PREV
Preface

NEXT
2. User-Defined Types

1. The Basics

The first thing we do, let's
kill all the language lawyers.

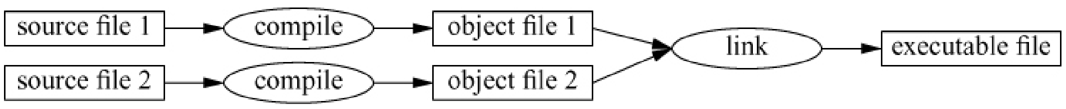
– Henry VI, Part II

1.1. INTRODUCTION

This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called *procedural programming*.

1.2. PROGRAMS

C++ is a compiled language. For a program to run, its source text has to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program. A C++ program typically consists of many source code files (usually simply called *source files*).



An executable program is created for a specific hardware/system combination; it is not portable, say, from a Mac to a Windows PC. When we talk about portability of C++ programs, we usually mean portability of source code; that is, the source code can be successfully compiled and run on a variety of systems.

The ISO C++ standard defines two kinds of entities:

• *Core language features*, such as built-in types (e.g., char and int) and loops (e.g., for-statements and while-statements)

• *Standard-library components*, such as containers (e.g., vector and map) and I/O operations (e.g., << and getline())

The standard-library components are perfectly ordinary C++ code provided by every C++ implementation. That is, the C++ standard library can be implemented in C++ itself (and is with very minor uses of machine code for things such as thread context switching). This implies that C++ is sufficiently expressive and efficient for the most demanding systems programming tasks.

C++ is a statically typed language. That is, the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use. The type of an object determines the set of operations applicable to it.

1.3. HELLO, WORLD!

The minimal C++ program is

Click here to view code image

```
int main() { }            // the minimal C++ program
```

This defines a function called main, which takes no arguments and does nothing.

Curly braces, { }, express grouping in C++. Here, they indicate the start and end of the function body. The double slash, //, begins a comment that extends to the end of the line. A comment is for the human reader; the compiler ignores comments.

Every C++ program must have exactly one global function named main(). The program starts by executing that function. The int value returned by main(), if any, is the program's return value to "the system." If no value is returned, the system will receive a value indicating successful completion. A nonzero value from main() indicates failure. Not every operating system and execution environment make use of that return value: Linux/Unix-based environments often do, but Windows-based environments rarely do.

Typically, a program produces some output. Here is a program that writes Hello, World!:

Click here to view code image

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

The line #include <iostream> instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in iostream. Without these declarations, the expression

```
std::cout << "Hello, World!\n"
```

would make no sense. The operator << ("put to") writes its second argument onto its first. In this case, the string literal "Hello, World!\n" is written onto the standard output stream std::cout. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character \ followed by another character denotes a single "special character." In this case, \n is the newline character, so that the characters written are Hello, World! followed by a newline.

The std:: specifies that the name cout is to be found in the standard-library namespace (§3.3). I usually leave out the std:: when discussing standard features; §3.3 shows how to make names from a namespace visible without explicit qualification.

Essentially all executable code is placed in functions and called directly or indirectly from main(). For example:

Click here to view code image

```
#include <iostream>                // include ("import") the declara

using namespace std;               // make names from std visible

double square(double x)            // square a double precision floa
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
        print_square(1.234);       // print: the square of 1.234 is 1.5
}
```

A "return type" void indicates that a function does not return a value.

1.4. FUNCTIONS

The main way of getting something done in a C++ program is to call a function to do it. Defining a function is the way you specify how an operation is to be done. A function cannot be called unless it has been previously declared.

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call. For example:

Click here to view code image

```
Elem* next_elem();          // no argument; return a pointer to Elem
void exit(int);             // int argument; return nothing
double sqrt(double);        // double argument; return a double
```

In a function declaration, the return type comes before the name of the function and the argument types after the name enclosed in parentheses.

The semantics of argument passing are identical to the semantics of copy initialization. That is, argument types are checked and implicit argument type conversion takes place when necessary (§1.5). For example:

Click here to view code image

```
double s2 = sqrt(2);            // call sqrt() with the argument dou
double s3 = sqrt("three");      // error: sqrt() requires an argument
```

The value of such compile-time checking and type conversion should not be underestimated.

A function declaration may contain argument names. This can be a help to the reader of a program, but unless the declaration is also a function definition, the compiler simply ignores such names. For example:

Click here to view code image

```
double sqrt(double d);      // return the square root of d
double square(double);      // return the square of the argument
```

The type of a function consists of the return type and the argument types. For class member functions (§2.3, §4.2.1), the name of the class is also part of the function type. For example:

Click here to view code image

```
double get(const vector<double>& vec, int index);      // type: do
char& String::operator[](int index);                    // type: ch
```

We want our code to be comprehensible, because that is the first step on the way to maintainability. The first step to comprehensibility is to break computational tasks into comprehensible chunks (represented as functions and classes) and name those. Such functions then provide the basic vocabulary of computation, just as the types (built-in and user-defined) provide the basic vocabulary of data. The C++ standard algorithms (e.g., `find`, `sort`, and `iota`) provide a good start (Chapter 10). Next, we can compose functions representing common or specialized tasks into larger computations.

The number of errors in code correlates strongly with the amount of code and the complexity of the code. Both problems can be addressed by using more and shorter functions. Using a function to do a specific task often saves us from writing a specific piece of code in the middle of other code; making it a function forces us to name the activity and document its dependencies.

If two functions are defined with the same name, but with different argument types, the compiler will choose the most appropriate function to invoke for each call. For example:

Click here to view code image

```
void print(int);        // takes an integer argument
void print(double);     // takes a floating-point argument
void print(string);     // takes a string argument

void user()
{
        print(42);                      // calls print(int)
        print(9.65);                    // calls print(double)
        print("D is for Digital"); // calls print(string)
}
```

If two alternative functions could be called, but neither is better than the other, the call is deemed ambiguous and the compiler gives an error. For example:

Click here to view code image

```
void print(int,double);
void print(double,int);

void user2()
{
        print(0,0);      // error: ambiguous
}
```

This is known as function overloading and is one of the essential parts of generic programming (§5.4). When a function is overloaded, each function of the same name should implement the same semantics. The print() functions are an example of this; each print() prints its argument.

1.5. TYPES, VARIABLES, AND ARITHMETIC

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that inch is of type int; that is, inch is an integer variable.

A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

• A *type* defines a set of possible values and a set of operations (for an object).

• An *object* is some memory that holds a value of some type.

• A *value* is a set of bits interpreted according to a type.

• A *variable* is a named object.

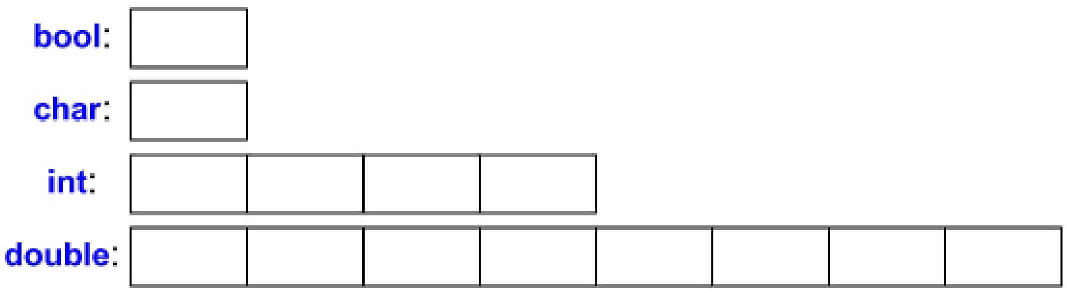
C++ offers a variety of fundamental types. For example:

Click here to view code image

```
bool                // Boolean, possible values are true and false
char                // character, for example, 'a', 'z', and '9'
int                 // integer, for example, –273, 42, and 1066
```

```
double          // double-precision floating-point number, for exa
unsigned        // non-negative integer, for example, 0, 1, and 99
```

Each fundamental type corresponds directly to hardware facilities and has a fixed size that determines the range of values that can be stored in it:



A char variable is of the natural size to hold a character on a given machine (typically an 8-bit byte), and the sizes of other types are quoted in multiples of the size of a char. The size of a type is implementation-defined (i.e., it can vary among different machines) and can be obtained by the sizeof operator; for example, sizeof(char) equals 1 and sizeof(int) is often 4.

The arithmetic operators can be used for appropriate combinations of these types:

Click here to view code image

```
x+y        // plus
+x         // unary plus
x-y        // minus
-x         // unary minus
x*y        // multiply
x/y        // divide
x%y        // remainder (modulus) for integers
```

So can the comparison operators:

Click here to view code image

```
x==y       // equal
x!=y       // not equal
x<y        // less than
x>y        // greater than
x<=y       // less than or equal
x>=y       // greater than or equal
```

Furthermore, logical operators are provided:

Click here to view code image

```
x&y        // bitwise and
x|y        // bitwise or
x^y        // bitwise exclusive or
~x         // bitwise complement
x&&y       // logical and
x||y       // logical or
```

A bitwise logical operator yield a result of their operand type for which the operation has been performed on each bit. The logical operators `&&` and `||` simply return `true` or `false` depending on the values of their operands.

In assignments and in arithmetic operations, C++ performs all meaningful conversions between the basic types so that they can be mixed freely:

Click here to view code image

```
void some_function()      // function that doesn't return a value
{
        double d = 2.2;   // initialize floating–pointinitializefloati
        int i = 7;        // initialize integer
        d = d+i;          // assign sum to d
        i = d*i;          // assign product to i (truncating the c
}
```

The conversions use in expressions are called *the usual arithmetic conversions* and aim to ensure that expressions are computed at the highest precision of its operands. For example, an addition of a `double` and an `int` is calculated using double-precision floating-point arithmetic.

Note that `=` is the assignment operator and `==` tests equality.

C++ offers a variety of notations for expressing initialization, such as the `=` used above, and a universal form based on curly-brace-delimited initializer lists:

Click here to view code image

```
double d1 = 2.3;               // initialize d1 to 2.3
double d2 {2.3};               // initialize d2 to 2.3
```

Click here to view code image

```
complex<double> z = 1;                // a complex number with c
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2};          // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6};         // a vector of ints
```

The = form is traditional and dates back to C, but if in doubt, use the general {}-list form. If nothing else, it saves you from conversions that lose information:

Click here to view code image

```
int i1 = 7.2;             // i1 becomes 7 (surprise?)
int i2 {7.2};             // error: floating-point to integer conversion
int i3 = {7.2};           // error: floating-point to integer conversion (t
```

Unfortunately, conversions that lose information, *narrowing conversions*, such as double to int and int to char are allowed and implicitly applied. The problems caused by implicit narrowing conversions is a price paid for C compatibility (§14.3).

A constant (§1.7) cannot be left uninitialized and a variable should only be left uninitialized in extremely rare circumstances. Don't introduce a name until you have a suitable value for it. User-defined types (such as string, vector, Matrix, Motor_controller, and Orc_warrior) can be defined to be implicitly initialized (§4.2.1).

When defining a variable, you don't actually need to state its type explicitly when it can be deduced from the initializer:

Click here to view code image

```
auto b = true;            // a bool
auto ch = 'x';            // a char
auto i = 123;             // an int
auto d = 1.2;           // a double
auto z = sqrt(y);      // z has the type of whatever sqrt(y) returns
```

With auto, we use the = because there is no potentially troublesome type conversion involved.

We use auto where we don't have a specific reason to mention the type explicitly. "Specific reasons" include:

• The definition is in a large scope where we want to make the type clearly visible to readers of our code.

• We want to be explicit about a variable's range or precision (e.g., double rather than float).

Using auto, we avoid redundancy and writing long type names. This is especially important in generic programming where the exact type of an object can be hard for the programmer to know and the type names can be quite long (§10.2).

In addition to the conventional arithmetic and logical operators, C++ offers more specific operations for modifying a variable:

Click here to view code image

```
x+=y          // x = x+y
++x           // increment: x = x+1
x−=y          // x = x−y
−−x           // decrement: x = x−1
x*=y          // scaling: x = x*y
x/=y          // scaling: x = x/y
x%=y          // x = x%y
```

These operators are concise, convenient, and very frequently used.

1.6. SCOPE AND LIFETIME

A declaration introduces its name into a scope:

• *Local scope*: A name declared in a function (§1.4) or lambda (§5.5) is called a *local name*. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A *block* is delimited by a { } pair. Function argument names are considered local names.

• *Class scope*: A name is called a *member name* (or a *class member name*) if it is defined in a class (§2.2, §2.3, Chapter 4), outside any function (§1.4), lambda (§5.5), or `enum class` (§2.5). Its scope extends from the opening `{` of its enclosing declaration to the end of that declaration.

• *Namespace scope*: A name is called a *namespace member name* if it is defined in a name-space (§3.3) outside any function, lambda (§5.5), class (§2.2, §2.3, Chapter 4), or `enum class` (§2.5). Its scope extends from the point of declaration to the end of its namespace.

A name not declared inside any other construct is called a *global name* and is said to be in the *global namespace*.

In addition, we can have objects without names, such as temporaries and objects created using `new` (§4.2.2). For example:

Click here to view code image

```
vector<int> vec;      // vec is global (a global vector of integers)

struct Record {
        string name;     // name is a member (a string member)
        // ...
};

void fct(int arg)     // fct is global (a global function)
                          // arg is local (an integer argument)
{
        string motto {"Who dares win"};           // motto is local
        auto p = new Record{"Hume"};               // p points to
        // ...
}
```

An object must be constructed (initialized) before it is used and will be destroyed at the end of its scope. For a namespace object the point of destruction is the end of the program. For a member, the point of destruction is determined by the point of destruction of the object of which it is a member. An object created by `new` "lives" until destroyed by `delete` (§4.2.2).

1.7. CONSTANTS

C++ supports two notions of immutability:

• const: meaning roughly "I promise not to change this value." This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by const.

• constexpr: meaning roughly "to be evaluated at compile time." This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted) and for performance.

For example:

Click here to view code image

```
const int dmv = 17;                            // dmv is a
int var = 17;                                  // var is n

constexpr double max1 = 1.4*square(dmv);       // OK if squa
constexpr double max2 = 1.4*square(var);       // error: var is
const double max3 = 1.4*square(var);           // OK, may be

double sum(const vector<double>&);             // sum will
vector<double> v {1.2, 3.4, 4.5};              // v is not a cor
const double s1 = sum(v);                      // OK: evalu
constexpr double s2 = sum(v);                  // error: sur
```

For a function to be usable in a *constant expression*, that is, in an expression that will be evaluated by the compiler, it must be defined constexpr. For example:

Click here to view code image

```
constexpr double square(double x) { return x∗x; }
```

To be constexpr, a function must be rather simple: just a return-statement computing a value. A constexpr function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a constexpr function to be called with non-constant-expression arguments in contexts that do not require constant expressions, so that we don't have to define essentially the same function twice: once for constant expressions and once for variables.

In a few places, constant expressions are required by language rules (e.g., array bounds (§1.8), case labels (§1.9), template value arguments (§5.2), and constants declared using constexpr). In other cases, compile-time evaluation is important for performance. Independently of performance issues, the notion of immutability (of an object with an unchangeable state) is an important design concern.

1.8. POINTERS, ARRAYS, AND REFERENCES

An array of elements of type char can be declared like this:

Click here to view code image

```
char v[6];          // array of 6 characters
```

Similarly, a pointer can be declared like this:

Click here to view code image

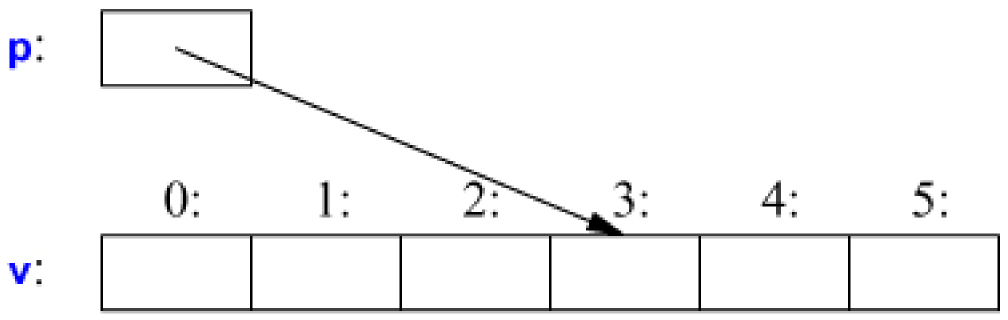
```
char* p;            // pointer to character
```

In declarations, [ ] means "array of" and * means "pointer to." All arrays have 0 as their lower bound, so v has six elements, v[0] to v[5]. The size of an array must be a constant expression (§1.7). A pointer variable can hold the address of an object of the appropriate type:

Click here to view code image

```
char* p = &v[3];          // p points to v's fourth element
char x = *p;              // *p is the object that p points to
```

In an expression, prefix unary * means "contents of" and prefix unary & means "address of." We can represent the result of that initialized definition graphically:



Consider copying ten elements from one array to another:

Click here to view code image

```
void copy_fct()
{
        int v1[10] = {0,1,2,3,4,5,6,7,8,9};
        int v2[10];                           // to become a copy of v1

        for (auto i=0; i!=10; ++i)    // copy elements
                v2[i]=v1[i];
        // ...
}
```

This for-statement can be read as "set i to zero; while i is not 10, copy the ith element and increment i." When applied to an integer variable, the increment operator, ++, simply adds 1. C++ also offers a simpler for-statement, called a range-for-statement, for loops that traverse a sequence in the simplest way:

Click here to view code image

```
void print()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto x : v)                        // for each x in v
                cout << x << '\n';

        for (auto x : {10,21,32,43,54,65})
                cout << x << '\n';
        // ...
}
```

The first range-for-statement can be read as "for every element of v, from the first to the last, place a copy in x and print it." Note that we don't have to specify an array bound when we initialize it with a list. The range-for-statement can be used for any sequence of elements (§10.1).

If we didn't want to copy the values from v into the variable x, but rather just have x refer to an element, we could write:

Click here to view code image

```
void increment()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};
        for (auto& x : v)
                ++x;
        // ...
}
```

In a declaration, the unary suffix & means "reference to." A reference is similar to a pointer, except that you don't need to use a prefix * to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization.

References are particularly useful for specifying function arguments. For example:

Click here to view code image

```
void sort(vector<double>& v);        // sort v
```

By using a reference, we ensure that for a call sort(my_vec), we do not copy my_vec and that it really is my_vec that is sorted and not a copy of it.

When we don't want to modify an argument, but still don't want the cost of copying, we use a const reference. For example:

Click here to view code image

```
double sum(const vector<double>&)
```

Functions taking const references are very common.

When used in declarations, operators (such as &, *, and [ ]) are called *declarator operators*:

Click here to view code image

```
T a[n];     // T[n]: array of n Ts
T* p;        // T*: pointer to T
T& r;        // T&: reference to T
T f(A);     // T(A): function taking an argument of type A returning
```

We try to ensure that a pointer always points to an object, so that dereferencing it is valid. When we don't have an object to point to or if we need to represent the notion of "no object available" (e.g., for an end of a list), we give the pointer the value nullptr ("the null pointer"). There is only one nullptr shared by all pointer types:

Click here to view code image

```
double* pd = nullptr;
Link<Record>* lst = nullptr;    // pointer to a Link to a Record
int x = nullptr;                  // error: nullptr is a pointer not an
```

It is often wise to check that a pointer argument that is supposed to point to something, actually points to something:

Click here to view code image

```
int count_x(char* p, char x)
        // count the number of occurrences of x in p[]
        // p is assumed to point to a zero–terminated array of char
{
        if (p==nullptr) return 0;
        int count = 0;
        for (;p!=nullptr; ++p)
                if (*p==x)
                        ++count;
        return count;
}
```

Note how we can move a pointer to point to the next element of an array using ++ and that we can leave out the initializer in a for-statement if we don't need it.

The definition of count_x() assumes that the char * is a *C-style string*, that is, that the pointer points to a zero-terminated array of char.

In older code, 0 or NULL is typically used instead of nullptr. However, using nullptr eliminates potential confusion between integers (such as 0 or NULL) and pointers (such as nullptr).

The count_if() example is unnecessarily complicated. We can simplify it by testing for the nullptr in one place only. We are not using the initializer part of the for-statement, so we can use the simpler while-statement:

Click here to view code image

```
int count_x(char* p, char x)
        // count the number of occurrences of x in p[]
        // p is assumed to point to a zero-terminated array of char (
{
        int count = 0;
        while (p) {
                if (*p==x)
                        ++count;
                ++p;
        }
        return count;
}
```

The while-statement executes until its condition becomes false.

A test of a pointer (e.g., while (p)) is equivalent to comparing the pointer to the null pointer (e.g., while (p!=nullptr)).

1.9. TESTS

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

Click here to view code image

```
bool accept()
{
        cout << "Do you want to proceed (y or n)?\n";     // write q

        char answer = 0;
        cin >> answer;                                         //
        if (answer == 'y')
                return true;
        return false;
}
```

To match the << output operator ("put to"), the >> operator ("get from") is used for input; cin is the standard input stream (Chapter 8). The type of the right-hand operand of >> determines what input is accepted, and its right-hand operand is the target of the input operation. The \n character at the end of the output string represents a newline (§1.3).

Note that the definition of answer appears where it is needed (and not before that). A declaration can appear anywhere a statement can.

The example could be improved by taking an n (for "no") answer into account:

Click here to view code image

```
bool accept2()
{
        cout << "Do you want to proceed (y or n)?\n";     // write q

        char answer = 0;
        cin >> answer;                                          //

        switch (answer) {
        case 'y':
                return true;
        case 'n':
                return false;
        default:
                cout << "I'll take that for a no.\n";
                return false;
        }
}
```

A switch-statement tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the default is chosen. If no default is provided, no action is taken if the value doesn't match any case constant.

We don't have to exit a case by returning from the function that contains its switch-statement. Often, we just want to continue execution with the statement following the switch-statement. We can do that using a break statement. As an example, consider an overly clever, yet primitive, parser for a trivial command video game:

Click here to view code image

```
void action()
{
        while (true) {
                cout << "enter action:\n";                // request a
                string act;
                cin >> act;                 // rear characters into a string
                Point delta {0,0};      // Point holds an {x,y} pair

                for (char ch : act) {
                        switch (ch) {
                        case 'u': // up
                        case 'n': // north
                                ++delta.y;
                                break;
                        case 'r': // right
                        case 'e': // east
                                ++delta.x;
                                break;
                        // ... more actions ...

                        default:
                              cout << "I freeze!\n";
                        }
                        move(current+delta*scale);
                        update_display();
                }
        }
}
```

1.10. ADVICE

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 5- 6 , 9 -10, and 12 of [Stroustrup,2013].

[2] Don't panic! All will become clear in time; §1.1.

[3] You don't have to know every detail of C++ to write good programs.

[4] Focus on programming techniques, not on language features.

[5] For the final word on language definition issues, see the ISO C++ standard; §14.1.3.

[6] "Package" meaningful operations as carefully named functions; §1.4.

[7] A function should perform a single logical operation; §1.4.

[8] Keep functions short; §1.4.

[9] Use overloading when functions perform conceptually the same task on different types; §1.4.

[10] If a function may have to be evaluated at compile time, declare it `constexpr`; §1.7.

[11] Avoid "magic constants;" use symbolic constants; §1.7.

[12] Declare one name (only) per declaration.

[13] Keep common and local names short, and keep uncommon and nonlocal names longer.

[14] Avoid similar-looking names.

[15] Avoid `ALL_CAPS` names.

[16] Prefer the `{}`-initializer syntax for declarations with a named type; §1.5.

[17] Prefer the `=` syntax for the initialization in declarations using `auto`; §1.5.

[18] Avoid uninitialized variables; §1.5.

[19] Keep scopes small; §1.6.

[20] Keep use of pointers simple and straightforward; §1.8.

[21] Use `nullptr` rather than `0` or `NULL`; §1.8.

[22] Don't declare a variable until you have a value to initialize it with; §1.8, §1.9.

[23] Don't say in comments what can be clearly stated in code.

[24] State intent in comments.

[25] Maintain a consistent indentation style.

[26] Avoid complicated expressions.

[27] Avoid narrowing conversions; §1.5.