Continue

From cppreference.com #define stdin /* implementation-defined */ (1) #define stdout /* implementation-defined */ (2) #define stderr /* implementation-defined */ (3) Three text streams are predefined. These streams are implicitly opened and unoriented at program startup.1) Associated with the standard input stream, used for reading conventional input. At program startup, the stream is fully buffered if and only if the stream can be determined not to refer to an interactive device.2) Associated with the standard output stream, used for writing conventional output. At program startup, the stream is fully buffered if and only if the stream can be determined not to refer to an interactive device.3) Associated with the standard error stream, used for writing diagnostic output. At program startup, the stream is not fully buffered.What constitutes an interactive device is implementation-defined.These macros are expanded to expressions of type std::FILE*.[edit] NotesAlthough not mandated by POSIX, the UNIX convention is that stdin and stdout are line-buffered if associated with a terminal and stderr is unbuffered.These macros may be expanded to modifiable lvalues. If any of these std::FILE* lvalue is modified, subsequent operations on the corresponding stream result in unspecified or undefined behavior.[edit] ExampleThis example shows a function similar to std::printf.#include #include #include templateconcept IsPrintable = std::integral or std::floating_point or std::is_pointer_v;int my_printf(char const* const format, IsPrintable auto const ... arguments){ return std::fprintf(stdout, format, arguments...);}int main(int argv, char*[]){ my_printf("Strings and chars:\t%s%c", "hello", 'x'); my_printf("Rounding:\t\t%f%.0f%.32f", 1.5, 1.5, 1.3); my_printf("Padding:\t\t%05.2f%.2f%5.2f", 1.5, 1.5, 1.5); my_printf("Scientific:\t\t%E%e", 1.5, 1.5); my_printf("Hexadecimal:\t\t%a%A 0x%X", 1.5, 1.5, &argv);}Possible output:Strings and chars: hello xRounding: 1.500000 2 1.30000000000000004440892098500626Padding: 01.50 1.50 1.50Scientific: 1.500000E+00 1.500000e+00Hexadecimal: 0x1.8p+0 0X1.8P+0 0x2CFB41BC [edit] See also From cppreference.comAccesses a member of its operand.OperatornameSyntaxOverloadablePrototype examples (for class T)Inside class definitionOutside class definitionsubscripta[b]YesR& T::operator[](S b); N/Aa[...] (since C++23)R& T::operator[](...);indirection*aYesR& T::operator*();R& operator*(T a);address-of&aYesR* T::operator&();R* operator&(T a);member of objecta.b No N/A N/Amember of pointera->bYesR* T::operator->(); N/Apointer to member of objecta.*b No N/A N/Apointer to member of pointera->*bYesR& T::operator->*(S b);R& operator->*(T a, S b);Notes As with most user-defined overloads, return types should match return types provided by the built-in operators so that the user-defined operators can be used in the same manner as the built-ins. However, in a user-defined operator overload, any type can be used as return type (including void). One exception is operator->, which must return a pointer or another class with overloaded operator-> to be realistically usable.[edit] ExplanationBuilt-in subscript operator provides access to an object pointed-to by the pointer or array operand.Built-in indirection operator provides access to an object or function pointed-to by the pointer operand.Built-in address-of operator creates a pointer pointing to the object or function operand.Member of object and pointer to member of object operators provide access to a data member or member function of the object operand.Built-in member of pointer and pointer to member of pointer operators provide access to a data member or member function of the class pointed-to by the pointer operand.[edit] Built-in subscript operatorThe subscript operator expressions have the form expr1[expr2] (1) expr1[{expr, ...}] (2) (since C++11) expr1[expr2, expr, ...] (3) (since C++23)1) For the built-in operator, one of the expressions (either expr1 or expr2) must be a glvalue of type array of T or a prvalue of type pointer to T, while the other expression (expr2 or expr1, respectively) must be a prvalue of unscoped enumeration or integral type. The result of this expression has the type T. expr2 cannot be an unparenthesized comma expression.(since C++23)2) The form with brace-enclosed list inside the square brackets is only used to call an overloaded operator[].3) The form with comma-separated expression list inside the square brackets is only used to call an overloaded operator[].The built-in subscript expression E1[E2] is exactly identical to the expression *(E1 + E2) except for its value category (see below) and evaluation order(since C++17): the pointer operand (which may be a result of array-to-pointer conversion, and which must point to an element of some array or one past the end) is adjusted to point to another element of the same array, following the rules of pointer arithmetic, and is then dereferenced.When applied to an array, the subscript expression is an lvalue if the array is an lvalue, and an xvalue if it isn't(since C++11). When applied to a pointer, the subscript expression is always an lvalue.The type T is not allowed to be an incomplete type, even if the size or internal structure of T is never used, as in &x[0].Using an unparenthesized comma expression as second (right) argument of a subscript operator is deprecated.For example, a[b, c] is deprecated and a[(b, c)] is not.(since C++20)(until C++23)An unparenthesized comma expression cannot be second (right) argument of a subscript operator. For example, a[b, c] is either ill-formed or equivalent to a.operator[](b, c).Parentheses are needed to for using a comma expression as the subscript, e.g., a[(b, c)].(since C++23)In overload resolution against user-defined operators, for every object type T (possibly cv-qualified), the following function signature participates in overload resolution: T& operator[](T*, std::ptrdiff_t); T& operator[](std::ptrdiff_t, T*);#include #include #include int main(){ int a[4] = {1, 2, 3, 4}; int* p = &a[2]; std::cout () { return uPtr; }};int main(){ A a(1); std::cout