Creates and initializes objects with dynamic [storage duration](http://en.cppreference.com/w/cpp/language/storage_duration), that is, objects whose lifetime is not limited by the scope in which they were created.

**Syntax**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | |
| **::**(optional) **new** **(***placement\_params***)**(optional) **(** *type* **)** *initializer*(optional) | (1) |  |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| **::**(optional) **new** **(***placement\_params***)**(optional) *type* *initializer*(optional) | (2) |  |  |  |  |  |  |  |  |
|  | | | | | | | | | |

1) Attempts to create an object of type, denoted by the [type-id](http://en.cppreference.com/w/cpp/language/type#Type_naming) type, which may be array type, and may include the type specifier auto (since C++11)or the placeholder type decltype(auto) (since C++17)

2) Same, but type cannot include parentheses:

new int(\*[10])(); // error: parsed as (new int) (\*[10]) ()

new (int (\*[10])()); // okay: allocates an array of 10 pointers to functions

In addition, unparenthesized *type* is greedy: it will include every token that can be a part of a declarator:

new int + 1; // okay: parsed as (new int) + 1, increments a pointer returned by new int

new int \* 1; // error: parsed as (new int\*) (1)

Note: If auto is used in type, then initializer is not optional: it is required to deduce the type to use in place ofauto: auto p = new auto('c'); // creates a single object of type char. p is a char\*

**Explanation**

The new expression attempts to allocate storage and then attempts to construct and initialize either a single unnamed object, or an unnamed array of objects in the allocated storage. The new-expression returns a prvalue pointer to the constructed object or, if an array of objects was constructed, a pointer to the initial element of the array.

If type is an array type, all dimensions other than the first must be specified as positive [integral constant expression](http://en.cppreference.com/w/cpp/language/constant_expression)(until C++14)[converted constant expression](http://en.cppreference.com/w/cpp/language/constant_expression) of type [std::size\_t](http://en.cppreference.com/w/cpp/types/size_t) (since C++14), but the first dimension may be any expression convertible to [std::size\_t](http://en.cppreference.com/w/cpp/types/size_t). This is the only way to directly create an array with size defined at runtime, such arrays are often referred to as *dynamic arrays*:

int n = 42;

double a[n][5]; // error

auto p1 = new double[n][5]; // okay

auto p2 = new double[5][n]; // error

In the following cases the expression specifying the first dimension is erroneous:

* the expression is of non-class type and its value before conversion to [std::size\_t](http://en.cppreference.com/w/cpp/types/size_t) is negative;
* the expression is of class type and its value after user-defined conversion function and before the [second standard conversion](http://en.cppreference.com/w/cpp/language/implicit_cast) is negative;
* the value of the expression is larger than some implementation-defined limit;
* the value is smaller than the number of array elements provided in [the brace-enclosed initializer](http://en.cppreference.com/w/cpp/language/aggregate_initialization) (including the terminating '\0' on a [string literal](http://en.cppreference.com/w/cpp/language/string_literal)).

If the value in the first dimension is erroneous for any of these reasons,

* if, after conversion to [std::size\_t](http://en.cppreference.com/w/cpp/types/size_t), the first dimension is a [core constant expression](http://en.cppreference.com/w/cpp/language/constant_expression), the program is ill-formed (a compile-time error is issued);
* Otherwise, the new-expression does not call the allocation function, and instead throws an exception of type[std::bad\_array\_new\_length](http://en.cppreference.com/w/cpp/memory/new/bad_array_new_length) or derived from it (since C++11).

The first dimension of zero is acceptable, and the allocation function is called.

Note: [std::vector](http://en.cppreference.com/w/cpp/container/vector) offers similar functionality for one-dimensional dynamic arrays.

**Allocation**

The new-expression allocates storage by calling the appropriate [allocation function](http://en.cppreference.com/w/cpp/memory/new/operator_new). If type is a non-array type, the name of the function is operator new. If type is an array type, the name of the function is operator new[].

As described in [allocation function](http://en.cppreference.com/w/cpp/memory/new/operator_new), the C++ program may provide global and class-specific replacements for these functions. If the new-expression begins with the optional :: operator, as in ::new T or ::new T[n], class-specific replacements will be ignored (the function is [looked up](http://en.cppreference.com/w/cpp/language/lookup) in global [scope](http://en.cppreference.com/w/cpp/language/scope)). Otherwise, if T is a class type, it is looked up in the scope of T first.

When calling the allocation function, the new-expression passes the number of bytes requested as the first argument, of type [std::size\_t](http://en.cppreference.com/w/cpp/types/size_t), which is exactly sizeof(T) for non-array T.

Array allocation may supply unspecified overhead, which may vary from one call to new to the next. The pointer returned by the new-expression will be offset by that value from the pointer returned by the allocation function. Many implementations use the array overhead to store the number of objects in the array which is used by the [delete[]](http://en.cppreference.com/w/cpp/language/delete)expression to call the correct number of destructors. In addition, if the new-expression is used to allocate an array of char or an array unsigned char, it may request additional memory from the allocation function if necessary to guarantee correct alignment of objects of all types no larger than the requested array size, if one is later placed into the allocated array.

|  |  |
| --- | --- |
| New-expressions are allowed to elide or combine allocations made through replaceable allocation functions. In case of elision, the storage may be provided by the compiler without making the call to an allocation function (this also permits optimizing out unused new-expression). In case of combining, the allocation made by a new-expression E1 may be extended to provide additional storage for another new-expression E2 if all of the following is true:  1) The lifetime of the object allocated by E1 strictly contains the lifetime of the object allocated by E2,  2) E1 and E2 would invoke the same replaceable global allocation function  3) For a throwing allocation function, exceptions in E1 and E2 would be first caught in the same handler. | (since C++14) |

If placement\_params are provided, they are passed to the allocation function as additional arguments:

new T; // calls operator new(sizeof(T))

new T[5]; // calls operator new[](sizeof(T)\*5 + overhead)

new(2,f) T; // calls operator new(sizeof(T), 2, f)

Such allocation functions are known as "placement new", after the standard allocation function void\* [operator new](http://en.cppreference.com/w/cpp/memory/new/operator_new)([std::size\_t](http://en.cppreference.com/w/cpp/types/size_t), void\*), which simply returns its second argument unchanged. This is used to construct objects in allocated storage:

char\* ptr = new char[sizeof(T)]; // allocate memory

T\* tptr = new(ptr) T; // construct in allocated storage ("place")

tptr->~T(); // destruct

delete[] ptr; // deallocate memory

Note: this functionality is encapsulated by the member functions of the [Allocator](http://en.cppreference.com/w/cpp/concept/Allocator) classes.

If the allocation function return a null pointer, which is possible if the non-throwing overload was selected, e.g. with new([std::nothrow](http://en.cppreference.com/w/cpp/memory/new/nothrow)) T;, then the new-expression returns immediately, it does not attempt to initialize an object or to call a deallocation function. If the standard placement allocation function returns a null pointer, which is possible if the user passes a null pointer as the argument, the behavior is undefined. (since C++17)

**Construction**

The object created by a new-expression is initialized according to the following rules:

* For non-array type, the single object is constructed in the acquired memory area.
* If *initializer* is absent, the object is [default-initialized](http://en.cppreference.com/w/cpp/language/default_initialization).
* If *initializer* is a parenthesized list of arguments, the object is [direct-initialized](http://en.cppreference.com/w/cpp/language/direct_initialization).

|  |  |
| --- | --- |
| * If *initializer* is a brace-enclosed list of arguments, the object is [list-initialized](http://en.cppreference.com/w/cpp/language/list_initialization). | (since C++11) |

* If *type* is an array type, an array of objects is initialized.
* If *initializer* is absent, each element is [default-initialized](http://en.cppreference.com/w/cpp/language/default_initialization)
* If *initializer* is an empty pair of parentheses, each element is [value-initialized](http://en.cppreference.com/w/cpp/language/value_initialization).

|  |  |
| --- | --- |
| * If *initializer* is a brace-enclosed list of arguments, the array is [aggregate-initialized](http://en.cppreference.com/w/cpp/language/aggregate_initialization). | (since C++11) |

If initialization terminates by throwing an exception (e.g. from the constructor), if new-expression allocated any storage, it calls the appropriate [deallocation function](http://en.cppreference.com/w/cpp/memory/new/operator_delete): [operator delete](http://en.cppreference.com/w/cpp/memory/new/operator_delete) for non-array type, [operator delete](http://en.cppreference.com/w/cpp/memory/new/operator_delete)[]for array type. The deallocation function is looked up in global scope if the new-expression used the ::new syntax, otherwise it is looked up in the scope of T first, if T is a class type. If no deallocation function is found, memory is not deallocated. The value obtained earlier from the allocation function is passed as the first argument to the deallocation function, and placement\_params, if any, are passed as the additional arguments.

**Memory leaks**

The objects created by new-expressions (objects with dynamic storage duration) persist until the pointer returned by the new-expression is used in a matching [delete-expression](http://en.cppreference.com/w/cpp/language/delete). If the original value of pointer is lost, the object becomes unreachable and cannot be deallocated: a *memory leak* occurs.

This may happen if the pointer is assigned to:

int\* p = new int(7); // dynamically allocated int with value 7

p = nullptr; // memory leak

or if the pointer goes out of scope:

void f()

{

int\* p = new int(7);

} // memory leak

or due to exception

void f()

{

int\* p = new int(7);

g(); // may throw

delete p; // okay if no exception

} // memory leak if g() throws

To simplify management of dynamically-allocated objects, the result of a new-expression is often stored in a [*smart pointer*](http://en.cppreference.com/w/cpp/memory#Smart_pointers): [std::auto\_ptr](http://en.cppreference.com/w/cpp/memory/auto_ptr) (until C++17)[std::unique\_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr), or [std::shared\_ptr](http://en.cppreference.com/w/cpp/memory/shared_ptr) (since C++11). These pointers guarantee that the delete expression is executed in the situations shown above.