Initialization

- process that provides the initial value of a variable at its time of construction
- occurs in:
  - declarator
  - new expression
  - function parameters
  - return values
- forms:
  - (expression-list)
    - comma-separated list of arbitrary expressions and braced-init-lists in parentheses
  - = expression
  - { initializer-list }
    - braced-init-list: possibly empty, comma-separated list of expressions and other braced-init-lists

Value initialization

- performed for construction with empty initializer

- forms:

  i. T();
  ii. new T();
  iii. Class::Class(...) : member() { ... }
  iv. T object{}; (C++11)
  v. T{}; (C++11)
  vi. new T{}; (C++11)
  vii. Class::Class(...) : member{} { ... } (C++11)

- occurs in initialization of:

  - 1 & 5 - nameless temporary object
  - 2 & 6 - object with dynamic storage duration
  - 3 & 7 - non-static member or base class initialization using member initialization
  - 4 - named variable (automatic, static, or thread-local) is declared with braced initialization
  - aggregate initialization is used when braces {} are used for an aggregate type
  - list initialization is performed for class types with no default ctor but a ctor taking a std::initializer_list

- effects:
  - calls defaults ctor if any user-provided ctor is defined (until C++11)
  - default-initialized for class types with no default ctor or a user-provided or deleted default ctor
  - for non-union types with no user provided ctors, all non-static data members and base-class components are value-initialized
  - for class types with a defaulted or implicit default ctor, the object is zero-initialized then default-initialized
  - for array types, each element is value-initialized
  - otherwise, zero-initialized
- notes:
  - IMPORTANT!!! T object(); does not initalize an object, declares a function
  - T object = T(); was the preferred format before C++11 (value-initializes a temporary and then copy initializes object, usually optimized out)
  - references cannot be value-initialized
  - see functional cast for use with arrays
  - all standard containers value initialize element when called with size_type or .resize()
  - since C++11, value-initializing a class without a user-provided ctor that has a member with a user-provided ctor zeroes out the member then calls its ctor

Direct initialization

- initializes with explicit set of ctor args
- forms:
  1. T object (arg);
  2. T object (arg1, arg2, ...);
  3. T object { arg }; (since C++11)
  4. T object { arg1, arg2, ... }; (since C++11)
  5. T (other)
  6. T (arg1, arg2, ...);
  7. static_cast(other)
  8. new T(args, ...)
  9. Class::Class() : member(args, ...) {...
  10. `[arg](){...` (since C++11)
- occurs in initialization of:
  1. with a nonempty argument list (expressions) in parentheses

ii. using braced init list if no initializer-list constructors are provided, a matching constructor is accessible, and all necessary implicit conversions are non-narrowing
iii. a prvalue temporary by functional cast or with a parenthesized expression list
iv. a prvalue temporary by a static_cast expression
v. an object with dynamic storage duration by a new-expression with a non-empty initializer
vi. a base or a non-static member by constructor initializer list
vii. closure object members from the variables caught by copy in a lambda-expression

- effects:
  - ctors are inspected and best match is used to initialize the object for class types
  - otherwise, standard conversions are used to convert from other to T (item 3.)

Copy initialization

- forms:
  - T object = other;
  - T object = {other}; (until C++11)
  - f(other)
  - return other;
  - throw object;
  - catch (T object)
  - `T array[N] = {other};`
- occurs:

i. when a named variable (automatic, static, or thread-local) of a non-reference type T is declared with the initializer consisting of an equals sign followed by an expression
ii. (until C++11) when a named variable of a scalar type T is declared with the initializer consisting of an equals sign followed by a brace-enclosed expression (Note: as of C++11, this is classified as list initialization, and narrowing conversion is not allowed)
iii. when passing an argument to a function by value
iv. when returning from a function that returns by value
v. when throwing or catching an exception by value
vi. as part of aggregate initialization, to initialize each element for which an initializer is provided

- effects:
  - finds best match of ctors for class types (when cv type of other is T or a derived class)
  - finds a user-defined conversion for class types when cv type of other is not T or derived and use direct initialization generally optimizing out any temporaries created to convert from

  - otherwise standard conversion are used (see implicit conversione)

List initialization

direct list initialization

```
1. T object { arg1, arg2, ... };
2. T { arg1, arg2, ... };
3. new T { arg1, arg2, ... }
4. Class { T member { arg1, arg2, ... }; };
5. Class::Class() : member{arg1, arg2, ...} {...
```

- occurs:
  i. named variable with braced init list
  ii. unnamed object with braced init list
  iii. dynamic object with new expression and braced init list
  iv. non-static member initializer without "="
  v. member initializer list of a ctor with braced init list

copy-list-initialization

```
6. T object = {arg1, arg2, ...};
7. function( { arg1, arg2, ... } );
8. return { arg1, arg2, ... };
9. `object[ { arg1, arg2, ... } ];`
10. object = { arg1, arg2, ... };
11. U( { arg1, arg2, ... } )
12. Class { T member = { arg1, arg2, ... }; };
```

- occurs: 6. named variable after "=" with braced init list 7. in function call as argument, braced init list initializes the corresponding argument 8. in a return statement, braced init list initializes the returned object 9. with user defined `operator[]` where list initialization initializes a parameter 10. in overloaded operator= where list initialization initializes a parameter 11. in a functional cast or other ctor type where list initialization initializes a ctor argument 12. in a non-static data member initializer using "="
- effects:

prevents narrowing conversions

- float type to int type
- long double to double
- long double to float
- double to float (except with constant expressions with no overflow)
- int type to float type (except with constant expressions with value that fits exactly in target)

- int type or unscoped enum type to integer type of smaller size (except with constant expressions with value that fits exactly in target)

Reference initialization

- forms:

  i. T & ref = object ;
  ii. T & ref = { arg1, arg2, ... };
  iii. T & ref (object) ;
  iv. T & ref { arg1, arg2, ... } ;
  v. T && ref = object ; (since C++11)
  vi. T && ref = { arg1, arg2, ... }; (since C++11)
  vii. T && ref (object) ; (since C++11)
  viii. T && ref { arg1, arg2, ... } ; (since C++11)
  ix. given R fn (T & arg); or R fn (T && arg);
  x. fn (object)
  xi. fn ({ arg1, arg2, ... })
  xii. given T & fn () { or T && fn () {
  xiii. return object ;
  xiv. Class::Class(...) : refmember(expr) {...}

- occurs when:

  i. a named lvalue reference variable is declared with an initializer
  ii. a named rvalue reference variable is declared with an initializer
  iii. a function call expression, when the function parameter has reference type
  iv. the return statement, when the function returns a reference type
  v. a non-static data member of reference type is initialized using a member initializer

lifetime of a temporary

- lifetime of a reference is extended past the lifetime of a corresponding temporary to match reference lifetime
- exceptions:
  - temporaries returned from functions (exists until function exits, dangling reference) !!!
  - a temporary bound to a reference member in a constructor initializer list (persists until ctor exits)
  - temporary bound to a function call parameter (exists until function exits, dangling reference) !!!

    - temporary bound to a reference in initializer in new expression

Default initialization

- initialization performed when a variable is constructed with no initializer
- forms:
    1. T object ;
    2. new T ;
    3. new T ( ) ; (until c++03)
- occurs when:
    1. a variable with automatic, static, or thread-local storage duration is declared with no initializer
    2. an object with dynamic storage duration is created by a new-expression with no initializer or when an object is created by a new-expression with the initializer consisting of an empty pair of parentheses (until C++03)
    3. a base class or a non-static data member is not mentioned in a constructor initializer list and that constructor is called
- effects:
    - for class types, chooses a default construct (user-defined, implicit, or defaulted) using an empty argument list
    - for array types, default initializes all elements
    - otherwise, nothing -> automatic storage objects have undefined values and use is undefined

Zero initialization

- sets initial value to 0
- forms:
    1. static T object ;
    2. T () ;
    3. T t = {} ;
    4. T {} ; (since C++11)
    5. `char array [ n ] = "";`
- occurs:
    1. for every named variable with static or thread-local storage duration that is not subject to constant initialization (since C++14), before any other initialization

    ii. as part of value-initialization sequence for non-class types and for members of value-initialized class types that have no constructors, including value initialization of elements of aggregates for which no initializers are provided
    iii. when a character array is initialized with a string literal that is too short, the remainder of the array is zero-initialized.

- effects:
    - if T is a scalar type, the object's initial value is the integral constant zero explicitly converted to T
    - if T is an non-union class type, all base classes and non-static data members are zero-initialized, and all padding is initialized to zero bits. The constructors, if any, are ignored.
    - if T is a union type, the first non-static named data member is zero-initialized and all padding is initialized to zero bits.
    - if T is array type, each element is zero-initialized
    - if T is reference type, nothing is done.

Constant initialization

- sets initial values of constants

- forms:

    i. static T & ref = constexpr;
    ii. static T object = constexpr;

- NOTE: until C++14, constant initialization was performed after zero initialization and after C++14 it is performed instead of zero initialization

- guaranteed to be complete before any other initialization of a static or thread-local object begins (may be performed at compile time)

- occurs for:

    i. static or thread-local references, bound to static glvalue, temporary object, or function (requires every expression in the initializer to be a constant expression)
    ii. static or thread-local object of calls type with constant expression ctor
    iii. static or thread-local object (not necessarily class type) without ctor call (value initialized or if every expression in initializer is a constant expression)

Aggregate initialization

- form of list initialization for aggregate types

- forms:
    - T object = { arg1, arg2, ... };
    - T object{ arg1, arg2, ... }; (C++11)

- aggregate type:

  - array type
  - class type (usually struct or union)
    - no private or protected non-static data members
    - no user provided ctors (defaulted or deleted ctors are allowed)
    - no virtual, private, or protected base classes (C++17)
    - no virtual member functions
    - no default member initializers (C++1 to C++14)

```
struct S {
    int x;
    struct Foo {
        int i;
        int j;
        int a[3];
    } b;
};

S s1 = { 1, { 2, 3, {4, 5, 6} } };
S s3{1, {2, 3, {4, 5, 6} } };
```

Member initializer lists

- before the body of a constructor and after a colon following the (parameter-list)
- forms:
  - class-or-identifier(expression-list)
    - uses direct initialization or value initialization if the expression-list is empty
  - class-or-identifier{ braced-init-list } (C++11)
    - uses list-initialization (value init for empty, aggregate otherwise)
  - parameter-pack...
    - initializes multiple base classes using parameter pack expansion

```
template<class... Mixins>
class X : public Mixins... {
 public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

Additional Notes

Vectors and defaults after reserve

- if the default ctor is defined, it is called

- if the default ctor does not do anything, all primitives are garbage
- if the default ctor is deleted, results in a compiler error