# Utility library

C++ includes a variety of utility libraries that provide functionality ranging from bit-counting to partial function application. These libraries can be broadly divided into two groups:

- language support libraries, and
- general-purpose libraries.

## Language support

Language support libraries provide classes and functions that interact closely with language features and support common language idioms.

### Type support

Basic types (e.g. `std::size_t`, `std::nullptr_t`), RTTI (e.g. `std::type_info`), type traits (e.g. `std::is_integral`, `std::rank`)

### Dynamic memory management

Smart pointers (e.g. `std::shared_ptr`), allocators (e.g. `std::allocator`), C-style memory management (e.g. `std::malloc`)

### Error handling

Exceptions (e.g. `std::exception`, `std::logic_error`), assertions (e.g. `assert`)

### Initializer lists

| | |
|---|---|
| **`initializer_list`** (C++11) | allows the use of initializer list syntax to initialize non-aggregate types<br>(class template) |

### Variadic functions

Support for functions that take an arbitrary number of parameters (via e.g. `va_start`, `va_arg`, `va_end`)

## General-purpose utilities

### Program utilities

Termination (e.g. `std::abort`, `std::atexit`), environment (e.g. `std::system`), signals (e.g. `std::raise`)

### Date and time

Time tracking (e.g. `std::chrono::time_point`, `std::chrono::duration`), C-style date and time (e.g. `std::time`, `std::clock`)

### Bitset

| | |
|---|---|
| **`bitset`** | implements constant length bit array<br>(class) |

### Function objects

Partial function application (e.g. `std::bind`) and related utilities: utilities for binding such as `std::ref` and `std::placeholders`, polymorphic function wrappers: `std::function`, predefined functors (e.g. `std::plus`, `std::equal_to`), method to function converters `std::mem_fn`.

### Pairs and tuples

| | |
|---|---|
| **`pair`** | implements binary tuple, i.e. a pair of values<br>(class template) |
| **`tuple`** (C++11) | implements fixed size container, which holds elements of possibly different types<br>(class template) |
| **`piecewise_construct_t`** (C++11) | tag type used to select correct function overload for piecewise construction<br>(class) |
| **`piecewise_construct`** (C++11) | an object of type `piecewise_construct_t` used to disambiguate functions for piecewise construction<br>(constant) |
| **`integer_sequence`** (C++14) | implements compile-time sequence of integers<br>(class template) |

### Swap, forward and move

| | |
|---|---|
| **`swap`** | swaps the values of two objects<br>(function template) |
| **`exchange`** (C++14) | replaces the argument with a new value and returns its previous value<br>(function template) |
| **`forward`** (C++11) | forwards a function argument<br>(function template) |
| **`move`** (C++11) | obtains an rvalue reference<br>(function template) |
| **`move_if_noexcept`** (C++11) | obtains an rvalue reference if the move constructor does not throw<br>(function template) |
| **`declval`** (C++11) | obtains the type of expression in unevaluated context<br>(function template) |

### Relational operators

Defined in namespace `std::rel_ops`

| | |
|---|---|
| **`operator!=`**<br>**`operator>`**<br>**`operator<=`**<br>**`operator>=`** | automatically generates comparison operators based on user-defined `operator==` and `operator<`<br>(function template) |

### Hash support

| | |
|---|---|
| **`hash`** (C++11) | hash function object<br>(class template) |

# Program support utilities

## Program termination

The following functions manage program termination and resource cleanup.

Defined in header <cstdlib>

| | |
|---|---|
| **abort** | causes abnormal program termination (without cleaning up) (function) |
| **exit** | causes normal program termination with cleaning up (function) |
| **quick_exit** (C++11) | causes quick program termination without completely cleaning up (function) |
| **_Exit** (C++11) | causes normal program termination without cleaning up (function) |
| **atexit** | registers a function to be called on `std::exit()` invocation (function) |
| **at_quick_exit** (C++11) | registers a function to be called on `quick_exit` invocation (function) |
| **EXIT_SUCCESS**<br>**EXIT_FAILURE** | indicates program execution execution status (macro constant) |

## Communicating with the environment

| | |
|---|---|
| **system** | calls the host environment's command processor (function) |
| **getenv** | access to the list of environment variables (function) |

## Signals

Several functions and macro constants for signal management are provided.

Defined in header <csignal>

| | |
|---|---|
| **signal** | sets a signal handler for particular signal (function) |
| **raise** | runs the signal handler for particular signal (function) |
| **sig_atomic_t** | the integer type that can be accessed as an atomic entity from an asynchronous signal handler (typedef) |
| **SIG_DFL**<br>**SIG_IGN** | defines signal handling strategies (macro constant) |
| **SIG_ERR** | return value of `signal` specifying that an error was encountered (macro constant) |

**Signal types**

| | |
|---|---|
| **SIGABRT**<br>**SIGFPE**<br>**SIGILL**<br>**SIGINT**<br>**SIGSEGV**<br>**SIGTERM** | defines signal types (macro constant) |

## Non-local jumps

Defined in header <csetjmp>

| | |
|---|---|
| **setjmp** | saves the context (function macro) |
| **longjmp** | jumps to specified location |

(function)

**Types**

| | |
|---|---|
| **jmp_buf** | execution context type<br>(typedef) |

## See also

| |
|---|
| **C documentation** for **Program support utilities** |

# Variadic functions

Variadic functions are functions (e.g. `std::printf`) which take a variable number of arguments.

To declare a variadic function, an ellipsis is used as the last parameter, e.g. `int printf(const char* format, ...);`. See Variadic arguments for additional detail on the syntax, automatic argument conversions and the alternatives.

To access the variadic arguments from the function body, the following library facilities are provided:

Defined in header `<cstdarg>`

| | |
|---|---|
| **va_start** | enables access to variadic function arguments (function macro) |
| **va_arg** | accesses the next variadic function argument (function macro) |
| **va_copy** (C++11) | makes a copy of the variadic function arguments (function macro) |
| **va_end** | ends traversal of the variadic function arguments (function macro) |
| **va_list** | holds the information needed by va_start, va_arg, va_end, and va_copy (typedef) |

### Example

Run this code

```
#include <iostream>
#include <cstdarg>

void simple_printf(const char* fmt...)
{
    va_list args;
    va_start(args, fmt);

    while (*fmt != '\0') {
        if (*fmt == 'd') {
            int i = va_arg(args, int);
            std::cout << i << '\n';
        } else if (*fmt == 'c') {
            // note automatic conversion to integral type
            int c = va_arg(args, int);
            std::cout << static_cast<char>(c) << '\n';
        } else if (*fmt == 'f') {
            double d = va_arg(args, double);
            std::cout << d << '\n';
        }
        ++fmt;
    }

    va_end(args);
}

int main()
{
    simple_printf("dcff", 3, 'a', 1.999, 42.5);
}
```

Output:

```
3
a
1.999
42.5
```

### See also

**C documentation** for **`Variadic functions`**

# Function objects

A *function object* is any object for which the function call operator is defined. C++ provides many built-in function objects as well as support for creation and manipulation of new function objects.

## Polymorphic function wrappers

`std::function` provides support for storing arbitrary function objects.

| | |
|---|---|
| **`function`** (C++11) | wraps callable object of any type with specified function call signature<br>(class template) |
| **`mem_fn`** (C++11) | creates a function object out of a pointer to a member<br>(function template) |
| **`bad_function_call`** (C++11) | the exception thrown when invoking an empty `std::function`<br>(class) |
| **`invoke`** (C++17) | invokes any `Callable` object with given arguments<br>(function template) |

## Bind

`std::bind` provides support for partial function application , i.e. binding arguments to functions to produce new functions.

| | |
|---|---|
| **`bind`** (C++11) | binds one or more arguments to a function object<br>(function template) |
| **`is_bind_expression`** (C++11) | indicates that an object is `std::bind` expression or can be used as one<br>(class template) |
| **`is_placeholder`** (C++11) | indicates that an object is a standard placeholder or can be used as one<br>(class template) |
| Defined in namespace `std::placeholders` | |
| **`_1, _2, _3, _4, ...`** (C++11) | placeholders for the unbound arguments in a `std::bind` expression<br>(constant) |

## Reference wrappers

Reference wrappers allow reference arguments to be stored in copyable function objects:

| | |
|---|---|
| **`reference_wrapper`** (C++11) | `CopyConstructible` and `CopyAssignable` reference wrapper<br>(class template) |
| **`ref`** (C++11)<br>**`cref`** (C++11) | creates a `std::reference_wrapper` with a type deduced from its argument<br>(function template) |

## Function objects

C++ defines several function objects that represent common arithmetic and logical operations:

#### Arithmetic operations

| | |
|---|---|
| **`plus`** | function object implementing `x + y`<br>(class template) |
| **`minus`** | function object implementing `x - y`<br>(class template) |
| **`multiplies`** | function object implementing `x * y`<br>(class template) |
| **`divides`** | function object implementing `x / y`<br>(class template) |
| **`modulus`** | function object implementing `x % y`<br>(class template) |

| | |
|---|---|
| **negate** | function object implementing `-x`<br>(class template) |

#### Comparisons

| | |
|---|---|
| **equal_to** | function object implementing `x == y`<br>(class template) |
| **not_equal_to** | function object implementing `x != y`<br>(class template) |
| **greater** | function object implementing `x > y`<br>(class template) |
| **less** | function object implementing `x < y`<br>(class template) |
| **greater_equal** | function object implementing `x >= y`<br>(class template) |
| **less_equal** | function object implementing `x <= y`<br>(class template) |

#### Logical operations

| | |
|---|---|
| **logical_and** | function object implementing `x && y`<br>(class template) |
| **logical_or** | function object implementing `x \|\| y`<br>(class template) |
| **logical_not** | function object implementing `!x`<br>(class template) |

#### Bitwise operations

| | |
|---|---|
| **bit_and** | function object implementing `x & y`<br>(class template) |
| **bit_or** | function object implementing `x \| y`<br>(class template) |
| **bit_xor** | function object implementing `x ^ y`<br>(class template) |
| **bit_not** (C++14) | function object implementing `~x`<br>(class template) |

#### Negators

| | |
|---|---|
| **unary_negate** | wrapper function object returning the complement of the unary predicate it holds<br>(class template) |
| **binary_negate** | wrapper function object returning the complement of the binary predicate it holds<br>(class template) |
| **not1** | constructs custom `std::unary_negate` object<br>(function template) |
| **not2** | constructs custom `std::binary_negate` object<br>(function template) |

### Deprecated in C++11

Several utilities that provided early functional support are deprecated in C++11:

#### Base

| | |
|---|---|
| **unary_function** (until C++17) | adaptor-compatible unary function base class<br>(class template) |
| **binary_function** (until C++17) | adaptor-compatible binary function base class<br>(class template) |

**Binders**

| | |
|---|---|
| **binder1st** (until C++17)<br>**binder2nd** (until C++17) | function object holding a binary function and one of its arguments<br>(class template) |
| **bind1st** (until C++17)<br>**bind2nd** (until C++17) | binds one argument to a binary function<br>(function template) |

**Function adaptors**

| | |
|---|---|
| **pointer_to_unary_function** (until C++17) | adaptor-compatible wrapper for a pointer to unary function<br>(class template) |
| **pointer_to_binary_function** (until C++17) | adaptor-compatible wrapper for a pointer to binary function<br>(class template) |
| **ptr_fun** (until C++17) | creates an adaptor-compatible function object wrapper from a pointer to function<br>(function template) |
| **mem_fun_t** (until C++17)<br>**mem_fun1_t** (until C++17)<br>**const_mem_fun_t** (until C++17)<br>**const_mem_fun1_t** (until C++17) | wrapper for a pointer to nullary or unary member function, callable with a pointer to object<br>(class template) |
| **mem_fun** (until C++17) | creates a wrapper from a pointer to member function, callable with a pointer to object<br>(function template) |
| **mem_fun_ref_t** (until C++17)<br>**mem_fun1_ref_t** (until C++17)<br>**const_mem_fun_ref_t** (until C++17)<br>**const_mem_fun1_ref_t** (until C++17) | wapper for a pointer to nullary or unary member function, callable with a reference to object<br>(class template) |
| **mem_fun_ref** (until C++17) | creates a wrapper from a pointer to member function, callable with a reference to object<br>(function template) |

# std::initializer_list

(not to be confused with member initializer list)

Defined in header <initializer_list>

```
template< class T >
class initializer_list;
```
(since C++11)

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-initialization, including function-call list initialization and assignment expressions
- a *braced-init-list* is bound to `auto`, including in a ranged for loop

Initializer lists may be implemented as a pair of pointers or pointer and length. Copying a `std::initializer_list` does not copy the underlying objects.

| | |
|---|---|
| The underlying array is not guaranteed to exist after the lifetime of the original initializer list object has ended. The storage for `std::initializer_list` is unspecified (i.e. it could be automatic, temporary, or static read-only memory, depending on the situation). | (until C++14) |
| The underlying array is a temporary array, in which each element is copy-initialized (except that narrowing conversions are invalid) from the corresponding element of the original initializer list. The lifetime of the underlying array is the same as any other temporary object, except that initializing an initializer_list object from the array extends the lifetime of the array exactly like binding a reference to a temporary (with the same exceptions, such as for initializing a non-static class member). The underlying array may be allocated in read-only memory. | (since C++14) |
| The program is ill-formed if an explicit or partial specialization of `std::initializer_list` is declared. | (since C++17) |

## Member types

| Member type | Definition |
|---|---|
| `value_type` | `T` |
| `reference` | `const T&` |
| `const_reference` | `const T&` |
| `size_type` | `std::size_t` |
| `iterator` | `const T*` |
| `const_iterator` | `const T*` |

## Member functions

| | |
|---|---|
| (constructor) | creates an empty initializer list<br>(public member function) |

### Capacity

| | |
|---|---|
| **`size`** | returns the number of elements in the initializer list<br>(public member function) |

### Iterators

| | |
|---|---|
| **`begin`** | returns a pointer to the first element<br>(public member function) |
| **`end`** | returns a pointer to one past the last element<br>(public member function) |

### Non-member functions

| | |
|---|---|
| **std::begin**(std::initializer_list) (C++11) | specializes std::begin (function template) |
| **std::end**(std::initializer_list) (C++11) | specializes std::end (function template) |
| Defined in header <iterator> | |
| **rbegin**(std::initializer_list) (C++14) | specializes std::rbegin (function) |
| **rend**(std::initializer_list) (C++14) | specializes std::rend (function) |

### Example

Run this code

```
#include <iostream>
#include <vector>
#include <initializer_list>

template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) {
         std::cout << "constructed with a " << l.size() << "-element list\n";
    }
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
    std::pair<const T*, std::size_t> c_arr() const {
        return {&v[0], v.size()};  // copy list-initialization in return statement
                                   // this is NOT a use of std::initializer_list
    }
};

template <typename T>
void templated_fn(T) {}

int main()
{
    S<int> s = {1, 2, 3, 4, 5}; // copy list-initialization
    s.append({6, 7, 8});      // list-initialization in function call

    std::cout << "The vector size is now " << s.c_arr().second << " ints:\n";

    for (auto n : s.v)
        std::cout << n << ' ';
    std::cout << '\n';

    std::cout << "Range-for over brace-init-list: \n";

    for (int x : {-1, -2, -3}) // the rule for auto makes this ranged-for work
        std::cout << x << ' ';
    std::cout << '\n';

    auto al = {10, 11, 12};   // special rule for auto

    std::cout << "The list bound to auto has size() = " << al.size() << '\n';

//    templated_fn({1, 2, 3}); // compiler error! "{1, 2, 3}" is not an expression,
                             // it has no type, and so T cannot be deduced
    templated_fn<std::initializer_list<int>>({1, 2, 3}); // OK
    templated_fn<std::vector<int>>({1, 2, 3});           // also OK
}
```

Output:

```
constructed with a 5-element list
The vector size is now 8 ints:
1 2 3 4 5 6 7 8
Range-for over brace-init-list:
-1 -2 -3
The list bound to auto has size() = 3
```

# std::bitset

Defined in header <bitset>

```
template< std::size_t N >
class bitset;
```

The class template `bitset` represents a fixed-size sequence of `N` bits. Bitsets can be manipulated by standard logic operators and converted to and from strings and integers.

`bitset` meets the requirements of `CopyConstructible` and `CopyAssignable`.

## Template parameters

**N** - the number of bits to allocate storage for

## Member types

| | |
|---|---|
| **reference** | proxy class representing a reference to a bit (class) |

## Member functions

| | |
|---|---|
| (constructor) | constructs the bitset (public member function) |
| **operator==**<br>**operator!=** | compares the contents (public member function) |

### Element access

| | |
|---|---|
| **operator[]** | accesses specific bit (public member function) |
| **test** | accesses specific bit (public member function) |
| **all** (C++11)<br>**any**<br>**none** | checks if all, any or none bits are set to `true` (public member function) |
| **count** | returns the number of bits set to `true` (public member function) |

### Capacity

| | |
|---|---|
| **size** | returns the size number of bits that the bitset can hold (public member function) |

### Modifiers

| | |
|---|---|
| **operator&=**<br>**operator\|=**<br>**operator^=**<br>**operator~** | performs binary AND, OR, XOR and NOT (public member function) |
| **operator<<=**<br>**operator>>=**<br>**operator<<**<br>**operator>>** | performs binary shift left and shift right (public member function) |
| **set** | sets bits to `true` or given value (public member function) |
| **reset** | sets bits to `false` (public member function) |

| | |
|---|---|
| **flip** | toggles the values of bits<br>(public member function) |

**Conversions**

| | |
|---|---|
| **to_string** | returns a string representation of the data<br>(public member function) |
| **to_ulong** | returns an `unsigned long` integer representation of the data<br>(public member function) |
| **to_ullong** (C++11) | returns an `unsigned long long` integer representation of the data<br>(public member function) |

## Non-member functions

| | |
|---|---|
| **operator&**<br>**operator\|**<br>**operator^** | performs binary logic operations on bitsets<br>(function template) |
| **operator<<**<br>**operator>>** | performs stream input and output of bitsets<br>(function template) |

## Helper classes

| | |
|---|---|
| **std::hash**<std::bitset> (C++11) | hash support for **std::bitset**<br>(class template specialization) |

## Notes

If the size of the bitset is not known at compile time, std::vector<bool> or boost::dynamic_bitset (http%3A//www.boost.org/doc/libs/release/libs/dynamic_bitset/dynamic_bitset.html) may be used.

# std::hash

Defined in header <functional>

```
template< class Key >
struct hash;
```
(since C++11)

The hash template defines a function object that implements a hash function . Instances of this function object satisfy `Hash`. In particular, they define an `operator()` that:

1. Accepts a single parameter of type `Key`.

2. Returns a value of type `size_t` that represents the hash value of the parameter.

3. Does not throw exceptions when called.

4. For two parameters `k1` and `k2` that are equal, `std::hash<Key>()(k1) == std::hash<Key>()(k2)`.

5. For two different parameters `k1` and `k2` that are not equal, the probability that `std::hash<Key>()(k1) == std::hash<Key>()(k2)` should be very small, approaching `1.0/std::numeric_limits<size_t>::max()`.

The hash template is `DefaultConstructible`, `CopyAssignable`, `Swappable` and `Destructible`.

The unordered associative containers `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, `std::unordered_multimap` use specializations of the template **`std::hash`** as the default hash function.

## Notes

The actual hash functions are implementation-dependent and are not required to fulfill any other quality criteria except those specified above. Notably, some implementations use trivial (identity) hash functions which map an integer to itself. In other words, these hash functions are designed to work with unordered associative containers, but not as cryptographic hashes, for example.

Hash functions are only required to produce the same result for the same input within a single execution of a program; this allows salted hashes that prevent collision DoS attacks. (since C++14)

There is no specialization for C strings. `std::hash<const char*>` produces a hash of the value of the pointer (the memory address), it does not examine the contents of any character array.

## Member types

| Member type | Definition |
|---|---|
| `argument_type` | `Key` |
| `result_type` | `std::size_t` |

## Member functions

| | |
|---|---|
| (constructor) | constructs a hash function object<br>(public member function) |
| **`operator()`** | calculate the hash of the argument<br>(public member function) |

## Standard specializations for basic types

Defined in header <functional>

```
template<> struct hash<bool>;
```

```
template<> struct hash<char>;
template<> struct hash<signed char>;
template<> struct hash<unsigned char>;
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<short>;
template<> struct hash<unsigned short>;
template<> struct hash<int>;
template<> struct hash<unsigned int>;
template<> struct hash<long>;
template<> struct hash<long long>;
template<> struct hash<unsigned long>;
template<> struct hash<unsigned long long>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;
template< class T > struct hash<T*>;
```

In addition to the above, standard library provides specializations for all (scoped and unscoped) enumeration types (which are not required, but usually are implemented as `std::hash<std::underlying_type<Enum>::type>`) (since C++14)

### Standard specializations for library types

| | |
|---|---|
| **std::hash**<std::string> (C++11)<br>**std::hash**<std::u16string> (C++11)<br>**std::hash**<std::u32string> (C++11)<br>**std::hash**<std::wstring> (C++11) | hash support for strings<br>(class template specialization) |
| **std::hash**<std::error_code> (C++11) | hash support for `std::error_code`<br>(class template specialization) |
| **std::hash**<std::bitset> (C++11) | hash support for `std::bitset`<br>(class template specialization) |
| **std::hash**<std::unique_ptr> (C++11) | hash support for `std::unique_ptr`<br>(class template specialization) |
| **std::hash**<std::shared_ptr> (C++11) | hash support for `std::shared_ptr`<br>(class template specialization) |
| **std::hash**<std::type_index> (C++11) | hash support for `std::type_index`<br>(class template specialization) |
| **std::hash**<std::vector<bool>> (C++11) | hash support for `std::vector<bool>`<br>(class template specialization) |
| **std::hash**<std::thread::id> (C++11) | hash support for `std::thread::id`<br>(class template specialization) |

Note: additional specializations for `std::pair` and the standard container types, as well as utility functions to compose hashes are available in boost.hash (http%3A//www.boost.org/doc/libs/release/doc/html/hash/reference.html)

### Examples

Demonstrates the computation of a hash for `std::string`, a type that already has a hash specialization.

Run this code

```
#include <iostream>
#include <functional>
#include <string>

int main()
{
    std::string str = "Meet the new boss...";
    std::hash<std::string> hash_fn;
    std::size_t str_hash = hash_fn(str);

    std::cout << str_hash << '\n';
```

```
}
```

Possible output:

```
391070135
```

Demonstrates creation of a hash function for a user defined type. Using this as a template parameter for `std::unordered_map`, `std::unordered_set`, etc. also requires specialization of `std::equal_to`.

Run this code

```
#include <iostream>
#include <functional>
#include <string>

struct S
{
    std::string first_name;
    std::string last_name;
};

template <class T>
class MyHash;

template<>
class MyHash<S>
{
public:
    std::size_t operator()(S const& s) const
    {
        std::size_t h1 = std::hash<std::string>()(s.first_name);
        std::size_t h2 = std::hash<std::string>()(s.last_name);
        return h1 ^ (h2 << 1);
    }
};

int main()
{
    std::string s1 = "Hubert";
    std::string s2 = "Farnsworth";
    std::hash<std::string> h1;

    S n1;
    n1.first_name = s1;
    n1.last_name =  s2;

    std::cout << "hash(s1) = " << h1(s1) << "\n"
              << "hash(s2) = " << std::hash<std::string>()(s2) << "\n"
              << "hash(n1) = " << MyHash<S>()(n1) << "\n";

}
```

Possible output:

```
hash(s1) = 6119893563398376542
hash(s2) = 14988020022735710972
hash(n1) = 17649170831080298918
```

Demonstrates how to specialize **`std::hash`** for a user defined type.

Run this code

```
#include <iostream>
#include <functional>
#include <string>
```

```
struct S
{
    std::string first_name;
    std::string last_name;
};

namespace std
{
    template<>
    struct hash<S>
    {
        typedef S argument_type;
        typedef std::size_t result_type;

        result_type operator()(argument_type const& s) const
        {
            result_type const h1 ( std::hash<std::string>()(s.first_name) );
            result_type const h2 ( std::hash<std::string>()(s.last_name) );
            return h1 ^ (h2 << 1);
        }
    };
}

int main()
{
    S s;
    s.first_name = "Bender";
    s.last_name =  "Rodriguez";
    std::hash<S> hash_fn;

    std::cout << "hash(s) = " << hash_fn(s) << "\n";
}
```

Possible output:

```
hash(s) = 32902390710
```

# std::rel_ops::operator!=,>,<=,>=

Defined in header <utility>

```
template< class T >
bool operator!=( const T& lhs, const T& rhs );
```
(1)

```
template< class T >
bool operator>( const T& lhs, const T& rhs );
```
(2)

```
template< class T >
bool operator<=( const T& lhs, const T& rhs );
```
(3)

```
template< class T >
bool operator>=( const T& lhs, const T& rhs );
```
(4)

Given a user-defined `operator==` and `operator<` for objects of type `T`, implements the usual semantics of other comparison operators.

1) Implements `operator!=` in terms of `operator==`.

2) Implements `operator>` in terms of `operator<`.

3) Implements `operator<=` in terms of `operator<`.

4) Implements `operator>=` in terms of `operator<`.

### Parameters

**lhs** - left-hand argument
**rhs** - right-hand argument

### Return value

1) Returns `true` if `lhs` is *not equal* to `rhs`.

2) Returns `true` if `lhs` is *greater* than `rhs`.

3) Returns `true` if `lhs` is *less or equal* to `rhs`.

4) Returns `true` if `lhs` is *greater or equal* to `rhs`.

### Possible implementation

**First version**

```
namespace rel_ops {
    template< class T >
    bool operator!=( const T& lhs, const T& rhs )
    {
        return !(lhs == rhs);
    }
}
```

**Second version**

```
namespace rel_ops {
    template< class T >
    bool operator>( const T& lhs, const T& rhs )
    {
        return rhs < lhs;
```

```
    }
}
```

**Third version**

```
namespace rel_ops {
    template< class T >
    bool operator<=( const T& lhs, const T& rhs )
    {
        return !(rhs < lhs);
    }
}
```

**Fourth version**

```
namespace rel_ops {
    template< class T >
    bool operator>=( const T& lhs, const T& rhs )
    {
        return !(lhs < rhs);
    }
}
```

## Notes

Boost.operators (http%3A//www.boost.org/doc/libs/release/libs/utility/operators.htm) provides a more versatile alternative to `std::rel_ops`

## Example

Run this code

```
#include <iostream>
#include <utility>

struct Foo {
    int n;
};

bool operator==(const Foo& lhs, const Foo& rhs)
{
    return lhs.n == rhs.n;
}

bool operator<(const Foo& lhs, const Foo& rhs)
{
    return lhs.n < rhs.n;
}

int main()
{
    Foo f1 = {1};
    Foo f2 = {2};
    using namespace std::rel_ops;

    std::cout << std::boolalpha;
    std::cout << "not equal?     : " << (f1 != f2) << '\n';
    std::cout << "greater?       : " << (f1 > f2) << '\n';
    std::cout << "less equal?    : " << (f1 <= f2) << '\n';
    std::cout << "greater equal? : " << (f1 >= f2) << '\n';
}
```

Output:

```
not equal?     : true
greater?       : false
less equal?    : true
greater equal? : false
```

# std::pair

Defined in header <utility>

```
template<
    class T1,
    class T2
> struct pair;
```

std::pair is a struct template that provides a way to store two heterogeneous objects as a single unit. A pair is a specific case of a std::tuple with two elements.

### Template parameters

**T1, T2** - the types of the elements that the pair stores.

### Member types

| Member type | Definition |
|---|---|
| first_type | T1 |
| second_type | T2 |

### Member objects

| Member name | Type |
|---|---|
| first | T1 |
| second | T2 |

### Member functions

| | |
|---|---|
| (constructor) | constructs new pair<br>(public member function) |
| **operator=** | assigns the contents<br>(public member function) |
| **swap** (C++11) | swaps the contents<br>(public member function) |

### Non-member functions

| | |
|---|---|
| **make_pair** | creates a pair object of type, defined by the argument types<br>(function template) |
| **operator==**<br>**operator!=**<br>**operator<**<br>**operator<=**<br>**operator>**<br>**operator>=** | lexicographically compares the values in the pair<br>(function template) |
| **std::swap**(std::pair) (C++11) | specializes the std::swap algorithm<br>(function template) |
| **std::get**(std::pair) (C++11) | accesses an element of a pair<br>(function template) |

### Helper classes

| | |
|---|---|
| **std::tuple_size**<std::pair> (C++11) | obtains the size of a pair<br>(class template specialization) |

| | |
|---|---|
| **std::tuple_element**<std::pair> (C++11) | obtains the type of the elements of `pair`<br>(class template specialization) |

## See also

| | |
|---|---|
| **tuple** (C++11) | implements fixed size container, which holds elements of possibly different types<br>(class template) |

# std::tuple

Defined in header <tuple>

```
template< class... Types >
class tuple;
```
(since C++11)

Class template `std::tuple` is a fixed-size collection of heterogeneous values. It is a generalization of `std::pair`.

### Template parameters

**`Types...`** - the types of the elements that the tuple stores. Empty list is supported.

### Member functions

| | |
|---|---|
| (constructor) | constructs a new `tuple`<br>(public member function) |
| **`operator=`** | assigns the contents of one `tuple` to another<br>(public member function) |
| **`swap`** | swaps the contents of two `tuples`<br>(public member function) |

### Non-member functions

| | |
|---|---|
| **`make_tuple`** | creates a `tuple` object of the type defined by the argument types<br>(function template) |
| **`tie`** | creates a `tuple` of lvalue references or unpacks a tuple into individual objects<br>(function template) |
| **`forward_as_tuple`** | creates a `tuple` of rvalue references<br>(function template) |
| **`tuple_cat`** | creates a `tuple` by concatenating any number of tuples<br>(function template) |
| **`std::get`**`(std::tuple)` | tuple accesses specified element<br>(function template) |
| **`operator==`**<br>**`operator!=`**<br>**`operator<`**<br>**`operator<=`**<br>**`operator>`**<br>**`operator>=`** | lexicographically compares the values in the tuple<br>(function template) |
| **`std::swap`**`(std::tuple)` (C++11) | specializes the `std::swap` algorithm<br>(function template) |

### Helper classes

| | |
|---|---|
| **`tuple_size`** | obtains the size of `tuple` at compile time<br>(class template specialization) |
| **`tuple_element`** | obtains the type of the specified element<br>(class template specialization) |
| **`std::uses_allocator`**`<std::tuple>` (C++11) | specializes the `std::uses_allocator` type trait<br>(class template specialization) |
| **`ignore`** | placeholder to skip an element when unpacking a `tuple` using `tie`<br>(constant) |

### Notes

Until C++17, a function could not return a tuple using list-initialization:

```
std::tuple<int, int> foo_tuple()
{
  return {1, -1};  // Error until C++17
  return std::make_tuple(1, -1); // Always works
}
```

### Example

Run this code

```
#include <tuple>
#include <iostream>
#include <string>
#include <stdexcept>

std::tuple<double, char, std::string> get_student(int id)
{
    if (id == 0) return std::make_tuple(3.8, 'A', "Lisa Simpson");
    if (id == 1) return std::make_tuple(2.9, 'C', "Milhouse Van Houten");
    if (id == 2) return std::make_tuple(1.7, 'D', "Ralph Wiggum");
    throw std::invalid_argument("id");
}

int main()
{
    auto student0 = get_student(0);
    std::cout << "ID: 0, "
              << "GPA: " << std::get<0>(student0) << ", "
              << "grade: " << std::get<1>(student0) << ", "
              << "name: " << std::get<2>(student0) << '\n';

    double gpa1;
    char grade1;
    std::string name1;
    std::tie(gpa1, grade1, name1) = get_student(1);
    std::cout << "ID: 1, "
              << "GPA: " << gpa1 << ", "
              << "grade: " << grade1 << ", "
              << "name: " << name1 << '\n';
}
```

Output:

```
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson
ID: 1, GPA: 2.9, grade: C, name: Milhouse Van Houten
```

### References

- C++11 standard (ISO/IEC 14882:2011):
  - 20.4 Tuples [tuple]

# std::piecewise_construct_t

Defined in header <utility>

```
struct piecewise_construct_t { };    (since C++11)
```

**std::piecewise_construct_t** is an empty struct tag type used to disambiguate between different functions that take two tuple arguments.

The overloads that do not use **std::piecewise_construct_t** assume that each tuple argument becomes the element of a pair. The overloads that use **std::piecewise_construct_t** assume that each tuple argument is used to construct, piecewise, a new object of specified type, which will become the element of the pair.

## Example

Run this code

```
#include <iostream>
#include <utility>
#include <tuple>

struct Foo {
    Foo(std::tuple<int, float>)
    {
        std::cout << "Constructed a Foo from a tuple\n";
    }
    Foo(int, float)
    {
        std::cout << "Constructed a Foo from an int and a float\n";
    }
};

int main()
{
    std::tuple<int, float> t(1, 3.14);
    std::pair<Foo, Foo> p1(t, t);
    std::pair<Foo, Foo> p2(std::piecewise_construct, t, t);
}
```

Output:

```
Constructed a Foo from a tuple
Constructed a Foo from a tuple
Constructed a Foo from an int and a float
Constructed a Foo from an int and a float
```

## See also

| | |
|---|---|
| **piecewise_construct** (C++11) | an object of type piecewise_construct_t used to disambiguate functions for piecewise construction<br>(constant) |
| (constructor) | constructs new pair<br>(public member function of std::pair) |

# std::piecewise_construct

```
constexpr piecewise_construct_t piecewise_construct = std::piecewise_construct_t();
```
(since C++11)

The constant **std::piecewise_construct** is an instance of an empty struct tag type std::piecewise_construct_t.

## Example

Run this code

```
#include <iostream>
#include <utility>
#include <tuple>

struct Foo {
    Foo(std::tuple<int, float>)
    {
        std::cout << "Constructed a Foo from a tuple\n";
    }
    Foo(int, float)
    {
        std::cout << "Constructed a Foo from an int and a float\n";
    }
};

int main()
{
    std::tuple<int, float> t(1, 3.14);
    std::pair<Foo, Foo> p1(t, t);
    std::pair<Foo, Foo> p2(std::piecewise_construct, t, t);
}
```

Output:

```
Constructed a Foo from a tuple
Constructed a Foo from a tuple
Constructed a Foo from an int and a float
Constructed a Foo from an int and a float
```

## See also

| | |
|---|---|
| **piecewise_construct_t** (C++11) | tag type used to select correct function overload for piecewise construction<br>(class) |
| (constructor) | constructs new pair<br>(public member function of std::pair) |

# std::integer_sequence

Defined in header <utility>

```
template< class T, T... Ints >
class integer_sequence;
```
(since C++14)

The class template `std::integer_sequence` represents a compile-time sequence of integers. When used as an argument to a function template, the parameter pack `Ints` can be deduced and used in pack expansion.

### Template parameters

| | | |
|---|---|---|
| **T** | - | an integer type to use for the elements of the sequence |
| **...Ints** | - | a non-type parameter pack representing the sequence |

### Member types

| Member type | Definition |
|---|---|
| value_type | T |

### Member functions

| | |
|---|---|
| **size** [static] | returns the number of elements in `Ints`<br>(public static member function) |

#### std::integer_sequence::size

```
static constexpr std::size_t size();
```

Returns the number of elements in `Ints`. Equivalent to `sizeof...(Ints)`

### Parameters

(none)

### Return value

The number of elements in `Ints`.

### Exceptions

`noexcept` specification: `noexcept`

### Helper templates

A helper alias template **`std::index_sequence`** is defined for the common case where `T` is `std::size_t`.

```
template<std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

A helper alias template **std::make_integer_sequence** is defined to simplify creation of std::integer_sequence and std::index_sequence types with 0, 1, 2, ..., N-1 as Ints:

```
template<class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;
```

```
template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

The program is ill-formed if N is negative. If N is zero, the indicated type is integer_sequence<T>.

A helper alias template **std::index_sequence_for** is defined to convert any type parameter pack into an index sequence of the same length

```
template<class... T>
using index_sequence_for = std::make_index_sequence<sizeof...(T)>;
```

### Example

Run this code

```
#include <tuple>
#include <iostream>
#include <array>
#include <utility>

// Convert array into a tuple
template<typename Array, std::size_t... I>
decltype(auto) a2t_impl(const Array& a, std::index_sequence<I...>)
{
    return std::make_tuple(a[I]...);
}

template<typename T, std::size_t N, typename Indices = std::make_index_sequence<N>>
decltype(auto) a2t(const std::array<T, N>& a)
{
    return a2t_impl(a, Indices());
}

// pretty-print a tuple (from http://stackoverflow.com/a/6245777/273767)

template<class Ch, class Tr, class Tuple, std::size_t... Is>
void print_tuple_impl(std::basic_ostream<Ch,Tr>& os,
                      const Tuple & t,
                      std::index_sequence<Is...>)
{
    using swallow = int[]; // guaranties left to right order
    (void)swallow{0, (void(os << (Is == 0? "" : ", ") << std::get<Is>(t)), 0)...};
}

template<class Ch, class Tr, class... Args>
decltype(auto) operator<<(std::basic_ostream<Ch, Tr>& os,
                          const std::tuple<Args...>& t)
{
    os << "(";
    print_tuple_impl(os, t, std::index_sequence_for<Args...>{});
    return os << ")";
}

int main()
{
    std::array<int, 4> array = {1,2,3,4};

    // convert an array into a tuple
    auto tuple = a2t(array);
    static_assert(std::is_same<decltype(tuple),
                               std::tuple<int, int, int, int>>::value, "");
```

```
    // print it to cout
    std::cout << tuple << '\n';
}
```

Output:

```
(1, 2, 3, 4)
```

### Example

This example shows how a `std::tuple` can be converted into arguments for a function invocation, see `std::experimental::apply`.

Run this code

```
#include <iostream>
#include <tuple>
#include <utility>

template<typename Func, typename Tup, std::size_t... index>
decltype(auto) invoke_helper(Func&& func, Tup&& tup, std::index_sequence<index...>)
{
    return func(std::get<index>(std::forward<Tup>(tup))...);
}

template<typename Func, typename Tup>
decltype(auto) invoke(Func&& func, Tup&& tup)
{
    constexpr auto Size = std::tuple_size<typename std::decay<Tup>::type>::value;
    return invoke_helper(std::forward<Func>(func),
                         std::forward<Tup>(tup),
                         std::make_index_sequence<Size>{});
}

void foo(int a, const std::string& b, float c)
{
    std::cout << a << " , " << b << " , " << c << '\n';
}

int main()
{
    auto args = std::make_tuple(2, "Hello", 3.5);
    invoke(foo, args);
}
```

Output:

```
2 , Hello , 3.5
```

# std::exchange

Defined in header <utility>

```
template< class T, class U = T >
T exchange( T& obj, U&& new_value );
```
(since C++14)

Replaces the value of `obj` with `new_value` and returns the old value of `obj`.

## Parameters

**obj** - object whose value to replace

**new_value** - the value to assign to `obj`

### Type requirements

- `T` must meet the requirements of `MoveConstructible`. Also, it must be possible to move-assign objects of type `U` to objects of type `T`

## Return value

The old value of `obj`

## Exceptions

(none)

## Possible implementation

```
template<class T, class U = T>
T exchange(T& obj, U&& new_value)
{
    T old_value = std::move(obj);
    obj = std::forward<U>(new_value);
    return old_value;
}
```

## Example

Run this code

```
#include <iostream>
#include <utility>
#include <vector>
#include <iterator>

class stream
{
  public:

   using flags_type = int;

  public:

    flags_type flags() const
    { return flags_; }
```

```
    ///Replaces flags_ by newf, and returns the old value.
    flags_type flags(flags_type newf)
    { return std::exchange(flags_, newf); }

  private:

    flags_type flags_ = 0;
};

void f() { std::cout << "f()"; }

int main()
{
   stream s;

   std::cout << s.flags() << '\n';
   std::cout << s.flags(12) << '\n';
   std::cout << s.flags() << "\n\n";

   std::vector<int> v;

   //Since the second template parameter has a default value, it is possible
   //to use a braced-init-list as second argument. The expression below
   //is equivalent to std::exchange(v, std::vector<int>{1,2,3,4});

   std::exchange(v, {1,2,3,4});

   std::copy(begin(v),end(v), std::ostream_iterator<int>(std::cout,", "));

   std::cout << "\n\n";

   void (*fun)();

   //the default value of template parameter also makes possible to use a
   //normal function as second argument. The expression below is equivalent to
   //std::exchange(fun, std::static_cast<void(*)()>(f))
   std::exchange(fun,f);
   fun();
}
```

Output:

```
0
0
12

1, 2, 3, 4,

f()
```

## See also

| | | |
|---|---|---|
| **swap** | | swaps the values of two objects<br>(function template) |
| **atomic_exchange**<br>**atomic_exchange_explicit** | (C++11)<br>(C++11) | atomically replaces the value of the atomic object with non-atomic argument and returns the old value of the atomic<br>(function template) |

# std::forward

Defined in header <utility>

| | | |
|---|---|---|
| `template< class T >`<br>`T&& forward( typename std::remove_reference<T>::type& t );` | (1) | (since C++11) (until C++14) |
| `template< class T >`<br>`constexpr T&& forward( typename std::remove_reference<T>::type& t );` | (1) | (since C++14) |
| `template< class T >`<br>`T&& forward( typename std::remove_reference<T>::type&& t );` | (2) | (since C++11) (until C++14) |
| `template< class T >`<br>`constexpr T&& forward( typename std::remove_reference<T>::type&& t );` | (2) | (since C++14) |

When used according to the following recipe in a function template, forwards the argument to another function with the value category it had when passed to the calling function.

For example, if used in wrapper such as the following, the template behaves as described below:

```
template<class T>
void wrapper(T&& arg)
{
    foo(std::forward<T>(arg)); // Forward a single argument.
}
```

- If a call to `wrapper()` passes an rvalue `std::string`, then `T` is deduced to `std::string` (not `std::string&`, `const std::string&`, or `std::string&&`), and `std::forward` ensures that an rvalue reference is passed to `foo`.
- If a call to `wrapper()` passes a const lvalue `std::string`, then `T` is deduced to `const std::string&`, and `std::forward` ensures that a const lvalue reference is passed to `foo`.
- If a call to `wrapper()` passes a non-const lvalue `std::string`, then `T` is deduced to `std::string&`, and `std::forward` ensures that a non-const lvalue reference is passed to `foo`.

## Notes

Attempting to forward an rvalue as an lvalue, such as by instantiating the form (2) with lvalue reference type T, is a compile-time error.

See template argument deduction for the special rules behind `T&&` used as a function parameter.

## Parameters

**t** - the object to be forwarded

## Return value

`static_cast<T&&>(t)`

## Exceptions

`noexcept` specification: `noexcept`

## Example

This example demonstrates perfect forwarding of the parameter(s) to the argument of the constructor of class `T`. Also, perfect forwarding of parameter packs is demonstrated.

Run this code

```
#include <iostream>
#include <memory>
#include <utility>
#include <array>

struct A {
    A(int&& n) { std::cout << "rvalue overload, n=" << n << "\n"; }
    A(int& n)  { std::cout << "lvalue overload, n=" << n << "\n"; }
};

class B {
public:
    template<class T1, class T2, class T3>
    B(T1&& t1, T2&& t2, T3&& t3) :
        a1_{std::forward<T1>(t1)},
        a2_{std::forward<T2>(t2)},
        a3_{std::forward<T3>(t3)}
    {
    }

private:
    A a1_, a2_, a3_;
};

template<class T, class U>
std::unique_ptr<T> make_unique1(U&& u)
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)));
}

template<class T, class... U>
std::unique_ptr<T> make_unique(U&&... u)
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)...));
}

int main()
{
    auto p1 = make_unique1<A>(2); // rvalue
    int i = 1;
    auto p2 = make_unique1<A>(i); // lvalue

    std::cout << "B\n";
    auto t = make_unique<B>(2, i, 3);
}
```

Output:

```
rvalue overload, n=2
lvalue overload, n=1
B
rvalue overload, n=2
lvalue overload, n=1
rvalue overload, n=3
```

## Complexity

Constant

## See also

| | |
|---|---|
| **move** (C++11) | obtains an rvalue reference<br>(function template) |

| | |
|---|---|
| **move_if_noexcept** (C++11) | obtains an rvalue reference if the move constructor does not throw<br>(function template) |

# std::move

Defined in header <utility>

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t );          (since C++11)
                                                                  (until C++14)
template< class T >
constexpr typename std::remove_reference<T>::type&& move( T&& t );   (since C++14)
```

`std::move` obtains an rvalue reference to its argument and converts it to an xvalue.

Code that receives such an xvalue has the opportunity to optimize away unnecessary overhead by *moving* data out of the argument, leaving it in a valid but unspecified state.

### Parameters

**t** - the object to be moved

### Return value

`static_cast<typename std::remove_reference<T>::type&&>(t)`

### Exceptions

`noexcept` specification: `noexcept`

### Notes

The functions that accept rvalue reference parameters (including move constructors, move assignment operators, and regular member functions such as `std::vector::push_back`) are selected, by overload resolution, when called with rvalue arguments (either prvalues such as a temporary objects or xvalues such as the one produced by `std::move`). If the argument identifies a resource-owning object, these overloads have the option, but aren't required, to *move* any resources held by the argument. For example, a move constructor of a linked list might copy the pointer to the head of the list and store `nullptr` in the argument instead of allocating and copying individual nodes.

Names of rvalue reference variables are lvalues and have to be converted to xvalues to be bound to the function overloads that accept rvalue reference parameters, which is why move constructors and move assignment operators typically use `std::move`:

```
// Simple move constructor
A(A&& arg) : member(std::move(arg.member)) // the expression "arg.member" is lvalue
{}
// Simple move assignment operator
A& operator=(A&& other) {
     member = std::move(other.member);
     return *this;
}
```

One exception is when the type of the function parameter is rvalue reference to type template parameter ("forwarding reference" or "universal reference"), in which case `std::forward` is used instead.

Unless otherwise specified, all standard library functions that accept rvalue reference parameters (such as `std::vector::push_back`) are guaranteed to leave the moved-from argument in valid but unspecified state. That is, only the functions without preconditions, such as the assignment operator, may be used on the object after it was moved into a standard library container:

```
std::vector<std::string> v;
```

```
std::string str = "example";
v.push_back(std::move(str)); // str is now valid but unspecified
str[0]; // undefined behavior: operator[](size_t n) has a precondition size() > n
str.clear(); // OK, clear() has no preconditions
```

Also, the standard library functions called with xvalue arguments may assume the argument is the only reference to the object; if it was constructed from an lvalue with `std::move`, no aliasing checks are made. In particular, this means that standard library move assignment operators do not have to perform self-assignment checks:

```
std::vector<int> v = {2, 3, 3};
v = std::move(v); // undefined behavior
```

### Example

Run this code

```
#include <iostream>
#include <utility>
#include <vector>
#include <string>

int main()
{
    std::string str = "Hello";
    std::vector<std::string> v;

    // uses the push_back(const T&) overload, which means
    // we'll incur the cost of copying str
    v.push_back(str);
    std::cout << "After copy, str is \"" << str << "\"\n";

    // uses the rvalue reference push_back(T&&) overload,
    // which means no strings will be copied; instead, the contents
    // of str will be moved into the vector.  This is less
    // expensive, but also means str might now be empty.
    v.push_back(std::move(str));
    std::cout << "After move, str is \"" << str << "\"\n";

    std::cout << "The contents of the vector are \"" << v[0]
                                    << "\", \"" << v[1] << "\"\n";

    // string move assignment operator is often implemented as swap,
    // in this case, the moved-from object is NOT empty
    std::string str2 = "Good-bye";
    std::cout << "Before move from str2, str2 = '" << str2 << "'\n";
    v[0] = std::move(str2);
    std::cout << "After move from str2, str2 = '" << str2 << "'\n";
}
```

Possible output:

```
After copy, str is "Hello"
After move, str is ""
The contents of the vector are "Hello", "Hello"
Before move from str2, str2 = 'Good-bye'
After move from str2, str2 = 'Hello'
```

### See also

| | |
|---|---|
| **forward** (C++11) | forwards a function argument (function template) |
| | obtains an rvalue reference if the move |

| | |
|---|---|
| **move_if_noexcept** (C++11) | constructor does not throw<br>(function template) |
| **move** (C++11) | moves a range of elements to a new location<br>(function template) |
| **std::experimental::parallel::move** (parallelism TS) | parallelized version of `std::move`<br>(function template) |

# std::move_if_noexcept

Defined in header <utility>

```
template< class T >
typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T& x);
```
(since C++11) (until C++14)

```
template< class T >
constexpr typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T& x);
```
(since C++14)

`move_if_noexcept` obtains an rvalue reference to its argument if its move constructor does not throw exceptions, otherwise obtains an lvalue reference to its argument. It is typically used to combine move semantics with strong exception guarantee.

### Parameters

**x** - the object to be moved or copied

### Return value

`std::move(x)` or `x`, depending on exception guarantees.

### Exceptions

`noexcept` specification: `noexcept`

### Notes

This is used, for example, by `std::vector::resize`, which may have to allocate new storage and then move or copy elements from old storage to new storage. If an exception occurs during this operation, `std::vector::resize` undoes everything it did to this point, which is only possible if `std::move_if_noexcept` was used to decide whether to use move construction or copy construction. (unless copy constructor is not available, in which case move constructor is used either way and the strong exception guarantee may be waived)

### Example

Run this code

```
#include <iostream>
#include <utility>

struct Bad
{
    Bad() {}
    Bad(Bad&&)  // may throw
    {
        std::cout << "Throwing move constructor called\n";
    }
    Bad(const Bad&) // may throw as well
    {
        std::cout << "Throwing copy constructor called\n";
    }
```

```
};

struct Good
{
    Good() {}
    Good(Good&&) noexcept // will NOT throw
    {
        std::cout << "Non-throwing move constructor called\n";
    }
    Good(const Good&) noexcept // will NOT throw
    {
        std::cout << "Non-throwing copy constructor called\n";
    }
};

int main()
{
    Good g;
    Bad b;
    Good g2 = std::move_if_noexcept(g);
    Bad b2 = std::move_if_noexcept(b);
}
```

Output:

```
Non-throwing move constructor called
Throwing copy constructor called
```

### Complexity

Constant

### See also

| | |
|---|---|
| **forward** (C++11) | forwards a function argument<br>(function template) |
| **move** (C++11) | obtains an rvalue reference<br>(function template) |

# std::declval

Defined in header <utility>

```
template< class T >
typename std::add_rvalue_reference<T>::type declval();
```
(since C++11)

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.

`declval` is commonly used in templates where acceptable template parameters may have no constructor in common, but have the same member function whose return type is needed. Note that because no definition exists for `declval`, it can only be used in unevaluated contexts; it is an error to evaluate an expression that contains this function.

## Parameters

(none)

## Return value

Cannot be called and thus never returns a value. The return type is `T&&` unless `T` is (possibly cv-qualified) `void`, in which case the return type is `T`.

## Exceptions

`noexcept` specification: `noexcept`

## Example

Run this code

```
#include <utility>
#include <iostream>

struct Default {
    int foo() const {return 1;}
};

struct NonDefault {
    NonDefault(const NonDefault&) {}
    int foo() const {return 1;}
};

int main()
{
    decltype(Default().foo()) n1 = 1; // int n1
//  decltype(NonDefault().foo()) n2 = n1; // will not compile
    decltype(std::declval<NonDefault>().foo()) n2 = n1; // int n2
    std::cout << "n2 = " << n2 << '\n';
}
```

Output:

```
n2 = 1
```

## See also

| | |
|---|---|
| decltype specifier | defines a type equivalent to the type of an expression (C++11) |
| **result_of** (C++11) | deduces the return type of a function call expression (class template) |

# std::bind

Defined in header <functional>

```
template< class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );                    (1)  (since C++11)

template< class R, class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );                    (2)  (since C++11)
```

The function template `bind` generates a forwarding call wrapper for `f`. Calling this wrapper is equivalent to invoking `f` with some of its arguments bound to `args`.

## Parameters

**f** - `Callable` object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments

**args** - list of arguments to bind, with the unbound arguments replaced by the placeholders `_1, _2, _3...` of namespace `std::placeholders`

## Return value

A function object of unspecified type `T`, for which `std::is_bind_expression<T>::value == true`, and which can be stored in `std::function`. It has the following members:

### std::bind *return type*

#### Member objects

The return type of `std::bind` holds a member object of type `std::decay<F>::type` constructed from `std::forward<F>(f)`, and one object per each of `args...`, of type `std::decay<Arg_i>::type`, similarly constructed from `std::forward<Arg_i>(arg_i)`.

#### Constructors

The return type of `std::bind` is `CopyConstructible` if all of its member objects (specified above) are CopyConstructible, and is `MoveConstructible` otherwise. The type defines the following members:

#### Member type `result_type`

1) If `F` is a pointer to function or a pointer to member function, `result_type` is the return type of `F`. If `F` is a class type with nested typedef `result_type`, then `result_type` is `F::result_type`. Otherwise no `result_type` is defined.

2) `result_type` is exactly `R`.

#### Member function `operator()`

Given an object `g` obtained from an earlier call to `bind`, when it is invoked in a function call expression `g(u1, u2, ... uM)`, an invocation of the stored object of type `std::decay<F>::type` takes place, with arguments defined as follows:

- If the argument is of type `std::reference_wrapper<T>` (for example, `std::ref` or `std::cref` was used in the initial call to `bind`), then the reference `T&` stored in the bound argument is passed to the invocable object.
- If `std::is_bind_expression<T>::value == true` (i.e. another `bind` subexpression was used as an argument in the initial call to `bind`), then that bind subexpression is invoked immediately and its result is passed to the invocable object. If the bind subexpression has any placeholder arguments, they are picked from `u1, u2, ...`.
- If `std::is_placeholder<T>::value != 0` (i.e., `std::placeholders::_1, _2, _3, ...` was used as the argument to the initial call to `bind`), then the argument indicated by the placeholder (`u1` for `_1`, `u2` for `_2`, etc) is passed to the invocable object as

`std::forward<Uj>(uj)`.

- Otherwise, the stored argument is passed to the invocable object as-is.

If some of the arguments that are supplied in the call to `g()` are not matched by any placeholders stored in `g`, the unused arguments are evaluated and discarded.

### Exceptions

Only throws if construction of `std::decay<F>::type` from `std::forward<F>(f)` throws, or any of the constructors for `std::decay<Arg_i>::type` from the corresponding `std::forward<Arg_i>(arg_i)` throws where `Arg_i` is the ith type and `arg_i` is the ith argument in `Args... args`.

### Notes

As described in `Callable`, when invoking a pointer to non-static member function or pointer to non-static data member, the first argument has to be a reference or pointer (including, possibly, smart pointer such as `std::shared_ptr` and `std::unique_ptr`) to an object whose member will be accessed.

The arguments to bind are copied or moved, and are never passed by reference unless wrapped in `std::ref` or `std::cref`.

Duplicate placeholders in the same bind expression (multiple `_1`'s for example) are allowed, but the results are only well defined if the corresponding argument (`u1`) is an lvalue or non-movable rvalue.

### Example

Run this code

```
#include <random>
#include <iostream>
#include <memory>
#include <functional>

void f(int n1, int n2, int n3, const int& n4, int n5)
{
    std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}

int g(int n1)
{
    return n1;
}

struct Foo {
    void print_sum(int n1, int n2)
    {
        std::cout << n1+n2 << '\n';
    }
    int data = 10;
};

int main()
{
    using namespace std::placeholders;  // for _1, _2, _3...

    // demonstrates argument reordering and pass-by-reference
    int n = 7;
    // (_1 and _2 are from std::placeholders, and represent future
    // arguments that will be passed to f1)
    auto f1 = std::bind(f, _2, _1, 42, std::cref(n), n);
    n = 10;
    f1(1, 2, 1001); // 1 is bound by _2, 2 is bound by _1, 1001 is unused
```

```
    // nested bind subexpressions share the placeholders
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12);

    // common use case: binding a RNG with a distribution
    std::default_random_engine e;
    std::uniform_int_distribution<> d(0, 10);
    std::function<int()> rnd = std::bind(d, e); // a copy of e is stored in rnd
    for(int n=0; n<10; ++n)
        std::cout << rnd() << ' ';
    std::cout << '\n';

    // bind to a member function
    Foo foo;
    auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
    f3(5);

    // bind to member data
    auto f4 = std::bind(&Foo::data, _1);
    std::cout << f4(foo) << '\n';

    // smart pointers can be used to call members of the referenced objects, too
    std::cout << f4(std::make_shared<Foo>(foo)) << '\n'
              << f4(std::unique_ptr<Foo>(new Foo(foo))) << '\n';}
```

Output:

```
2 1 42 10 7
12 12 12 4 5
1 5 0 2 0 8 2 2 10 8
100
10
10
10
```

## See also

| | |
|---|---|
| **_1, _2, _3, _4, ...** (C++11) | placeholders for the unbound arguments in a `std::bind` expression (constant) |
| **mem_fn** (C++11) | creates a function object out of a pointer to a member (function template) |

# va_start

Defined in header <cstdarg>

```
void va_start( va_list ap, parm_n );
```

The `va_start` macro enables access to the variable arguments following the named argument `parm_n`.

`va_start` should be invoked with an instance to a valid `va_list` object `ap` before any calls to `va_arg`.

If `parm_n` is declared with reference type, with `register` storage class specifier, or with a type not compatible with the type that results from default argument promotions, the behavior is undefined.

### Parameters

**ap** - an instance of the `va_list` type

**parm_n** - the named parameter preceding the first variable parameter

### Expanded value

(none)

### Example

Run this code

```
#include <iostream>
#include <cstdarg>

int add_nums(int count, ...)
{
    int result = 0;
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        result += va_arg(args, int);
    }
    va_end(args);
    return result;
}

int main()
{
    std::cout << add_nums(4, 25, 25, 50, 50) << '\n';
}
```

Output:

```
150
```

### See also

| | |
|---|---|
| **va_arg** | accesses the next variadic function argument (function macro) |
| **va_end** | ends traversal of the variadic function arguments (function macro) |

**C documentation** for **va_start**

# va_arg

Defined in header <cstdarg>

```
T va_arg( va_list ap, T );
```

The va_arg macro expands to an expression of type T that corresponds to the next parameter from the va_list ap.

Prior to calling va_arg, ap must be initialized by a call to either va_start or va_copy, with no intervening call to va_end. Each invocation of the va_arg macro modifies ap to point to the next variable argument.

If va_arg is called when there are no more arguments in ap, or if the type of the next argument in ap (after promotions) is not compatible with T, the behavior is undefined, unless:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types; or
- one type is pointer to `void` and the other is a pointer to a character type (`char`, `signed char`, or `unsigned char`).

### Parameters

**ap** - an instance of the va_list type
**T** - the type of the next parameter in ap

### Expanded value

the next variable parameter in ap

### Example

Run this code

```
#include <iostream>
#include <cstdarg>
#include <cmath>

double stddev(int count, ...)
{
    double sum = 0;
    double sum_sq = 0;
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        double num = va_arg(args, double);
        sum += num;
        sum_sq += num*num;
    }
    va_end(args);
    return std::sqrt(sum_sq/count - (sum/count)*(sum/count));
}

int main()
{
    std::cout << stddev(4, 25.0, 27.3, 26.9, 25.7) << '\n';
}
```

Output:

```
0.920258
```

### See also

| | |
|---|---|
| **va_start** | enables access to variadic function arguments<br>(function macro) |
| **va_copy** (C++11) | makes a copy of the variadic function arguments<br>(function macro) |
| **va_end** | ends traversal of the variadic function arguments<br>(function macro) |

**C documentation** for **va_arg**

# va_end

Defined in header <cstdarg>

```
void va_end( va_list ap );
```

The va_end macro performs cleanup for an ap object initialized by a call to va_start or va_copy. va_end may modify ap so that it is no longer usable.

If there is no corresponding call to va_start or va_copy, or if va_end is not called before a function that calls va_start or va_copy returns, the behavior is undefined.

### Parameters

**ap** - an instance of the va_list type to clean up

### Expanded value

(none)

### See also

| | |
|---|---|
| **va_start** | enables access to variadic function arguments (function macro) |
| **va_copy** (C++11) | makes a copy of the variadic function arguments (function macro) |
| **va_arg** | accesses the next variadic function argument (function macro) |

**C documentation** for **va_end**

# std::abort

Defined in header <cstdlib>

| | |
|---|---|
| `void abort();` | (until C++11) |
| `[[noreturn]] void abort();` | (since C++11) |

Causes abnormal program termination unless `SIGABRT` is being caught by a signal handler passed to signal and the handler does not return.

Destructors of variables with automatic, thread local and static storage durations are not called. Functions passed to `std::atexit()` are also not called. Whether open resources such as files are closed is implementation defined. Implementation defined status is returned to the host environment that indicates unsuccessful execution.

### Parameters

(none)

### Return value

(none)

### Exceptions

| | | |
|---|---|---|
| (none) | | (until C++11) |
| `noexcept` specification: | `noexcept` | (since C++11) |

### Example

Run this code

```
#include <csignal>
#include <iostream>
#include <cstdlib>

class Tester {
public:
    Tester()  { std::cout << "Tester ctor\n"; }
    ~Tester() { std::cout << "Tester dtor\n"; }
};

Tester static_tester; // Destructor not called

void signal_handler(int signal)
{
    if (signal == SIGABRT) {
        std::cerr << "SIGABRT received\n";
    } else {
        std::cerr << "Unexpected signal " << signal << " received\n";
    }
    std::_Exit(EXIT_FAILURE);
}

int main()
{
    Tester automatic_tester; // Destructor not called

    // Setup handler
    auto previous_handler = std::signal(SIGABRT, signal_handler);
    if (previous_handler == SIG_ERR) {
```

```
        std::cerr << "Setup failed\n";
        return EXIT_FAILURE;
    }

    std::abort();  // Raise SIGABRT
    std::cout << "This code is unreachable\n";
}
```

Output:

```
Tester ctor
Tester ctor
SIGABRT received
```

## See also

| | |
|---|---|
| **exit** | causes normal program termination with cleaning up (function) |
| **atexit** | registers a function to be called on `std::exit()` invocation (function) |
| **quick_exit** (C++11) | causes quick program termination without completely cleaning up (function) |
| **signal** | sets a signal handler for particular signal (function) |
| **C documentation** for **abort** | |

# std::atexit

Defined in header <cstdlib>

```
extern "C"   int atexit( void (*func)() );
extern "C++" int atexit( void (*func)() );
```

Registers the function pointed to by `func` to be called on normal program termination (via `std::exit()` or returning from the cpp/language/main function)

| | |
|---|---|
| The functions will be called during the destruction of the static objects, in reverse order: if A was registered before B, then the call to B is made before the call to A. Same applies to the ordering between static object constructors and the calls to `atexit`: see `std::exit` | (until C++11) |
| The functions may be called concurrently with the destruction of the objects with static storage duration and with each other, maintaining the guarantee that if registration of A was sequenced-before the registration of B, then the call to B is sequenced-before the call to A, same applies to the sequencing between static object constructors and calls to `atexit`: see `std::exit` | (since C++11) |

The same function may be registered more than once.

If a function exits via an exception, `std::terminate` is called.

`atexit` is thread-safe: calling the function from several threads does not induce a data race.

The implementation is guaranteed to support the registration of at least `32` functions. The exact limit is implementation-defined.

## Parameters

**`func`** - pointer to a function to be called on normal program termination

## Return value

`0` if the registration succeeds, nonzero value otherwise.

## Notes

The two overloads are distinct because the types of the parameter `func` are distinct (language linkage is part of its type)

## Exceptions

| | | |
|---|---|---|
| (none) | | (until C++11) |
| `noexcept` specification: | `noexcept` | (since C++11) |

## Example

Run this code

```
#include <iostream>
#include <cstdlib>

void atexit_handler_1()
{
    std::cout << "at exit #1\n";
}

void atexit_handler_2()
```

```
{
    std::cout << "at exit #2\n";
}

int main()
{
    const int result_1 = std::atexit(atexit_handler_1);
    const int result_2 = std::atexit(atexit_handler_2);

    if ((result_1 != 0) or (result_2 != 0)) {
        std::cerr << "Registration failed\n";
        return EXIT_FAILURE;
    }

    std::cout << "returning from main\n";
    return EXIT_SUCCESS;
}
```

Output:

```
returning from main
at exit #2
at exit #1
```

## See also

| | |
|---|---|
| **at_quick_exit** (C++11) | registers a function to be called on `quick_exit` invocation (function) |
| **C documentation** for **atexit** | |

# std::system

Defined in header <cstdlib>

```
int system( const char *command );
```

Calls the host environment's command processor with command parameter. Returns implementation-defined value (usually the value that the invoked program returns).

If command is the NULL pointer, checks if host environment has a command processor and returns nonzero value only if the command processor exists.

## Parameters

**command** - character string identifying the command to be run in the command processor. If NULL pointer is given, command processor is checked for existence

## Return value

Implementation-defined value. If command is NULL returns nonzero value only if command processor exists.

## Notes

Related POSIX function popen (http%3A//pubs.opengroup.org/onlinepubs/9699919799/functions/popen.html) makes the output generated by command available to the caller.

## Example

In this example there is a system call of the unix command **ls -l >test.txt**:

Run this code

```
#include <cstdlib>

int main()
{
    std::system("ls -l >test.txt");
}
```

## See also

**C documentation** for **system**

# std::raise

Defined in header <csignal>

```
int raise( int sig );
```

Sends signal sig to the program. The signal handler (specified using the `std::signal()` function) is invoked.

If the user-defined signal handling strategy is not set using `std::signal()` yet, it is implementation-defined whether the signal will be ignored or default handler will be invoked.

## Parameters

**sig** - the signal to be sent. It can be an implementation-defined value or one of the following values:

| | |
|---|---|
| **SIGABRT** | |
| **SIGFPE** | |
| **SIGILL** | defines signal types |
| **SIGINT** | (macro constant) |
| **SIGSEGV** | |
| **SIGTERM** | |

## Return value

0 upon success, non-zero value on failure.

## Example

Run this code

```
#include <csignal>
#include <iostream>

void signal_handler(int signal)
{
    std::cout << "Received signal " << signal << '\n';
}

int main()
{
    // Install a signal handler
    std::signal(SIGTERM, signal_handler);

    std::cout << "Sending signal " << SIGTERM << '\n';
    std::raise(SIGTERM);
}
```

Possible output:

```
Sending signal 15
Received signal 15
```

## See also

| | |
|---|---|
| **signal** | sets a signal handler for particular signal (function) |

**C documentation** for **`raise`**

# std::ref, std::cref

Defined in header <functional>

| | | |
|---|---|---|
| `template< class T > std::reference_wrapper<T> ref(T& t);` | (1) | (since C++11) |
| `template< class T > std::reference_wrapper<T> ref( std::reference_wrapper<T> t );` | (2) | (since C++11) |
| `template <class T> void ref(const T&&) = delete;` | (3) | (since C++11) |
| `template< class T > std::reference_wrapper<const T> cref( const T& t );` | (4) | (since C++11) |
| `template< class T > std::reference_wrapper<const T> cref( std::reference_wrapper<T> t )` | (5) | (since C++11) |
| `template <class T> void cref(const T&&) = delete;` | (6) | (since C++11) |

Function templates `ref` and `cref` are helper functions that generate an object of type `std::reference_wrapper`, using template argument deduction to determine the template argument of the result.

### Parameters

**t** - lvalue reference to object that needs to be wrapped or an instance of `std::reference_wrapper`

### Return value

1) `std::reference_wrapper<T>(t)`
2) `ref(t.get())`
4) `std::reference_wrapper<const T>(t)`
5) `cref(t.get())`

### Exceptions

`noexcept` specification: `noexcept`

### Example

Run this code

```
#include <functional>
#include <iostream>

void f(int& n1, int& n2, const int& n3)
{
    std::cout << "In function: " << n1 << ' ' << n2 << ' ' << n3 << '\n';
    ++n1; // increments the copy of n1 stored in the function object
    ++n2; // increments the main()'s n2
    // ++n3; // compile error
}

int main()
{
    int n1 = 1, n2 = 2, n3 = 3;
    std::function<void()> bound_f = std::bind(f, n1, std::ref(n2), std::cref(n3));
    n1 = 10;
    n2 = 11;
```

```
    n3 = 12;
    std::cout << "Before function: " << n1 << ' ' << n2 << ' ' << n3 << '\n';
    bound_f();
    std::cout << "After function: " << n1 << ' ' << n2 << ' ' << n3 << '\n';
}
```

Output:

```
Before function: 10 11 12
In function: 1 11 12
After function: 10 12 12
```

## See also

| | |
|---|---|
| **reference_wrapper** (C++11) | `CopyConstructible` and `CopyAssignable` reference wrapper (class template) |

# std::placeholders::_1, std::placeholders::_2, ..., std::placeholders::_N

Defined in header <functional>

```
/*see below*/ _1;
/*see below*/ _2;
.
.
/*see below*/ _N;
```

The **std::placeholders** namespace contains the placeholder objects [_1, . . . _N] where N is an implementation defined maximum number.

When used as an argument in a std::bind expression, the placeholder objects are stored in the generated function object, and when that function object is invoked with unbound arguments, each placeholder _N is replaced by the corresponding Nth unbound argument.

| | |
|---|---|
| Each placeholder is declared as if by `extern /*unspecified*/ _1;` | (until C++17) |
| Implementations are encouraged to declare the placeholders as if by `constexpr /*unspecified*/ _1;`, although declaring them by `extern /*unspecified*/ _1;` is still allowed by the standard. | (since C++17) |

The types of the placeholder objects are DefaultConstructible and CopyConstructible, their default copy/move constructors do not throw exceptions, and for any placeholder _N, the type `std::is_placeholder<decltype(_N)>` is defined and is derived from `std::integral_constant<int, N>`.

## Example

The following code shows the creation of function objects with a placeholder argument.

Run this code

```
#include <functional>
#include <string>
#include <iostream>

void goodbye(const std::string& s)
{
    std::cout << "Goodbye " << s << '\n';
}

class Object {
public:
    void hello(const std::string& s)
    {
        std::cout << "Hello " << s << '\n';
    }
};

int main(int argc, char* argv[])
{
    typedef std::function<void(const std::string&)> ExampleFunction;
    Object instance;
    std::string str("World");
    ExampleFunction f = std::bind(&Object::hello, &instance,
                                  std::placeholders::_1);

    // equivalent to instance.hello(str)
    f(str);
    f = std::bind(&goodbye, std::placeholders::_1);

    // equivalent to goodbye(str)
    f(str);
```

```
    return 0;
}
```

Output:

```
Hello World
Goodbye World
```

### See also

| | |
|---|---|
| **bind** (C++11) | binds one or more arguments to a function object<br>(function template) |
| **is_placeholder** (C++11) | indicates that an object is a standard placeholder or can be used as one<br>(class template) |

# std::function

Defined in header <functional>

```
template< class >
class function; /* undefined */
```
(since C++11)

```
template< class R, class... Args >
class function<R(Args...)>
```
(since C++11)

Class template `std::function` is a general-purpose polymorphic function wrapper. Instances of `std::function` can store, copy, and invoke any `Callable` *target* -- functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members.

The stored callable object is called the *target* of `std::function`. If a `std::function` contains no target, it is called *empty*. Invoking the *target* of an *empty* `std::function` results in `std::bad_function_call` exception being thrown.

`std::function` satisfies the requirements of `CopyConstructible` and `CopyAssignable`.

## Member types

| Type | Definition |
|---|---|
| `result_type` | `R` |
| `argument_type` | `T` if `sizeof...(Args)==1` and `T` is the first and only type in `Args...` |
| `first_argument_type` | `T1` if `sizeof...(Args)==2` and `T1` is the first of the two types in `Args...` |
| `second_argument_type` | `T2` if `sizeof...(Args)==2` and `T2` is the second of the two types in `Args...` |

## Member functions

| | |
|---|---|
| (constructor) | constructs a new `std::function` instance<br>(public member function) |
| (destructor) | destroys a `std::function` instance<br>(public member function) |
| **`operator=`** | assigns a new target<br>(public member function) |
| **`swap`** | swaps the contents<br>(public member function) |
| **`assign`** | assigns a new target<br>(public member function) |
| **`operator bool`** | checks if a valid target is contained<br>(public member function) |
| **`operator()`** | invokes the target<br>(public member function) |

### Target access

| | |
|---|---|
| **`target_type`** | obtains the `typeid` of the stored target<br>(public member function) |
| **`target`** | obtains a pointer to the stored target<br>(public member function) |

## Non-member functions

| | |
|---|---|
| **`std::swap`**`(std::function)` (C++11) | specializes the `std::swap` algorithm<br>(function template) |
| **`operator==`**<br>**`operator!=`** | compares an **`std::function`** with `nullptr`<br>(function template) |

### Helper classes

| | |
|---|---|
| **std::uses_allocator**<std::function> (C++11) | specializes the std::uses_allocator type trait (class template specialization) |

### Example

Run this code

```
#include <functional>
#include <iostream>

struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
    int num_;
};

void print_num(int i)
{
    std::cout << i << '\n';
}

struct PrintNum {
    void operator()(int i) const
    {
        std::cout << i << '\n';
    }
};

int main()
{
    // store a free function
    std::function<void(int)> f_display = print_num;
    f_display(-9);

    // store a lambda
    std::function<void()> f_display_42 = []() { print_num(42); };
    f_display_42();

    // store the result of a call to std::bind
    std::function<void()> f_display_31337 = std::bind(print_num, 31337);
    f_display_31337();

    // store a call to a member function
    std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
    const Foo foo(314159);
    f_add_display(foo, 1);

    // store a call to a member function and object
    using std::placeholders::_1;
    std::function<void(int)> f_add_display2= std::bind( &Foo::print_add, foo, _1 );
    f_add_display2(2);

    // store a call to a member function and object ptr
    std::function<void(int)> f_add_display3= std::bind( &Foo::print_add, &foo, _1 );
    f_add_display3(3);

    // store a call to a function object
    std::function<void(int)> f_display_obj = PrintNum();
    f_display_obj(18);
}
```

Output:

```
-9
42
```

```
31337
314160
314161
314162
18
```

## See also

| | |
|---|---|
| **bad_function_call** (C++11) | the exception thrown when invoking an empty **std::function** (class) |
| **mem_fn** (C++11) | creates a function object out of a pointer to a member (function template) |

# std::plus

Defined in header <functional>

```
template< class T >
struct plus;                          (until C++14)

template< class T = void >
struct plus;                          (since C++14)
```

Function object for performing addition. Effectively calls `operator+` on two instances of type `T`.

## Specializations

The standard library provides a specialization of `std::plus` when `T` is not specified, which leaves the parameter types and return type to be deduced. (since C++14)

| | |
|---|---|
| **plus**<void> | function object implementing `x + y` deducing argument and return types (class template specialization) |

## Member types

| Type | Definition |
|---|---|
| result_type | T |
| first_argument_type | T |
| second_argument_type | T |

## Member functions

| | |
|---|---|
| **operator()** | returns the sum of two arguments (public member function) |

### std::plus::operator()

```
T operator()( const T& lhs, const T& rhs ) const;          (until C++14)
constexpr T operator()( const T& lhs, const T& rhs ) const;    (since C++14)
```

Returns the sum of `lhs` and `rhs`.

#### Parameters

**lhs, rhs** - values to sum

#### Return value

The result of `lhs + rhs`.

#### Exceptions

(none)

#### Possible implementation

```
constexpr T operator()(const T &lhs, const T &rhs) const
{
    return lhs + rhs;
}
```

# std::equal_to

Defined in header <functional>

```
template< class T >
struct equal_to;                          (until C++14)

template< class T = void >
struct equal_to;                          (since C++14)
```

Function object for performing comparisons. Unless specialised, invokes `operator==` on type `T`.

## Specializations

The standard library provides a specialization of `std::equal_to` when `T` is not specified, which leaves the parameter types and return type to be deduced.

| | | |
|---|---|---|
| **`equal_to`**<void> | function object implementing `x == y` deducing argument and return types<br>(class template specialization) | (since C++14) |

## Member types

| type | definition |
|---|---|
| `result_type` | `bool` |
| `first_argument_type` | `T` |
| `second_argument_type` | `T` |

## Member functions

| | |
|---|---|
| **`operator()`** | checks if the arguments are *equal*<br>(public member function) |

### std::equal_to::operator()

```
bool operator()( const T& lhs, const T& rhs ) const;          (until C++14)
constexpr bool operator()( const T& lhs, const T& rhs ) const;    (since C++14)
```

Checks whether `lhs` is *equal* to `rhs`.

#### Parameters

**`lhs, rhs`** - values to compare

#### Return value

`true` if `lhs == rhs`, `false` otherwise.

#### Exceptions

(none)

#### Possible implementation

```
constexpr bool operator()(const T &lhs, const T &rhs) const
{
    return lhs == rhs;
}
```

# std::mem_fn

Defined in header <functional>

```
template< class R, class T >
/*unspecified*/ mem_fn(R T::* pm);                                          (1)  (since C++11)

template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...));
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) const);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) volatile);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) const volatile);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) &);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) const &);                        (2)  (since C++11)
template< class R, class T, class... Args >                                     (until C++14)
/*unspecified*/ mem_fn(R (T::* pm)(Args...) volatile &);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) const volatile &);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) &&);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) const &&);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) volatile &&);
template< class R, class T, class... Args >
/*unspecified*/ mem_fn(R (T::* pm)(Args...) const volatile &&);
```

Function template `std::mem_fn` generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a `std::mem_fn`.

The overloads (2) were introduced in C++11 but removed in C++14 as defect #2048 (http%3A//cplusplus.github.io/LWG/lwg-defects.html#2048)

### Parameters

**pm** - pointer to member that will be wrapped

### Return value

`std::mem_fn` returns a call wrapper of unspecified type that has the following members:

## std::mem_fn Return type

### Member types

| type | definition |
|---|---|
| `result_type` | the return type of `pm` if `pm` is a pointer to member function, not defined for pointer to member object |
| `argument_type` | `T*`, possibly cv-qualified, if `pm` is a pointer to member function taking no arguments |
| `first_argument_type` | `T*` if `pm` is a pointer to member function taking one argument |
| `second_argument_type` | `T1` if `pm` is a pointer to member function taking one argument of type `T1` |

### Member function

| | |
|---|---|
| **operator()** | invokes the target on a specified object, with optional parameters |

(public member function)

### Exceptions

None.

### Example 1

Use `mem_fn` to store and execute a member function and a member object:

Run this code

```
#include <functional>
#include <iostream>

struct Foo {
    void display_greeting() {
        std::cout << "Hello, world.\n";
    }
    void display_number(int i) {
        std::cout << "number: " << i << '\n';
    }
    int data = 7;
};

int main() {
    Foo f;

    auto greet = std::mem_fn(&Foo::display_greeting);
    greet(f);

    auto print_num = std::mem_fn(&Foo::display_number);
    print_num(f, 42);

    auto access_data = std::mem_fn(&Foo::data);
    std::cout << "data: " << access_data(f) << '\n';
}
```

Output:

```
Hello, world.
number: 42
data: 7
```

### Example 2

Demonstrates the effect of the C++14 changes to the specification of std::mem_fn

Run this code

```
#include <iostream>
#include <functional>

struct X {
    int x;

    int&       easy()       {return x;}
    int&       get()        {return x;}
    const int& get() const {return x;}
};


int main(void)
```

```
{
    auto a = std::mem_fn        (&X::easy); // no problem at all
//  auto b = std::mem_fn<int&  >(&X::get ); // no longer works in C++14
    auto c = std::mem_fn<int&()>(&X::get ); // works with both C++11 and C++14
    auto d = [] (X& x) {return x.get();};   // another approach to overload resolution

    X x = {33};
    std::cout << "a() = " << a(x) << '\n';
    std::cout << "c() = " << c(x) << '\n';
    std::cout << "d() = " << d(x) << '\n';
}
```

Output:

```
a() = 33
c() = 33
d() = 33
```

## See also

| | |
|---|---|
| **function** (C++11) | wraps callable object of any type with specified function call signature<br>(class template) |
| **bind** (C++11) | binds one or more arguments to a function object<br>(function template) |