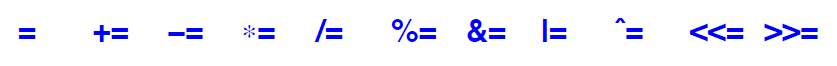
* **Introduction:** In C++, an *assignment* is an expression, a *function call* is an expression, the *construction of an object* is an expression, etc.
* The following assignment operators are possible in C++.



* **isspace(c):** Provides standard test for whitespace. Returns a non-zero value if c is a whitespace character and 0 otherwise.
* **isdigit(c):** Tests if a character is a digit.
* **isalpha(c):** Tests if a character is an alphabet.
* **isalnum(c):** Test if a character is a digit or a letter.
* These test functions are available in #include<cctype>
* **Constant Expressions –**
* There are a variety of reasons why someone might want a named constant rather than a literal or a value stored in a variable –
* Named constants make the code easier to read, understand and maintain.
* A variable might be changed. So, we have to be more careful in our reasoning than a constant.
* The language requires constant expressions for array sizes, case labels and template value arguments.
* Embedded-system programmers like to put immutable data into read-only memory. This is because read-only memory is cheaper than dynamic memory (in terms of cost and energy consumption) and often more plentiful. Also, data in read-only memory is immune to most system crashes.
* If initialisation is done at compile time, there can be no data races in that object in a multi-threaded system.
* Sometimes, evaluating something once at compile time gives significantly better performance than doing so a million times at runtime.
* **Floating Point Conversions –**
* A floating-point value is converted to another floating point type.
* If the float value can be exactly represented in the destination type, the result is the original numeric value.
* If the source value is between two adjacent destination values, the result is one of those values.

A white background with blue text

Description automatically generated

* DBL\_MAX and FLT\_MAX are defined in <climits>.
* Numeric\_limits is defined in <limits>.
* **Pointer and Reference Conversions –**
* Any pointer to an object type can be implicitly converted to a *void\**.
* A constant expression that evaluates to 0 can be implicitly converted to a null pointer of any pointer type.
* A *T\** can be implicitly converted to a *const T\*.* Similarly, a *T&* can be implicitly converted to a *const T&.*
* **Boolean Conversions –**
* Pointer, integral, and floating-point values can be implicitly converted to bool.
* A non-zero value converts to true.
* A zero value converts to false.
* **Floating-Integral Conversions –**
* When a floating-point value is converted to an integer value, the fractional part is discarded.
* Conversion from integer to floating types are as mathematically correct as the hardware allows.
* It is best to avoid potentially value destroying implicit conversions.
* To truncate in a way that is guaranteed to be portable requires the use of numeric\_limits.
* In initialisations, truncation can be avoided by using the {}-initialiser notation.
* **Usual Arithmetic Conversions –**
* If either operand is of type long double, the other is converted to long double.
* Otherwise, if either operand is double, the other is converted to double.
* Otherwise, if either operand is float, the other is converted to float.
* Otherwise, if either operand is unsigned long long, the other is converted to unsigned long long.
* Otherwise, if one operand is a long long int and the other is an unsigned long int, then if the long long int operand can convert all the values of an unsigned long int the unsigned long int is converted to long long int.
* Otherwise, both the operands are converted to unsigned long long int.
* Otherwise, if one operand is a long int and the other is an unsigned int, then if a long int can represent all the values of an unsigned int, the unsigned int is converted to long int.
* Otherwise, both operands are converted to unsigned long int.
* Otherwise, if either operand is long, the other is converted to long.
* Otherwise, if either operand is unsigned, the other is converted to unsigned.
* Otherwise, both operands are int.
* It is best to avoid mixing signed and unsigned integers for obvious reasons as can be seen from above.
* **Advice –**
* Use character level input only when you have to.
* When reading, always consider well-formed input.
* Prefer suitable abstractions (classes, algorithms, etc) to direct use of language features (e.g. ints, statements, etc)
* Avoid complicated expressions.
* If in doubt about operator precedence, parenthesize.
* Avoid expressions with undefined order of evaluation.
* Avoid narrowing conversions.
* Define symbolic constants to avoid “magic constants”.