* **T\*:** It is read as ‘pointer to T’. A variable of type T\* can hold the address of an object of type T.

A black rectangle with a black line

Description automatically generated

* **Dereferencing**: The fundamental operation on a pointer. Refers to the object pointed to by a pointer. Also called **indirection.**

A close-up of a message

Description automatically generated

* To store smaller values more compactly, one can use bitwise logical operations, bit-fields in structures, or a bitset.
* \* as a prefix is a dereferencing operator.
* \* as a suffix means ‘pointer to’ a type name.

A blue text on a white background

Description automatically generated

* **void\*:** Read as ‘pointer to an object of unknown type’. Used when we occasionally need to store or pass along an address of a memory location without actually knowing what type of object is stored there.
* A pointer to any type of *object* can be assigned to a variable of type *void\**, but a pointer to a *function* or a *pointer* to a member cannot.
* A *void* can be assigned to another void.
* *void\**s can be compared for equality and inequality.
* A *void\** can be explicitly converted to another type.
* In general, it is not safe to use a pointer that has been converted to a type that differs from the object being pointed to. Consequently, the notation used, *static\_cast* was designed to be ugly and easy to find in code.

A computer code with blue text

Description automatically generated

* Occurrences of *void\**s at higher levels of the system should be viewed with great suspicion because they are likely indicators of design errors.
* Before nullptr was used, 0 was used to denote nullptr.

A blue text on a white background

Description automatically generated

* Also, it has been popular to define a macro NULL to represent a null pointer. However, definitions of NULL are different in different implementations.
* In C, NULL is typically (void\*)0, which makes it illegal in C++.



* Using nullptr makes code more readable than alternatives and avoids confusion.
* **T[size]**: An array of size elements of type T.
* The elements are indexed from 0 to *size-1*.
* The number of elements in the array, the array bound, should be a constant expression.

Blue text on a white background

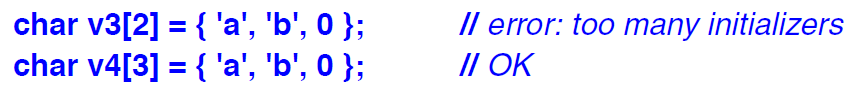
Description automatically generated

* If what one wants is a simple fixed-length sequence of objects of a given type in memory, an array is the ideal solution. For any other need, an array has serious problems.
* Arrays can be allocated statically, on the stack, or on free store.

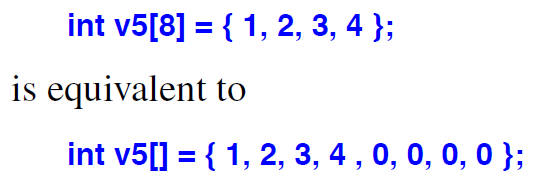
A close-up of a white background

Description automatically generated

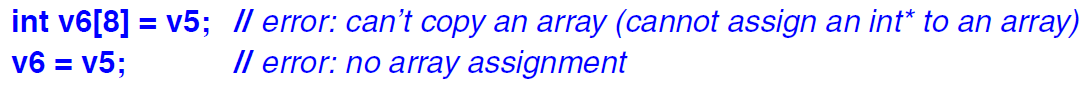
* Avoid arrays in interfaces, e.g. as function arguments, because implicit conversion to pointer is the root cause of many errors in C code and C-style C++ code.
* If an array is allocated on free store, be sure to *delete[]* its pointer once only and only after its last use.
* One of the most widely used kinds of arrays is the zero-terminated array of *char*, also called C-style string.
* Often a *char\** or a *const char\** is assumed to point to a zero-terminated sequence of characters, even in C++.
* An **array** can be **initialised** by a list of values.
* When an array is initialised without a specific size but with an initialiser list, the size of the array is calculated by counting the number of elements in the initialiser list.
* If the size is explicitly specified, it is an error to give surplus elements in an initialiser list.



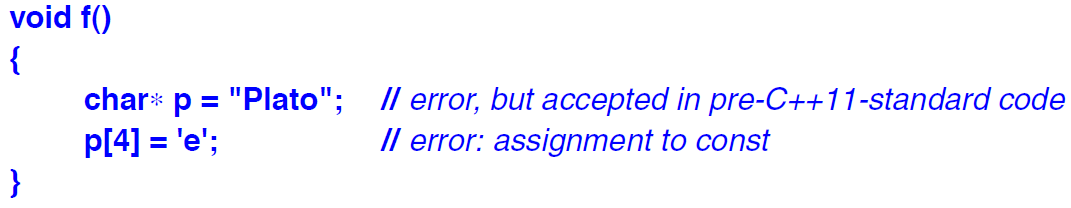
* If the initialiser supplies too few elements for the list, the rest is populated with zero.



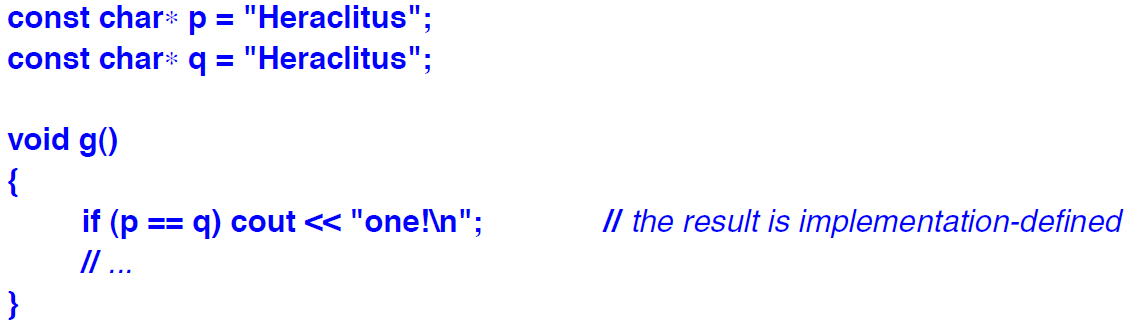
* There is no built-in copy operation for arrays.
* One array cannot be assigned with another, not even of exactly the same type.
* There is no array assignment.



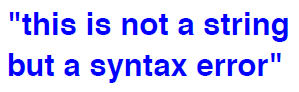
* Arrays cannot be passed by value.
* **String literal:** A character sequence enclosed within double quotes.
* A string literal contains one more character than it appears to have. It is terminated by the ‘\0’ character with a value 0.
* The type of string literal is ‘an array of appropriate number of const characters.’

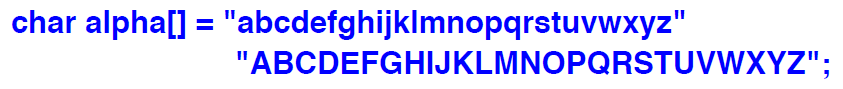


* Whether two identical string literals are allocated as one array or as two is implementation-defined.



* The empty string is represented as a pair of adjacent “” and has the type const char[1]. The one character is ‘\0’.
* The backslash convention makes it possible to use characters such as double quotes (“) or backslash (\) within a string.
* It is mostly used to represent non-graphic characters, such as \n (newline) or \a (alert with a beep sound).
* Long strings can be broken by white space to make the program neat. The first example is wrong whereas the second is correct.

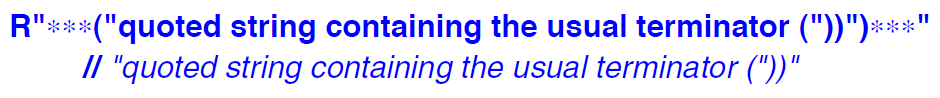




* It is possible to have null character in a string but most programs will ignore any characters after it. For example, “Khazad\0Dum” will be read as “Khazad” by functions such as strcpy() and strlen().
* **Raw string literal:** A string literal where a blackslash is just a backslash and a double quote is just a double quote.



* We can add personalised delimiters before and after () in a raw string.



* Unless one works with regular expressions, raw string literals are probably just a curiosity.

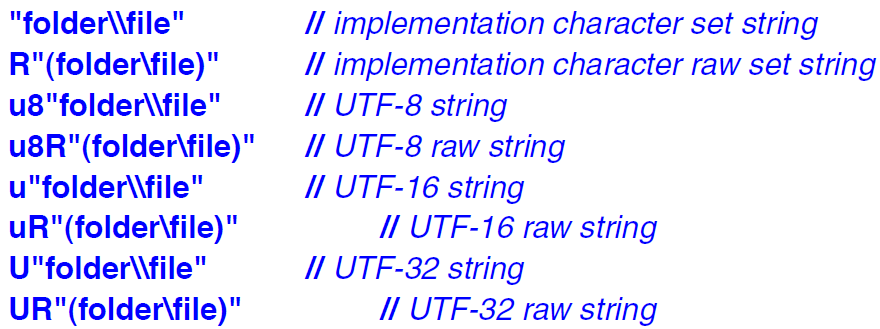


* In contrast to non-raw string literals, raw string literals can be spread over multiple lines.

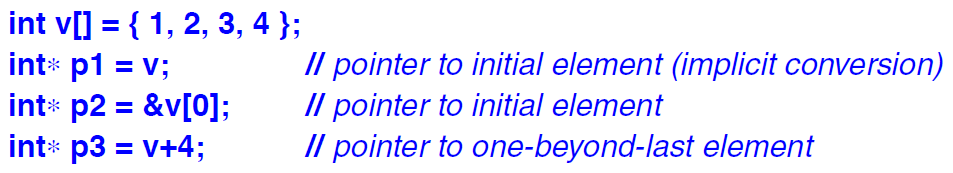
A white background with blue text

Description automatically generated

* **Larger Character Sets**
* A string with a prefix L, such as L”Dumbledore”, is a string of wide characters. Its type is *const wchar\_t[]*.
* A string with prefix LR, LR”(Gandalf)”, is a raw string of wide characters. Its type is also *const wchar\_t[]*.
* Such strings are terminated by L’\0’.
* There are 3 major encodings of Unicode: UTF-8, UTF-16, UTF-32.
* UTF is a variable length encoding. Common characters fit into 1 byte, less frequently used characters into 2 bytes, and rarer characters into 3 or 4 bytes.
* In particular, ASCII characters fit into 1 byte with the same encodings.
* The various Latin alphabets, Greek, Cyrillic, Hebrew, Arabic and more into 2 bytes.
* A UTF-8 string is terminated by ‘\0’. UTF-16 string by u’\0’. UTF-32 string by U’\0’.
* An ordinary English character can be represented in a variety of ways. All these strings will look the same, but the representations will differ.



* The order of u and R and their cases are significant when denoting raw Unicode strings.
* **Pointers into arrays:** The name of an array can be used as a pointer to its initial element.

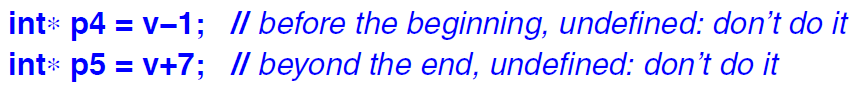


* Taking a pointer into the element one beyond the end of an array is guaranteed to work.

A diagram of a number

Description automatically generated

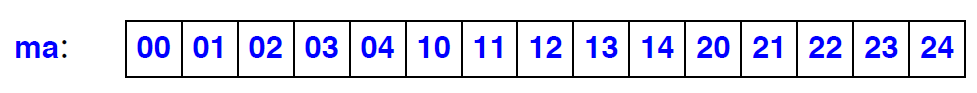
* The result of taking the address of the pointer before the first element of the array and after the one-past-the-last element of the array is undefined.



* The implicit conversion of the array argument to a pointer means that the size of the array is lost to the called function.
* This problem is handled by standard library types, such as vector, array, and string, which give number of elements as their *size()* without having to count the number of elements each time.
* **Navigating arrays:** Access can be achieved either through a pointer to an array plus and index, e.g. *a[i]*, or through a pointer to an element, e.g. *\*(p+2)*.
* No one version is faster than the other. It’s simply a matter of the programmer’s preference and aesthetics.
* For every integer array a and an integer j within the range a, we have –



* Equivalences like *a[j] == j[a]* are pretty low-level and do not hold for standard library functions such as *vector* and *array*.
* Subtraction of pointers is defined only when both pointers point to elements of the same array. q-p refers to the range [p:q) of the array, if the range is valid. Otherwise, it is undefined.
* One can add/subtract an integer from a pointer, resulting in a new pointer. E.g. *q = p+2*, *r = p-2*.
* The array concept is inherently low-level. Safer alternatives are *array* and *vector*.
* **Multi-dimensional arrays:** Represented as arrays of arrays.

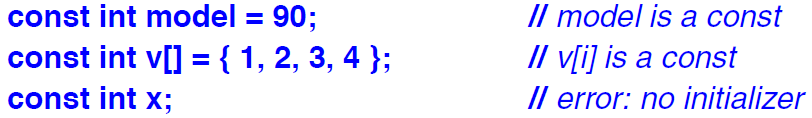


* Multi-dimensional arrays cannot be declared using commas in C++.

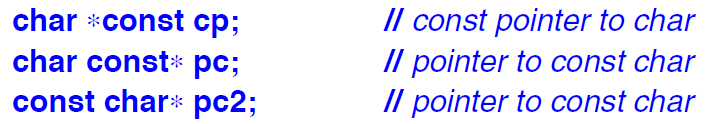
A close-up of blue text

Description automatically generated

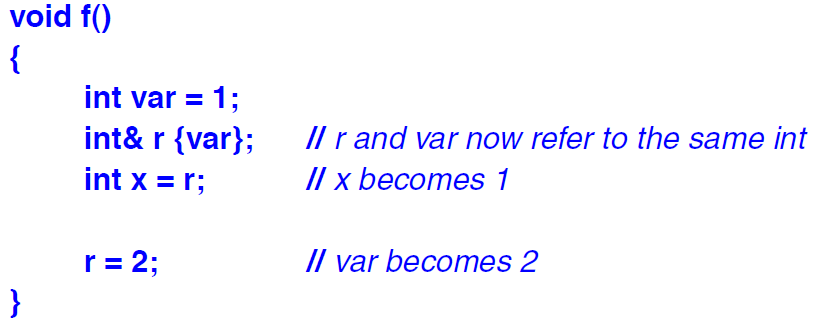
* **Passing arrays –**
* Arrays cannot be directly passed by value in functions.
* In order to avoid confusions related to dimensions, it is best to use the standard library *vector*.
* **Pointers and const –**
* Many objects don’t have their values changed during initialisation –
* Symbolic constants lead to more maintainable code, rather than using literals in the code directly.
* Many pointers are often read through but never written through.
* Most function parameters are often read but never written to.
* For all the above cases, *const* objects is a good choice.



* Since a *const* object cannot be assigned to later in the program, it must definitely be initialised during the declaration process.
* **Prefixing** a declaration of a pointer with *const* makes the object, but not the pointer, a constant.
* To declare a pointer itself, rather than the object pointed to, to be constant, we use *\*const*.



* **Pointers and ownership –**
* A resource is something that has to be acquired and later released. E.g. –
* Memory acquired by *new* and released by *delete*.
* Files opened by *fopen()* and closed by *fclose()*.
* This can be most confusing because a pointer can be passed around in the system and there is no way to distinguish between an pointer which is in ownership and a pointer which is not.
* The standard library unique\_ptr, vector, and string are the best to use in this regard.
* A pointer allows us to use potentially large amounts of data at a low cost.
* However, using a pointer differs from using the name of the object in various ways –
* Syntax used is different. E.g. \*p vs ob, p->m vs ob.m, etc.
* A pointer can be made to point to different objects at different times.
* A pointer can be a *nullptr* or point to an object that wasn’t expected.
* **Reference:** An alias for an object.
* A reference differs from a pointer in the following ways –
* A reference can be accessed exactly with the same syntax as the name of the object.
* A reference always refers to the object to which it was initialised.
* There is no ‘null reference’.
* The main use of references is for specifying arguments and return values for functions.
* There are three kinds of references –
* *lvalue reference*: To refer to objects whose value we want to change.
* *const reference*: To refer to objects whose value we do not want to change.
* *rvalue reference:* To refer to objects whose value we do not need to preserve after we have used it.
* Notation X& means **‘lvalue reference of X’.**



* To ensure that a reference is a name for something, we must initialise the reference. Otherwise, it will throw an error.
* No operator operates on a reference.

`A close-up of a white background

Description automatically generated

* ++r does not increment the reference itself. It increments the value to which the reference is pointing.
* The value of a reference cannot be changed after initialisation.
* We cannot have a pointer to a reference. We can write &rr to get a pointer to the object referenced to.
* References can also be used as return types in cases where a type may be too expensive to copy.
* Having more than one kind of reference is to support different uses of objects –
* A non-const lvalue reference refers to an object, to which the user of the reference can write.
* A const lvalue reference refers to a constant, which is immutable from the point of view of the user of the reference.
* An **rvalue reference** refers to a temporary object, which the user of the reference can modify, assuming that it will not be used again.
* && declarator operator means ‘rvalue reference’.
* We do not use const rvalue references.
* Both a constant lvalue reference and an rvalue reference can bind to an rvalue. But the purposes are different –
* rvalue reference is used to implement a ‘destructive read’ for optimisation. Without it, a copy operation would have been required which can be expensive.
* We use const lvalue reference to prevent modification of an argument.

A close-up of a white background

Description automatically generated

A close-up of a white background

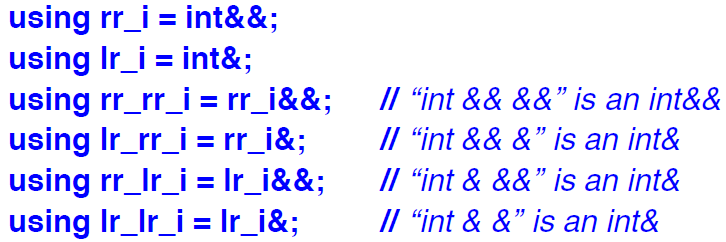
Description automatically generated

* move(x) does not really move x. It simply produces an rvalue reference to x and hence should be used instead of costly copy operations. In this sense, it would have been better to call move() as rval(), but now move() has been used for many years and has already become a standard.

A close-up of a computer code

Description automatically generated

* **References to references:** When we take a reference to a reference to a type, we simply get a reference to that type, rather than some special reference to reference type.
* lvalue reference always wins here. This phenomenon is known as reference collapse.



* Reference to reference can only happen as a result of an alias or a template type argument.
* **Advice –**
* Keep use of pointers simple and straightforward.
* Avoid non-trivial pointer arithmetic.
* Take care not to write beyond the bounds of an array.
* Avoid multi-dimensional arrays; define suitable containers instead.
* Use nullptr rather than 0 or NULL.
* Use containers, such as vector, array, valarray, rather than built-in containers.
* Use string rather than 0 terminated arrays of char.
* Use raw strings for string literals with complicated use of backslash.
* Prefer const reference arguments to plain reference arguments.
* Use rvalue reference only for forwarding and move semantics.
* Keep pointers that represent ownership inside handle classes.
* Avoid void\* except in low-level code.
* Use const pointers and const references to express immutability in interfaces.
* Prefer references to pointers as arguments, except where ‘no object’ is a reasonable option.