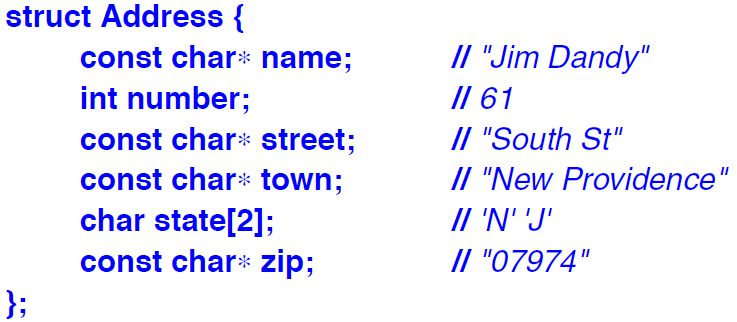
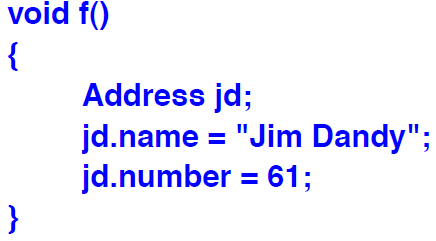
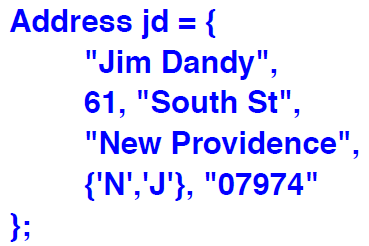
* Most primitive variants of the notion of a user-defined class are –
* **struct:** A sequence of elements/members of arbitrary type. It is a simple form of a class.
* **union:** A struct that holds the value of just one of its members at any one time.
* **enum:** A type with a set of named constants called enumerators.
* **enum class:** An enum where the enumerators are within the scope of the enumeration and no implicit conversions to other types are provided.
* **Structures –**



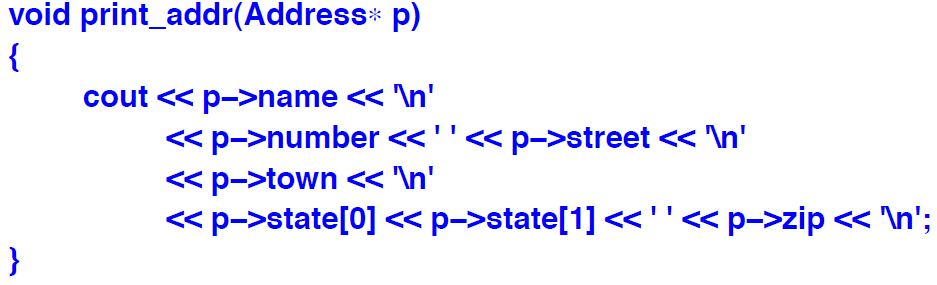
* The members of a struct can be accessed/initialised in various ways –
* By the use of a . (dot) operator.



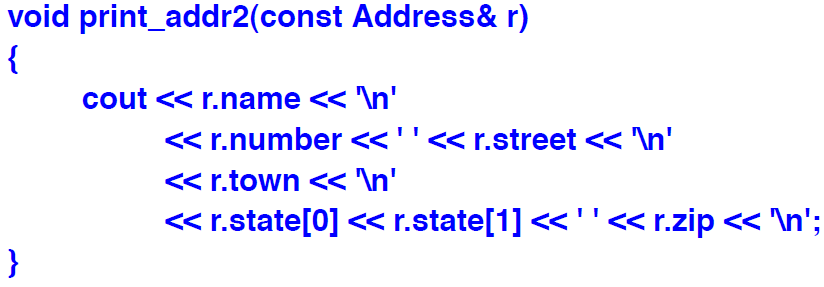
* By the use of {} to initialise only.



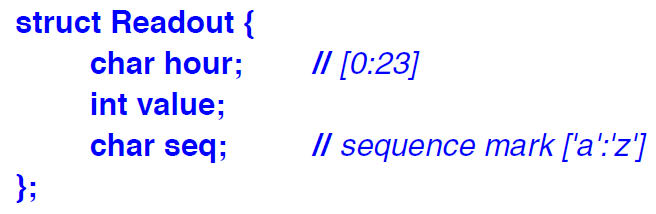
* By the use of -> (pointer).



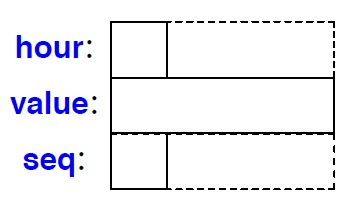
* By the use of a reference and . (dot) operator.



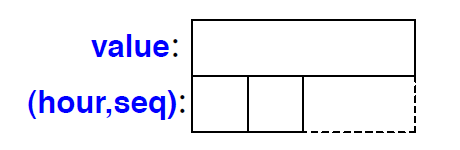
* Structures can be passed as function arguments and returned as a result of a function.
* Other plausible operators, such as comparison, are not available by default, but can be defined by the user as and when required.
* **struct layout:** An object of a struct holds its members in the order they are declared.



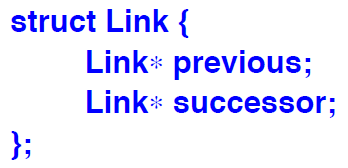
* Members are allocated in memory in declaration order.
* The size of an object of a struct is not necessarily the sum of sizes of its members.
* This is because many machine require object of certain types to be allocated on architecture dependent boundaries.
* This leads to “holes” in the structures.



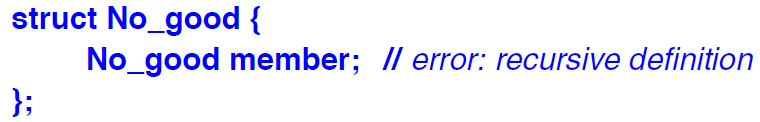
* The wasted space can be minimised by simply ordering members by decreasing order of their size.



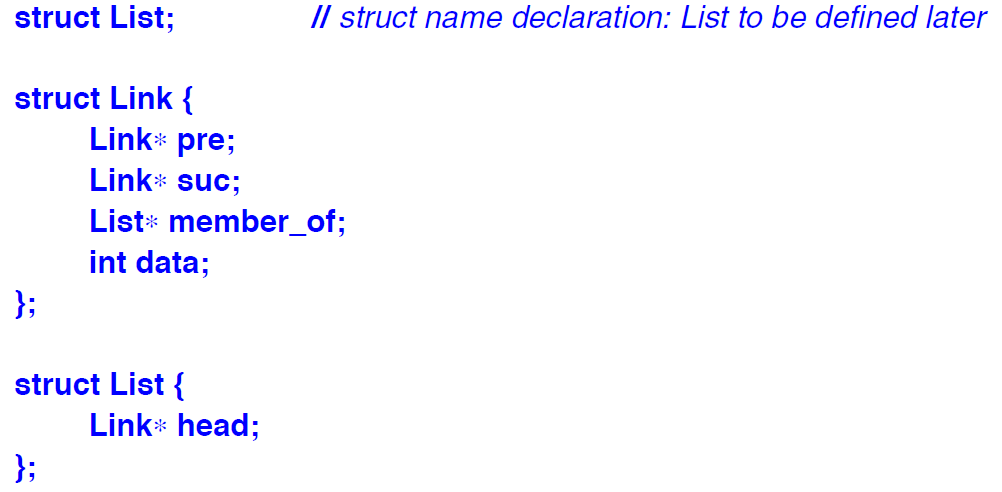
* It is best to order members by readability.
* Members should only be sorted by size if there is a demonstrated need to optimise memory.
* The **name of a struct** becomes available immediately after it has been encountered.



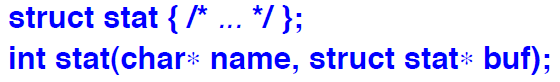
* However, it is not possible to declare objects of a struct until it is completely declared.



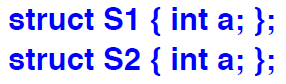
* This gives an error because the compiler is not able to determine the size of the object declared.
* To allow two or more structs to refer to each other, one struct can be just declared and defined later.



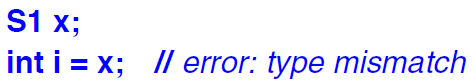
* A struct and a non-struct can be declared with the same name in the same scope. The plain name will be referring to the non-struct and the one with struct prefix will be referring to the struct.



* However, it is best to avoid similar names and hence, confusion.
* **Structures and classes:** A struct is simply a class where the members are public by default. So, a struct can have member functions and constructors.
* **Structures and arrays:** We can have arrays of structs and structs containing arrays.
* **Type Equivalence:** Two structs are different types even when they have the same members.

****

* A struct is also a different type from a type used as a member.

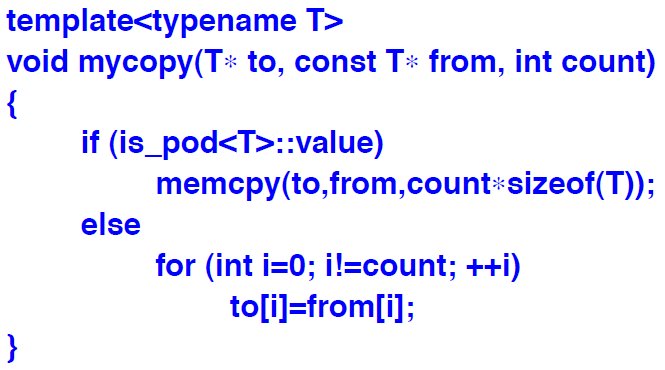
****

* **Plain Old Data (POD):** An object that can be manipulated as just data, without worrying about complications of class layout or user-defined semantics for construction, copy, and move.

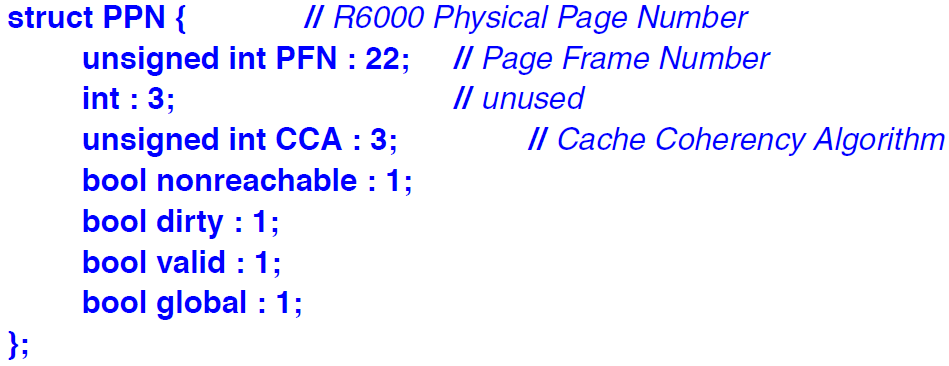
**A white background with blue text

Description automatically generated**

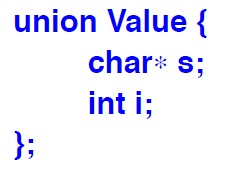
* Copying a 100-element array using 100 calls of a copy constructor is unlikely to be as fast as calling std::memcpy, which typically uses block-move machine instruction.

****

* The best thing about is\_pod<T> is that it relieves us from remembering what the exact rules for a POD are.
* **Fields:** It is possible to bundle several tiny variables together as fields in a struct. It is also often called as bit-field.
* Unnamed fields are allowed. They do not affect the meaning of the named fields in any way, but they can be used to make the layout better in some machine-dependent way.

****

* A field must be of an integral or enumeration type.
* A bool field can be represented by a single bit.
* It is not possible to take the address of a field. Other than that, it can be used exactly as other variables.
* Using fields to pack several variables do not necessarily space more space.
* It saves data space, but the size of the code needed to manipulate the variables increases on most machines.
* Also, it is typically faster to use a char or an int in a machine than to access a field.
* Fields are simply a convenient shorthand for using bitwise logical operators.
* **Unions:** A struct in which all members are allocated the same address, so that it occupies only as much space as the largest member.
* It can hold a value for only one member at a time.

****

* **Unions and Classes:** Many non-trivial unions have a member that is much larger than the most frequently used members. Hence, space is wasted.
* This waste of space can be eliminated by using a set of derived classes, instead of a union.
* Technically, a union is a kind of struct, which in turn is a kind of class.
* There are many facilities available for classes which are not relevant for unions, so some restrictions are imposed on unions, including –
* Cannot have virtual functions.
* Cannot have members of reference type.
* Cannot have base classes.
* If a union has a member with a user-defined constructor/destructor, copy/move operations, then that special function is deleted for this union.
* At most, one member of a union can have in-class initialiser.
* Cannot be used as a base class.
* **Enumeration:** A type that can hold a set of integer values specified by the user. Some of enumeration’s possible values are named and are called enumerators.

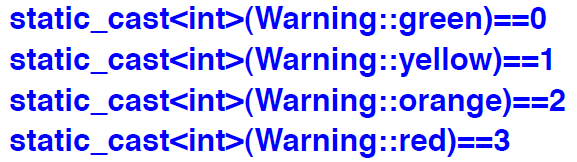
****

* There are two types of enumerations –
* **Enum class:** Enumerator names are local to the enum and their values do not implicitly convert to other types.
* **Enum:** Enumerator names are in the same scope as the enum and their values implicitly converted to integers.
* In general, enum classes cause fewer surprises.
* **Enum class:**  It is a scoped and strongly typed enumeration.

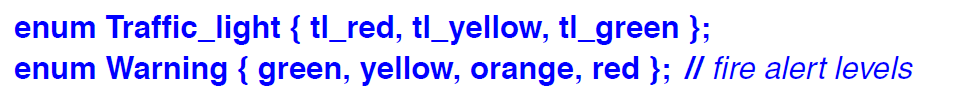
**A white background with blue text

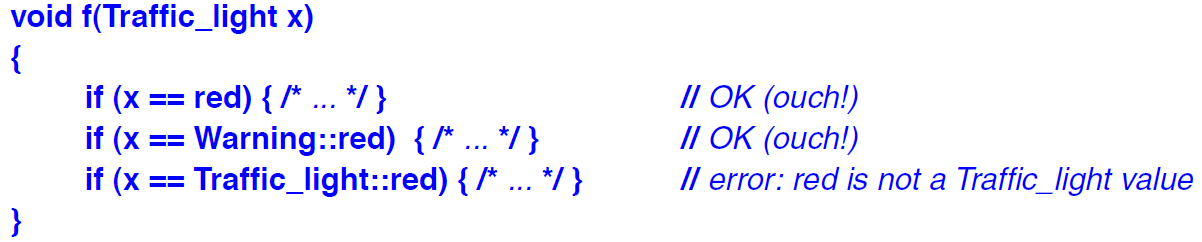
Description automatically generated**

* Enumerator values of enum classes can be explicitly converted.

****

* The size of a enum class is the sizeof() of its underlying type.
* If the underlying type is not explicitly specified, the size is sizeof(int).
* Plain **Enum:** The enumerators of a plain enum are exported into the enum’s scope and they implicitly convert the values of some integer type.

****

****

* The compiler accepts x==red, which is almost certainly a bug!
* **Unnamed enum:** A plain enum can be unnamed.

****

* We use unnamed enum when all we need is a set of variables rather than a type to use for variables.
* **Advice –**
* When compactness of data is important, lay out structure data members with larger members before smaller ones.
* Use bit fields to represent hardware-imposed data layouts.
* Don’t naively try to optimise memory consumption by packing several values into a single byte.
* Use unions to save space and never for type conversions.
* Use enumerations to represent sets of named constants.
* Prefer class enums over plain enums to minimise surprises.
* Define operations on enumerations for safe and simple use.