```
#define SIG_DFL see below
#define SIG_ERR see below
#define SIG_IGN see below
#define SIGABRT see below
#define SIGFPE see below
#define SIGILL see below
#define SIGINT see below
#define SIGSEGV see below
#define SIGTERM see below
```

1 The contents of the header `<csignal>` are the same as the C standard library header `<signal.h>`.

17.13.5 Signal handlers [support.signal]

1 A call to the function `signal` synchronizes with any resulting invocation of the signal handler so installed.

2 A *plain lock-free atomic operation* is an invocation of a function `f` from 33.5, such that:

(2.1) — `f` is the function `atomic_is_lock_free()`, or

(2.2) — `f` is the member function `is_lock_free()`, or

(2.3) — `f` is a non-static member function of class `atomic_flag`, or

(2.4) — `f` is a non-member function, and the first parameter of `f` has type *cv* `atomic_flag*`, or

(2.5) — `f` is a non-static member function invoked on an object `A`, such that `A.is_lock_free()` yields `true`, or

(2.6) — `f` is a non-member function, and for every pointer-to-atomic argument `A` passed to `f`, `atomic_is_lock_free(A)` yields `true`.

3 An evaluation is *signal-safe* unless it includes one of the following:

(3.1) — a call to any standard library function, except for plain lock-free atomic operations and functions explicitly identified as signal-safe;

[*Note 1*: This implicitly excludes the use of `new` and `delete` expressions that rely on a library-provided memory allocator. — *end note*]

(3.2) — an access to an object with thread storage duration;

(3.3) — a `dynamic_cast` expression;

(3.4) — throwing of an exception;

(3.5) — control entering a *try-block* or *function-try-block*;

(3.6) — initialization of a variable with static storage duration requiring dynamic initialization (6.9.3.3, 8.8)[202] ; or

(3.7) — waiting for the completion of the initialization of a variable with static storage duration (8.8).

A signal handler invocation has undefined behavior if it includes an evaluation that is not signal-safe.

4 The function `signal` is signal-safe if it is invoked with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

See also: ISO C 7.14

17.14 C headers [support.c.headers]

17.14.1 General [support.c.headers.general]

1 For compatibility with the C standard library, the C++ standard library provides the *C headers* shown in Table 40. The intended use of these headers is for interoperability only. It is possible that C++ source files need to include one of these headers in order to be valid ISO C. Source files that are not intended to also be valid ISO C should not use any of the C headers.

[*Note 1*: The C headers either have no effect, such as `<stdbool.h>` (17.14.5) and `<stdalign.h>` (17.14.4), or otherwise the corresponding header of the form `<cname>` provides the same facilities and assuredly defines them in namespace `std`. — *end note*]

202) Such initialization can occur because it is the first odr-use (6.3) of that variable.

[*Example 1*: The following source file is both valid C++ and valid ISO C. Viewed as C++, it declares a function with C language linkage; viewed as C it simply declares a function (and provides a prototype).

```
#include <stdbool.h>     // for bool in C, no effect in C++
#include <stddef.h>      // for size_t

#ifdef __cplusplus       // see 15.11
extern "C"               // see 9.11
#endif
void f(bool b[], size_t n);
```

— *end example*]

Table 40: C headers [tab:c.headers]

<assert.h>	<inttypes.h>	<signal.h>	<stdint.h>	<uchar.h>
<complex.h>	<iso646.h>	<stdalign.h>	<stdio.h>	<wchar.h>
<ctype.h>	<limits.h>	<stdarg.h>	<stdlib.h>	<wctype.h>
<errno.h>	<locale.h>	<stdatomic.h>	<string.h>	
<fenv.h>	<math.h>	<stdbool.h>	<tgmath.h>	
<float.h>	<setjmp.h>	<stddef.h>	<time.h>	

17.14.2 Header <complex.h> synopsis [complex.h.syn]

```
#include <complex>
```

1 The header <complex.h> behaves as if it simply includes the header <complex> (28.4.2).

2 [*Note 1*: Names introduced by <complex> in namespace std are not placed into the global namespace scope by <complex.h>. — *end note*]

17.14.3 Header <iso646.h> synopsis [iso646.h.syn]

1 The C++ header <iso646.h> is empty.

[*Note 1*: and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in C++ (5.11). — *end note*]

17.14.4 Header <stdalign.h> synopsis [stdalign.h.syn]

1 The contents of the C++ header <stdalign.h> are the same as the C standard library header <stdalign.h>, with the following changes: The header <stdalign.h> does not define a macro named alignas.

See also: ISO C 7.15

17.14.5 Header <stdbool.h> synopsis [stdbool.h.syn]

1 The contents of the C++ header <stdbool.h> are the same as the C standard library header <stdbool.h>, with the following changes: The header <stdbool.h> does not define macros named bool, true, or false.

See also: ISO C 7.18

17.14.6 Header <tgmath.h> synopsis [tgmath.h.syn]

```
#include <cmath>
#include <complex>
```

1 The header <tgmath.h> behaves as if it simply includes the headers <cmath> (28.7.1) and <complex> (28.4.2).

2 [*Note 1*: The overloads provided in C by type-generic macros are already provided in <complex> and <cmath> by "sufficient" additional overloads. — *end note*]

3 [*Note 2*: Names introduced by <cmath> or <complex> in namespace std are not placed into the global namespace scope by <tgmath.h>. — *end note*]

17.14.7 Other C headers [support.c.headers.other]

1 Every C header other than <complex.h> (17.14.2), <iso646.h> (17.14.3), <stdalign.h> (17.14.4), <stdatomic.h> (33.5.12), <stdbool.h> (17.14.5), and <tgmath.h> (17.14.6), each of which has a name of the form <*name*.h>, behaves as if each name placed in the standard library namespace by the corresponding <c*name*> header is placed within the global namespace scope, except for the functions described in 28.7.6,

```
operator<<(basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);
```

1 *Constraints*: `treat_as_floating_point_v<typename Duration::rep>` is `false`, and `Duration{1} < days{1}` is `true`.

2 *Effects*: Equivalent to:

```
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{:L%F %T}"), tp);
```

3 [*Example 1*:

```
cout << sys_seconds{0s} << '\n';                // 1970-01-01 00:00:00
cout << sys_seconds{946'684'800s} << '\n';      // 2000-01-01 00:00:00
cout << sys_seconds{946'688'523s} << '\n';      // 2000-01-01 01:02:03
```

— *end example*]

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_days& dp);
```

4 *Effects*: `os << year_month_day{dp}`.

5 *Returns*: `os`.

```
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                sys_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);
```

6 *Effects*: Attempts to parse the input stream `is` into the `sys_time tp` using the format flags given in the NTCTS `fmt` as specified in 29.13. If the parse fails to decode a valid date, `is.setstate(ios_base::failbit)` is called and `tp` is not modified. If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null. Additionally, the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that difference to `tp`.

7 *Returns*: `is`.

29.7.3 Class `utc_clock` [time.clock.utc]

29.7.3.1 Overview [time.clock.utc.overview]

```
namespace std::chrono {
  class utc_clock {
  public:
    using rep                        = a signed arithmetic type;
    using period                     = ratio<unspecified, unspecified>;
    using duration                   = chrono::duration<rep, period>;
    using time_point                 = chrono::time_point<utc_clock>;
    static constexpr bool is_steady = unspecified;

    static time_point now();

    template<class Duration>
      static sys_time<common_type_t<Duration, seconds>>
        to_sys(const utc_time<Duration>& t);
    template<class Duration>
      static utc_time<common_type_t<Duration, seconds>>
        from_sys(const sys_time<Duration>& t);
  };
}
```

1 In contrast to `sys_time`, which does not take leap seconds into account, `utc_clock` and its associated `time_point`, `utc_time`, count time, including leap seconds, since 1970-01-01 00:00:00 UTC.

[*Note 1*: The UTC time standard began on 1972-01-01 00:00:10 TAI. To measure time since this epoch instead, one can add/subtract the constant `sys_days{1972y/1/1} - sys_days{1970y/1/1}` (`63'072'000s`) from the `utc_time`. — *end note*]

```
function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

```
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.
Difficulty of converting: Simple.
How widely used: Programs which depend upon `sizeof('x')` are probably rare.

3 **Affected subclause:** 5.13.5
Change: Concatenated *string-literal*s can no longer have conflicting *encoding-prefix*es.
Rationale: Removal of non-portable feature.
Effect on original feature: Concatenation of *string-literal*s with different *encoding-prefix*es is now ill-formed.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

4 **Affected subclause:** 5.13.5
Change: String literals made const.
The type of a *string-literal* is changed from "array of `char`" to "array of `const char`". The type of a UTF-8 string literal is changed from "array of `char`" to "array of `const char8_t`". The type of a UTF-16 string literal is changed from "array of *some-integer-type*" to "array of `const char16_t`". The type of a UTF-32 string literal is changed from "array of *some-integer-type*" to "array of `const char32_t`". The type of a wide string literal is changed from "array of `wchar_t`" to "array of `const wchar_t`".
Rationale: This avoids calling an inappropriate overloaded function, which might expect to be able to modify its argument.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Syntactic transformation. The fix is to add a cast:

```
char* p = "abc";                // valid in C, invalid in C++
void f(char*) {
  char* p = (char*)"abc";       // OK, cast added
  f(p);
  f((char*)"def");              // OK, cast added
}
```

How widely used: Programs that have a legitimate reason to treat string literal objects as potentially modifiable memory are probably rare.

C.6.3 Clause 6: basics [diff.basic]

1 **Affected subclause:** 6.2
Change: C++ does not have "tentative definitions" as in C.
E.g., at file scope,

```
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local objects with static storage duration, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X* next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for fundamental types and user-defined types.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. In C++, the initializer for one of a set of mutually-referential file-local objects with static storage duration must invoke a function call to achieve the initialization.
How widely used: Seldom.

2 **Affected subclause:** 6.4
Change: A `struct` is a scope in C++, not in C. For example,