C++1x features cheatsheet

This file is not (at all) intended to be a thorough introduction to the main features of `C++1x`. Instead, it is supposed to be a guide to help you get started learning them. The detailed information about these features can be found, among other websites, at cppreference.com. You might want to also consider checking out the information provided in the Standard C++ foundation and ISO C++ standards committee.

Default constructor

A default constructor is a constructor which can be called with no arguments (either defined with an empty parameter list, or with default arguments provided for every parameter).

Explicit constructor

Specifies that a constructor or conversion function is explicit, that is, it cannot be used for implicit conversions and copy-initialization. Avoids behind-the-scenes type conversions without explicit casting operations.

`explicit`

The explicit specifier may be used with a constant expression. The function is explicit if and only if that constant expression evaluates to true.

Non-static member functions

A non-static member function is a function that is declared in a member specification of a class without a static or friend specifier.

Value initialization

This is the initialization performed when a variable is constructed with an empty initializer.

List initialization

Initializes an object from braced-init-list. Use this type of initialization as opposed to assignment statements when class attributes are objects with complex initializations. If a data member is const, there is no other way to initialize it. Use braces instead of parentheses.

Pointers

Pointer declarator: the declaration `S* D;` declares `D` as a pointer to the type determined by `decl-specifier-seq S`. Pointer to member declarator: the declaration `S C::* D;` declares `D` as a pointer to non-static member of `C` of type determined by `decl-specifier-seq S`.

References

Binds a reference to an object. A reference to `T` can be initialized with an object of type `T`, a function of type `T`, or an object implicitly convertible to `T`. Once initialized, a reference cannot be changed to refer to another object.

Lvalue and rvalue references

An lvalue is an expression that identifies a non-temporary object. An rvalue is an expression that identifies a temporary object or is a value (like a literal constant) not associated with any object. As a general rule, if there is a name for a variable, it is an lvalue, regardless whether it is modifiable. Intuitively, if the value of an expression did not exist before, it is likely a rvalue.

An lvalue reference becomes an alias for the object it references. An rvalue reference has the same characteristics as an lvalue reference except that, unlike an lvalue reference, it can also reference an rvalue (a temporary). Lvalue reference declarator: the declaration `S& D;` declares `D` as an lvalue reference to the type determined by `decl-specifier-seq S`. Rvalue reference declarator: the declaration `S&& D;` declares `D` as an rvalue reference to the type determined by `decl-specifier-seq S`. Its most common uses are for `operator=` and constructors.

Move constructors

A move constructor of `class T` is a non-template constructor whose first parameter is `T&&` or `const T&&`, and either there are no other parameters, or the rest of the parameters all have default values.

Move assignment operator

A move assignment operator of `class T` is a non-template non-static member function with the name `operator=` that takes exactly one parameter of type `T&&` or `const T&&`.

`std::move`

`std::move` is used to indicate that an object `t` may be ”moved from”, i.e. allowing the efficient transfer of resources from `t` to another object. In particular, `std::move` produces an xvalue expression that identifies its argument `t`. It is exactly equivalent to a `static_cast` to an rvalue reference type.

Copy elision

Related to `return` by value optimization. Omits copy and move (since C++11) constructors, resulting in zero-copy pass-by-value semantics.

Smart pointers

Library type that acts like a pointer but can be checked to see whether it is safe to use. The type takes care of deleting memory when appropriate.

Other language new features

1. Range `for`;
2. `auto`, `constexpr` and `nullptr`;
3. `decltype` and type aliases;
4. `std::tie`, `std::tuple`, and `<tuple>`;
5. Variadic `template`s, and `noexcept`;
6. Scoped `enum`s;
7. Literal types;
8. Static assertion;

Short `string` optimization

Follow other links therein.