
```

The Preprocessor

- Preprocessor directives can be embedded in C++ source code. They comprise a text processing language (separate from the C++ language itself) used to modify C++ source file text prior to compilation.
- **The preprocessor supports several categories of functionality:** File inclusion, macro definition and expansion, conditional source code inclusion, and diagnostic messaging.
- **The following standard preprocessor directives are supported:** #include, #define, #undef, #if, #ifdef, #ifndef, #else, #elif, #endif, #line, #pragma, #error.
  - Additional nonstandard directives (e.g., #warning) may also be supported.
    - **The C++ standard requires that the preprocessor predefines the following macros:** __cplusplus, __DATE__, __FILE__, __LINE__, __STDC_HOSTED__, __STDCPP_DEFAULT_NEW_ALIGNMENT__, __TIME__.
  - **The following macros may be defined, conditionally, depending on implementation-specific details:** __STDC__, __STDC_MB_MIGHT_NEQ_WC__, __STDC_VERSION__, __STDC_ISO_10646__, __STDCPP_STRICT_POINTER_SAFETY__, __STDCPP_THREADS__.

Line Continuation

When a backslash character \ is immediately followed by a newline character, both characters are deleted before compilation. This effectively allows the current line to be continued on the next line by ending the current line with a backslash.

Keywords

alignas	continue	friend	register*	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

*Unused, but reserved for possible future use

Fundamental Types

- Alternate ways to specify a given type (e.g., signed short int) are not listed.
- The **int keyword** may be omitted when modifiers (e.g., signed, unsigned, short, and long) are specified.
- The C++ standard does not strictly specify the widths of the fundamental types. The widths shown below are typical for modern systems. Some platforms may have different widths/ranges.
- Use the **std::is_fundamental** type trait to determine if a type is fundamental.
- Use **std::numeric_limits** to determine ranges.

FUNDAMENTAL TYPES

Type	Width (Bits)	Range
bool	8	0,1 (false, true)
char	8	-128…127
signed char	8	-128…127
unsigned char	8	0…255
short int	16	-32768…32767
unsigned short int	16	0…65535
int	32	-2147483648…2147483647
unsigned int	32	0…4294967295
long int	64	-9223372036854775808…9223372036854775807
unsigned long int	64	0…18446744073709551615
long long int	64	-9223372036854775808…9223372036854775807
unsigned long long int	64	0…18446744073709551615
float	32	-3.40282e+38…3.40282e+38
double	64	-1.79769e+308…1.79769e+308
long double	128	-1.18973e+4932…1.18973e+4932
wchar_t	32	-2147483648…2147483647
char16_t	16	0…65535
char32_t	32	0…4294967295
void	N/A	N/A
std::nullptr_t	64	N/A

Enumerations

A type whose values are restricted to a set of named constants. The named constants themselves are of an integral type referred to as the underlying type.

```
#include <iostream>

// unscoped enum (deprecated)
enum Color : char { Red, Green, Blue };

// scoped enum
enum class Role : char { Mom=1, Dad=3, Son, Daughter };

int main() {
    Role role{Role::Son};

    std::cout << "role type = [" << typeid(Role).name() << "]\n";
    std::cout << "role size = [" << sizeof(role) << "]\n";
    std::cout << "role value = [" << static_cast<int>(role) << "]\n";

    return 0;
}
```

Output:

```
role type = [4Role]
role size = [1]
role value = [4]
```

Incomplete Types

Types for which objects cannot be declared, since the compiler cannot determine the size of the object. Void, forward declarations of structs, classes, unions, arrays of unknown size, and arrays whose elements are of an incomplete type are incomplete types.

Storage Class Specifiers

Used in declarations to indicate storage duration and linkage properties

- **static:** Indicates the object is allocated when the program starts and deallocated when the program terminates.
  - When used to declare a **class member**, it indicates the member is a class member instead of a class instance member.
  - When used in a declaration at **namespace scope**, it indicates **internal linkage**.
- **extern:** Indicates that the object being declared is not being defined but only declared and that it is has external linkage.
  - Objects declared as extern must be defined (once) in a separate definition with **external linkage**, or a **linker error** will occur.
- **thread_local:** Indicates that the object being declared is allocated (for each thread) when the thread is created and is deallocated when the thread is destroyed. Each thread has its own instance of the object.

Objects

Instances of types

- **Variable:** A named object.
- **Unnamed objects (temporary objects):** Created during implicit type conversion and when returning a value from a function.
- When an object is created, a region of memory must be reserved to store the object's state information. The amount of memory needed to store an object's state is known as the object's **size**.
  - The size of an object or type can be determined using the **sizeof operator**.
- An object of a type having at least one virtual function is known as a **polymorphic object**.
  - Polymorphic objects should almost always have virtual destructors.
- Objects have **alignment requirements** that impose restrictions on an object's memory address.

Object Lifetime

- Significant events in an object's life cycle:
  - Reservation of storage (memory) for the object
  - Initialization/construction (may be a trivial constructor that performs no action)
  - Object in live/normal state
  - Destruction (may be a trivial destructor that performs no action)
  - Release of storage
- During construction and destruction, member functions including virtual member functions may be called.
  - When a virtual function is called during these times, it is always called on the object being constructed or destructed. Care must be taken when calling member functions from within constructors and destructors, since the object may be in a state that violates class invariants, which could lead to undefined behavior.

Attributes

Attributes (*attribute-specifier-seq*$_{opt}$) can be used to specify information about types, variables, names, blocks, and translation units. Various tools in the toolchain may implement their own implementation-specific attributes. However, the standard does define the following standard attributes: **noreturn**, **carries_dependency**, **deprecated**, **fallthrough**, **nodiscard**, **maybe_unused**.

Examples:

```
[[ noreturn ]] void onError() { throw std::runtime_error("Error."); }
[[ maybe_unused, deprecated ]] void oldImplementation();
```

Object Initialization

The act of assigning an initial state to an object during creation

- Generally, all objects should be initialized, either implicitly (via a default constructor) or explicitly. Objects can be explicitly initialized in three ways:
  - Parenthesized expression-list, in which the expression-list is interpreted as arguments to a constructor

EX:
```
std::complex<double> point(1,0);
```

  - Assignment expression

EX:
```
std::complex<double> point = 1;
```

  - Initialization list

EX:
```
std::complex<double> point = {1,0};
```

Compound Types

Defined in terms of other types. Arrays, functions, pointers to objects or functions, pointers to nonstatic class data or function members, references, classes, unions, and enumerations are compound types. Use the **std::is_compound** trait type to determine if a type is a compound type.

```
#include <string>

int main(int argc, char **argv) {
    int intArray[10]; // array of 10 integers
    int * pInt = nullptr; // pointer to an integer
    int ** ppInt = nullptr; // pointer to a pointer to an integer
    // function pointer that takes a double and returns an integer
    int (*fp)(double) = nullptr;
    // pointer to char that points to an array of 6 characters
    // containing the null-terminated string "Scott"
    char const * name = "Scott";
    // union
    union String
    {
        std::string str;
        std::wstring wstr;
    };

    return 0;
}
```

Type Aliases

Type aliases are declared using the following syntax:

using *identifier attribute-specifier-seq*$_{opt}$ = *type-id*;

Example:

```
using Age = unsigned char;
using FirstName = std::string;
using Date = struct {
   unsigned char day;
   unsigned char month;
   unsigned int year;
};
```

Alias template declarations are declared using the following syntax:

template *<template-parameter-list>* **using** *identifier attribute-specifier-seq*$_{opt}$ = *type-id*;

Example:

```
template <typename T> using MyVector = std::vector<T, MyAllocator<T>>;
```

CV Qualifiers

CV (const and volatile) type qualifiers are used in declarations to indicate if objects of the declared type can be modified or if they may be modified outside the current thread of execution.

- **const:** Objects or subobjects of objects declared as const cannot be modified except by using **const_cast** or **mutable**.
- **volatile:** Objects or subobjects of objects declared as volatile may have their value changed outside the current thread of execution (prevents the compiler from performing certain optimizations).

Expressions

A sequence of operators and operands that specify a computation

- The evaluation of an expression may cause side effects and may result in a value (e.g., the result of the expression 1+1 is the value 2).
  - Expressions have a type and a value category. The C++ standard defined the following value categories:
  - **prvalue (pure rvalue):** Does not have identity; can be moved from
  - **xvalue (expiring value):** Has identity; can be moved from
  - **lvalue (can appear on the left of an assignment operator):** Has identity; cannot be moved from
  - **glvalue (generalized lvalue):** Has identity; general category that applies to xvalue and lvalue
  - **rvalue (can appear on the right of an assignment operator):** Can be moved from; general category that applies to prvalue and xvalue; address may not be taken; can't be assigned to; can be used to initialize a const lvalue reference

Literals

Constant values embedded into the source code

- Literal prefixes and suffixes are used to indicate the type of literal value.
  - The type of integer literals is assumed to be **int** if no suffix is specified.
  - The type of floating-point literals is assumed to be **double** if no suffix is specified.

LITERAL PREFIXES & SUFFIXES

Category	Type	Prefix	Suffix	Example
bool	bool			true, false
character	char	'		'a'
	char16_t	u'		u'a'
	char32_t	U'		U'a'
	wchar_t	L'		L'a'
integer	int			42
	int (octal)			052
	int (hex)			0x2a
	unsigned	0		42U, 42u, 0x21u
	long	0X, 0x	L, l	42L, 42l, 0x21L
	long long	U, u	LL, ll	42LL, 42ll, 0x21LL
	unsigned long		UL, ul	42UL, 42ul, 0x21UL
	unsigned long long		ULL, ull	42ULL, 42ull, 0x21ULL
floating-point	float		F, f	42.0F, 42.f, 1.0e10f
	double			42.0, 42., 1.0e10
	long double		L, l	42.0L, 42.l, 1.0e10L
string	string	"		"foo"
	raw string	R"		R"(\n)"
	UTF-8 string	u8"		u8"foo", u8R"foo\n"
	UTF-16 string	u"		u"foo", uR"foo\n"
	UTF-32 string	U"		U"foo", UR"foo\n"
	wchar_t string	L"		L"foo", LR"foo\n"

- The **octal (0)** and **hex (0x)** prefixes can be used with any literal integer suffix.
- The **raw string prefix (R)** can be used with any string literal prefix to disable escape sequence processing.
- Character and string literals may contain **escape sequences** that are used to represent commonly used nonprintable characters.

ESCAPE SEQUENCES

Name	Esc. Seq.
new line	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert/beep	\a
backslash	\\
single quote	\'
double quote	\"
any char value (octal)	\000
any char value (hex)	\xhhh

User-defined literal suffixes are supported, which allow literals to produce values of user-defined types.

Example of a user-defined literal:

```
#include <iostream>

using meters = double;

constexpr meters operator"" _feet (long double n) {
    return 0.3048 * n;
}

constexpr meters operator"" _inches (long double n) {
    return 0.0254 * n;
}

int main() {
    std::cout << "100 feet = [" << 100.0_feet << "] meters.\n";
    std::cout << "100 inches = [" << 100.0_inches << "] meters.\n";

    return 0;
}
```

Output:

```
100 feet = [30.48] meters.
100 inches = [2.54] meters.
```

Classes

User-defined types that represent concepts

- Classes encapsulate state and behavior, which are defined by the member data and member functions.
- External accessibility of member data and functions can be controlled via access specifiers (**private**, **protected**, and **public**).
- User-defined classes can be made to behave just like the built-in fundamental types due to the operator overloading capabilities provided in C++.

There are three different kinds of classes:

- **Class-key:** Specifies which kind of class is being declared.
- **Struct:** Similar to a **class**, except the default access specifier is **public** instead of **private**.
- **Union:** A class in which all of the nonstatic data members occupy the same memory location, and therefore only one member of a union can be active/valid at any given time. The active/valid member of a union is the member most recently assigned a value. Accessing the value of an inactive/invalid member of a union is undefined behavior.
  - Unions have the following limitations:
    - » A union may not have virtual functions.
    - » A union may not have a base class.
    - » A union may not be used as a base class.
    - » A union may not contain a nonstatic data member of reference type.
    - » If any nonstatic data members of a union have nontrivial special member functions, the corresponding member function of the union must be user-provided or it will be deleted for the union.
  - A union is declared as:
    union { *member-specification* };
    - » This is called an anonymous union. Anonymous unions have additional limitations:
      - ♦ All members must be public (**private** and **protected** access specifiers are not allowed).
      - ♦ If declared in a named namespace or global namespace, the union must be declared to have **static** storage class.

Access Specifiers

Public, **protected**, and **private** access specifiers are used to limit or grant access to member data and functions.

- This is done is two ways:
  - When used in the **member-specification** of a class, struct, or union, the access specifier indicates the accessibility of members that follow in the class declaration.
  - When used in the **base-specifier** of a derived class, the access specifier indicates the accessibility the derived class has to the members of the base class.
- The default access level for members of a type declared as a class is **private**.
- The default access level for members of a type declared as a struct or union is **public**.

Statements

An ordered sequence of one or more instructions that the program may execute

- C++ supports expression statements, null statements, compound statements (blocks), selection statements, iteration statements (loops), jump statements, declaration statements, and try-block statements.
- Statements may be given labels for use with **goto** and **switch** statements.

Expression Statements

attribute-specifier-seq$_{opt}$ *expression*;
EX: Assignment statements and function calls

Null Statements

;
Can be thought of as an expression statement in which no expression is specified. Used to provide an empty **init-statement** for a for loop or as a statement to associate a label with at the end of a compound statement.

Compound Statements (Blocks)

attribute-specifier-seq$_{opt}$ { *statement-seq*$_{opt}$ }
A sequence of statements enclosed by paired braces. Variables defined inside a compound statement are destroyed at the end of the block.

Selection Statements

attribute-specifier-seq$_{opt}$ **if** *constexpr*$_{opt}$ (*init-statement*$_{opt}$ *condition*) *statement*
attribute-specifier-seq$_{opt}$ **if** *constexpr*$_{opt}$ (*init-statement*$_{opt}$ *condition*) *statement* **else** *statement*
attribute-specifier-seq$_{opt}$ **switch** (*init-statement*$_{opt}$ *condition*) *statement*

Iteration Statements (Loops)

attribute-specifier-seq$_{opt}$ **for** (*for-range-decl* : *for-range-init*) *statement*
attribute-specifier-seq$_{opt}$ **for** (*init-statement*$_{opt}$ *condition*$_{opt}$; *expression*$_{opt}$) *statement*
attribute-specifier-seq$_{opt}$ **while** *constexpr*$_{opt}$ (*init-statement*$_{opt}$ *condition*$_{opt}$; *expression*$_{opt}$) *statement*
attribute-specifier-seq$_{opt}$ **do** *statement* **while** (*expression*);

QuickStudy

Jump Statements

break;
continue;
return *expression*;
return *initializer-list*;
goto *identifier*;

Declaration Statements

simple-declaration:
attribute-specifier-seq$_{opt}$ *decl-specifier-seq init-declarator-list*$_{opt}$;
decl-specifier-seq:
decl-specifier attribute-specifier-seq$_{opt}$
decl-specifier decl-specifier-seq
decl-specifier:
storage-class-specifier
defining-type-specifier
function-specifier
friend
typedef
constexpr
inline
init-declarator-list:
init-declarator
init-declarator-list, init-declarator
init-declarator:
declarator initializer$_{opt}$
declarator:
ptr-declarator
noptr-declarator parameters-and-qualifier trailing-return-type
ptr-declarator:
noptr-declarator
ptr-operator ptr-declarator
noptr-declarator:
declarator-id attribute-specifier-seq$_{opt}$
noptr-declarator parameters-and-qualifiers
noptr-declarator [constant-expression$_{opt}$ *] attribute-specifier-seq*$_{opt}$
(ptr-declarator)
parameters-and-qualifiers:
(parameter-declaration-clause) cv-qualifier-seq$_{opt}$
ref-qualifier$_{opt}$ *noexcept-specifier*$_{opt}$ *attribute-specifier-seq*$_{opt}$
trailing-return-type:
-> type-id
ptr-operator:
** attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$
& attribute-specifier-seq$_{opt}$
&& att-seq$_{opt}$
*nested-name-specifier * attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$
cv-qualifier-seq:
cv-qualifier cv-qualifier-seq$_{opt}$
cv-qualifier:
const
volatile
ref-qualifier:
&
&&
declarator-id:
...$_{opt}$ *id-expression*

Try-Block Statements

try-block:
attribute-specifier-seq$_{opt}$ **try** *compound-statement handler-seq*
function-try-block:
try ctor-initializer$_{opt}$ *compound-statement handler-seq*
handler-seq:
handler handler-seq$_{opt}$
handler:
catch *(exception-declaration) compound-statement*
exception-declaration:
attribute-specifier-seq$_{opt}$ *type-specifier-seq declarator*
attribute-specifier-seq$_{opt}$ *type-specifier-seq abstract-declarator*$_{opt}$
...

Labeled Statements

attribute-specifier-seq$_{opt}$ *identifier* : *statement*
attribute-specifier-seq$_{opt}$ **case** *constant-expression* : *statement*
attribute-specifier-seq$_{opt}$ **default** : *statement*

Functions

Named blocks of code that can be called

- Function parameters can be passed to the function as inputs. Functions may return void (nothing), or some other object, according to the declaration.
- All function parameters (including references and pointers) are passed by value.

Basic function declaration syntax:

return-type function-name (*parameter-list*) { *function-body* }

Example:

```
int add(int a, int b)
{
    return a + b;
}
```

Main Function

- All C++ programs must contain a function named **main**, which is the function at which the program starts executing user-defined code.
- The main function is called just after nonlocal objects with static storage duration are initialized.
- The main function can be defined in one of two ways:
  - int main() { function-*body* }
  - int main(int argc, char *argv[]) { function-*body* }

Example:

```
#include <iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Operators

Precedence Group	Associativity	Operator Description	Operator	Alias
1	N/A	Scope resolution	::	
2	Left to right	Array subscript Function call Member selection Postfix decrement Postfix increment Constant type conversion Dynamic type conversion Reinterpret type conversion Static type conversion Type name	[ ] ( ) . or ->* -- ++ const_cast dynamic_cast reinterpret_cast static_cast typeid	<: :>
3	Right to left	Address-of Cast Create object Destroy object Indirection Logical not One's complement Prefix decrement Prefix increment Unary negation Unary plus Size of an object or type	& ( ) new delete * ! ~ -- ++ - + sizeof	not compl
4	Left to right	Pointer-to-member	.* or ->*	
5	Left to right	Division Modulus Multiplication	/ % *	
6	Left to right	Addition Subtraction	+ -	
7	Left to right	Left shift Right shift	<< >>	
8	Left to right	Greater than Greater than or equal to Less than Less than or equal to	> >= < <=	
9	Left to right	Equality Inequality	== !=	not_eq
10	Left to right	Bitwise AND	&	bitand
11	Left to right	Bitwise exclusive OR	^	xor
12	Left to right	Bitwise inclusive OR	|	bitor
13	Left to right	Logical AND	&&	and
14	Left to right	Logical OR	||	or
15	Right to left	Conditional	?:	
16	Right to left	Addition assignment Assignment Division assignment Bitwise ADD assignment Bitwise exclusive OR assignment Bitwise inclusive OR assignment Left-sift assignment Modulus assignment Multiplication assignment Right-shift assignment Subtraction assignment	+= = /= &= ^= |= <<= >>= *= >>= -=	 and_eq xor_eq or_eq
17	Right to left	Throw expression	throw	
18	Left to right	Comma	,	

Namespaces

Named blocks that prevent accidental name duplication/collision in large projects

Note: Multiple namespace blocks that have the same name are allowed.

Name Hiding

Since the name lookup process stops looking for candidates once a nonempty set of candidates is found in a particular scope, names can be hidden by declarations in nested scopes. This can result in unexpected behavior and misunderstanding, so it is best avoided, if possible.

Polymorphism

A feature of a computer language that allows a single interface to be used to access functionality having multiple implementations, where the specific implementation invoked is based on the type or types being acted on

- C++ supports the three kinds of polymorphism: Ad hoc polymorphism, parametric polymorphism, and subtype polymorphism.
  - **Ad hoc polymorphism:** Supported via function and operator overloading; an example of static polymorphism (compile-time polymorphism).
  - **Parametric polymorphism:** Supported via templates; an example of static polymorphism (compile-time polymorphism).
  - **Subtype polymorphism:** Supported via interfaces (abstract classes) and derived concrete classes; an example of dynamic polymorphism (run-time polymorphism).

Name Resolution & Scopes

- As a program is compiled, the compiler must determine the declaration associated with each variable name, function name, class name, and namespace name it encounters. This process is known as **name resolution**.
- Since C++ allows multiple functions to have the same name in the same scope, name resolution of functions and function templates may take two steps: **name lookup** (which may result in multiple candidates being found) followed by **overload resolution**.
  - **Overload resolution:** Process by which the compiler determines which one of multiple candidate functions is the best match, based on the number and types of the function parameters. After the compiler selects the best candidate, implicit type conversion may then be employed to convert arguments to the types that the selected function requires.
  - Name resolution of variable names, class names, and namespace names only requires a **name lookup**, which must result in a single candidate being found.
- There are two kinds of name lookups:
  - **Qualified:** Performed when a name appears on the right-hand side of the scope resolution operator :: .
    - » In these cases, the identifier on the left of the scope resolution operator can be a class name, struct name, enum name, or namespace name. If there is no identifier to the left of the scope resolution operator, it indicates that the name is in the global namespace. Explicitly specifying that the name is in the global namespace is required to refer to names in the global namespace that are *hidden* by another declaration.
  - **Unqualified:** Performed when a name appears without a preceding scope resolution operator. In these cases, the compiler will try to resolve the name by searching the following scopes, stopping when it finds the first match:
    - » **Block scope:** Declarations in a compound statement

- » **Local scope (a kind of block scope):** Declarations in a function or lambda body
- » **Statement scope:** Declarations in **if**, **for**, **while**, and **switch** statements
- » **Function parameter scope**
- » **Function scope:** Labels are at function scope and are valid anywhere inside the function, not just following declarations
- » **Namespace scope:** Declarations made inside the current namespace
- » **Class scope:** Declarations made inside the current class declaration
- » **Enumeration scope:** Declarations made inside the current scoped enum declaration
- » **Template parameter scope**
- » **Global scope:** Global declarations outside any class, function, or namespace (file scope)

- Unqualified function names (including overloaded operators) can be looked up using **argument-dependent lookup** (Koenig lookup).
  - This allows unqualified functions to be used to call functions in a namespace, as long as one or more of the function argument types are in the same namespace.

Example of argument-dependent lookup:

```
#include <iostream>

namespace exmpl {
  class MyClass {};
  void myFunc( MyClass const & ) {};
}

int main() {
   // myFunc resolved via argument-dependent lookup (ADL)
   // since it is the same namespace as the argument
   myFunc( exmpl::MyClass() );
   // In the following statement, the << operator is resolved
   // via ADL.  Since one of its arguments (std::cout) is in
   // the namespace 'std', the compiler is able to resolve
   // the operator to std::operator<<(std::ostream, const char *)
   std::cout << "Goodbye\n";
   return 0;
}
```

Lambda Expressions

Used to create function objects (functors)

- The return type of a lambda expression can be either specified explicitly via the **trailing-return-type** or deduced by the compiler from the **return statement**.
- The type of a lambda expression is a unique, unnamed, nonunion class type called the **closure type**, which overrides the function call operator.
  - The closure type will have a nonstatic data member for each object that the lambda expression captures by value.
- In the example that follows, a lambda function is used as a predicate in a call to the standard template library algorithm **std::count_if()**.
- The capture clause is empty, meaning that no variables are being captured.
- The lambda expression has a single parameter in its parameter list: an **int**.
  - An int must be used in this example because int is the value type of the iterators passed to count_if().
  - The return type of the lambda expression is deduced (as bool) from its return statement.

Example:

```
int countPositiveIntegers(std::vector<int> const & vec) {
    return std::count_if(vec.begin(), vec.end(),
                  [](int i){ return i > 0;} //lambda function
        );
}
```

The *lambda-introducer* [*lambda-capture*$_{opt}$] specifies the variables that are "captured" by the lambda expression. The *lambda-capture* may be empty or may contain one or more captures. Each capture may be one of the following:

Capture	Meaning
[&]	Capture any used variables by reference
[=]	Capture any used variables by value/copy
[*identifier*]	Capture variable by value
[*identifier*...]	Capture parameter pack by value
[& *identifier*]	Capture variable by reference
[& *identifier*...]	Capture parameter pack by reference
[& *identifier*= *initializer*]	Capture variable by reference and initialization
[this]	Capture current object by reference
[*this]	Capture current object by value

Copy/Move

Copy/Move Semantics

- Objects can be copied by using a **copy constructor** or by using a **copy assignment operator**.
  - For large objects, a copy can be an expensive operation. Therefore, the compiler will do its best to eliminate unnecessary copies in a process called **copy elison**.
- There are many situations in which the original object is not needed after the copy is made. In such cases, the object can be moved instead of copied.
  - Temporary objects are **rvalues** and can be implicitly moved from.
  - **Lvalues** can only be moved from explicitly by using the **std::move() function**, which essentially implements a cast to an **rvalue**.
  - Objects are moved by using a **move constructor** or a **move assignment operator**.
  - In order to gain a performance benefit from moving, an object must have member pointers to other dynamically allocated objects.

Copy/Move Elision

- Under certain circumstances, the compiler can automatically remove unnecessary object copy/move operations in order to make the resulting code more efficient. This removal is called **copy/move elision**. A copy/move elision is usually performed in the following situations:
  - When returning an object from a function
    - » This is known as **return value optimization**.
  - When a temporary object is passed to a function by value
  - When throwing an exception object by value
  - When catching an exception object by value

Special Member Functions

There are six special member functions that the compiler can declare and define automatically, in some cases. Special member functions should not be explicitly defined, unless necessary.

Special Member Function	Form for Class T	Default Implementation	Conditions Required for the Compiler to Implicitly Define
Default constructor	T::T();	Does nothing	No explicitly declared constructors
Copy constructor	T::T(const T&);	Copies all members	No explicitly declared move constructor or move assignment
Copy assignment	T& T::operator=(const T&);	Copies all members	No explicitly declared move constructor or move assignment
Move constructor	T::T(T&&);	Moves all members	No explicitly declared copy constructor, copy assignment, move assignment, or destructor
Move assignment	T& T::operator=(T&&);	Moves all members	No explicitly declared copy constructor, copy assignment, move constructor, or destructor
Destructor	T::~T();	Does nothing	No explicitly declared destructor

The **default** and **delete** keywords can be used when declaring special member functions (e.g., default constructor, destructor, copy constructor, copy assignment, move constructor, and move assignment).

```
class Complex
{
public:
    Complex() = default; // default constructor
    Complex(Complex const&) = default; // copy constructor
    Complex& operator=(Complex&) = default; // copy assignment
    Complex(Complex&&) = delete; // move constructor
    Complex& operator=(Complex&&) = delete; // move assignment
    ~Complex() = default;
private:
    double r;
    double i;
};
```

Function Overloading

- A function's **signature** is determined by its name and the number, order, and type of its parameters. A function's **return type** is not part of its signature.
- **Function overloading** is accomplished by declaring more than one function in the same scope having the same name but different signatures.
  - Overloaded functions must differ by the number and/or types of parameters, or in the case of member functions, by their constness.
  - Overloaded functions are allowed to have different return types; however, functions cannot be overloaded based on return type. Declaring multiple functions with the same name in the same scope that differ only in return type is not allowed.
- Overloaded functions must be in the same **scope**. A member function in a base class cannot be overloaded by adding a member function to one of its derived classes, since the two functions are in different class scopes.

Example:

```
#include <vector>

class Room {};

// type alias
using RoomId = int;

class Building
{
public:
    // overloading based on number and/or
    // types of parameters
    Room& addRoom();
    Room& addRoom(Room const& room);

    // overloading based on constness
    Room& getRoom(RoomId roomId);
    Room const& getRoom(RoomId roomId) const;
private:
    std::vector<Room> rooms;
};
```

Type Conversion

- C++ is a strongly/strictly typed language; however, C++ gives users great power to control (and even dictate) **type transformations**.
  - Type transformations may be performed automatically and silently by the compiler (**implicit type conversion**) without warning the user (unless such warnings are enabled), leading some to mistakenly believe that C++ is weakly typed.
  - There are several mechanisms that allow users to control how and when types can be transformed.
  - **Explicit type conversion** allows users to explicitly convert an object of one type to an object of another type.
    - » Use of explicit casting should be avoided, when possible. Excessive use of casting should be considered a **code smell** and usually indicates a design problem.
    - » The following explicit type conversion mechanisms are supported:
      - ♦ **C-style cast:** (
      - ♦ **Functional cast** (alternate syntax for C-style cast)
      - ♦ **const_cast:** For converting a const type to a mutable type (removes constness)
      - ♦ **static_cast:** For explicitly specifying an implicit conversion (or reversing such a conversion); able to utilize **explicit** constructors to perform conversion
      - ♦ **dynamic_cast:** For safe down-casting from a base class pointer/reference to a derived class pointer/reference
      - ♦ **reinterpret_cast:** For low-level casts between unrelated types (can be dangerous and should be avoided)
  - C++ allows implicit (**automatic**) type conversion between different types of fundamental and user-defined types. Implicit type conversion is what allows an **int** (integer) to be passed as an argument to a function that takes a **long int** without having to explicitly convert the argument.
    - » Implicit conversions between **fundamental types** do carry a certain degree of risk. For example, assigning an integer having a value of -1 to a variable of type 32-bit unsigned int is equivalent of assigning a value of 4294967295.
      - ♦ This is probably not the intended behavior. By default, modern compilers do not warn of potential issues that can be caused by implicit type conversions between fundamental types; however, such warnings may be enabled via compiler command line options (e.g., -Wconversion in the case of gcc).
    - » Implicit type conversion is enabled for user-defined types via:
      - ♦ **Conversion constructors:** Constructors declared without the **explicit** keyword
      - ♦ **User-defined conversion functions**
    - » Implicit type conversion of user-defined types can be disabled via the **explicit** keyword. If the explicit keyword is added to a constructor or user-defined conversion function, the constructor or conversion must be explicitly specified.
      - ♦ User-defined conversion functions should almost always be declared as **explicit** to avoid accidental implicit conversion.

Example of the various type conversion mechanisms:

```
void example()
{
    class Complex
    {
    public:
        // Note use of C-style explicit conversion
        // of some arguments from int to double
        Complex() : real{0.0}, img{0.0} {};
        Complex(double r) : real{r}, img{0.0} {};
        Complex(double r, double i) : real{r}, img{i} {};
        explicit Complex(int r, int i) :
            real{(double)r}, img{(double)i} {};
        explicit Complex(int r) :
            real{(double)r}, img{0.0} {};
        // user-defined conversion to int
        explicit operator int() const {
            return (int)sqrt(real*real+img*img); };
        // virtual destructor is needed because we
        // use of dynamic_cast below
        virtual ~Complex() {};
    private:
        double real, img;
    };

    // implicit conversion of initializer list
    // of double to complex
    Complex c = {1.0,1.0};

    // implicit conversion of initializer list
    // of int to complex
    // c = {1,1}; // error: will not compile
    // c = {1}; // error: will not compile

    // explicit construction of complex from
    // initializer list of int
    c = Complex{1,1};

    c = Complex{1};

    // explicit conversion from complex to int
    int i = (int)Complex();
    i = static_cast<int>(Complex{1,1});

    // implicit conversion from complex to int
    // i = complex(); // error: will not compile

    class SpecialComplex : public Complex
    {
    public:
        SpecialComplex(double r, double i) :
         Complex(r,i) {};
    };

    SpecialComplex sc{1,1};

    // implicit conversion (upcast) from SpecialComplex ref
    // to Complex ref
    Complex const & cr = sc;

    // downcast from base to derived class
    SpecialComplex const & sc2 =
     dynamic_cast<SpecialComplex const &>(cr);

// assign base class ref to derived class ref
    // sp2 = cr; // error: will not compile

    // initialize a non-const ref from a const ref
    // by removing constness
    SpecialComplex & sc3(const_cast<SpecialComplex &>(sc2));

    // initialize a const ref from non-const ref
    // SpecialComplex & sc4(sc2); // error: will not compile
}
```

Operator Overloading

OPERATOR	CLASS MEMBER	NONMEMBER
preincrement	T& T::operator++();	friend T& operator++(T&);
postincrement	T T::operator++(int);	friend T operator++(T&,int);
predecrement	T& T::operator--();	friend T& operator--(T&);
postdecrement	T T::operator--(int);	friend T operator--(T&,int);
unary plus	T T::operator+() const;	friend T operator+(T const &);
unary minus	T T::operator-() const;	friend T operator-(T const &);
addition	T T::operator+(T2 const &) const;	friend T operator+(T const &, T2 const &);
subtraction	T T::operator-(T2 const &) const;	friend T operator-(T const &, T2 const &);
multiplication	T T::operator*(T2 const &) const;	friend T operator*(T const &, T2 const &);
division	T T::operator/(T2 const &) const;	friend T operator/(T const &, T2 const &);
modulo	T T::operator%(T2 const &) const;	friend T operator%(T const &, T2 const &);
bitwise AND	T T::operator&(T2 const &) const;	friend T operator&(T const &, T2 const &);
bitwise OR	T T::operator|(T2 const &) const;	friend T operator|(T const &, T2 const &);
bitwise NOT	T T::operator~() const;	friend T operator~(T const &);
bitwise XOR	T T::operator^(T2 const &) const;	friend T operator^(T const &, T2 const &);
bitwise left shift	T T::operator<<(T2 const &) const;	friend T operator<<(T const &, T2 const &);
bitwise right shift	T T::operator>>(T2 const &) const;	friend T operator>>(T const &, T2 const &);
copy assignment	T& T::operator=(T const &);	N/A
move assignment	T& T::operator=(T const &&) noexcept;	N/A
stream insertion	N/A	friend std::ostream& operator<<(std::ostream&, T const &);
stream extraction	N/A	friend std::istream& operator>>(std::istream&, T&);
function call	T2 operator()(…); **Note:** Function call operator overloads may have arbitrary return types and arguments. Used to implement functors (aka function objects)	N/A
less than	bool operator<(T const &) const;	friend bool operator<(T const&, T const &);
less or equal	bool operator<=(T const &) const;	friend bool operator<=(T const&, T const &);
greater than	bool operator>(T const &) const;	friend bool operator>(T const&, T const &);
greater or equal	bool operator>=(T const &) const;	friend bool operator>=(T const&, T const &);
equal	bool operator==(T const &) const;	friend bool operator==(T const&, T const &);
not equal	bool operator!=(T const &) const;	friend bool operator!=(T const&, T const &);
array subscript	T::value_type& operator[](std::size_t); T::value_type const & operator[](std::size_t) const;	N/A
not	bool operator!() const; **Note:** Should return the inverse of the user-defined conversion function: explicit operator bool() const;	N/A

Operator Overloading (*continued*)

- C++ operators for operands of user-defined types may be overloaded in order to allow user-defined types to behave like fundamental types in regard to the operators that are overloaded.
  - When a user-defined type overloads an operator, the behavior of that operator should be consistent with expected/conventional behavior for that operator.
  - It is best not to overload an operator unless there is an expectation that the operator should apply naturally to the user-defined type.
  - Most operators can be overloaded. The only operators that cannot be overloaded are **sizeof :: . ?:**
- Defining new operators is not supported.
- Operator overloads may be declared as **class members** or as **nonmember friends**.
  - When overloading an operator for a class using a nonmember function, the function is often made a friend function so that it may access internals of the class.
  - To avoid having to make a nonmember operator overload a friend, the nonmember operator overload can be implemented in terms of the corresponding member operator overload.
  - The table on p. 5 shows how to declare various kinds of operator overloads for a class T.

Operator overloading example:

```
#include <iostream>
#include <vector>
#include <algorithm>

template <typename T> class vec
{
    std::vector<T> v;

public:
    // default constructor
    vec() {};
    // constructor to reserve size
    vec(size_t size) : v(size) {};
    // construct from initializer list
    vec(std::initializer_list<T> l) : v{l} {};
    // construct from std::vector
    vec(std::vector<T> arg) : v{arg} {};
    // copy constructor
    vec(vec const & arg) : v{arg.v} {};
    // move constructor
    vec(vec const && arg) : v{std::move(arg.v)} {};
    // copy assignment
    vec& operator=(vec const & rhs) {
        if (this != &rhs) {
            v = rhs;
        }
        return *this;
    }
    // move assignment
    vec& operator=(vec && rhs) noexcept {
        if (this != &rhs) {
            v = std::move(rhs.v);
        }
        return *this;
    }
    // addition
    friend vec operator+(vec const& lhs, vec const & rhs) {
        if (lhs.v.size() != rhs.v.size())
            throw std::invalid_argument("");
        vec result(lhs.v.size());
        std::transform( lhs.v.begin(), lhs.v.end(),
            rhs.v.begin(), result.v.begin(),
            [](T l, T r){return l + r;}
        );
        return result;
    }
    // stream insertion
    friend std::ostream&
    operator<<(std::ostream& os, vec const & v) {
        if (!v.v.empty()) {
            auto i{v.v.begin()};
            os << *i;
            while (++i!=v.v.end()) {
                os << "," << *i;
            }
        }
        return os;
    }

};

int main() {

    vec<double> v1{1,2,3};
    vec<double> v2{1,1,1};

    auto v3 = v1 + v2;
    v1 = std::move(v3);

    std::cout << "v1 = [" << v1 << "]\n";
    std::cout << "v3 = [" << v3 << "]\n";

    return 0;
}
```

Output:

```
v1 = [2,3,4]
v3 = []
```

Virtual Functions, Function Overriding & Interfaces

- **Virtual function (virtual method):** Member function that is invoked via *dynamic dispatch*
- **Dynamic dispatch:** Mechanism by which an implementation of a function is selected and called at run-time based on the type of the object on which the virtual member function is invoked

Example of a virtual function declared using the virtual keyboard:

```
// Shape is an abstract class
// containing a pure virtual function
class Shape
{
    virtual void draw() = 0;
};
```

- In the example above, the Shape class declares a virtual method called **draw**. The "=0;" at the end of the virtual function declaration indicates that the Shape class will not provide an implementation of the draw method. Rather, classes derived from Shape must provide the draw function in order to be instantiable.
- By declaring a member function to be virtual in a base class, Shape forces that member function to be virtual in derived classes whether or not the function is declared as virtual in the derived class.
- A class containing at least one pure virtual function is known as an **abstract class**, meaning that it cannot be instantiated.
  - Abstract classes are used to define *interfaces*.
    - » An **interface** is a set of method signatures that define the ways in which subtype polymorphic code can interact with objects that implement the interface.
- Derived concrete classes that implement the interface defined by the Shape abstract class can be declared as follows:

```
class Circle : public Shape
{
    void draw() override final;
};
```

Override & Final Keywords

- Use of the **override** keyword is optional but recommended, since it allows the compiler to catch common mistakes, such as declaring the virtual function in the derived class using a function signature that does not match the signature of the virtual function in the base class.
- In order to override a virtual method, the function signature of the virtual method in the derived class must exactly match the signature of the virtual method in the base class. Specifying the override keyword allows the compiler to enforce this rule.
- The use of the *final* keyword is also optional. The **final** keyword specifies that any classes derived from Circle are not allowed to override the draw method.
- If the object model calls for different types of circles (e.g., SolidCircle and HollowCircle) that are derived from the Circle class and are drawn differently, then you would not want to use the *final* keyword here.

Polymorphic Destruction & Virtual Destructors

- Subtype polymorphic code is written in terms of pointers and/or references to base classes that point or refer to objects of derived classes.
  - In such code, it is common to delete objects of derived classes via pointer to base classes, in which case it is critically important to declare the base class's destructor as a virtual function.
- In the following example, two derived objects (a Circle and a Square) are added to a vector of a unique pointer. Note that the unique pointers contain pointers to the base class (Shape).
  - When the main function returns, the vector is destroyed, which results in the unique pointers contained in the vector being destroyed, which in turn causes the shape objects (allocated on the heap) to be deleted.
  - When the unique_ptr deletes its contained object, it does so by invoking the delete operator on a pointer to the base class (Shape).
    - » **The use of a virtual destructor is required here in order to assure that all resources associated with the derived objects get cleaned up properly.** By declaring the base class as having a virtual destructor, it ensures that when the destructor is called, the derived object's destructor is called first. This can be seen in the output of the example, which shows that first the derived object's destructor is called, then the base object's destructor is called.
    - » **When deleting derived objects via pointers to their base classes, resource leaks will occur unless the base class is declared to have a virtual destructor.**

Example:

```
struct Shape
{
    // virtual destructor
    virtual ~Shape() { std::cout << "~Shape\n"; }
};

struct Circle : public Shape
{
    // virtual destructor
   ~Circle() override { std::cout << "~Circle\n"; }
};

struct Square : public Shape
{
    // virtual destructor
    ~Square() override { std::cout << "~Square\n"; }
};

int main()
{
    std::vector<std::unique_ptr<Shape>> shapes;

    shapes.emplace_back(std::make_unique<Circle>());
    shapes.emplace_back(std::make_unique<Square>());
}
```

Output:

```
~Circle
~Shape
~Square
~Shape
```

Templates

- Generic code that can act on any type in a type-safe manner as long as the type meets certain requirements implied by the template
- Can be used to define classes, functions (which may be member functions), and variables
  - Template member functions cannot be virtual functions.

```
#include <iostream>
#include <type_traits>

// template function
template<typename T> T factorial(const T& n) {
    return n<2 ? 1 : n * factorial(n-1); }

// template class
template<typename T> class Point
{
public:
    Point(T _x, T _y) : x(_x), y(_y) {}

    T x;
    T y;
};

// template variable
template<class T>
constexpr T pi = T(3.1415926535897932385L);

int main(int argc, char **argv) {

    double PI = pi<double>;

    const Point origin(0,0);

    for (long i=1; i<10; ++i)
    {
        std::cout << "Factorial of [" << i
           << "] = [" << factorial(i)
           << "]" << std::endl;
    }
    return 0;
}
```

U.S. $7.95 Author: Scott Smith
ISBN-13: 978-142324217-8
ISBN-10: 142324217-3
9 781423 242178 50795

6 54614 04217 0