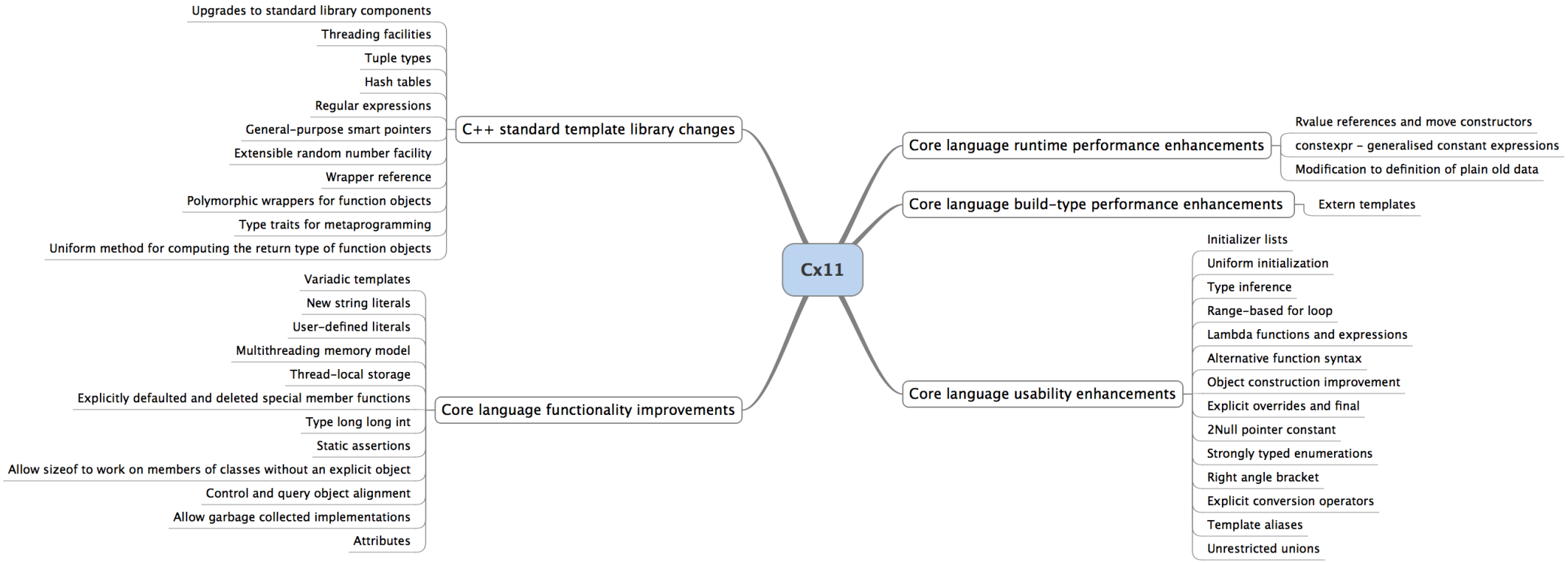
Effective C++

powered by C++11

Overview of C++11



source: https://en.wikipedia.org/wiki/C%2B%2B11

Performance Enhancements

Move Operators…

`class_name(class_name&&)`

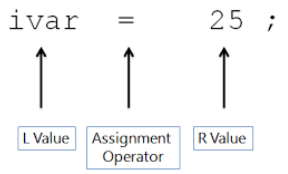
The move constructor is called whenever an object is initialised from rvalue reference.

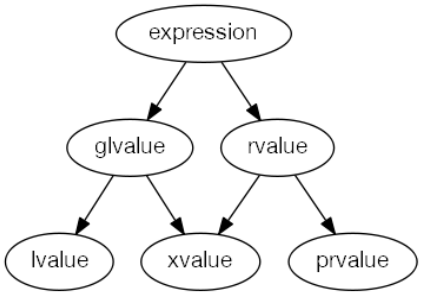
`class_name& operator=(class_name&&)`

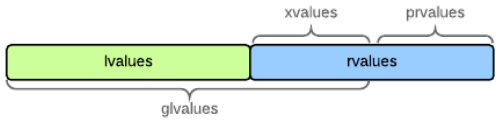
The move assignment operator is called when an object appears on the left-hand side of an assignment expression and the right-hand side is rvalue reference.

(r, l, gl, x, p) Value References

- An lvalue (lefthand value) represents an object that occupies some identifiable location in memory (i.e. has an address).
- An rvalue does not represent an object occupying some identifiable location in memory.
- An xvalue (an “eXpiring” value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references. For example: *The result of calling a function whose return type is an rvalue reference is an xvalue.*
- A glvalue (“generalized” lvalue) is an lvalue or an xvalue.
- A rvalue (“pure” rvalue) is an rvalue that is not an xvalue. For example: *The result of calling a function whose return type is not a reference is a prvalue*.







Universal References

- There are two meanings of `&&` in type declarations.
- If a function template parameter has type `T&&` for a deduced type `T`, or if an object is declared using `auto&&`, the parameter or object is a universal reference.
- If the expression initialising a universal reference is an lvalue, the universal reference becomes an lvalue reference.
- If the expression initialising the universal reference is an rvalue, the universal reference becomes an rvalue reference.

Default Member Function Generation

- The special member functions are those compilers may generate on their own: default constructor, destructor, copy operations, and move operations.
- Move operations are generated only for classes lacking explicitly declared move operations, copy operations and a destructor.
- The copy constructor is generated only for classes lacking an explicitly declared copy constructor, and it's deleted if a move operation is declared.
- The copy assignment operator is generated only for classes lacking an explicitly declared copy assignment operator, and it's deleted if a move operation is declared. Generation of the copy operations in classes with an explicitly declared destructor is deprecated.
- Member function templates never suppress generation of special member functions.

Understand std::move and std::forward

- `std::move` and `std::forward` are just cast operations, they don't move anything and they don't do anything in runtime.
- `std::move` performs an unconditional cast to an rvalue.
- `std::forward` performs conditional cast to an rvalue.
- `std::move` is used to indicate that an object may be "moved from", i.e. allowing the efficient transfer of resources from one to another object.
- `std::forward` forwards the argument to another function with the value category it had when passed to the calling function.

Use std::move and std::forward with care

- Don’t declare objects const if you want to be able to move from them.
- Remember, function parameters are lvalue all the time.
- Apply `std::move` to rvalue references and `std::forward` to universal references the last time each is used.
- Functions taking universal references are the greediest functions in C++. Avoid overloading on universal references.

constexpr

- constexpr objects are const and are initialised with values known during compilation.
- constexpr functions can produce compile-time results when called with arguments whose values are known during compilation.
- Use constexpr everywhere where is possible, constexpr objects and functions may be used in a wider range of contexts than non-constexpr objects and functions.
- be careful, constexpr is part of an object's or function's interface so removing of constexpr may lead to compile errors.

Modification to the definition of plain old data

- C++11 relaxed several of the POD (Plain Old Data) rules, by dividing the POD concept into two separate concepts: trivial and standard-layout.
- A type that is trivial can be statically initialized. It also means that it is legal to copy data around via memcpy, rather than having to use a copy constructor. The lifetime of a trivial type begins when its storage is defined, not when a constructor completes.
- A type that is standard-layout means that it orders and packs its members in a way that is compatible with C.
- A class/struct/union is considered POD if it is trivial, standard-layout, and all of its non-static data members and base classes are PODs.

Extern template

- In C++03, the compiler must instantiate a template whenever a fully specified template is encountered in a translation unit. If the template is instantiated with the same types in many translation units, this can dramatically increase compile times. There is no way to prevent this in C++03, so C++11 introduced extern template declarations, analogous to extern data declarations.

- `extern` in C++11 tells compiler to not instantiate a template.

Usability Enhancements

Type Inference

- `auto`, specifies that the type of the variable that is being declared will be automatically deduced from its initializer.
- `decltype`, inspects the declared type of an entity or the type and value category of an expression.
- `declval`, converts any type T to a reference type, making it possible to use member functions in decltype expressions without the need to go through constructors.

Type Deduction

- During type deduction, arguments that are references are treated as non-references, i.e., their reference-ness is ignored.
- When deducing types for by-value parameters, const and/or volatile arguments are treated as non-const and non-volatile.
- When deducing types for universal reference parameters, lvalue arguments get special treatment.
- During type deduction, arguments that are array or function names decay to pointers, unless they're used to initialize references.
- auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.
- auto x3 = { 27 }; is deduced to std::initializer_list<int> containing a single element with value 27.

Alternative Function Syntax

- Attribute specifier sequence, attributes provide the unified standard syntax for implementation-defined language extensions, such as the GNU and IBM language extensions __attribute__((...)), Microsoft extension __declspec(), etc.

- Trailing return types, allows specifying the function return type after the declaration of parameter declarations. The auto keyword is placed before the function identifier, which is the placeholder of the return type specifier.

Lambda Expressions

- C++11 provides support for anonymous functions, called lambda expressions. A lambda expression has the form:

```
[capture](parameters) -> return_type { body }
```

- Closures are defined between square brackets [and] in the declaration of lambda expression. The mechanism allows these variables to be captured by value or by reference.

```
[]        //no variables defined. Attempting to use any external variables in the lambda is an error.
[x, &y]   //x is captured by value, y is captured by reference
[&]       //any external variable is implicitly captured by reference if used
[=]       //any external variable is implicitly captured by value if used
[&, x]    //x is explicitly captured by value. Other variables will be captured by reference
[=, &z]   //z is explicitly captured by reference. Other variables will be captured by value
```

Object Construction Improvement

- C++11 allows constructors to call other peer constructors (termed delegation). This allows constructors to utilise another constructor's behaviour with a minimum of added code.

- C++03 considers an object to be constructed when its constructor finishes executing, but C++11 considers an object constructed once any constructor finishes execution.

- C++11 allows a class to specify that base class constructors will be inherited. Thus, the C++11 compiler will generate code to perform the inheritance, the forwarding of the derived class to the base class.

Strongly Typed Enumerations

- Strong typed enumeration solves type-safe problems with C++03 enumerations.
- Two enums from different type can not be compared.
- Enumeration values are scoped to the enclosing scope. With this two different enumeration types may have same member names.
- The default enum member type is `int`. This can be overridden to a different integral type and with that to make code more portable.

Initializer Lists and Uniform Initialization

- An object of type `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type const T.
- A `std::initializer_list` object is automatically constructed when:
  - a braced-init-list is used in list-initialisation, including function-call list initialisation and assignment expressions
  - a braced-init-list is bound to auto, including in a ranged for loop
- Initialiser lists may be implemented as a pair of pointers or pointer and length. Copying a `std::initializer_list` does not copy the underlying objects

Range-based for Loop

- Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

- This form of for, called the “range-based for”, will iterate over each element in the list. It will work for C-style arrays, initializer lists, and any type that has begin() and end() functions defined for it that return iterators.

- All the standard library containers that have begin/end pairs will work with the range-based for statement.

Explicit overrides and final

- The `override` special identifier means that the compiler will check the base classes to see if there is a virtual function with this exact signature. And if there is not, the compiler will indicate an error.
- The `final` adds the ability to prevent inheriting from classes or simply preventing overriding methods in derived classes.
- Neither override nor final are language keywords. They are technically identifiers for declarator attributes.

nullptr

- If NULL is defined as 0, the statement foo(NULL); will call foo(int), which is almost certainly not what the programmer intended, and not what a superficial reading of the code suggests.
- C++11 corrects this by introducing a new keyword to serve as a distinguished null pointer constant: nullptr. It is of type nullptr_t, which is implicitly convertible and comparable to any pointer type or pointer-to-member type.
- It is not implicitly convertible or comparable to integral types, except for bool.

Right Angle Bracket

- C++11 improves the specification of the parser so that multiple right angle brackets will be interpreted as closing the template argument list where it is reasonable. This can be overridden by using parentheses around parameter expressions using the “>”, “>=” or “>>” binary operators.

```
template<bool Test> class SomeType;
std::vector<SomeType<1>2>> x1;  // Interpreted as a std::vector of SomeType<true>,
    // followed by "2 >> x1", which is not legal syntax for a declarator. 1 is true.
std::vector<SomeType<(1>2)>> x1;  // Interpreted as std::vector of SomeType<false>,
    // followed by the declarator "x1", which is legal C++11 syntax. (1>2) is false.
```

Explicit Conversion Operators

- C++98 added the explicit keyword as a modifier on constructors to prevent single-argument constructors from being used as implicit type conversion operators.

- Now in C++11, the explicit keyword can now be applied to conversion operators. This feature solves cleanly the safe bool issue.

- Language contexts that specifically need a boolean value (if-statements, loops,...) count as explicit conversions and can thus use a bool conversion operator.

Type and Template Aliases

- In C++11 aliases can be declared using `using` keyword.
- There is no difference between a type alias declaration and typedef declaration.
- `using` allows creation of typedef templates.

Unrestricted Unions

- In C++11, there are no restrictions on what types of objects can be members of a union. For example, now unions can contain any objects that define a non-trivial constructor or destructor.

```
#include <new> // Needed for placement 'new'.

struct Point {
    Point() {}
    Point(int x, int y): x_(x), y_(y) {}
    int x_, y_;
};

union U {
    int z;
    double w;
    Point p; // Illegal in C++03; legal in C++11.
    U() {} // Due to the Point member, a constructor definition is now needed.
    U(const Point& pt) : p(pt) {} // Construct Point object using initializer list.
    U& operator=(const Point& pt) { new(&p) Point(pt); return *this; } // Assign
Point object using placement 'new'.
};
```