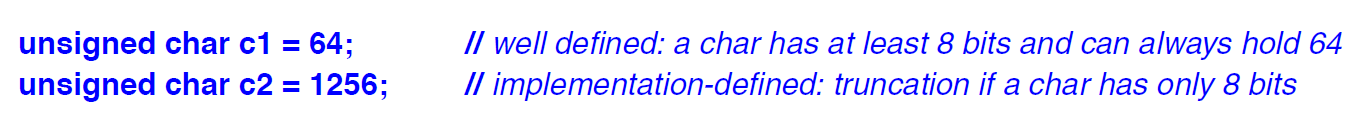
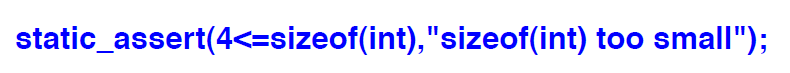
* Many important things are deemed as implementation-defined by the standard. For example,

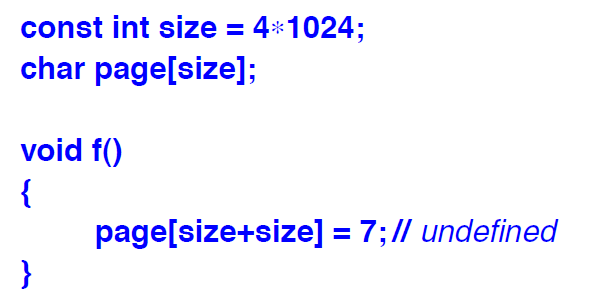


If the char has only 8 bits, then 1256 will be converted to 232. (No idea how?)

* Many assumptions about implementation-defined features can be checked by stating them as static assertions. For example,

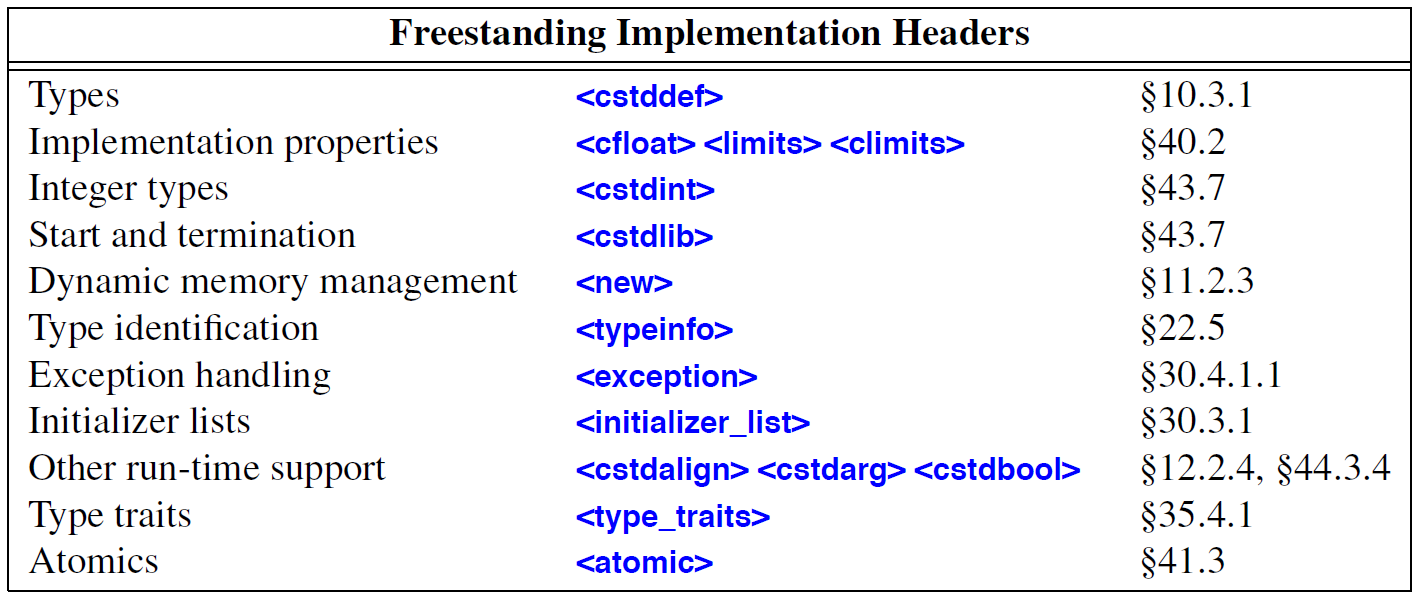


* A construct is deemed undefined by the standard if no reasonable behaviour is required by the implementation. For example,

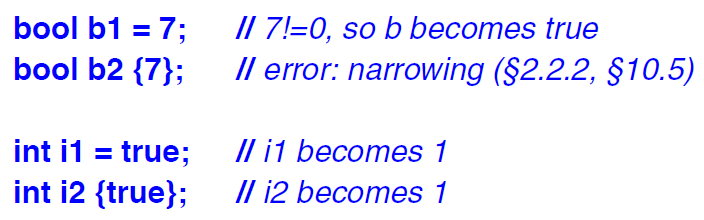


Plausible outcomes of this code fragment include overwriting unrelated data and triggering a hardware error/exception.

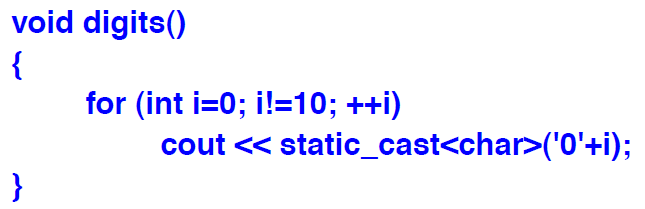
An implementation is not required to choose among plausible outcomes.



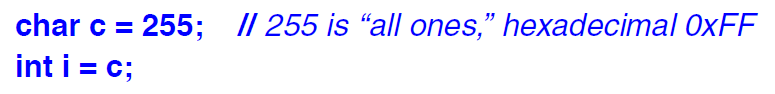
* C++ has a set of fundamental types –
* Boolean type (bool)
* Character types (char, wchar\_t )
* Integer types (int, long long)
* Floating point types (double, long double)
* Type to signify absence of information (void)
* Other types, constructed from the fundamental types, using declarator operators –
* Pointer types (int\*)
* Array types (char[])
* Reference types (double&, vector<int>&&)
* User defined additional types –
* Data structures and classes
* Enumeration types for representing specific sets of values (enum, enum class)
* **Integral types:** Boolean, character, integer types.
* **Arithmetic types:** Integral and floating-point types.
* **User-defined types:** Must be defined by users rather than being available for use without previous declaration, e.g. enums and classes.
* **Built-in types:** Fundamental types, pointers and references.
* **Booleans:** By definition, true has the value 1 when converted to integer, false has the value 0. Conversely, integers can be converted to bool values: non-zero integers convert to true and zero integers convert to false.



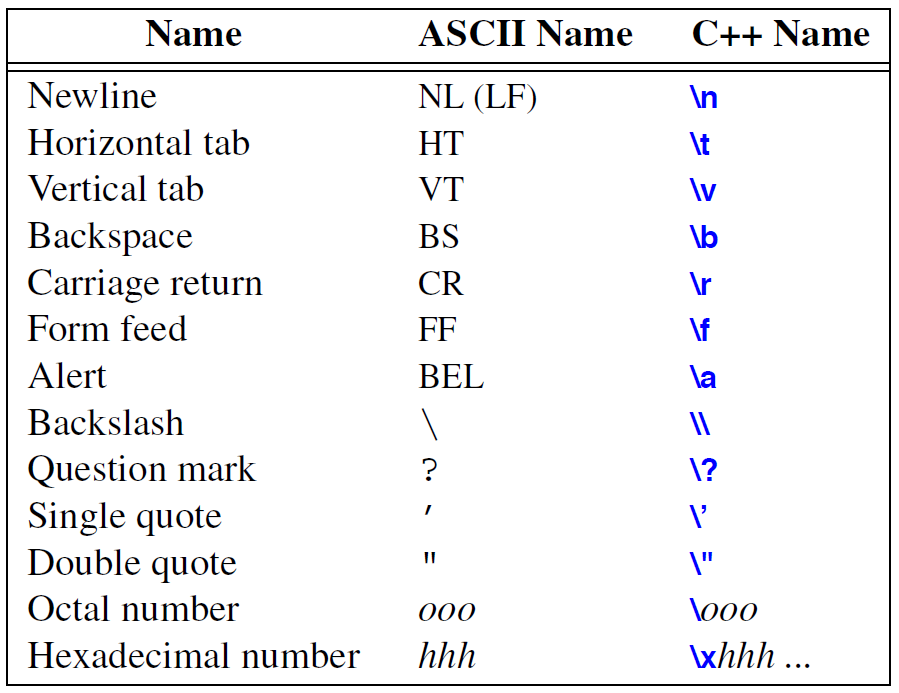
* **Character types:** C++ offers a variety of character types –
* *char*: Default character type. Usually 8 bits.
* *signed* *char*: Capable of holding both positive and negative values.
* *unsigned* *char*: A char that is guaranteed to be unsigned.
* *wchar\_t:* Can hold characters of a larger character set such as Unicode. Its size is implementation dependent.
* *char16\_t*: Holds 16-bit character sets, such as UTF-16.
* *char32\_t*: Holds 32-bit character sets, such as UTF-32.
* The following is an example of the use of *static\_cast –*



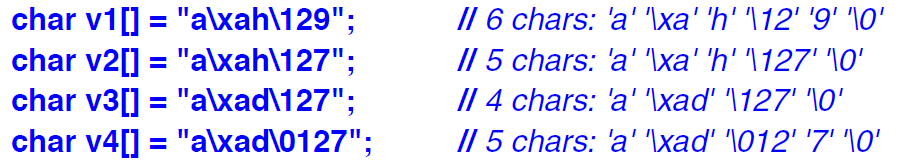
* The above is a program to print the 10 integer numbers, i.e. 0, 1, 2 and so on.
* By leaving out the *static\_cast*, the output will be 48, 49, 50 and so on.
* **Signed and unsigned chars:** It is implementation dependent whether a plain char is considered as signed or unsigned. This leads to confusions, e.g. –



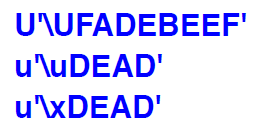
* On a machine where a char is unsigned, the answer is 255.
* On a machine where a char is signed, the answer is -1.
* Pointers of the types char, unsigned char and signed char cannot be mixed.
* Variables of the three char types can be freely assigned to one another. However, assigning too large value to a signed char is undefined.
* None of the potential problems and confusions arise if plain char is used throughout and negative values are not used.
* **Character literals:** A single character enclosed in single quotes, e.g. ‘a’, ‘B’, ‘4’. The type of a character literal is char.
* Use of character literals rather than decimal notations make programs more portable.
* The following are special characters. Despite their appearance, they are considered as single characters.



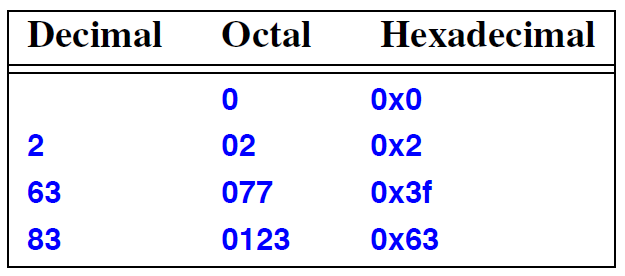
* A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal or hexadecimal digit respectively.



* For octal constants, always use 3 digits to represent a number.
* For hexadecimal constants, always use 2 digits to represent a number.
* **Wide character literals** are of the form L’ab’ and are of type wchar\_t. The number of characters between the quotes and their meanings are implementation-defined.
* Literals of larger character sets, such as **Unicode,** are presented as sequences of 4 or 8 hexadecimal digits preceded by a U or a u, e.g.



* A number of hexadecimal digits different from 4 or 8 is a lexical error.
* **Universal character names:** The values of hexadecimal number, defined by the ISO/IEC 10646 standard.
* **Integer types:** Integers come in 3 forms –
* plain int, referred to as ‘int’
* signed int, referred to as ‘signed’
* unsigned int, referred to as 'unsigned’
* **Integer sizes:** Integers come in 4 sizes –
* short int, referred to as ‘short’
* plain int, referred to as ‘int’
* long int, referred to as ‘long’
* long long int, referred to as 'long long’
* Plain ints are always signed.
* To get more detailed control over integer sizes, the following aliases from <cstdint> can be used –
* *int64\_t*: A signed integer with exactly 64 bits.
* *uint\_fast16\_t*: An unsigned integer with exactly 16 bits, supposedly the fastest such integer.
* *int\_least32\_t*: A signed integer with at least 32 bits, just like plain int.
* *extended integer types*: Behave exactly like integers but usually have greater range and occupy more space.
* **Decimal** numbers are represented as they are.
* **Octal** numbers are represented with a 0 in the starting.
* **Hexadecimal** numbers are represented with a 0x or a 0X in the starting.
* A compiler does warn when literals are too long to represent, but this can only be guaranteed for {} initialisers.



* **Suffixes** can be used to **explicitly mark literals.**
* **U**  is used to mark an unsigned integer.
* **L** is used to mark a long integer.
* Combination of suffixes can also be used. For example,

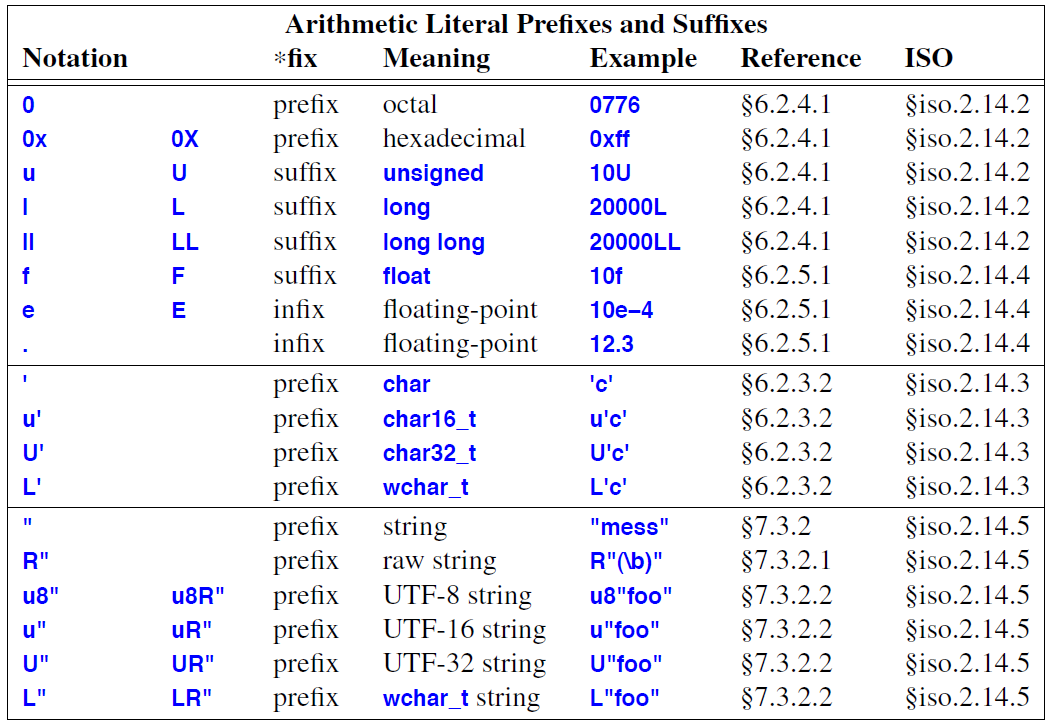
A blue arrow pointing to the right

Description automatically generated

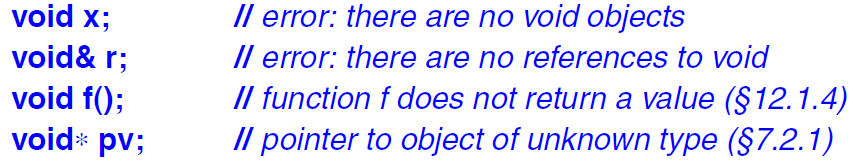
* It is a **good idea to limit** the use of nonobvious constants to a few well-commented *const*, *constexpr*, and *enumerator* initialisers.
* **Types of Integer literals –**
* If it is a decimal and has no suffix: *int, long int, long long int.*
* If it is octal or hexadecimal and has no suffix: *int, unsigned int, long int, unsigned long int, long long int, unsigned long long int.*
* If it is suffixed by a U or u:  *unsigned int, unsigned long int, unsigned long long int.*
* If it is decimal and suffixed by I or L: *long int, long long int.*
* If it is octal or hexadecimal and suffixed by I or L: *long int, unsigned long int, long long int, unsigned long long int.*
* If it is suffixed b ul, lu, uL, Lu, UI, IU, UL, LU: *unsigned long int, unsigned long long int.*
* If it is decimal and suffixed by II or LL: *long long int.*
* If it is octal or hexadecimal and suffixed by II or LL: *long long int, unsigned long long int.*
* Implementation dependencies can be avoided by using proper suffixes.
* **Floating point types –**
* float (single precision)
* double (double precision)
* long double (extended precision)
* The exact meaning of single, double and extended precision is implementation-defined.
* Some examples are –



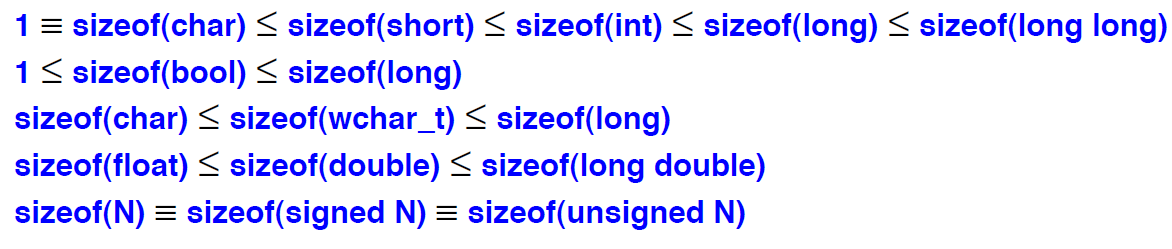
* A space cannot occur between a floating-point literal. It will lead to syntax errors.
* **float:** floating-point literal with suffix f or F.
* **double:** floating-point literal without a suffix. It is the default.
* **long double:** floating-point literal with suffix I or L.



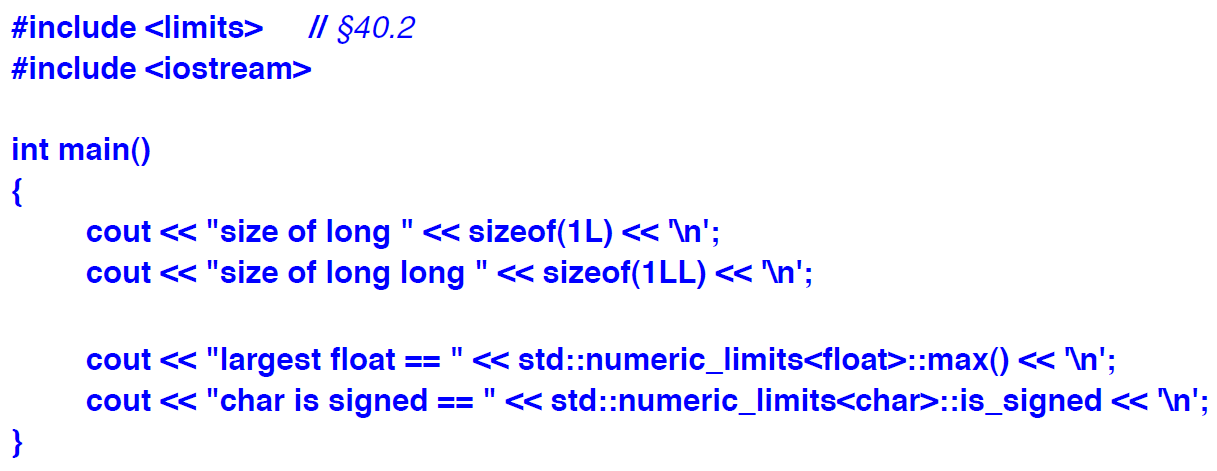
* There are no objects of type **void.**
* It is used to specify that a function does not return a value. It is a *pseudo* return type.
* It is also used as a base type for pointers to objects of unknown type.



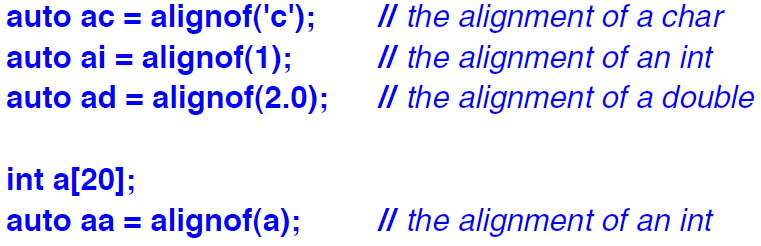
* Sizes of many data types are implementation dependent.
* Writing truly portable low-level code is hard.
* **Comparison of sizes –**



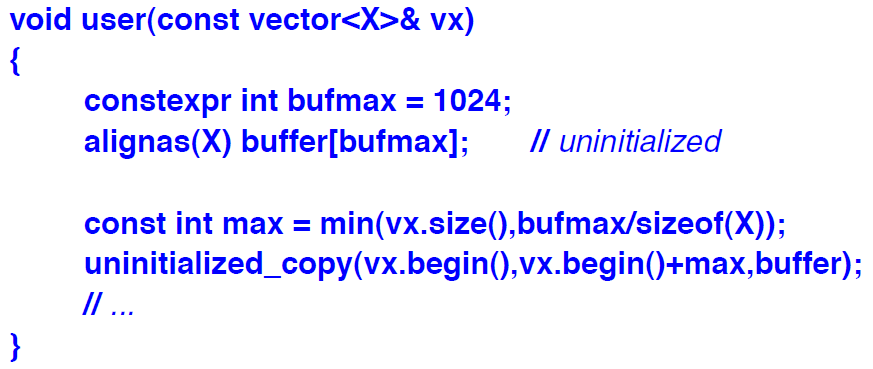
* It is guaranteed that a char has at least 8 bits, a short has at least 16 bits, a long has at least 32 bits.
* The sizes and signs of literals of different types are found by writing appropriate code. For example –



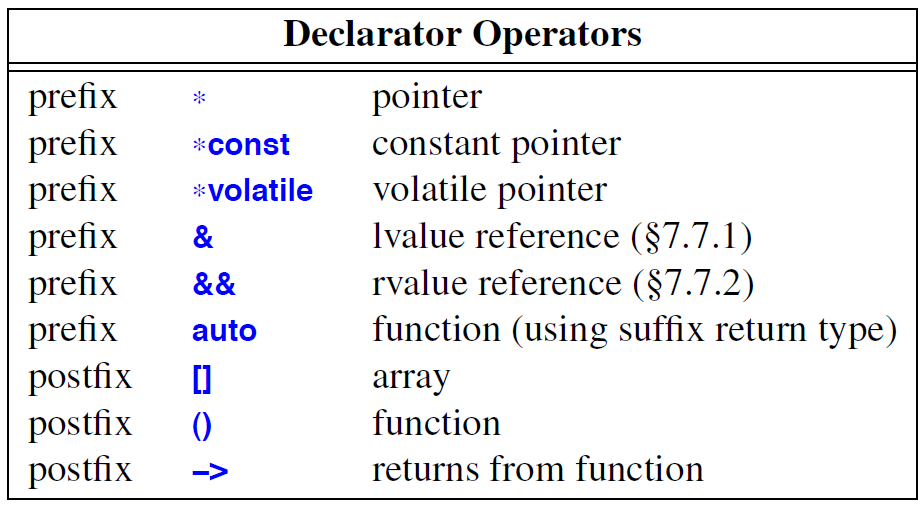
* **Alignment:** An object doesn’t just need enough storage to hold its representation. On some machine architectures, the bytes used to hold it must have proper alignment for the hardware to access it efficiently.
* Alignment is extremely implementation specific.
* Sometimes struct contains holes to improve its alignment.
* Use of the alignof() operator –



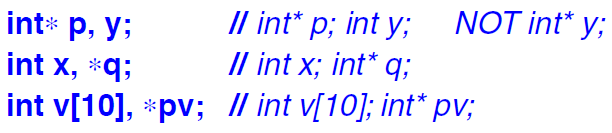
* Sometimes we have to use alignment in a declaration. Like so –



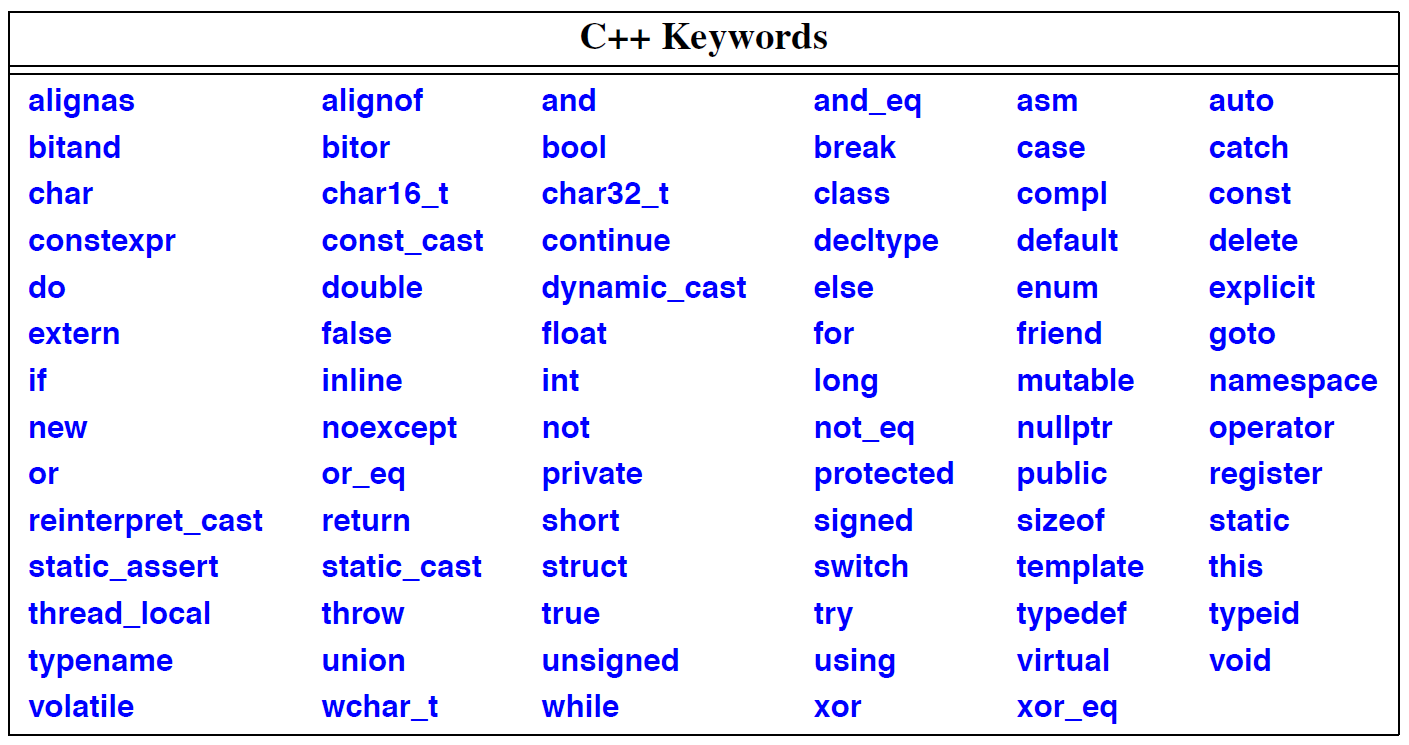
* **Declarations:** Before an identifier can be used in a program, it must be declared.
* Declarations are deemed part of an interface.
* Definitions are deemed part of an implementation.
* Any declaration that specifies a default value is a definition.
* For example, int num; (Assuming num is a global variable, it is equivalent to the statement “int num = 0;”)
* **Structure of declarations:** A declaration has five parts –
* (Optional) prefix specifiers (e.g. static, virtual, extern, constexpr etc)
* Base type (e.g. vector<double>, const int, etc)
* A declarator, (optionally) including a name (e.g. p[7], n, \*(\*)[], etc)
* (Optional) suffix function specifiers (e.g. const, noexcept, etc)
* (Optional) initialiser or function body (e.g. = {7, 5, 3}, {return x;})
* A declaration is terminated by a semi-colon, except for function and namespace definitions.



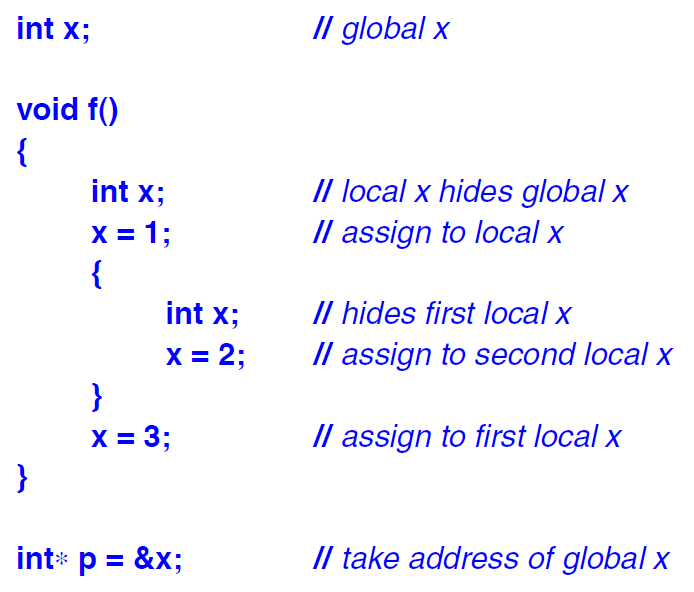
* Postfix declarator operators bind tighter than the prefix ones. So –
* **char\*kings[]:** An array of pointers to char.
* **char(\*kings)[]:** A pointer to an array of chars.
* Standard C++ differs from early versions of C and C++ in the sense that these would consider int to be the default type when no type was mentioned. This implicit int rule was dropped to avoid errors and confusion. No implicit data type is assumed and not specifying a type leads to syntax errors.



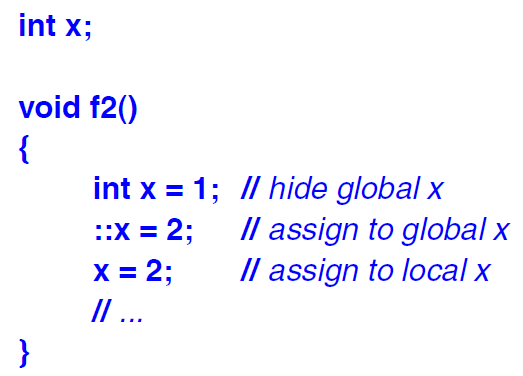
* Declarations of the above type with multiple names and non-trivial declarators make a program harder to read and should be avoided.
* **Names:** The following are the rules used for names (identifiers) –
* Consists of a sequence of letters and digits.
* The first character must be a letter.
* The underscore (\_) character is considered a letter.
* C++ imposes no limit on the number of characters in a name, but a linker might. So, it is unwise to use exceptionally long names.
* Extensions (e.g. using $ character in a name) yield non-portable programs.
* A C++ keyword (e.g. new, int) cannot be used as an identifier.
* Non-local names starting with an underscore are reserved for special facilities in the implementation and run-time environments.
* Names starting with double underscore, or an underscore followed by an uppercase letter are reserved.
* Some combinations of characters should be avoided for better readability of names. E.g. uppercase ‘O’ and ‘0’, lowercase ‘l’, uppercase 'I’ and ‘1’, etc.
* It is often useful to keep frequently used names relatively short and reserve really long names for infrequently used entities.
* Choose names to reflect the meaning of an entity rather than its implementation, e.g. phone\_book rather than number\_vector.
* Do not encode type information in an name, e.g. pc name for a name that is a char\* or icount for a count that is an int.
* Choosing good names is an art.
* Captialise names of user-defined types, e.g. Shape.
* Start names of non-type entities with a lower-case letter, e.g. current\_token.
* Use all capitals for macros, e.g. HACK. Do not use all capitals for anything else.
* The language and the standard library use lowercase for types, e.g. number\_of\_elements rather than numberOfElements. So, use of underscores to separate words is recommended.
* Here are my personal favourite rules for C++ names –
* Pascal’s case for type name, e.g. LandVehicles.
* Camel case for function or method names, e.g. openCar().
* All lowercase letter with underscore to separate words for variables, e.g. my\_institute.
* All uppercase letters for macros, e.g. PRIME.
* **C++ keywords –**
* The word *export* is reserved for future use, along with the available keywords mentioned in the picture below.

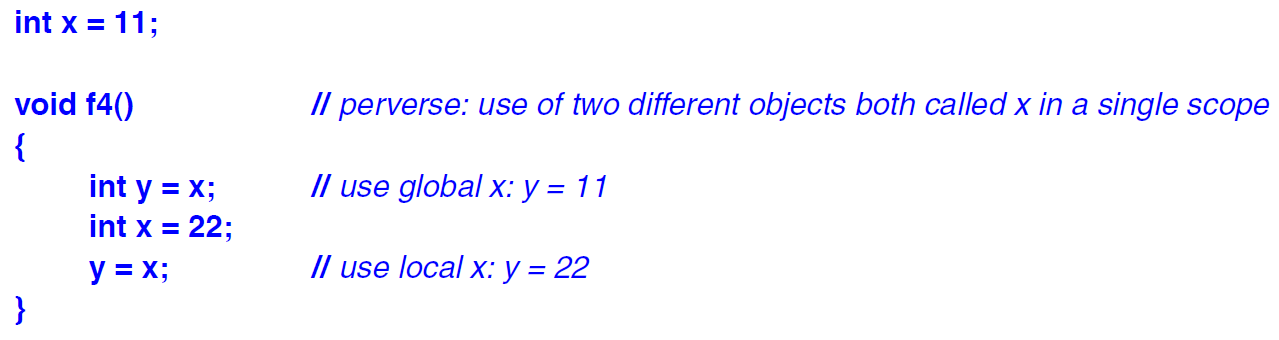


* Types of scopes –
* **Local scope:** A name declared in a function or lambda.
* **Class scope:** Defined in a class outside any function, class, enum class or other namespace.
* **Global scope:** Defined outside any function, class, enum class or namespace.
* **Statement scope:** Defined within the () of a for-, while-, if-, switch statement.
* **Function scope:** Defined within a function.
* A declaration of a name in a block can hide a declaration in an enclosing block or a global name. After exit from a block, the name resumes its previous meaning. E.g. –

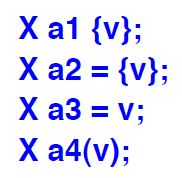


* A hidden global name can be referred to using the scope resolution operator :: or without it.

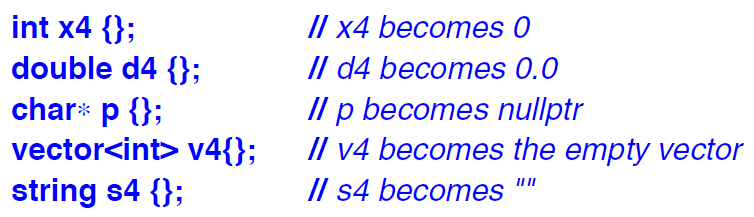




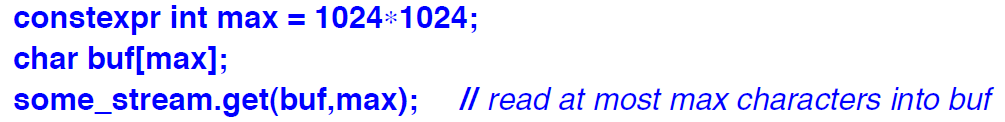
* These kinds of situations are best avoided. They lead to unreadable code.
* An **initialisation** in C++ can have one of the four following styles –



* The first type is the most robust.
* Initialisation using {} has the following characteristics –
* An integer cannot be converted to another integer that can’t hold its value. E.g. – char to int is allowed, but int to char is not.
* A floating-point value cannot be converted to another floating-point type that cannot hold its value. E.g. – float to double is allowed, but double to float is not.
* A floating-point value cannot be converted to an integer type.
* An integer value cannot be converted to a floating-point type.
* = is preferred when using ***auto***.
* An empty initialiser list {} is preferred when a default value is intended. E.g. –



* **Missing initialisers:** The only really good case for an uninitialized variable is a large input buffer.

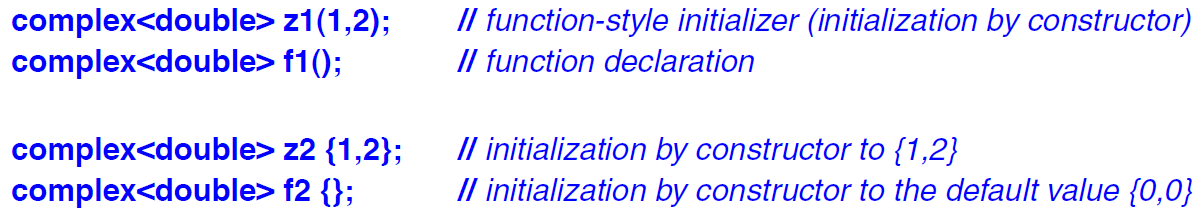


* Initialising the above buffer redundantly might have resulted in a significant performance hit.
* If no initialisation is specified –
* Global, namespace, local static and static member variables are initialised by default. These types are called as static types.
* Local variables and objects are not initialised by default. These are called dynamic objects or heap objects.
* More complicated values are handled by **initialiser lists.**

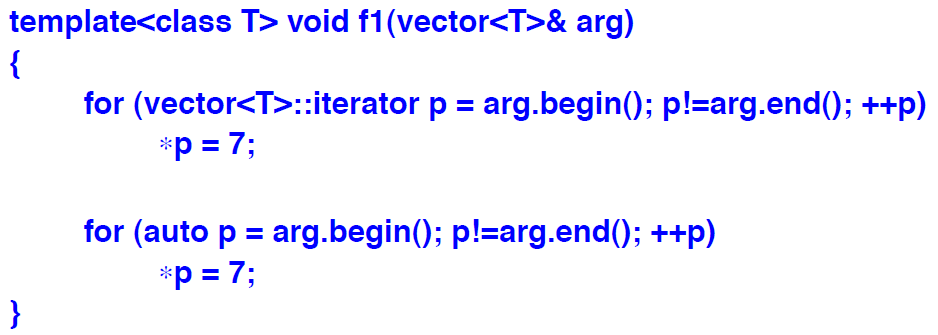
A close-up of a math equation

Description automatically generated

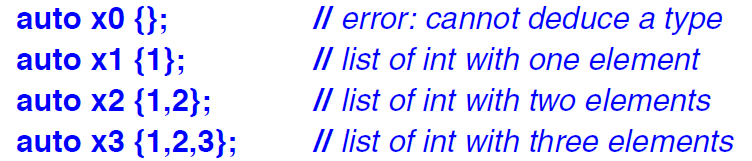
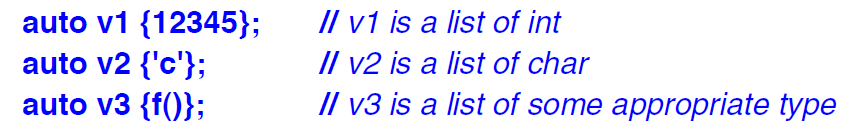
* In the cases above, the = is redundant.
* In a declaration, an empty pair of parentheses () always means a function. So, {} are instead used for default initialisation.



* There are two mechanisms for **deducing a type** from an expression –
* **auto:** Deducing type of an object from its initialiser. It can be the type of a variable, a const, a constexpr.
* **decltype(expr):** Deducing the type of something that is not a simple initialiser, e.g. return type of a function, type of a class member.
* The harder the type is to write, the harder the type is to know, the more useful **auto** becomes.
* **auto** should be used in small scopes like loops.
* The second loop in the example below is easier to read and is more resilient to code changes.



* However, if a scop is large, using **auto** can delay the detection of type errors.
* For objects specified as **auto**, = is recommended to be used rather than {} wherever a list is not meant.



* **decltype(expr)** is the declared type of **expr** and is mostly used in generic programming.

A white background with blue text

Description automatically generated

* **lvalue:** An expression that refers to an object. The term was originally coined to mean “something that can be on the left-hand side of an assignment”.
* **modifiable lvalue:** An lvalue that has not been declared ***const***.
* There are two properties that matter for an object when it comes to addressing, copying and moving –
* ***Has identity:*** The program has the name of, pointer to, or reference to the object so that it is possible to determine if two objects are the same, whether the value of the object has changed, etc.
* ***Movable:*** The object can be moved from. In other words, we are allowed to move its value to another location and leave the object in a valid but unspecified state, rather than copying.
* Using ‘m’ for movable and ‘i’ for identity, we can represent this classification of expressions graphically –

A black arrows pointing to a white background

Description automatically generated

* **glvalue:** Generalised lvalue.
* **prvalue:** Pure rvalue.
* **xvalue:** Extraordinary or expert value. The meaning of x is imaginative.
* Every expression is either an lvalue or an rvalue, but not both.