# C++ Primer

- Input and Output
  - **iostream library**
    - A stream is a sequence of characters read from or written to an IO device → sequentially over time
- Note: Headers from the standard library are encloses in angle brackets (< >) and those that are not part of the library are enclosed in double quotes (”“) [eg header files that you create]
- endl is a special value called **a manipulator.** → ending of the current line and flushing the **buffer.**
  - Debugging statements should **always** end with endl to flush the stream. Otherwise, if the program crashes, outputs may be left in the buffer, leading to incorrect inferences about where the program crashed.
- Using of Namespaces: Allows is to avoid unnecessary collisions between the names we define and uses of those same names inside a library.
- Writing **comments:** When using delimeters it is best to begin each line in the comment with an **asterisk,** thus indicating that the entire range is part of a multiline comment

```
#include <iostream>
/*
 * Simple main function:
 * Read two numbers and write their sum
 */
int main()
{
    // prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;   // variables to hold the input we read
    std::cin >> v1 >> v2; // read input
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

- **Classes**
  - a class is a user defined data type along with a collection of operations that are related to that type

- Every class has a type. The type name is the same as the name of the class
- Member Function: Is a function that is defined as part of a class - sometimes referred to as **methods**
  - To call a member function on behalf of an object we use the **dot(.)** operator [**applies only to objects of class type]**
  - The left operand must be an object of class type and the right operand must name a member of that type.
  - When we use the dot operator to access a member function we usually do so to *call that function ; using the call operator (the **() operator)*** → *can be empty or enclose a list of **arguments***

> **USING FILE REDIRECTION**
>
> **It can be tedious to repeatedly type these transactions as input to the programs you are testing. Most operating systems support file redirection, which lets us associate a named file with the standard input and the standard output:**
>
> ```
> $ addItems <infile >outfile
> ```
>
> **Assuming `$` is the system prompt and our addition program has been compiled into an executable file named `addItems.exe` (or `addItems` on UNIX systems), this command will read transactions from a file named `infile` and write its output to a file named `outfile` in the current directory.**

- **Built-in Types**
  - *void: is a special type that has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value*
  - Arithmetic types:
    - integral types ( which include character and boolean types)
    - floating types

**Table 2.1: C++: Arithmetic Types**

| Type | Meaning | Minimum Size |
|---|---|---|
| `bool` | boolean | NA |
| `char` | character | 8 bits |
| `wchar_t` | wide character | 16 bits |
| `char16_t` | Unicode character | 16 bits |
| `char32_t` | Unicode character | 32 bits |
| `short` | short integer | 16 bits |
| `int` | integer | 16 bits |
| `long` | long integer | 32 bits |
| `long long` | long integer | 64 bits |
| `float` | single-precision floating-point | 6 significant digits |
| `double` | double-precision floating-point | 10 significant digits |
| `long double` | extended-precision floating-point | 10 significant digits |

        - Unicode is a standard for representing characters used in essentially any natural language
    - Floating types - represent single-32bits , double-64bits , and extended precision values (long doubles - 96/128bits)
- Signed and Unsigned Types
    - Only for integral types
    - Signed: represents negative or positive numbers (including zero)
        - int, long and long long are all signed
    - Unsigned: represents **only** values greater than or equal to zero

### ADVICE: DECIDING WHICH TYPE TO USE

**C++, like C, is designed to let programs get close to the hardware when necessary. The arithmetic types are defined to cater to the peculiarities of various kinds of hardware. Accordingly, the number of arithmetic types in C++ can be bewildering. Most programmers can (and should) ignore these complexities by restricting the types they use. A few rules of thumb can be useful in deciding which type to use:**

- **Use an unsigned type when you know that the values cannot be negative.**
- **Use `int` for integer arithmetic. `short` is usually too small and, in practice, `long` often has the same size as `int`. If your data values are larger than the minimum guaranteed size of an `int`, then use `long long`.**
- **Do not use plain `char` or `bool` in arithmetic expressions. Use them *only* to hold characters or truth values. Computations using `char` are especially problematic because `char` is `signed` on some machines and `unsigned` on others. If you need a tiny integer, explicitly specify either `signed char` or `unsigned char`.**
- **Use `double` for floating-point computations; `float` usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable run-time cost.**

- Expressions involving Unsigned Types - pg. 66

### CAUTION: DON'T MIX SIGNED AND UNSIGNED TYPES

**Expressions that mix signed and unsigned values can yield surprising results when the signed value is negative. It is essential to remember that signed values are automatically converted to unsigned. For example, in an expression like `a * b`, if `a` is `-1` and `b` is `1`, then if both `a` and `b` are `int`s, the value is, as expected `-1`. However, if `a` is `int` and `b` is an `unsigned`, then the value of this expression depends on how many bits an `int` has on the particular machine. On our machine, this expression yields `4294967295`.**

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl;   //  prints -84
std::cout << u + i << std::endl;   //  if 32-bit ints, prints 4294967264
```

- Literals
  - Integral and Floating-Point Literals
    - using decimal, octa, or hexadecimal notation
    - Integers that begin with 0 (zero) are interpreted as octal

        - 20 /* decimal / *024* / octal / *0x14* / hexadecimal */
    - Character and Character String Literals

      'a' // character literal
      "Hello World!" // string literal

        - The type of a string literal is **array** of constants *chars* → The compiler appends a null character ('\0') to every string literal.
            - Thus the actual size of a string literal is **one more** than its apparent size.

- Escape Sequences

| | | | | | |
|---|---|---|---|---|---|
| newline | \n | horizontal tab | \t | alert (bell) | \a |
| vertical tab | \v | backspace | \b | double quote | \" |
| backslash | \\ | question mark | \? | single quote | \' |
| carriage return | \r | formfeed | \f | | |

**Table 2.2: Specifying the Type of a Literal**

**Character and Character String Literals**

| Prefix | Meaning | Type |
|---|---|---|
| u | Unicode 16 character | char16_t |
| U | Unicode 32 character | char32_t |
| L | wide character | wchar_t |
| u8 | utf-8 (string literals only) | char |

| Integer Literals | | Floating-Point Literals | |
|---|---|---|---|
| **Suffix** | **Minimum Type** | **Suffix** | **Type** |
| u or U | unsigned | f or F | float |
| l or L | long | l or L | long double |
| ll or LL | long long | | |

- **Variables**

**TERMINOLOGY: WHAT IS AN OBJECT?**

**C++ programmers tend to be cavalier in their use of the term *object*. Most generally, an object is a region of memory that can contain data and has a type.**

**Some use the term *object* only to refer to variables or values of class types. Others distinguish between named and unnamed objects, using the term *variable* to refer to named objects. Still others distinguish between objects and values, using the term *object* for data that can be changed by the program and the term *value* for data that are read-only.**

**In this book, we'll follow the more general usage that an object is a region of memory that has a type. We will freely use the term *object* regardless of whether the object has built-in or class type, is named or unnamed, or can be read or written.**

- **Initialisation**


Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

- Default initialisation:
  - The default value depends on the type of the variable and may also depend on where the variable is defined.
- Note:

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

The generalised use of curly braces for initialisation was introduced as part of the new standard (C++ 11) [referred to as: list initialisation].

- When used in variables of built-in type, this form of initialisation has one important property. The compiler will **not** let us initialise variables of built-in type if the initialiser might lead to the loss of information.

```
long double ld = 3.1415926536;
int a{ld}, b = {ld}; // error: narrowing conversion required
int c(ld), d = ld;   // ok: but value will be truncated
```

The compiler rejects the initializations of `a` and `b` because using a `long double` to initialize an `int` is likely to lose data. At a minimum, the fractional part of `ld` will be truncated. In addition, the integer part in `ld` might be too large to fit in an `int`.

- **Variable Declarations and Definitions**
  - C++ supports what is commonly known as *separate complication*. [ which let us split out programs into several files, each of which can be compiled independently]
  - Therefore C++ Distinguishes between **declarations** and **definitions**
    - Declaration: makes a name known to the program.
    - Definition: creates the associated entity

    eg. a variable declaration specifies the type and name of a variable.
  - To obtain a declaration that is **not** a definition, we add the ***extern*** key-word and may not provide an explicit initilizer.

```
extern int i;    //  declares but does not define i
int j;           //  declares and defines j
```

Any declaration that includes an explicit initializer is a definition. We can provide an initializer on a variable defined as `extern`, but doing so overrides the `extern`. An `extern` that has an initializer is a definition:

```
extern double pi = 3.1416; //  definition
```

It is an error to provide an initializer on an `extern` inside a function.


Variables must be defined exactly once but can be declared many times.

  - Note: C++ is statically typed language, which mean that types are checked at compile time
    - Also C++ is case sensitive

**ADVICE: DEFINE VARIABLES WHERE YOU FIRST USE THEM**

**It is usually a good idea to define an object near the point at which the object is first used. Doing so improves readability by making it easy to find the definition of the variable. More importantly, it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used.**

- Nested and Global Scopes
  - Once a name has been declared in a scope, that name can be used by scopes nested inside that scope

- Names declared in the outer scope can also be redefined in an inner scope

- Compound Types
  - A **compound type** is a type that is defined in terms of another type
  - **References**
    - A reference defines an alternative name for an object
    - When we define a reference, instead of copying the initialiser's value, we **bind** the reference to its initialiser. [remains bound to tis initial object]
    - After a reference has been defined, **all** operations on that reference are actually operations on the object to which the reference is bound

```
// ok: refVal3 is bound to the object to which refVal is bound, i.e., to ival
int &refVal3 = refVal;
// initializes i from the value in the object to which refVal is bound
int i = refVal;  // ok: initializes i to the same value as ival
```

Because references are not objects, we may not define a reference to a reference.

    - Note that reference may be bound only to an object, **not a literal or to the result of a more general expression**
  - **Pointers**
    - A pointer is a compound type that "points" to another type
    - pointers are objects / they can be assigned and copied

We define a pointer type by writing a declarator of the form `*d`, where `d` is the name being defined. The `*` must be repeated for each pointer variable:

```
int *ip1, *ip2;  // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

    - Taking the Address of an Object

A pointer holds the address of another object. We get the address of an object by usin the address-of operator (the **& operator**):

```
int ival = 42;
int *p = &ival; //  p holds the address of ival; p is a pointer to ival
```

### Pointer Value

The value (i.e., the address) stored in a pointer can be in one of four states:

1. It can point to an object.
2. It can point to the location just immediately past the end of an object.
3. It can be a null pointer, indicating that it is not bound to any object.
4. It can be invalid; values other than the preceding three are invalid.

- Using a Pointer to Access an Object
  - When a pointer points to an object, we can use the dereference operator (the *** operator)** to access that object.

```
int ival = 42;
int *p = &ival; //  p holds the address of ival; p is a pointer to ival
cout << *p;     //  * yields the object to which p points; prints 42
```

Dereferencing a pointer yields the object to which the pointer points. We can assign to that object by assigning to the result of the dereference:

```
*p = 0;     //  * yields the object; we assign a new value to ival through p
cout << *p; //  prints 0
```

When we assign to `*p`, we are assigning to the object to which `p` points.


We may dereference only a valid pointer that points to an object.

**KEY CONCEPT: SOME SYMBOLS HAVE MULTIPLE MEANINGS**

**Some symbols, such as & and *, are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:**

```
int i = 42;
int &r = i;    //  & follows a type and is part of a declaration; r is a reference
int *p;        //  * follows a type and is part of a declaration; p is a pointer
p = &i;        //  & is used in an expression as the address-of operator
*p = i;        //  * is used in an expression as the dereference operator
int &r2 = *p; //  & is part of the declaration; * is the dereference operator
```

**In declarations, & and * are used to form compound types. In expressions, these same symbols are used to denote an operator. Because the same symbol is used with very different meanings, it can be helpful to ignore appearances and think of them as if they were different symbols.**

- Null Pointers
  - Assignment and Pointers
    - Both pointers and references give indirect access to other objects
    - Reminder:
      - Reference is not an object → no way to make that reference refer to a different object [we get the what was initially bound]
  - A null pointer does not point to any object.

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
int *p2 = 0;       // directly initializes p2 from the literal constant 0
// must #include cstdlib
int *p3 = NULL;    // equivalent to int *p3 = 0;
```

    - *nullptr* → C++ 11 new standard
      - is a literal that has a special type that can be converted to any other pointer type
- Pointer Operations
  - Two pointers are equal if they hold the same address and unequal otherwise.
    - Two pointers hold the same address (ie **equal**) if they are both null, if they address the same object or if they are both pointers one past the same object
- void* Pointers
  - Special type of pointer that can hold the address of any object

- How to write pointers correctly:

```
int* p1, p2; // p1 is a pointer to int; p2 is an int
```

There are two common styles used to define multiple variables with pointer or reference type. The first places the type modifier adjacent to the identifier:

```
int *p1, *p2; // both p1 and p2 are pointers to int
```

This style emphasizes that the variable has the indicated compound type.

The second places the type modifier with the type but defines only one variable per statement:

```
int* p1;       // p1 is a pointer to int
int* p2;       // p2 is a pointer to int
```

This style emphasizes that the declaration defines a compound type.


There is no single right way to define pointers or references. The important thing is to choose a style and use it consistently.

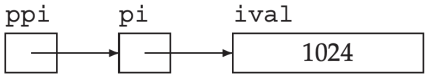
In this book we use the first style and place the * (or the &) with the variable name.

- Pointers to Pointers

Reminder: A pointer is an object in memory, so like any object it has an address

```
int ival = 1024;
int *pi = &ival;    // pi points to an int
int **ppi = &pi;    // ppi points to a pointer to an int
```

Here pi is a pointer to an int and ppi is a pointer to a pointer to an int. We might represent these objects as




Just as dereferencing a pointer to an int yields an int, dereferencing a pointer to a pointer yields a pointer. To access the underlying object, we must dereference the original pointer twice:

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

- Reference to Pointers

Reminder: A reference is not an object

the base type of the declaration says that r is a reference to a pointer to an int.


It can be easier to understand complicated pointer or reference declarations if you read them from right to left.

- constexpr Variable

```
constexpr int mf = 20;          //  20 is a constant expression
constexpr int limit = mf + 1; //  mf + 1 is a constant expression
constexpr int sz = size();     //  ok only if size is a constexpr function
```

- const Qualifier
  - a value that cannot be changed
  - are defined as local to the file

To define a single instance of a `const` variable, we use the keyword `extern` on both its definition and declaration(s):

```
//  file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
//  file_1.h
extern const int bufSize; //  same bufSize as defined in file_1.cc
```

  - Can use pointers to const

- **Dealing with Types**
  - Type Aliases
    - Define using *typedef*
    - Or: C++ 11 → via **alias declaration** (starting with the word: *using )*
  - *auto* Type Specifier
    - Tells the compiler to deduce the type from the initialiser
  - *decltype* Type Specifier
    - We want to define a variable with a type that the compiler deduces from an expression but *do not want to use that expression to initialise the variable.*
      - returns the type of its operand

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; //  x has type const int
decltype(cj) y = x; //  y has type const int& and is bound to x
decltype(cj) z;      //  error: z is a reference and must be initialized
```

Because `cj` is a reference, `decltype(cj)` is a reference type. Like any other reference, z must be initialized.

It is worth noting that `decltype` is the *only* context in which a variable defined as a reference is not treated as a synonym for the object to which it refers.


Remember that `decltype((variable))` (note, double parentheses) is always a reference type, but `decltype(variable)` is a reference type only if *variable* is a reference.

- **Defining Our Own Data Structures**
  - A data structure is a way to group together related data elements and a strategy for using those data
  - We can define our own data types by defining a class
  - using the word **struct** followed by the name of the class and a class body
  - Data members of a class define the contents of the objects of that class type
- **Writing Header Files**
  - Headers usually contain entities (such as class definitions and const and constexpr variables that can be defined only once in a given file)
  - The most common technique for making it safe to include a header multiple times relies on the **preprocessor** - a program that runs before the compiler and changes the source text of out programs eg #include
  - Preprocessor variable has 2 states
    - Defined: **#ifdef** is true if the variable has been defined
    - Undefined **#ifndef** is true if the variable has **not** been defined.
    - Both must end if the **#endif**

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

---

## Chapter 3: String, Vectors and Arrays

- *String* is a variable-length sequence of characters
- *Vector* holds a variable-length sequence of objects of a given type

## Namespace **using** Declarations

```
#include <iostream>
// using declaration; when we use the name cin, we get the one from the namespace std
using std::cin;
int main()
{
    int i;
    cin >> i;        // ok: cin is a synonym for std::cin
    cout << i;       // error: no using declaration; we must use the full name
    std::cout << i; // ok: explicitly use cout from namepsace std
    return 0;
}
```

- Note that Headers: **should not include using Declarations**
- **Strings**

```
#include <string>
using std::string;
```

  - Copy and Direct Forms of Initialization

```
string s5 = "hiya";   // copy initialization
string s6("hiya");    // direct initialization
string s7(10, 'c');   // direct initialization; s7 is cccccccccc
```

**Table 3.1: Ways to Initialize a `string`**

| | |
|---|---|
| `string s1` | Default initialization; `s1` is the empty string. |
| `string s2(s1)` | `s2` is a copy of `s1`. |
| `string s2 = s1` | Equivalent to `s2(s1)`, `s2` is a copy of `s1`. |
| `string s3("value")` | `s3` is a copy of the string literal, not including the null. |
| `string s3 = "value"` | Equivalent to `s3("value")`, `s3` is a copy of the string literal. |
| `string s4(n, 'c')` | Initialize `s4` with `n` copies of the character `'c'`. |

    - **Operations on strings**

| Table 3.2: `string` Operations | |
|---|---|
| `os << s` | Writes `s` onto output stream `os`. Returns `os`. |
| `is >> s` | Reads whitespace-separated string from `is` into `s`. Returns `is`. |
| `getline(is, s)` | Reads a line of input from `is` into `s`. Returns `is`. |
| `s.empty()` | Returns `true` if `s` is empty; otherwise returns `false`. |
| `s.size()` | Returns the number of characters in `s`. |
| `s[n]` | Returns a reference to the `char` at position `n` in `s`; positions start at 0. |
| `s1 + s2` | Returns a `string` that is the concatenation of `s1` and `s2`. |
| `s1 = s2` | Replaces characters in `s1` with a copy of `s2`. |
| `s1 == s2` | The `strings` `s1` and `s2` are equal if they contain the same characters. |
| `s1 != s2` | Equality is case-sensitive. |
| `<, <=, >, >=` | Comparisons are case-sensitive and use dictionary ordering. |

- Using *getline* to read an entire line
  - When we don't want to ignore white space in our output
  - stops when encounter new line even if its in the start
  - new line is not stored on in the string
- The string *empty* and *size* Operations
  - empty returns a bool whether the string is empty
    - we use the (dot) operator to specify the object on which we want to run the empty function
  - size returns the length of a string
- string::size_type
  - This is the return type when using size
- **Comparing strings**
  - Works by comparing the characters of the strings

> The equality operators (`==` and `!=`) test whether two `strings` are equal or unequal, respectively. Two `strings` are equal if they are the same length and contain the same characters. The relational operators `<, <=, >, >=` test whether one `string` is less than, less than or equal to, greater than, or greater than or equal to another. These operators use the same strategy as a (case-sensitive) dictionary:

- **Dealing with the Characters in a string**
  - handle by a set of library functions in **cctype** header [Note the c represents a .h] see note below

| Table 3.3: `cctype` Functions | |
|---|---|
| `isalnum(c)` | `true` if `c` is a letter or a digit. |
| `isalpha(c)` | `true` if `c` is a letter. |
| `iscntrl(c)` | `true` if `c` is a control character. |
| `isdigit(c)` | `true` if `c` is a digit. |
| `isgraph(c)` | `true` if `c` is not a space but is printable. |
| `islower(c)` | `true` if `c` is a lowercase letter. |
| `isprint(c)` | `true` if `c` is a printable character (i.e., a space or a character that has a visible representation). |
| `ispunct(c)` | `true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace). |
| `isspace(c)` | `true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed). |
| `isupper(c)` | `true` if `c` is an uppercase letter. |
| `isxdigit(c)` | `true` if `c` is a hexadecimal digit. |
| `tolower(c)` | If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged. |
| `toupper(c)` | If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged. |

- C++ 11 Standard introduced to manipulate characters in a string.

```
string str("some string");
// print the characters in str one character to a line
for (auto c : str)       // for every char in str
    cout << c << endl;   // print the current character followed by a newline
```

As a somewhat more complicated example, we'll use a range `for` and the `ispunct` function to count the number of punctuation characters in a `string`:

```
string s("Hello World!!!");
// punct_cnt has the same type that s.size returns; see § 2.5.3 (p. 70)
decltype(s.size()) punct_cnt = 0;
// count the number of punctuation characters in s
for (auto c : s)          // for every char in s
    if (ispunct(c))       // if the character is punctuation
        ++punct_cnt;      // increment the punctuation counter
cout << punct_cnt
     << " punctuation characters in " << s << endl;
```

The output of this program is

```
3 punctuation characters in Hello World!!!
```

**ADVICE: USE THE C++ VERSIONS OF C LIBRARY HEADERS**

**In addition to facilities defined specifically for C++, the C++ library incorporates the C library. Headers in C have names of the form *name*`.h`. The C++ versions of these headers are named c*name*—they remove the `.h` suffix and precede the *name* with the letter `c`. The `c` indicates that the header is part of the C library.**

**Hence, `cctype` has the same contents as `ctype.h`, but in a form that is appropriate for C++ programs. In particular, the names defined in the c*name* headers are defined inside the `std` namespace, whereas those defined in the `.h` versions are not.**

**Ordinarily, C++ programs should use the c*name* versions of headers and not the *name*`.h` versions. That way names from the standard library are consistently found in the `std` namespace. Using the `.h` headers puts the burden on the programmer to remember which library names are inherited from C and which are unique to C++.**

- if we want to change the value of the characters in a string → define a reference type
  - When we use a reference as our control variable, that variable is bound to each element in the sequence in turn → hence we can change the character to which the reference is bound

```
string s("Hello World!!!");
// convert s to uppercase
for (auto &c : s)    // for every char in s (note: c is a reference)
    c = toupper(c); //  c is a reference, so the assignment changes the char in s
cout << s << endl;
```

The output of this code is

```
HELLO WORLD!!!
```

- Processing only some characters
  - **We can use the subscript operator (the [] operator)**
  - subscripts for strings start at zero

The following example uses the subscript operator to print the first character in a `string`:

```
if (!s.empty())             // make sure there's a character to print
    cout << s[0] << endl;   // print the first character in s
```

- **Vector type**
  - A vector is a collection of objects, all of which have the same type.
  - Often refer to as a "**container**" because it contains other objects
  - A vector is a **class template**

- Templates are not functions or classes, they can be though as instructions tot he compiler for generating classes or functions
  - The process that the compiler uses to create classes or functions from templates is called **instantiation**

- Defining an Initialising vectors

```
vector<string> svec; //  default initialization; svec has no elements
```

- can use {} to initialise a vector
- We can also create a specified Number of Elements

```
vector<int> ivec(10, -1);        //  ten int elements, each initialized to -1
vector<string> svec(10, "hi!"); //  ten strings; each element is "hi!"
```

**Table 3.4: Ways to Initialize a `vector`**

| | |
|---|---|
| `vector<T> v1` | `vector` that holds objects of type `T`. Default initialization; `v1` is empty. |
| `vector<T> v2(v1)` | `v2` has a copy of each element in `v1`. |
| `vector<T> v2 = v1` | Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`. |
| `vector<T> v3(n, val)` | `v3` has `n` elements with value `val`. |
| `vector<T> v4(n)` | `v4` has `n` copies of a value-initialized object. |
| `vector<T> v5{a,b,c...}` | `v5` has as many elements as there are initializers; elements are initialized by corresponding initializers. |
| `vector<T> v5 = {a,b,c...}` | Equivalent to `v5{a,b,c...}`. |

- Difference between initialising with parenthesis or curly brackets

```
vector<int> v1(10);      //  v1 has ten elements with value 0
vector<int> v2{10};      //  v2 has one element with value 10
vector<int> v3(10, 1);   //  v3 has ten elements with value 1
vector<int> v4{10, 1};   //  v4 has two elements with values 10 and 1
```

```
vector<string> v5{"hi"}; //  list initialization: v5 has one element
vector<string> v6("hi"); //  error: can't construct a vector from a string literal
vector<string> v7{10};        //  v7 has ten default-initialized elements
vector<string> v8{10, "hi"}; //  v8 has ten elements with value "hi"
```

- **Adding** elements to a vector
  - When we don't know the size

- Therefore in such cases it is better to create an empty vector and use **push_back** to add elements at run time

```
vector<int> v2;          //  empty vector
for (int i = 0; i != 100; ++i)
    v2.push_back(i);     //  append sequential integers to v2
//  at end of loop v2 has 100 elements, values 0 . . . 99
```

```
//  read words from the standard input and store them as elements in a vector
string word;
vector<string> text;         //  empty vector
while (cin >> word) {
    text.push_back(word);    //  append word to text
}
```

Again, we start with an initially empty `vector`. This time, we read and store an unknown number of values in `text`.


The body of a range `for` must not change the size of the sequence over which it is iterating.

  - Other vector operations

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto &i : v)      //  for each element in v (note: i is a reference)
    i *= i;            //  square the element value
for (auto i : v)       //  for each element in v
    cout << i << " "; //  print the element
cout << endl;
```

**Table 3.5: `vector` Operations**

| | |
|---|---|
| `v.empty()` | Returns `true` if `v` is empty; otherwise returns `false`. |
| `v.size()` | Returns the number of elements in `v`. |
| `v.push_back(t)` | Adds an element with value `t` to end of `v`. |
| `v[n]` | Returns a reference to the element at position `n` in `v`. |
| `v1 = v2` | Replaces the elements in `v1` with a copy of the elements in `v2`. |
| `v1 = {a,b,c...}` | Replaces the elements in `v1` with a copy of the elements in the comma-separated list. |
| `v1 == v2`<br>`v1 != v2` | `v1` and `v2` are equal if they have the same number of elements and each element in `v1` is equal to the corresponding element in `v2`. |
| `<, <=, >, >=` | Have their normal meanings using dictionary ordering. |

- **Iterators**
  - Using **begin and end** associated with containers

**Table 3.6: Standard Container Iterator Operations**

| | |
|---|---|
| `*iter` | Returns a reference to the element denoted by the iterator `iter`. |
| `iter->mem` | Dereferences `iter` and fetches the member named `mem` from the underlying element. Equivalent to `(*iter).mem`. |
| `++iter` | Increments `iter` to refer to the next element in the container. |
| `--iter` | Decrements `iter` to refer to the previous element in the container. |
| `iter1 == iter2`<br>`iter1 != iter2` | Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container. |

- Moving Iterators form one element to anaother

```
// process characters in s until we run out of characters or we hit a whitespace
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
```

- Iterator Types

```
vector<int>::iterator it; // it can read and write vector<int> elements
string::iterator it2;     // it2 can read and write characters in a string
vector<int>::const_iterator it3; // it3 can read but not write elements
string::const_iterator it4;      // it4 can read but not write characters
```

- It is often a good practice to use a **const** type (such as **const_iterator)** when we need to read but do not need to write to an object
- Combining Dereference and Member Access
  - When we dereference an iterator we get the object that the iterator denotes. if the object has a class type we may want to access a member of that object.
    - Assuming **it** is an iterator into this **vector<string>** , we can check whether the string that **it denotes** is empty

```
(*it).empty()
```

```
(*it).empty() // dereferences it and calls the member empty on the resulting object
*it.empty()   // error: attempts to fetch the member named empty from it
              //        but it is an iterator and has no member named empty
```

- To simplify the expressions the language defined the ( **→ operator)**

- **The arrow combines dereference and member access into a single operations**

That is, `it->mem` is a synonym for `(*it).mem`.

```
// print each line in text up to the first blank line
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

- Implications of using vectors
  - cannot grow dynamically, eg cannot add elements to a vector inside a range *for loop*.
  - operations that change the size of vectors can potentially invalidate all iterators into that vector

- Iterator Arithmetic
  - Incrementing an iterator moves the iterator one element at a time.

**Table 3.7: Operations Supported by `vector` and `string` Iterators**

| | |
|---|---|
| `iter + n`<br>`iter - n` | Adding (subtracting) an integral value `n` to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container. |
| `iter1 += n`<br>`iter1 -= n` | Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`. |
| `iter1 - iter2` | Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container. |
| `>, >=, <, <=` | Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container. |

Note:

A classical algorithm that uses iterator arithmetic is **binary search**

A binary search looks for a particular value in a **sorted sequence.** It operates by looking at the element closest tot he middle of the sequence.

If the element is the one we want, done. Otherwise if the element is smaller than the one we want, we continue searching by looking only at the elements after the rejected one.

If the middle element is larger than the one we want, we continue by looking only in the first half

We compute a new middle element in the reduced range and continue looking until we either find the element or run out of elements.

We can do a binary search using iterators as follows:

```
// text must be sorted
// beg and end will denote the range we're searching
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // original midpoint
// while there are still elements to look at and we haven't yet found sought
while (mid != end && *mid != sought) {
    if (sought < *mid)      // is the element we want in the first half?
        end = mid;          // if so, adjust the range to ignore the second half
    else                    // the element we want is in the second half
        beg = mid + 1;      // start looking with the element just after mid
    mid = beg + (end - beg)/2;   // new midpoint
}
```

- **Arrays**
  - An array is a container of unnamed objects of a single type that can be access by position.
  - Arrays have a fixed size, we cannot add elements to an array
  - Sometimes have better run-time performance for specialised applications


If you don't know exactly how many elements you need, use a `vector`.

  - Defining and initialising arrays
    - The dimension must be known at compile time, which means that the dimension must be a constant expression

```
unsigned cnt = 42;           // not a constant expression
constexpr unsigned sz = 42; // constant expression
                             // constexpr see § 2.4.4 (p. 66)
int arr[10];               // array of ten ints
int *parr[sz];             // array of 42 pointers to int
string bad[cnt];           // error: cnt is not a constant expression
string strs[get_size()]; // ok if get_size is constexpr, error otherwise
```

- We must always specify the type for the array.
- We **cannot use auto** to specify a type for the array

- Explicitly Initialising Array Element
- Character Arrays Are Special
  - string literals end with character. That null character is copied into the array along with the characters in the literal

```
char a1[] = {'C', '+', '+'};        // list initialization, no null
char a2[] = {'C', '+', '+', '\0'}; // list initialization, explicit null
char a3[] = "C++";                  // null terminator added automatically
const char a4[6] = "Daniel";        // error: no space for the null!
```

- No copy or Assignment
  - We **cannot** initialise an array as a copy of another array, nor is it legal to assign one array to another:

```
int a[] = {0, 1, 2}; // array of three ints
int a2[] = a;        // error: cannot initialize one array with another
a2 = a;              // error: cannot assign one array to another
```


Some compilers allow array assignment as a **compiler extension**. It is usually a good idea to avoid using nonstandard features. Programs that use such features, will not work with a different compiler.

- Understanding Complicates Array Declarations
  - Arrays can hold objects of most any type.
  - Eg we can have an array of pointers and references

```
int *ptrs[10];             // ptrs is an array of ten pointers to int
int &refs[10] = /* ? */;   // error: no arrays of references
int (*Parray)[10] = &arr; // Parray points to an array of ten ints
int (&arrRef)[10] = arr;   // arrRef refers to an array of ten ints
```

TIPS: reading definitions of type modifiers from right to left is easier or inside out!

```
int *(&arry)[10] = ptrs; //  arry is a reference to an array of ten pointers
```

Reading this declaration from the inside out, we see that `arry` is a reference. Looking right, we see that the object to which `arry` refers is an array of size 10. Looking left, we see that the element type is pointer to `int`. Thus, `arry` is a reference to an array of ten pointers.


It can be easier to understand array declarations by starting with the array's name and reading them from the inside out.

- Accessing the Elements of an array
  - When we use a variable to subscript an array, we normally should define that variable to have type **size_t** (is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of ant object in memory) [ defined in the **cstddef header]**
  - With the exception that arrays are fixed size
- Checking subscript Values


The most common source of security problems are buffer overflow bugs. Such bugs occur when a program fails to check a subscript and mistakenly uses memory outside the range of an array or similar data structure.

- Pointers and Arrays
  - Pointers and arrays are closely intertwined.
    - When we use an array we ordinarily convert the array to a pointer.

```
string nums[] = {"one", "two", "three"};  //  array of strings
string *p = &nums[0];    //  p points to the first element in nums
```

However, arrays have a special property—in most places when we use an array, the compiler automatically substitutes a pointer to the first element:

```
string *p2 = nums;        //  equivalent to p2 = &nums[0]
```


In most expressions, when we use an object of array type, we are really using a pointer to the first element in that array.

    - When we use an array as an initialiser for a variable defined using **auto** the deduced type is a **pointer** not an array

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; //  ia is an array of ten ints
auto ia2(ia); //  ia2 is an int* that points to the first element in ia
ia2 = 42;     //  error: ia2 is a pointer, and we can't assign an int to a pointer
```

Although `ia` is an array of ten `int`s, when we use `ia` as an initializer, the compiler treats that initialization as if we had written

```
auto ia2(&ia[0]);  //  now it's clear that ia2 has type int*
```

It is worth noting that this conversion does not happen when we use `decltype` (§ 2.5.3, p. 70). The type returned by `decltype(ia)` is array of ten `int`s:

```
//  ia3 is an array of ten ints
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p;     //  error: can't assign an int* to an array
ia3[4] = i; //  ok: assigns the value of i to an element in ia3
```

- Pointers are Iterators
  - Pointers to array elements support the same operations as iterators on vectors or strings

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; //  p points to the first element in arr
++p;          //  p points to arr[1]
```

    eg: use the increment operator to move from one element in an array to the next
  - Just like we use iterators to traverse the elements in a vector we can use pointers to traverse the element in n array
  - We can obtain an off-the-end pointer by using another special property of arrays
    - We can take the address of the nonexistent element one past the last element of an array

```
int *e = &arr[10]; //  pointer just past the last element in arr
```

Using these pointers we can write a loop to print the elements in `arr` as follows:

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; //  print the elements in arr
```

- The library begin and end Functions
  - Although we can compute off-the-end pointer, doing so can be error prone

- C++ 11 offers **begin and end**
- Remember arrays are not class types so these functions are not member functions. Instead they take an argument that is an array:

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; //  ia is an array of ten ints
int *beg = begin(ia); //  pointer to the first element in ia
int *last = end(ia);  //  pointer one past the last element in ia
```

Using `begin` and `end`, it is easy to write a loop to process the elements in an array. For example, assuming `arr` is an array that holds `int` values, we might find the first negative value in `arr` as follows:

```
//  pbeg points to the first and pend points just past the last element in arr
int *pbeg = begin(arr),  *pend = end(arr);
//  find the first negative element, stopping if we've seen all the elements
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

- Pointer Arithmetic
  - Pointers that address array elements can use all the iterator operations

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr;      //  equivalent to int *ip = &arr[0]
int *ip2 = ip + 4; //  ip2 points to arr[4], the last element in arr
```

  - We can also **subtract two pointers** to give us the distance between those pointers

```
auto n = end(arr) - begin(arr); //  n is 5, the number of elements in arr
```

The result of subtracting two pointers is a library type named **`ptrdiff_t`**. Like `size_t`, the `ptrdiff_t` type is a machine-specific type and is defined in the `cstddef` header. Because subtraction might yield a negative distance, `ptrdiff_t` is a signed integral type.

- Interaction between Dereference and Pointer Arithmetic
  - The result of adding an integral value to a pointer is itsedl a pointer

```
int ia[] = {0,2,4,6,8};  //  array with 5 elements of type int
int last = *(ia + 4);  //  ok: initializes last to 8, the value of ia[4]
```

- Subscripts and Pointers

```
int i = ia[2];  // ia is converted to a pointer to the first element in ia
                // ia[2] fetches the element to which (ia + 2) points
int *p = ia;    // p points to the first element in ia
i = *(p + 2);   // equivalent to i = ia[2]
```

We can use the subscript operator on any pointer, as long as that pointer points to an element (or one past the last element) in an array:

```
int *p = &ia[2];  // p points to the element indexed by 2
int j = p[1];     // p[1] is equivalent to *(p + 1),
                  // p[1] is the same element as ia[3]
int k = p[-2];    // p[-2] is the same element as ia[0]
```

The last example points the difference between arrays and library types such as **vector** and **string** that have subscript operators. → Library types force the index used with a subscript to be an **unsigned value**

- **C-Style Character Strings**


Although C++ supports C-style strings, they should not be used by C++ programs. C-style strings are a surprisingly rich source of bugs and are the root cause of many security problems. They're also harder to use!

- Using an Array to Initialise a **vector**
  - To do so we specify the address of the first element and one past the last element that we wish to copy

```
int int_arr[] = {0, 1, 2, 3, 4, 5};
//  ivec has six elements; each is a copy of the corresponding element in int_arr
vector<int> ivec(begin(int_arr), end(int_arr));
```

  - 2 pointers are used to construct ivec mark the range of values to use to initialise the elements in ivec
  - The second pointer points one past the last element to be copied

The specified range can be a subset of the array:

```
// copies three elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
```

This initialization creates `subVec` with three elements. The values of these elements are copies of the values in `int_arr[1]` through `int_arr[3]`.

**ADVICE: USE LIBRARY TYPES INSTEAD OF ARRAYS**

**Pointers and arrays are surprisingly error-prone. Part of the problem is conceptual: Pointers are used for low-level manipulations and it is easy to make bookkeeping mistakes. Other problems arise because of the syntax, particularly the declaration syntax used with pointers.**

**Modern C++ programs should use `vector`s and iterators instead of built-in arrays and pointers, and use `string`s rather than C-style array-based character strings.**

- Multidimensional Array
  - Strictly speaking there are **no** multidimensional arrays in C++
    - What are commonly referred to is arrays of arrays

We define an array whose elements are arrays by providing two dimensions: the dimension of the array itself and the dimension of its elements:

```
int ia[3][4]; // array of size 3; each element is an array of ints of size 4
// array of size 10; each element is a 20-element array whose elements are arrays of 30 ints
int arr[10][20][30] = {0}; // initialize all elements to 0
```

  - Subscripting a Multidimensional Array

dimensions, then the result is the inner array element at the specified index.

```
// assigns the first element of arr to the last element in the last row of ia
ia[2][3] = arr[0][0][0];
int (&row)[4] = ia[1]; // binds row to the second four-element array in ia
```

**Using a Range `for` with Multidimensional Arrays**

Under the new standard we can simplify the previous loop by using a range `for`:

```
size_t cnt = 0;
for (auto &row : ia)          // for every element in the outer array
    for (auto &col : row) {   // for every element in the inner array
        col = cnt;            // give this element the next value
        ++cnt;                // increment cnt
    }
```

  - Pointers and Multidimensional Arrays

Because a multidimensional array is really an array of arrays, the pointer type to which the array converts is a pointer to the first inner array:

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

---

With the advent of the new standard, we can often avoid having to write the type of a pointer into an array by using `auto` or `decltype` (§ 2.5.2, p. 68):

C++ 11

```
//  print the value of each element in ia, with each inner array on its own line
//  p points to an array of four ints
for (auto p = ia; p != ia + 3; ++p) {
    //  q points to the first element of an array of four ints; that is, q points to an int
    for (auto q = *p; q != *p + 4; ++q)
         cout << *q << ' ';
    cout << endl;
}
```

Of course, we can even more easily write this loop using the library `begin` and `end` functions (§ 3.5.3, p. 118):

```
//  p points to the first array in ia
for (auto p = begin(ia); p != end(ia); ++p) {
    //  q points to the first element in an inner array
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' ';    //  prints the int value to which q points
    cout << endl;
}
```

### Type Aliases Simplify Pointers to Multidimensional Arrays

A type alias (§ 2.5.1, p. 67) can make it easier to read, write, and understand pointers to multidimensional arrays. For example:

```
using int_array = int[4]; //  new style type alias declaration; see § 2.5.1 (p. 68)
typedef int int_array[4]; //  equivalent typedef declaration; § 2.5.1 (p. 67)
//  print the value of each element in ia, with each inner array on its own line
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
         cout << *q << ' ';
    cout << endl;
}
```

Here we start by defining `int_array` as a name for the type "array of four `int`s." We use that type name to define our loop control variable in the outer `for` loop.

---

## CHAPTER 4: EXPRESSIONS

An expression is composed of one or more **operands** and yield a **result** when it is evaluated. The simplest form of an expression is a single literal or variable

- **Basic Concepts**

    - Unary operators such as address of (&) and dereference (*) act on one operand
    - Binary operators such as equality (==) and multiplication (*) act on two operands
- Overloaded Operators
    - We can also define what most operators mean when applied to class types. Because such definitions give an *alternative* meaning to an existing operator symbol → refer to them as **overloaded operators**
- Lvalues and Rvalues
    - Every expression in C++ is either an **rvalue or an lvalue**
        - When we use an object as an rvalue we use the object's value (its contents)
        - When we use an object as an lvalue we use the object's identity (its location in memory)
        - Note: we can use an lvalue when an rvalue is required, but we cannot use an rvalue when an lvalue (ie location is required)
- **Precedence and Associativity**
    - An expression with two or more operators is a **compound expression**
    - Evaluating a compound expression involves grouping the operands to the operators.
        - Precedence and associativity **determine how the operands are grouped**
        - In general the value of an expression depend on how the subexpressions are grouped.
            - Operands with higher precedence group more tightly than operands with the same precedence
            - Associativity determines how to group operands with the same precedence.

- Because of precedence, the expression `3+4*5` is `23`, not `35`.
- Because of associativity, the expression `20-15-3` is `2`, not `8`.

As a more complicated example, a left-to-right evaluation of the following expression yields 20:

```
6 + 3 * 4 / 2 + 2
```

Other imaginable results include 9, 14, and 36. In C++, the result is 14, because this expression is equivalent to

```
// parentheses in this expression match default precedence and associativity
((6 + ((3 * 4) / 2)) + 2)
```

- When precedence and Associativity Matter

```
int ia[] = {0,2,4,6,8}; // array with five elements of type int
int last = *(ia + 4);   // initializes last to 8, the value of ia[4]
last = *ia + 4;          // last = 4, equivalent to ia[0] + 4
```

If we want to access the element at the location `ia + 4`, then the parentheses around the addition are essential. Without parentheses, `*ia` is grouped first and `4` is added to the value in `*ia`.

The most common case that we've seen in which associativity matters is in input and output expressions. As we'll see in § 4.8 (p. 155), the operators used for IO are left associative. This associativity means we can combine several IO operations in a single expression:

```
cin >> v1 >> v2;  // read into v1 and then into v2
```

- Order of Evaluation
  - Precedence specifies **how the operands are grouped.** It says nothing about the **order** in which the operands are evaluated.

Note: For operators that do not specify evaluation order, it is an error for an expression to *refer to and change* the same object. Expressions the do so have undefined behaviour

As example the << operator makes **no** guarantees about when or how its operands are evaluated. As a result, the following output expression is undefined

```
int i = 0;
cout << i << " " << ++i << endl;  // undefined
```

Why? **Because** this program is undefined, we cannot draw any conclusions about how it might behave. The compiler might evaluate ++i before evaluating i, in which case the output will be 1 1. Or the compiler might evaluate i first in which case the output will be 0 1 . *Or the compiler might do something else entirely.*

- There are 4 operators that guarantee the order in which operands are evaluated
    - AND (&&) operator guarantees that its **left-hand operand is evalauted first**
    - OR (||)
    - conditional (? :)
    - ,

- Order of Evaluation, Precedence and Associativity

Order of operand evaluation is independent of precedence and associativity. In an expression such as `f() + g() * h() + j()`:

- Precedence guarantees that the results of `g()` and `h()` are multiplied.
- Associativity guarantees that the result of `f()` is added to the product of `g()` and `h()` and that the result of that addition is added to the value of `j()`.
- There are no guarantees as to the order in which these functions are called.

If `f`, `g`, `h`, and `j` are independent functions that do not affect the state of the same objects or perform IO, then the order in which the functions are called is irrelevant. If any of these functions do affect the same object, then the expression is in error and has undefined behavior.

**ADVICE: MANAGING COMPOUND EXPRESSIONS**

**When you write compound expressions, two rules of thumb can be helpful:**

1. **When in doubt, parenthesize expressions to force the grouping that the logic of your program requires.**
2. **If you change the value of an operand, don't use that operand elsewhere in the same expresion.**

**An important exception to the second rule occurs when the subexpression that changes the operand is itself the operand of another subexpression. For example, in `*++iter`, the increment changes the value of `iter`. The (now changed) value of `iter` is the operand to the dereference operator. In this (and similar) expressions, order of evaluation isn't an issue. The increment (i.e., the subexpression that changes the operand) must be evaluated before the dereference can be evaluated. Such usage poses no problems and is quite common.**

- **Aritmetic Operators**

| Table 4.1: Arithmetic Operators (Left Associative) | | |
|---|---|---|
| **Operator** | **Function** | **Use** |
| `+` | unary plus | `+ expr` |
| `-` | unary minus | `- expr` |
| `*` | multiplication | `expr * expr` |
| `/` | division | `expr / expr` |
| `%` | remainder | `expr % expr` |
| `+` | addition | `expr + expr` |
| `-` | subtraction | `expr - expr` |

- The unary minus operator returns the results of negating a (possibly promoted) copy of the value of its operand:

```
int i = 1024;
int k = -i; //  i is -1024
bool b = true;
bool b2 = -b; //  b2 is true!
```

For most operators, operands of type `bool` are promoted to `int`. In this case, the value of `b` is `true`, which promotes to the `int` value `1` (§ 2.1.2, p. 35). That (promoted) value is negated, yielding `-1`. The value `-1` is converted back to `bool` and used to initialize `b2`. This initializer is a nonzero value, which when converted to `bool` is `true`. Thus, the value of `b2` is `true`!

### CAUTION: OVERFLOW AND OTHER ARITHMETIC EXCEPTIONS

**Some arithmetic expressions yield undefined results. Some of these undefined expressions are due to the nature of mathematics—for example, division by zero. Others are undefined due to the nature of computers—for example, due to overflow. Overflow happens when a value is computed that is outside the range of values that the type can represent.**

**Consider a machine on which `short`s are 16 bits. In that case, the maximum `short` is 32767. On such a machine, the following compound assignment overflows:**

```
short short_value = 32767; //  max value if shorts are 16 bits
short_value += 1; //  this calculation overflows
cout << "short_value: " << short_value << endl;
```

**The assignment to `short_value` is undefined. Representing a signed value of 32768 requires 17 bits, but only 16 are available. On many systems, there is *no* compile-time or run-time warning when an overflow occurs. As with any undefined behavior, what happens is unpredictable. On our system the program completes and writes**

```
short_value: -32768
```

**The value "wrapped around": The sign bit, which had been 0, was set to 1, resulting in a negative value. On another system, the result might be different, or the program might behave differently, including crashing entirely.**

```
int ival1 = 21/6;   // ival1 is 3; result is truncated; remainder is discarded
int ival2 = 21/7;   // ival2 is 3; no remainder; result is an integral value
```

- The % operator , known as the “remainder” or the “modulus” operator, computes the remainder that results from dividing the left-hand operand by the right hand operand. **The operand to % must have integral type**

```
int ival = 42;
double dval = 3.14;
ival % 12;    // ok: result is 6
ival % dval; // error: floating-point operand
```

  - C++11 the quotient is rounded toward zero (ie truncated)

```
 21 % 6;    /* result is 3   */        21 / 6;    /* result is 3   */
 21 % 7;    /* result is 0   */        21 / 7;    /* result is 3   */
-21 % -8;   /* result is -5  */       -21 / -8;   /* result is 2   */
 21 % -5;   /* result is 1   */        21 / -5;   /* result is -4  */
```

- Logical and Relational Operators
  - Relational operators take operands of arithmetic or pointer type
  - Logical operators take operands of any type that can be converted to bool.
  - The operators all return values of type bool.

**Table 4.2: Logical and Relational Operators**

| Associativity | Operator | Function | Use |
|---|---|---|---|
| Right | ! | logical NOT | !expr |
| Left | < | less than | expr < expr |
| Left | <= | less than or equal | expr <= expr |
| Left | > | greater than | expr > expr |
| Left | >= | greater than or equal | expr >= expr |
| Left | == | equality | expr == expr |
| Left | != | inequality | expr != expr |
| Left | && | logical AND | expr && expr |
| Left | \|\| | logical OR | expr \|\| expr |

  - Logical AND and OR Operators

- The overall results of the logical AND operator is *true* if and only if both its operands to *true.*
- The logical OR (| |) operator evaluates as *true* if either of its operands evaluates as *true.*

  - The right side of an `&&` is evaluated if and only if the left side is `true`.
  - The right side of an `||` is evaluated if and only if the left side is `false`.

  As an example that uses the logical OR, imagine we have some text in a `vector` of `strings`. We want to print the `strings`, adding a newline after each empty `string` or after a `string` that ends with a period. We'll use a range-based `for` loop (§ 3.2.3, p. 91) to process each element:

  ```
  // note s as a reference to const; the elements aren't copied and can't be changed
  for (const auto &s : text) { // for each element in text
      cout << s;          // print the current element
      // blank lines and those that end with a period get a newline
      if (s.empty() || s[s.size() - 1] == '.')
          cout << endl;
      else
          cout << " ";  // otherwise just separate with a space
  }
  ```

  - Logical NOT operator
    - returns the inverse of the truth value of its operand

    ```
    // print the first element in vec if there is one
    if (!vec.empty())
        cout << vec[0];
    ```

    The subexpression

    ```
    !vec.empty()
    ```

    evaluates as `true` if the call to `empty` returns `false`.

- Increment and Decrement
  - Increment (++) and Decrement (- -) provide a convenient notational shorthand for adding or subtracting 1 from an object

    ```
    int i = 0, j;
    j = ++i; // j = 1, i = 1: prefix yields the incremented value
    j = i++; // j = 1, i = 2: postfix yields the unincremented value
    ```

  - Combining Dereference and Increment in a Single Expression

- Using postfix increment to write a loop to print the values in a vector up to but not including the first negative value

```
auto pbeg = v.begin();
//  print elements up to the first negative value
while (pbeg != v.end() && *beg >= 0)
    cout << *pbeg++ << endl; //  print the current value and advance pbeg
```

- The precedence of postfix increment is higher than that of the dereference operator so *pbeg++ is equivalent to *(pbeg++). The subexpression pbeg++ increments pbeg and yield a copy of the previous value of pbeg

**ADVICE: BREVITY CAN BE A VIRTUE**

**Expressions such as `*pbeg++` can be bewildering—at first. However, it is a useful and widely used idiom. Once the notation is familiar, writing**

```
cout << *iter++ << endl;
```

**is easier and less error-prone than the more verbose equivalent**

```
cout << *iter << endl;
++iter;
```

**It is worthwhile to study examples of such code until their meanings are immediately clear. Most C++ programs use succinct expressions rather than more verbose equivalents. Therefore, C++ programmers must be comfortable with such usages. Moreover, once these expressions are familiar, you will find them less error-prone.**

- The Member Access Operator
  - The **dot and arrow** operators provide for member access
    - The dot operator fetches a member from an object of class type
    - The arrow defined so that *ptr→mem is synonym to (*ptr).mem*

```
string s1 = "a string", *p = &s1;
auto n = s1.size(); //  run the size member of the string s1
n = (*p).size();    //  run size on the object to which p points
n = p->size();      //  equivalent to (*p).size()
```

Note: Because dereference has *lower* precedence than dot, we must parenthesise the dereference subexpression

- Conditional Operator
  - The conditional operator (the **? :** ) let us embed simple if-else logic inside an expression

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

```
string finalgrade = (grade < 60) ?  "fail" : "pass";
```

The condition checks whether `grade` is less than `60`. If so, the result of the expression is `"fail"`; otherwise the result is `"pass"`. Like the logical AND and logical OR (`&&` and `||`) operators, the conditional operator guarantees that only one of *expr1* or *expr2* is evaluated.

That result of the conditional operator is an lvalue if both expressions are lvalues or if they convert to a common lvalue type. Otherwise the result is an rvalue.

- Nesting Conditional Operations

```
finalgrade = (grade > 90) ? "high pass"
                          : (grade < 60) ? "fail" : "pass";
```

The first condition checks whether the grade is above `90`. If so, the expression after the `?` is evaluated, which yields `"high pass"`. If the condition fails, the `:` branch is executed, which is itself another conditional expression. This conditional asks whether the `grade` is less than `60`. If so, the `?` branch is evaluated and yields `"fail"`. If not, the `:` branch returns `"pass"`.

The conditional operator is right associative, meaning (as usual) that the operands group right to left. Associativity accounts for the fact that the right-hand conditional—the one that compares `grade` to `60`—forms the `:` branch of the left-hand conditional expression.


Nested conditionals quickly become unreadable. It's a good idea to nest no more than two or three.

- Using a Conditional Operator in an Output Expression
  - The conditional operator has fairly low precedence

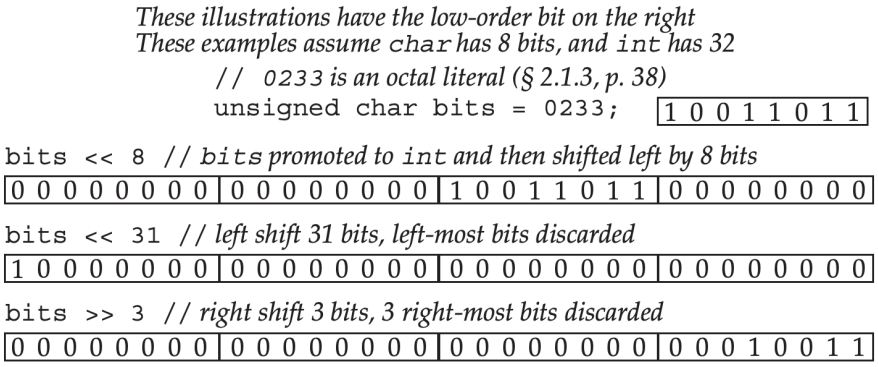
```
cout << ((grade < 60) ?  "fail" : "pass"); //  prints pass or fail
cout << (grade < 60) ?  "fail" : "pass";   //  prints 1 or 0!
cout << grade < 60 ?  "fail" : "pass"; //  error: compares cout to 60
```

- **The Bitwise Operators**
  - Take operands of integral type that they use as a collection of bits. These operators let us test and set individual bits
  - These operators let us test and set individual bits
  - As we'll see in we can also use these operators on a library type named **bitset** that represents a flexibility sized collection of bits.

**Table 4.3: Bitwise Operators (Left Associative)**

| Operator | Function | Use |
|---|---|---|
| ~ | bitwise NOT | ~expr |
| << | left shift | expr1 << expr2 |
| >> | right shift | expr1 >> expr2 |
| & | bitwise AND | expr1 & expr2 |
| ^ | bitwise XOR | expr1 ^ expr2 |
| \| | bitwise OR | expr1 \| expr2 |

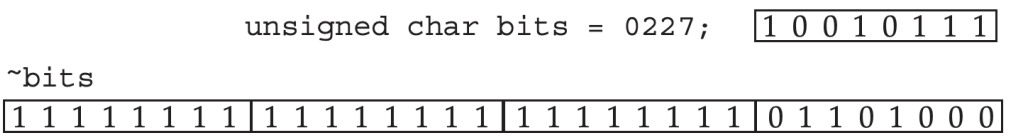
- If the operand is signed and its value is negative, then the way that the "sign bit" is handled in a number of the bitwise operations is machine dependent
- Bitwise Shift Operators



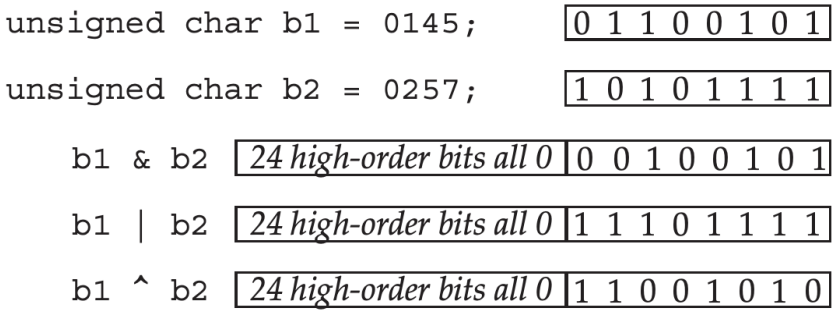

  - The left-shift operator ( << operand) inserts 0-valued bits on the right
- Bitwise NOT Operator

  (~ operand), generates a new value with the bits of its operand inverted.

  **Each 1 bit is set to 0 ; each 0 bit is set to 1**




- Bitwise AND, OR and XOR Operators




- **The *sizeof* Operator**
  - Returns the size, in bytes of an expression or a type anme
  - The operator is right associative

```
Sales_data data, *p;
sizeof(Sales_data); //  size required to hold an object of type Sales_data
sizeof data; //  size of data's type, i.e., sizeof(Sales_data)
sizeof p;    //  size of a pointer
sizeof *p;   //  size of the type to which p points, i.e., sizeof(Sales_data)
sizeof data.revenue; //  size of the type of Sales_data's revenue member
sizeof Sales_data::revenue; //  alternative way to get the size of revenue
```

The result of applying `sizeof` depends in part on the type involved:

- `sizeof char` or an expression of type `char` is guaranteed to be 1.
- `sizeof` a reference type returns the size of an object of the referenced type.
- `sizeof` a pointer returns the size needed hold a pointer.
- `sizeof` a dereferenced pointer returns the size of an object of the type to which the pointer points; the pointer need not be valid.
- `sizeof` an array is the size of the entire array. It is equivalent to taking the `sizeof` the element type times the number of elements in the array. Note that `sizeof` does not convert the array to a pointer.
- `sizeof` a `string` or a `vector` returns only the size of the fixed part of these types; it does not return the size used by the object's elements.

Because `sizeof` returns the size of the entire array, we can determine the number of elements in an array by dividing the array size by the element size:

```
//  sizeof(ia)/sizeof(*ia) returns the number of elements in ia
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr2[sz];  //  ok sizeof returns a constant expression § 2.4.4 (p. 65)
```

- **Comma operator**
  - Takes two operand, which it evaluates form left to right

One common use for the comma operator is in a `for` loop:

```
vector<int>::size_type cnt = ivec.size();
//  assign values from size... 1 to the elements in ivec
for(vector<int>::size_type ix = 0;
                ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

This loop increments `ix` and decrements `cnt` in the expression in the `for` header. Both `ix` and `cnt` are changed on each trip through the loop. As long as the test of `ix` succeeds, we reset the current element to the current value of `cnt`.

- **Type Conversion**
  - When two types are related we can use an object or value of one type where the operand of the related type is expected.
  - Two types are related if there is a **conversion** between them.
    - These conversions are carried out automatically without programmer intervention → For that reason they are referred to as **implicit conversions**
- **Arithmetic Conversions**
  - Convert one arithmetic type to another.
  - Integral Promotions convert the small integral types to a larger integral type
    - types bool, char , singed char, unsigned chat, short and unsigned short are **promoted to int** if all possible values of that type fit in an int
      - Otherwise the value is promoted to **unsigned int**
    - Operands of Unsigned Type
      - For example, given an unsigned int and an int, the int is converted to *unsigned int*
      - If the operands are *long* and *unsigned int,* and int and long have the same size, the long will be converted to *unsigned int.*
        - If the *long* type has more bits, then the *unsigned int* will be converted to *long*

```
bool      flag;        char           cval;
short     sval;        unsigned short usval;
int       ival;        unsigned int   uival;
long      lval;        unsigned long  ulval;
float     fval;        double         dval;
3.14159L + 'a'; // 'a' promoted to int, then that int converted to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, then that int converted to float
sval + cval;    // sval and cval promoted to int
```



```
cval + lval;    // cval converted to long
ival + ulval;   // ival converted to unsigned long
usval + ival;   // promotion depends on the size of unsigned short and int
uival + lval;   // conversion depends on the size of unsigned int and long
```

- Other Implicit Conversions
  - Array to Pointer Conversions: In most expressions when we use an array, the array is automatically converted to a pointer to the first element in that array

```
int ia[10];     // array of ten ints
int* ip = ia;   // convert ia to a pointer to the first element
```

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

```
char *cp = get_string();
if (cp) /* ... */      // true if the pointer cp is not zero
while (*cp) /* ... */ // true if *cp is not the null character
```

  - Conversion to const: We can convert a pointer to a non*const* type to a pointer to the corresponding *const* type and similarly for references. That is, if *T is a type,* we can convert a pointer or reference to T into a pointer or reference to *const T,* respectively.

```
int i;
const int &j = i;   //  convert a nonconst to a reference to const int
const int *p = &i; //  convert address of a nonconst to the address of a const
int &r = j, *q = p; //  error: conversion from const to nonconst not allowed
```

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

- Conversion Defined by Class Types: We use a class-type conversion when we use a C-style character string where a library *string* is expected, and when we read from an *istream* in a condition

```
string s, t = "a value";   //  character string literal converted to type string
while (cin >> s)           //  while condition converts cin to bool
```

  - The resulting *bool* value depend on the state of the stream. If the last read succeeded, then the conversion yield *true*. If the last attempt failed, then the conversion to *bool yield false*

- Explicit Conversions
  - Sometimes we want to explicitly force an objet to be converted to a different type. For example, we might want to use floating-point division in the following code:

```
int i, j;
double slope = i/j;
```

To do so, we'd need a way to explicitly convert `i` and/or `j` to `double`. We use a **cast** to request an explicit conversion.


Although necessary at times, casts are inherently dangerous constructs.

  - Named Casts

A named cast has the following form:

*cast-name*<*type*>(*expression*);

where *type* is the target type of the conversion, and *expression* is the value to be cast. If *type* is a reference, then the result is an lvalue. The *cast-name* may be one of **`static_cast`**, **`dynamic_cast`**, **`const_cast`**, and **`reinterpret_cast`**. We'll cover `dynamic_cast`, which supports the run-time type identification, in § 19.2 (p. 825). The *cast-name* determines what kind of conversion is performed.

    - *static_cast*
      - Forcing expression to use **floating-point** division by casting one of the operands to *double*

```
// cast used to force floating-point division
double slope = static_cast<double>(j) / i;
```

- Potential loss of precision // thus using cast we turn off the warning

```
void* p = &d;     // ok: address of any nonconst object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

When we store a pointer in a `void*` and then use a `static_cast` to cast the pointer back to its original type, we are guaranteed that the pointer value is preserved. That is, the result of the cast will be equal to the original address value. However, we must be certain that the type to which we cast the pointer is the actual type of that pointer; if the types do not match, the result is undefined.

- *const_cast*
  - Changes only a low-level const in its operand

```
const char *pc;
char *p = const_cast<char*>(pc); // ok: but writing through p is undefined
```

- *reinterpret_cast*
  - Generally performs a low-level reinterpretation of the bit patters of its operands.

the one in which pc is used to initialize a `string`.


A `reinterpret_cast` is inherently machine dependent. Safely using `reinterpret_cast` requires completely understanding the types involved as well as the details of how the compiler implements the cast.

## 4.12 Operator Precedence Table

| Associativity and Operator | | Function | Use | See Page |
|---|---|---|---|---|
| L | `::` | global scope | `::`name | 286 |
| L | `::` | class scope | class`::`name | 88 |
| L | `::` | namespace scope | namespace`::`name | 82 |
| L | `.` | member selectors | object`.`member | 23 |
| L | `->` | member selectors | pointer`->`member | 110 |
| L | `[]` | subscript | expr`[` expr `]` | 116 |
| L | `()` | function call | name`(`expr_list`)` | 23 |
| L | `()` | type construction | type`(`expr_list`)` | 164 |
| R | `++` | postfix increment | lvalue`++` | 147 |
| R | `--` | postfix decrement | lvalue`--` | 147 |
| R | `typeid` | type ID | `typeid(`type`)` | 826 |
| R | `typeid` | run-time type ID | `typeid(`expr`)` | 826 |
| R | explicit cast | type conversion | *cast_name*<type>(expr) | 162 |
| R | `++` | prefix increment | `++`lvalue | 147 |
| R | `--` | prefix decrement | `--`lvalue | 147 |
| R | `~` | bitwise NOT | `~`expr | 152 |
| R | `!` | logical NOT | `!`expr | 141 |
| R | `-` | unary minus | `-`expr | 140 |
| R | `+` | unary plus | `+`expr | 140 |
| R | `*` | dereference | `*`expr | 53 |
| R | `&` | address-of | `&`lvalue | 52 |
| R | `()` | type conversion | (type) expr | 164 |
| R | `sizeof` | size of object | `sizeof` expr | 156 |
| R | `sizeof` | size of type | `sizeof(` type `)` | 156 |
| R | `sizeof...` | size of parameter pack | `sizeof...(` name `)` | 700 |
| R | `new` | allocate object | `new` type | 458 |
| R | `new[]` | allocate array | `new` type[size] | 458 |
| R | `delete` | deallocate object | `delete` expr | 460 |
| R | `delete[]` | deallocate array | `delete[]` expr | 460 |
| R | `noexcept` | can expr throw | `noexcept (` expr `)` | 780 |
| L | `->*` | ptr to member select | ptr`->*`ptr_to_member | 837 |
| L | `.*` | ptr to member select | obj`.*`ptr_to_member | 837 |
| L | `*` | multiply | expr `*` expr | 139 |
| L | `/` | divide | expr `/` expr | 139 |
| L | `%` | modulo (remainder) | expr `%` expr | 139 |
| L | `+` | add | expr + expr | 139 |
| L | `-` | subtract | expr - expr | 139 |
| L | `<<` | bitwise shift left | expr << expr | 152 |
| L | `>>` | bitwise shift right | expr >> expr | 152 |
| L | `<` | less than | expr < expr | 141 |
| L | `<=` | less than or equal | expr <= expr | 141 |
| L | `>` | greater than | expr > expr | 141 |

*Continued on next page*

**Table 4.4: Operator Precedence (continued)**

| Associativity and Operator | | Function | Use | See Page |
|---|---|---|---|---|
| L | >= | greater than or equal | expr >= expr | 141 |
| L | == | equality | expr == expr | 141 |
| L | != | inequality | expr != expr | 141 |
| L | & | bitwise AND | expr & expr | 152 |
| L | ^ | bitwise XOR | expr ^ expr | 152 |
| L | \| | bitwise OR | expr \| expr | 152 |
| L | && | logical AND | expr && expr | 141 |
| L | \|\| | logical OR | expr \|\| expr | 141 |
| R | ?: | conditional | expr ? expr : expr | 151 |
| R | = | assignment | lvalue = expr | 144 |
| R | *=, /=, %=, | compound assign | lvalue += expr, etc. | 144 |
| R | +=, -=, | | | 144 |
| R | <<=, >>=, | | | 144 |
| R | &=, \|=, ^= | | | 144 |
| R | throw | throw exception | throw expr | 193 |
| L | , | comma | expr , expr | 157 |

## CHAPTER 5 - STATEMENTS

Statements are executed sequentially. Therefore C++ also defines a set of ***flow-of-control*** statements that allow more complicated execution paths.

### Simple Statements

- Null Statements
  - The simplest statement is the empty statement, also known as **null statement.** A null statement is a single semicolon

```
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
    ; // null statement
```


Null statements should be commented. That way anyone reading the code can see that the statement was omitted intentionally.

        - Beware of Missing or Extraneous Semicolons
    - Compound Statements (Blocks)
        - Surrounded by curly brackets
        - Not terminated by semicolons
    - Statement Scope
        - Variables defined in the control structure are visible only within that statement and are out of scope after the statement ends
- Conditional Statements
    - **If and switch statements** allow for conditional execution
        - If statement determines the flow of control based on a condition
        - Switch statement evaluates an integral expression and chooses one of several execution paths based on the expression's value
    - Switch statement
        - convenient way of selecting among a large number of fixed alternatives

We can solve our problem most directly using a `switch` statement:

```
// initialize counters for each vowel
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// print results
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

- Control Flow within a *switch*
  - After a case label is matched, execution starts at that label and continues across all the remaining cases or until the program explicitly interrupts it.
  - To avoid executing code for subsequent cases, we must explicitly tell the compiler to stop execution using **break**


Omitting a `break` at the end of a `case` happens rarely. If you do omit a `break`, include a comment explaining the logic.

  - The *default* Label
    - The default label are executed when no *case label matches the value of the* switch expression

```
        // if ch is a vowel, increment the appropriate counter
        switch (ch) {
            case 'a': case 'e': case 'i': case 'o': case 'u':
                ++vowelCnt;
                break;
            default:
                ++otherCnt;
                break;
        }
    }
```

we'll increment `otherCnt`.


It can be useful to define a `default` label even if there is no work for the `default` case. Defining an empty `default` section indicates to subsequent readers that the case was considered.

- Range *for* Statement
  - In C++ 11 new syntactic form of the range for statement is

```
for (declaration : expression)
    statement
```

  - Note: The easiest way to ensure types match is to use **auto** type specifie

We have already seen several such loops, but for completeness, here is one that doubles the value of each element in a `vector`:

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// range variable must be a reference so we can write to the elements
for (auto &r : v)    // for each element in v
    r *= 2;          // double the value of each element in v
```

The `for` header declares the loop control variable, `r`, and associates it with `v`. We use `auto` to let the compiler infer the correct type for `r`. Because we want to change the value of the elements in `v`, we declare `r` as a reference. When we assign to `r` inside the loop, that assignment changes the element to which `r` is bound.

A range `for` is defined in terms of the equivalent traditional `for`:

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // r must be a reference so we can change the element
    r *= 2;         // double the value of each element in v
}
```

- The for header declares the loop control variable ,r, and associates it with v. We can use auto to let the compiler infer the correct type of r. Because we want to change the value of the

elements in v, we declare r as a reference. When we assign to r inside the loop, that assignment changes the element to which r is bound

- Note: We can now understand **why** we said that we cannot use a range *for* to add elements to a *vector (or other container)*.

A range `for` is defined in terms of the equivalent traditional `for`:

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; //  r must be a reference so we can change the element
    r *= 2;         //  double the value of each element in v
}
```

In a range for, the value of *end()* is cached. If we add elements to or remove them from the sequence, the value of `end` might be invalidate.

- The `do while` statement
  - is like a while but the condition is tested after the statement body completes. Regardless of the value of the condition, **we execute the loop at least once**

```
do
        statement
while (condition);
```


A `do while` ends with a semicolon after the parenthesized condition.

  - condition **cannot be empty**
  - If the conditions evaluates **false** the loop terminates, otherwise the loop is repeated

We can write a program that (indefinitely) does sums using a `do while`:

```
// repeatedly ask the user for a pair of numbers to sum
string rsp;  // used in the condition; can't be defined inside the do
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin  >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
         << " = " << val1 + val2 << "\n\n"
         << "More? Enter yes or no: ";
    cin  >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

- **Jump Statements**
  - Jump statements interrupt the flow of execution
  - C++ **offers 4 jumps:** `break, continue, goto and return`
  - A `break` can appear only within an iteration statement or `switch` statement (including inside statements or blocks nested inside such loops.

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // process up to the first blank
        for (auto it = buf.begin()+1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #1, leaves the for loop
            // ...
        }
        // break #1 transfers control here
        // remaining '-' processing:
        break; // #2, leaves the switch statement
```



```
    case '+':
        // ...
    } // end switch
  // end of switch: break #2 transfers control here
} // end while
```

  - The `continue` statements terminates the current iteration of the nearest enclosing loop and immediately begins the next iteration.

- The `goto` statement: provides an unconditional jump from the goto to another statement in the same function
  - Programs should not use gotos makes programs hard to understand and hard to modify

> The syntactic form of a `goto` statement is
>
> ```
> goto label;
> ```
>
> where *label* is an identifier that identifies a statement. A **labeled statement** is any statement that is preceded by an identifier followed by a colon:
>
> ```
> end: return;   // labeled statement; may be the target of a goto
> ```

- **`try` Blocks and Exception Handling**
  - Exceptions are run-time anomalies - that exist outside the normal functioning of a program. Dealing with anomalous behavior can be one of the most difficult parts of designing any system

> - **`throw` expressions**, which the detecting part uses to indicate that it encountered something it can't handle. We say that a `throw` **raises** an exception.
> - **`try` blocks**, which the handling part uses to deal with an exception. A `try` block starts with the keyword `try` and ends with one or more **`catch` clauses**. Exceptions thrown from code executed inside a `try` block are usually handled by one of the `catch` clauses. Because they "handle" the exception, `catch` clauses are also known as **exception handlers**.
> - A set of **`exception` classes** that are used to pass information about what happened between a `throw` and an associated `catch`.

  - **A `throw` Expression**
    - Uses this expression to raise an exception

```
// first check that the data are for the same item
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// if we're still here, the ISBNs are the same
cout << item1 + item2 << endl;
```

    - Throwing an exception terminates the current function and transfers control to a handler that will know hot to handle the error. (eg the type `runtime_error` is defined in the `stdexcept` header
  - **A `try` Block**

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

- Following the try block is a list of one or more catch clauses.
- When a catch is selected to handle an exception, the associated block is executed. Once the catch finishes, execution continues with the statement immediately folloung the last `catch` clause of the try block

- Writing a Handler

```
while (cin >> item1 >> item2) {
    try {
        // execute code that will add the two Sales_items
        // if the addition fails, the code throws a runtime_error exception
    } catch (runtime_error err) {
        // remind the user that the ISBNs must match and prompt for another pair
        cout << err.what()
             << "\nTry Again?  Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break;       // break out of the while loop
    }
}
```

- Functions are exited during the search for a handler
  - In complicated systems, the execution path of a program may pass through multiple `try` blocks before encountering code that throws and exception
  - The search for a handler reverses the call chain. When an exception is thrown the function that threw the exception is searched first.
    - If no matching `catch` is found the function terminates
    - The function that called the one that threw is searched next.
    - If not handler is found, the function also exists

- That function's caller is searched next and so on back up the execution path until a `catch` of an appropriate type is found.
- If not appropriate catch is found, execution is transfered to a library function names `terminate`

> **CAUTION: WRITING EXCEPTION SAFE CODE IS *Hard***
>
> It is important to realize that exceptions interrupt the normal flow of a program. At the point where the exception occurs, some of the computations that the caller requested may have been done, while others remain undone. In general, bypassing part of the program might mean that an object is left in an invalid or incomplete state, or that a resource is not freed, and so on. Programs that properly "clean up" during exception handling are said to be *exception safe*. Writing exception safe code is surprisingly hard, and (largely) beyond the scope of this language Primer.
>
> Some programs use exceptions simply to terminate the program when an exceptional condition occurs. Such programs generally don't worry about exception safety.
>
> Programs that do handle exceptions and continue processing generally must be constantly aware of whether an exception might occur and what the program must do to ensure that objects are valid, that resources don't leak, and that the program is restored to an appropriate state.
>
> We will occasionally point out particularly common techniques used to promote exception safety. However, readers whose programs require robust exception handling should be aware that the techniques we cover are insufficient by themselves to achieve exception safety.

- Standard Exceptions
  - The C++ library defines several classes that is uses to report problems encountered in the function int he standard libary.

    - The `exception` header defines the most general kind of exception class named `exception`. It communicates only that an exception occurred but provides no additional information.
    - The `stdexcept` header defines several general-purpose exception classes, which are listed in Table 5.1.
    - The `new` header defines the `bad_alloc` exception type, which we cover in § 12.1.2 (p. 458).
    - The `type_info` header defines the `bad_cast` exception type, which we cover in § 19.2 (p. 825).

| Table 5.1: Standard Exception Classes Defined in `<stdexcept>` | |
|---|---|
| `exception` | The most general kind of problem. |
| `runtime_error` | Problem that can be detected only at run time. |
| `range_error` | Run-time error: result generated outside the range of values that are meaningful. |
| `overflow_error` | Run-time error: computation that overflowed. |
| `underflow_error` | Run-time error: computation that underflowed. |
| `logic_error` | Error in the logic of the program. |
| `domain_error` | Logic error: argument for which no result exists. |
| `invalid_argument` | Logic error: inappropriate argument. |
| `length_error` | Logic error: attempt to create an object larger than the maximum size for that type. |
| `out_of_range` | Logic error: used a value outside the valid range. |

## CHAPTER 6: FUNCTIONS

A function is a block of code with name.

- Function Basics
  - We execute a function through the **call operator** which is a pair of parentheses
  - Inside the parenthesis is a comm-separated list of **arguments**
  - The arguments are used to initialise the functions's parameters.

  A function call does two things:

    - It initialises the function's parameters from the corresponding arguments and it transfers control to that function

  - Parameters and Arguments
    - Arguments are the initialisers for a function's parameters. The first argument is the first parameter, the second arguments initialises the second etc
    - But we do not know the order in which arguments are evaluated

```
fact("hello");        // error: wrong argument type
fact();               // error: too few arguments
fact(42, 10, 0);      // error: too many arguments
fact(3.14);           // ok: argument is converted to int
```

  - Function Parameter List

        - A function's parameter list can be empty but cannot be omitted
        - we can use `void` to indicate that there are no parameters

```
void f1(){ /*  ... */ }      // implicit void parameter list
void f2(void){ /*  ... */ } // explicit void parameter list
```

        - Even though when the types of two parameters are the same, the type must be repeated.

```
int f3(int v1, v2) { /* ... */ }      // error
int f4(int v1, int v2) { /* ... */ } // ok
```

        - Note that parameter names are optional
    - A function return type
        - Most types can be used to return type of a function
        - **Except array type or function type**
        - Void does not return a value
- Local Objects
    - In C++, names have scope and objects have **lifetimes**. It is important to understand both of these concepts
        - The scope of a name is *the part of the program's text* in which that name is visible
        - The lifetime of an object is *the time during the program's execution* that the object exists
        - Parameters and variables defined inside a function body are refereed to as **local variables**
        - Objects created when the program start and are not destroyed until the program ends
    - Automatic Objects
        - The objects that correspond to ordinary local variables are created when the function's control path passes through the variable's definition.

- Objects that exist only while a block is executing are known as **automatic objects**
  - After execution exits a block, the values of the automatic objects created in that block are undefined.
- **Parameters are automatic objects.** Storage for the parameters is allocated when the function begins. Parameters are defined in the scope of the function body. Hence they are destroyed when the function terminates
- **Local `static` Objects**
  - It can be useful to have a local variable whose lifetime continues across calls to the function. → We obtain such objects by defining a local variables as `static` .
    - Each local `static` object is initialised before the *first time* execution passes though the object's definition.
    - **Local statics are not destroyed when the function ends**

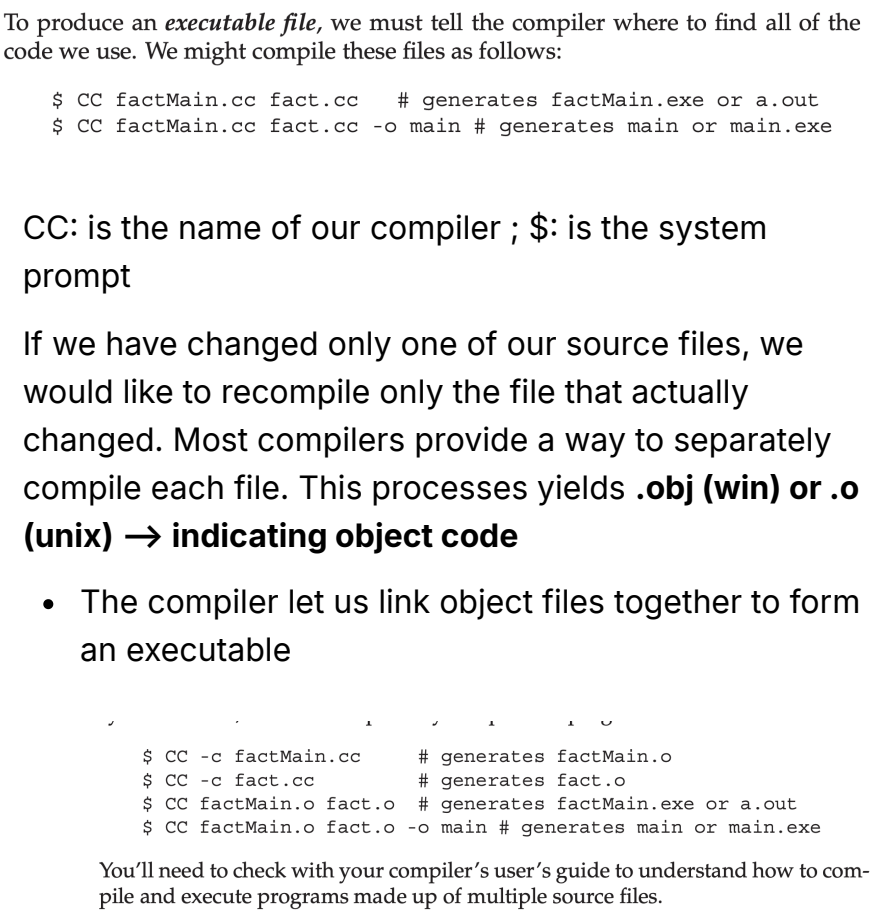
```
size_t count_calls()
{
    static size_t ctr = 0;   //  value will persist across calls
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

      - Each call increments *ctr* and return its new value.
  - Function Declarations
    - **Function declaration should be in header files and defined in source files**


The header that *declares* a function should be included in the source file that *defines* that function.

  - Separate Complication

- To allow programs t be written in logical parts, C++ supports what is commonly known as ***separate compilation:*** *Let us split our programs into several files, each of which can be compiled indepentenlby.*
- Compiling and Linking multiple Source Files



To produce an ***executable file,*** we must tell the compiler where to find all of the code we use. We might compile these files as follows:

```
$ CC factMain.cc fact.cc   # generates factMain.exe or a.out
$ CC factMain.cc fact.cc -o main # generates main or main.exe
```

  - CC: is the name of our compiler ; $: is the system prompt
  - If we have changed only one of our source files, we would like to recompile only the file that actually changed. Most compilers provide a way to separately compile each file. This processes yields **.obj (win) or .o (unix) → indicating object code**
    - The compiler let us link object files together to form an executable

```
$ CC -c factMain.cc     # generates factMain.o
$ CC -c fact.cc         # generates fact.o
$ CC factMain.o fact.o  # generates factMain.exe or a.out
$ CC factMain.o fact.o -o main # generates main or main.exe
```

You'll need to check with your compiler's user's guide to understand how to compile and execute programs made up of multiple source files.

- Argument Passing
  - When a parameter is a reference, we say that its corresponding argument is "**passed by reference"** or that the function is **"called by reference"** → meaning is an alias for the object to which it is bound
  - When the argument value is copied, the parameter and argument are independent objects. We say such arguments are **"passed by value"** or alternatively the function is "**called by value"**
  - Passing Arguments by Value

```
int n = 0;              // ordinary variable of type int
int i = n;              // i is a copy of the value in n
i = 42;                 // value in i is changed; n is unchanged
```

Passing an argument by value works exactly the same way; nothing the function does to the parameter can affect the argument. For example, inside `fact` (§ 6.1, p. 202) the parameter `val` is decremented:

```
ret *= val--;  // decrements the value of val
```

Although `fact` changes the value of `val`, that change has no effect on the argument passed to `fact`. Calling `fact(i)` does not change the value of `i`.

- Pointer Parameters
  - Pointers behave like any other non-reference type. When we copy a pointer the value of the pointer is copied. **After the copy, the two pointers are distinct → but we have indirect access to the object**

```
int n = 0, i = 42;
int *p = &n, *q = &i; // p points to n; q points to i
*p = 42;              // value in n is changed; p is unchanged
p = q;                // p now points to i; values in i and n are unchanged
```

The same behavior applies to pointer parameters:

```
// function that takes a pointer and sets the pointed-to value to zero
void reset(int *ip)
{
    *ip = 0;  // changes the value of the object to which ip points
    ip = 0;   // changes only the local copy of ip; the argument is unchanged
}
```

After a call to `reset`, the object to which the argument points will be `0`, but the pointer argument itself is unchanged:

```
int i = 42;
reset(&i);                      // changes i but not the address of i
cout << "i = "  << i << endl;  // prints i = 0
```


Programmers accustomed to programming in C often use pointer parameters to access objects outside a function. In C++, programmers generally use reference parameters instead.

- Passing Arguments by Reference

```
int n = 0, i = 42;
int &r = n;             // r is bound to n (i.e., r is another name for n)
r = 42;                 // n is now 42
r = i;                  // n now has the same value as i
i = r;                  // i has the same value as n
```

- Reference parameters exploit this behaviour. They are often used to allow a function to change the value of one or more of its arguments.

As one example, we can rewrite our `reset` program from the previous section to take a reference instead of a pointer:

```
// function that takes a reference to an int and sets the given object to zero
void reset(int &i)   // i is just another name for the object passed to reset
{
    i = 0;   // changes the value of the object to which i refers
}
```

- ## Using References to avoid Copies
  - It is inefficient to copy object of a large class type or large containers

As an example, we'll write a function to compare the length of two `string`s. Because `string`s can be long, we'd like to avoid copying them, so we'll make our parameters references. Because comparing two `string`s does not involve changing the `string`s, we'll make the parameters references to `const` (§ 2.4.1, p. 61):

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

As we'll see in § 6.2.3 (p. 213), functions should use references to `const` for reference parameters they do not need to change.


Reference parameters that are not changed inside a function should be references to `const`.

- ## Using Reference Parameters to Return Additional Information
  - A function can return **only** a single value
  - Reference parameters let us effectively return multiple results

```
// returns the index of the first occurrence of c in s
// the reference parameter occurs counts how often c occurs
string::size_type find_char(const string &s, char c,
                            string::size_type &occurs)
{
    auto ret = s.size();    // position of the first occurrence, if any
    occurs = 0;             // set the occurrence count parameter
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;    // remember the first occurrence of c
            ++occurs;       // increment the occurrence count
         }
    }
    return ret;             // count is returned implicitly in occurs
}
```

  - `const` Parameters and Arguments

```
const int ci = 42;     // we cannot change ci; const is top-level
int i = ci;            // ok: when we copy ci, its top-level const is ignored
int * const p = &i;    // const is top-level; we can't assign to p
*p = 0;                // ok: changes through p are allowed; i is now 0
```

- Top-level consts are ignored, as a result, top-level const on parameters are ignored. We can pass either a **const or a nonconst** object to a parameter that has a top level const:

```
void fcn(const int i) { /* fcn can read but not write to i */ }
```

  - We can call fcn passing it other a const int or a plain int.

```
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* ... */ }  // error: redefines fcn(int)
```

  - We can define several different functions that have the same name. Because top-level

- Pointer or Reference Parameters and const

```
int i = 42;
const int *cp = &i; // ok: but cp can't change i (§ 2.4.2 (p. 62))
const int &r = i;   // ok: but r can't change i (§ 2.4.1 (p. 61))
const int &r2 = 42; // ok: (§ 2.4.1 (p. 61))
int *p = cp;  // error: types of p and cp don't match (§ 2.4.2 (p. 62))
int &r3 = r;  // error: types of r3 and r don't match (§ 2.4.1 (p. 61))
int &r4 = 42; // error: can't initialize a plain reference from a literal (§ 2.3.1 (p. 50))
```

- Array Parameters
  - Arrays have 2 special properties that affect how we define and use function that operate on arrays:
    - We cannot copy an array → **cannot pass an array by value**
    - When we use an array it is (usually) converted to a pointer → actually passing a pointer to the array's first element

    Because arrays are passed as pointers, functions ordinarily **don't** know the size of the array are given. They must rely on additional information provided by the caller.

    **3 common techniques used to manage pointer parameters**

- Using a Marker to specify the extend of an Array
  - Requires the array itself to contain an end marker

```
void print(const char *cp)
{
    if (cp)           //  if cp is not a null pointer
        while (*cp)   //  so long as the character it points to is not a null character
            cout << *cp++;   //  print the character and advance the pointer
}
```

This convention works well for data where there is an obvious end-marker value (like the null character) that does not appear in ordinary data. It works less well with data, such as `int`s, where every value in the range is a legitimate value.

    eg: C-style characters is an example of this approach
- Using the Standard Library Conventions
  - Pass pointers to the first and one past the last element in the array
- Explicitly Passing a Size Parameter
  - Define a second parameter that indicates the size of the array
- Array parameters and const
  - All three versions of print function defined their array parameters as pointers to `const`
- Array Reference Parameters
  - We can define a parameter that is a reference to ana rray


The parentheses around `&arr` are necessary (§ 3.5.1, p. 114):

```
f(int &arr[10])    //  error: declares arr as an array of references
f(int (&arr)[10]) //  ok: arr is a reference to an array of ten ints
```

- Passing a Multidimensional Array
  - Recall that there are **no multidimensional arrays in C++ ==** Instead what appears to be a multidimensional array **is an array of arrays**

```
// matrix points to the first element in an array whose elements are arrays of ten ints
void print(int (*matrix)[10], int rowSize) { /* ... */ }
```

declares `matrix` as a pointer to an array of ten `int`s.


Again, the parentheses around `*matrix` are necessary:

```
int *matrix[10];   // array of ten pointers
int (*matrix)[10]; // pointer to an array of ten ints
```

- `main:` Handling Command-Line Options

  Such command-line options are passed to `main` in two (optional) parameters:

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

  - `argv` is an array of pointers to C-style character strings
  - `argc` passes the number of strings in that array

- **Functions with Varying Parameters**
  - Sometimes we don't know in advance how many arguments we need to pass to a function
    - 2 primary ways to write a function that takes a varying number of arguments:
      - If all arguments have the same type, we can pass the library type named `intializer_list`
      - If the argument types vary, we can write a special kind of function, known as a variadic template
      - C++ also has **special parameter type, ellipsis,** that can be sued to pass a varying number of arguments
  - `initializer_list` Parameters
    - We can write a function that takes an unknown number of arguments of a single type
    - initializer_list is a library type that represents an array of values of the specified type

| Table 6.1: Operations on `initializer_lists` | |
|---|---|
| `initializer_list<T> lst;` | Default initialization; an empty list of elements of type `T`. |
| `initializer_list<T> lst{a,b,c...};` | `lst` has as many elements as there are initializers; elements are copies of the corresponding initializers. Elements in the list are `const`. |
| `lst2(lst)`<br>`lst2 = lst` | Copying or assigning an `initializer_list` does not copy the elements in the list. After the copy, the original and the copy share the elements. |
| `lst.size()` | Number of elements in the list. |
| `lst.begin()`<br>`lst.end()` | Returns a pointer to the first and one past the last element in `lst`. |

```
initializer_list<string> ls; // initializer_list of strings
initializer_list<int> li;    // initializer_list of ints
```

Note: `initializer_list` are always `const` values unlike `vector` . Therefore there is no way to change the value of an element in this type of list

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

```
// expected, actual are strings
if (expected != actual)
    error_msg({"functionX", expected, actual});
else
    error_msg({"functionX", "okay"});
```

- Ellipsis Parameters
  - Are parameters in C++ to allow programs to interface to C code that uses a C library facility names `varargs`


Ellipsis parameters should be used only for types that are common to both C and C++. In particular, objects of most class types are not copied properly when passed to an ellipsis parameter.

```
void foo(parm_list, ...);
void foo(...);
```

- **Return Types and the `return` Statement**
  - A `return` statement terminates the function that is currently executing and returns control to the point from which the function was called.
  - Functions we No return value
    - A return with no value may be used only in a function that has a return type of void

```
void swap(int &v1, int &v2)
{
    //  if the values are already the same, no need to swap, just return
    if (v1 == v2)
        return;
    //  if we're here, there's work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    //  no explicit return necessary
}
```

This use of `return` is analogous to use of `break`

- Functions that return a value
  - Provides the functions result
- Never return a reference or Pointer to a Local Object
  - When a function completes, its storage is freed. After a function terminates, references to local object refer to memory that is no longer valid

```
//  disaster: this function returns a reference to a local object
const string &manip()
{
    string ret;
    //  transform ret in some way
    if (!ret.empty())
        return ret;      //  WRONG: returning a reference to a local object!
    else
        return "Empty"; //  WRONG: "Empty" is a local temporary string
}
```

Both return an **undefined** value ; refer to memory that is no longer available


One good way to ensure that the return is safe is to ask: To what *preexisting* object is the reference referring?

For the same reasons that it is wrong to return a reference to a local object, it is also wrong to return a pointer to a local object. Once the function completes, the local objects are freed. The pointer would point to a nonexistent object.

- Functions that Return Class Types and the Call operator
  - Like any other operator the call operator has **associativity and precedence (left associative ; use dot(.) and (→) arrow**
  - References returns are Lvalues

- Whether a function call is an lvalue depends on the return type of the function

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix]; // get_val assumes the given index is valid
}
int main()
{
    string s("a value");
    cout << s << endl;    // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A
    cout << s << endl;    // prints A value
    return 0;
}
```

    - **We can see a function call on the left-hand side of an assignment.** However nothing special is involved. The return value is a reference, so the call is an lvalue.

- List Initialising the return value
    - **C++ 11 allows functions to return a braced list of values**

```
vector<string> process()
{
    // ...
    // expected and actual are strings
    if (expected.empty())
        return {};  // return an empty vector
    else if (expected == actual)
        return {"functionX", "okay"}; // return list-initialized vector
    else
        return {"functionX", expected, actual};
}
```

    - Return from main
        - Return from `main` is treated as a status indicator. A 0 indicates success; most others indicate failure. A non-zero value has a machine-dependent meaning
        - To make return values machine independent, the `cstdlib` header defines two preprocessor variables that we use to indicate success or failure

```
int main()
{
    if (some_failure)
        return EXIT_FAILURE;  // defined in cstdlib
    else
        return EXIT_SUCCESS;  // defined in cstdlib
}
```

Because these are preprocessor variables, we must not precede them with `std::`, nor may we mention them in `using` declarations.

- **Recursion**
  - A function that call itself either directly or indirectly, is a ***recursive function*.**

```
// calculate val!, which is 1 * 2 * 3 . . . * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

**Trace of `factorial(5)`**

| Call | Returns | Value |
|---|---|---|
| `factorial(5)` | `factorial(4) * 5` | 120 |
| `factorial(4)` | `factorial(3) * 4` | 24 |
| `factorial(3)` | `factorial(2) * 3` | 6 |
| `factorial(2)` | `factorial(1) * 2` | 2 |
| `factorial(1)` | `1` | 1 |


The `main` function may *not* call itself.

- Returning a Pointer to an Array
  - Because we cannot copy an array, a function cannot return an arrayed code. **However** we can return a **pointer or reference to an array**

```
typedef int arrT[10]; // arrT is a synonym for the type array of ten ints
using arrT = int[10]; // equivalent declaration of arrT; see § 2.5.1 (p. 68)
arrT* func(int i);    // func returns a pointer to an array of ten ints
```

Here `arrT` is a synonym for an array of ten `int`s. Because we cannot return an array, we define the return type as a pointer to this type. Thus, `func` is a function that takes a single `int` argument and returns a pointer to an array of ten `int`s.

  - Declaring a Function that returns a Pointer to an Array

*Type* `(*`*function*`(`*parameter_list*`))[`*dimension*`]`

As in any other array declaration, *Type* is the type of the elements and *dimension* is the size of the array. The parentheses around `(*`*function*`(`*parameter_list*`))` are necessary for the same reason that they were required when we defined `p2`. Without them, we would be defining a function that returns an array of pointers.

As a concrete example, the following declares `func` without using a type alias:

```
int (*func(int i))[10];
```

To understand this declaration, it can be helpful to think about it as follows:

- `func(int)` says that we can call `func` with an `int` argument.
- `(*func(int))` says we can dereference the result of that call.
- `(*func(int))[10]` says that dereferencing the result of a call to `func` yields an array of size ten.
- `int (*func(int))[10]` says the element type in that array is `int`.

- Using a Trailing Return Type
  - C++ 11 new standard to simplify declaration of `func` is by using a **trailing return type.**
  - Follows the parameter list and is preceded by →
  - To signal that the return follows the parameter list, we use `auto` where the return type ordinarily appears

```
// fcn takes an int argument and returns a pointer to an array of ten ints
auto func(int i) -> int(*)[10];
```

Because the return type comes after the parameter list, it is easier to see that `func` returns a pointer and that that pointer points to an array of ten `int`s.

  - using `decltype`
    - As an alternative, if we know the arrays to which our function can return a pointer

```
int odd[] = {1,3,5,7,9};
int even[] = {0,2,4,6,8};
// returns a pointer to an array of five int elements
decltype(odd) *arrPtr(int i)
{
    return (i % 2) ? &odd : &even; // returns a pointer to the array
}
```

- Overloaded Functions
  - Functions that have the same name but different parameter lists and that appear in the same scope are **overloaded.**

```
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

- These function perform the same general action but apply to different parameter types.
- The compiler can deduce which function we want based on the argument type we pass:

```
int j[2] = {0,1};
print("Hello World");         // calls print(const char*)
print(j, end(j) - begin(j)); // calls print(const int*, size_t)
print(begin(j), end(j));      // calls print(const int*, const int*)
```

Function overloading eliminates the need to invent—and remember—names that exist only to help the compiler figure out which function to call.


The `main` function may *not* be overloaded.

- Defining Overloaded Functions - pg260
  - `const_cast` and Overloading

**ADVICE: WHEN NOT TO OVERLOAD A FUNCTION NAME**

**Although overloading lets us avoid having to invent (and remember) names for common operations, we should only overload operations that actually do similar things. There are some cases where providing different function names adds information that makes the program easier to understand. Consider a set of functions that move the cursor on a `Screen`.**

```
Screen& moveHome();
Screen& moveAbs(int, int);
Screen& moveRel(int, int, string direction);
```

**It might at first seem better to overload this set of functions under the name `move`:**

```
Screen& move();
Screen& move(int, int);
Screen& move(int, int, string direction);
```

**However, by overloading these functions, we've lost information that was inherent in the function names. Although cursor movement is a general operation shared by all these functions, the specific nature of that movement is unique to each of these functions. `moveHome`, for example, represents a special instance of cursor movement. Whether to overload these functions depends on which of these two calls is easier to understand:**

```
// which is easier to understand?
myScreen.moveHome(); // we think this one!
myScreen.move();
```

- Calling an Overloaded Function
  - **Function matching (overload resolution)** is the process by which a particular function call is associated with a specific function from set of overloaded functions.

- The compiler finds exactly one function that is a **best match** for the actual arguments and generates code to call that function.
- There is no function with parameters that match the arguments in the call, in which case the compiler issues an error message that there was **no match**.
- There is more than one function that matches and none of the matches is clearly best. This case is also an error; it is an **ambiguous call**.

- Overloading and Scope


Ordinarily, it is a bad idea to declare a function locally. However, to explain how scope interacts with overloading, we will violate this practice and use local function declarations.

```
string read();
void print(const string &);
void print(double);    // overloads the print function
void fooBar(int ival)
{
    bool read = false; // new scope: hides the outer declaration of read
    string s = read(); // error: read is a bool variable, not a function
    // bad practice: usually it's a bad idea to declare functions at local scope
    void print(int);   // new scope: hides previous instances of print
    print("Value: "); // error: print(const string &) is hidden
    print(ival);       // ok: print(int) is visible
    print(3.14);       // ok: calls print(int); print(double) is hidden
}
```

- When the compiler process the call to `read` , it find the local definition of `read` . That name is a `bool` variable, and we cannot call a `bool` . Hence the call is illefa

```
void print(const string &);
void print(double); // overloads the print function
void print(int);    // another overloaded instance
void fooBar2(int ival)
{
    print("Value: "); // calls print(const string &)
    print(ival);      // calls print(int)
    print(3.14);      // calls print(double)
}
```

- Features for Specialised Uses
  - Function related features which some of them can be used during debugging
  - Default argument
    - Some function have parameters that are given a particular value in most, but not all calls → in such cases we can declare that common value as **default argument** for a function
    - To accommodate both default and specified values we would declare our function to define the window as follows:

```
typedef string::size_type sz;   //  typedef see § 2.5.1 (p. 67)
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
```

- If we want to use the default argument, we omit that argument when we call the function

```
string window;
window = screen();   //  equivalent to screen(24,80,' ')
window = screen(66);//  equivalent to screen(66,80,' ')
window = screen(66, 256);       //  screen(66,256,' ')
window = screen(66, 256, '#'); //  screen(66,256,'#')
```

  - Arguments in the call are resolved by position.
  - The default are used for the trailing (right-most) argument of a call

```
window = screen(, , '?'); //  error: can omit only trailing arguments
window = screen('?');     //  calls screen('?',80,' ')
```

    - In this call, the char argument is implicitly converted to `string::size_type` and is passed as the argument to `height.` On our machine '?' has the hex value of 0x3F, which is decimal 63 → thus this call passes 63 to the height parameter

- Note: Part of the work of designing a function with default arguments is ordering the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last

- Default Argument Declarations

```
//  no default for the height or width parameters
string screen(sz, sz, char = ' ');
```

we cannot change an already declared default value:

```
string screen(sz, sz, char = '*'); //  error: redeclaration
```

but we can add a default argument as follows:

```
string screen(sz = 24, sz = 80, char);   //  ok: adds default arguments
```

Best Practices

Default arguments ordinarily should be specified with the function declaration in an appropriate header.

- Default Argument Initialiser

- Local variables may not be used as a default argument

```
// the declarations of wd, def, and ht must appear outside a function
sz wd = 80;
char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
string window = screen(); // calls screen(ht(), 80, ' ')
```

Names used as default arguments are resolved in the scope of the function declaration. The value that those names represent is evaluated at the time of the call:

```
void f2()
{
    def = '*';    // changes the value of a default argument
    sz wd = 100; // hides the outer definition of wd but does not change the default
    window = screen(); // calls screen(ht(), 80, '*')
}
```

## Inline and `constexpr` Functions

- Calling a function is apt to be slower than evaluating the equivalent expression
- `inline` Function avoid Function Call Overhead
  - A function specified as *`inline`* *(in line) at each call*

```
cout << shorterString(s1, s2) << endl;
```

(probably) would be expanded during compilation into something like

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

  - The run time overhead of making shorterString a function is thus removed

```
// inline version: find the shorter of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
        return s1.size() <= s2.size() ? s1 : s2;
}
```


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

## `constexpr` Function

- is a function that can be used in a constant expression

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz();  // ok: foo is a constant expression
```

        - Conditions: The `return` type and the type of each parameter in a must be a literal type
        - Function body must contain exactly oen `return` statement
        - constexpr is implicitly inline

    Note: Unlike other functions, `inline & constexpr` functions may be defined multiple times in the program as a result they are normally defined in headers

- **Aids for Debugging - pg 270**
    - Using two preprocessor facilities: `assert & NDEBUG`
    - Assert: defined in `cassert header`
        - is a **preprocessor macro:** meaning is a preprocessor variable that acts somewhat like an inline function

        ```
        assert(expr);
        ```

        - if the expression is false (ie zero) then assert writes a message and terminates the program
        - It is often used to check for conditions that "cannot happen".
    - The `NDEBUG` Preprocessor Variable
        - Is NDEBUG is defined, assert does nothing
        - By default, `NDEBUG` is **NOT** defined, so , by default assert performs a run-time check

        We can "turn off" debugging by providing a `#define` to define `NDEBUG`. Alternatively, most compilers provide a command-line option that lets us define preprocessor variables:

        ```
        $ CC -D NDEBUG main.C  # use /D with the Microsoft compiler
        ```

        has the same effect as writing `#define NDEBUG` at the beginning of `main.C`.

- Function Matching

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);  //  calls void f(double, double)
```

        - Determining the candidate and Viable function
            - The first step of a function matching identifies the set of overloaded functions considered for the call // functions in this set are the **candidate functions**
            - In the above example there are 4 candidate functions
    - Argument Type Conversions
        - In order to determine the best match, the compiler ranks the conversion that could be used to convert each argument to the type of tis corresponding parameter
- **Pointers to Functions**
    - A function pointer is just that-a pointer that denotes a function rather than an object

The function's name is not part of its type. For example:

```
//  compares lengths of two strings
bool lengthCompare(const string &, const string &);
```

has type `bool(const string&, const string&)`. To declare a pointer that can point at this function, we declare a pointer in place of the function name:

```
//  pf points to a function returning bool that takes two const string references
bool (*pf)(const string &, const string &);  //  uninitialized
```


The parentheses around `*pf` are necessary. If we omit the parentheses, then we declare `pf` as a function that returns a pointer to `bool`:

```
//  declares a function named pf that returns a bool*
bool *pf(const string &, const string &);
```

    - Using Function Pointer
        - When we use the name of a function as a value, the function is automatically converter to a pointer

```
pf = lengthCompare;  // pf now points to the function named lengthCompare
pf = &lengthCompare; // equivalent assignment: address-of operator is optional
```

### Using `auto` or `decltype` for Function Pointer Types

If we know which function(s) we want to return, we can use `decltype` to simplify writing a function pointer return type. For example, assume we have two functions, both of which return a `string::size_type` and have two `const string&` parameters. We can write a third function that takes a `string` parameter and returns a pointer to one of these two functions as follows:

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
// depending on the value of its string parameter,
// getFcn returns a pointer to sumLength or to largerLength
decltype(sumLength) *getFcn(const string &);
```

The only tricky part in declaring `getFcn` is to remember that when we apply `decltype` to a function, it returns a function type, not a pointer to function type. We must add a `*` to indicate that we are returning a pointer, not a function.

---

## CHAPTER 7: CLASSES

The fundamental idea behind **classes** are **data abstraction and encapsulation**

- Data abstraction is a programming & design technique that relies on the separation of **interface and implementation.**
- The interface of a class consist of the operations that users of the class can execute
- The implementation includes the class' data members, the bodies of the functions that constitute the interface and any function needed to define the class that are not intended for general use
- **Encapsulation:** enforces the separation of a class' interface and implementation.
    - A class that is encapsulated hides its implementation- users of the class can use the interface but have no access to the implementation

Note: **A class that uses data abstraction and encapsulation defines an abstract data type**.

- In abstract data types, the class designer worries about **how the class is implemented.** Programmers who use the class **do not need to know how the type works -** instead think *abstractly about what the type does*

- Defining Member Functions
  - Introducing `this`
    - Member function access the object on which they were called through an extra, implicit parameter names `this` . When we call a member function , this is initialised with the address of the object on which the function was invoked

> The `this` parameter is defined for us implicitly. Indeed, it is illegal for us to define a parameter or variable named `this`. Inside the body of a member function, we can use `this`. It would be legal, although unnecessary, to define `isbn` as
>
> ```
> std::string isbn() const { return this->bookNo; }
> ```
>
> Because `this` is intended to always refer to "this" object, `this` is a `const` pointer (§ 2.4.2, p. 62). We cannot change the address that `this` holds.

    - Introducing `const` Member Function
      - The purpose of that const is to modify the type of the implicit this pointer
      - We cannot call an ordinary member function on a const object


> Objects that are `const`, and references or pointers to `const` objects, may call only `const` member functions.

    - Defining a member function *outside* the class
      - As with any other function, when we define a member function outside the class body, the members definition must match its declaration → That is the return type, parameter list and name must match the declaration in the class body.

```
double Sales_data::avg_price() const {
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

        - here the member was declared as a const member function, then the definition must also specify const after the

parameter list.

- Defining a Function to return “this” object
  - The `combine` function is intended to act like the compound assignment operator += .

```
Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += rhs.units_sold; //  add the members of rhs into
    revenue += rhs.revenue;       //  the members of "this" object
    return *this; //  return the object on which the function was called
}
```



When our transaction-processing program calls

```
total.combine(trans);  //  update the running total
```

the address of `total` is bound to the implicit `this` parameter and `rhs` is bound to `trans`. Thus, when `combine` executes

```
units_sold += rhs.units_sold; //  add the members of rhs into
```

- Defining Nonmember Class-Related Function

```
//  input transactions contain ISBN, number of copies sold, and sales price
istream &read(istream &is, Sales_data &item)
{
    double price = 0;
    is >> item.bookNo >> item.units_sold >> price;
    item.revenue = price * item.units_sold;
    return is;
}
ostream &print(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
       << item.revenue << " " << item.avg_price();
    return os;
}
```

- **Constructors — more on ch 13**
  - Classes control object initialisation by defining one or more special member function known as **constructors** → the job of a constructor is to initialise the data members of a class object. A constructor is run whenever an object of a class type is created.

- Constructors have the same names as the class and no return type
- A class can have multiple constructors
- The Synthesised Default Constructor
  - eg

```
Sales_data total;      // variable to hold the running sum
Sales_data trans;      // variable to hold data for the next transaction
```

    - How are these initialised?
      - No supply of initialised therefore default initialised.
      - Classes control default initialisation initialisation by defining a special constructor, know as the **default constructor.[ takes no arguments]**

    Note: Only simple classes can rely on the synthesised default constructor. The compiler generates a default constructor automatically only if a class declares *no* constructors

> **What = `default` Means**
>
> We'll start by explaining the default constructor:
>
> ```
> Sales_data() = default;
> ```
>
> First, note that this constructor defines the default constructor because it takes no arguments. We are defining this constructor *only* because we want to provide other constructors as well as the default constructor. We want this constructor to do exactly the same work as the synthesized version we had been using.

- Constructor Initialiser List:

```
Sales_data(const std::string &s): bookNo(s) { }
Sales_data(const std::string &s, unsigned n, double p):
           bookNo(s), units_sold(n), revenue(p*n) { }
```

  - This new part is the **constructor initialiser list,** which specifies initial values for one or more data members of the object being created.
  - Basically is a list of member names either in parenthesis or braces


Constructors should not override in-class initializers except to use a different initial value. If you can't use in-class initializers, each constructor should explicitly initialize every member of built-in type.

- **Copy, Assignment and Destruction**
  - Classes also control what happens when we copy, assign, or destroy objects of the class type
- **Access Control and Encapsulation**
  - In C++ we use **access specifies** to enforce encapsulation:
    - Members defined after a **public specifier** are accessible to all parts of the program. The public members define the interface to the class
    - Members defined after **private specifier** are accessible to the member functions of the class but are not accessible to code that uses the class. The `private` sections encapsulate (ie hide) the implementation
- **Using the `class or struct` Keyword**
  - We use the class keyword rather than struct to open the class definition.
  - **The only difference between `struct` and `class` is the default access level**
  - Access to members depends on how the class is defined: `struct` keyword, the member defined before the first access specifier are `public` || `class` are `private`
- **Friends**
  - A class can allow another class or function to access its **nonpublic members** by making that class or function a **friend.**
  - A class makes a function its friend by including a declaration for that function preceded by the keyword friend

```
class Sales_data {
// friend declarations for nonmember Sales_data operations added
friend Sales_data add(const Sales_data&, const Sales_data&);
friend std::istream &read(std::istream&, Sales_data&);
friend std::ostream &print(std::ostream&, const Sales_data&);
// other members and access specifiers as before
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
               bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

**KEY CONCEPT: BENEFITS OF ENCAPSULATION**

**Encapsulation provides two important advantages:**

- **User code cannot inadvertently corrupt the state of an encapsulated object.**
- **The implementation of an encapsulated class can change over time without requiring changes in user-level code.**

**By defining data members as `private`, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what effect the change may have. User code needs to change only when the interface changes. If the data are `public`, then any code that used the old data members might be broken. It would be necessary to locate and rewrite any code that relied on the old representation before the program could be used again.**

**Another advantage of making data members `private` is that the data are protected from mistakes that users might introduce. If there is a bug that corrupts an object's state, the places to look for the bug are localized: Only code that is part of the implementation could be responsible for the error. The search for the mistake is limited, greatly easing the problems of maintenance and program correctness.**


**Although user code need not change when a class definition changes, the source files that use a class must be recompiled any time the class changes.**

- Class Types
  - Every class defines a unique type. Two different classes define two different types even if they define the same members.
  - Class Declaration

```
class Screen; // declaration of the Screen class
```

- declaration without definition

**ADVICE: USE CONSTRUCTOR INITIALIZERS**

**In many classes, the distinction between initialization and assignment is strictly a matter of low-level efficiency: A data member is initialized and then assigned when it could have been initialized directly.**

**More important than the efficiency issue is the fact that some data members must be initialized. By routinely using constructor initializers, you can avoid being surprised by compile-time errors when you have a class with a member that requires a constructor initializer.**

### 7.5.3 The Role of the Default Constructor

The default constructor is used automatically whenever an object is default or value initialized. Default initialization happens

- When we define nonstatic variables (§ 2.2.1, p. 43) or arrays (§ 3.5.1, p. 114) at block scope without initializers
- When a class that itself has members of class type uses the synthesized default constructor (§ 7.1.4, p. 262)
- When members of class type are not explicitly initialized in a constructor initializer list (§ 7.1.4, p. 265)

Value initialization happens

- During array initialization when we provide fewer initializers than the size of the array (§ 3.5.1, p. 114)
- When we define a local static object without an initializer (§ 6.1.1, p. 205)
- When we explicitly request value initialization by writing an expressions of the form `T()` where `T` is the name of a type (The `vector` constructor that takes a single argument to specify the `vector`'s size (§ 3.3.1, p. 98) uses an argument of this kind to value initialize its element initializer.)