Concepts of C++ Programming

winter term 2023/24

PD Dr. Tobias Lasser

Computational Imaging and Inverse Problems (CIIP)
Technical University of Munich

Tweedback today

The Tweedback session ID today is zjxa, the URL is:

`https://tweedback.de/zjxa`

Examination

- Written exam:
  - February 22, 2024
  - in presence
  - 90 minutes duration
  - no repeat exam!

- Homework:
  - weekly programming assignments
  - passing the automated tests yields 2 points per week (total of 24 points achievable, as currently planned)

- Final grade:
  - points from written exam and from homeworks added together
  - final grade will be derived from the sum of points
  - homework assignments are **NOT** bonus points!

Contents

Initialization revisited

There are two kinds of initialization:

- safe:

```
int a{42};     // OK, a == 42
int b{7.5};    // ERROR: no narrowing allowed
```

- unsafe:

```
int a = 42;    // OK, a == 42
int b = 7.5;   // OK, but b == 7
int c(42);     // OK, c == 42
int d(7.5);    // OK, but d == 7
```

Prefer initialization using `{}`!

Initialization subtleties

Initialization using `{}` is of type `std::initializer_list<T>`, where T is the type of the value inside `{}`

- this can lead to surprises:

```
std::vector<int> v1{99}; // v1 is a vector of 1 element
                         // with value 99
std::vector<int> v2(99); // v2 is a vector of 99 elements,
                         // each with default value 0
```

- or errors:

```
std::vector<std::string> v1{"hello"};
   // v1 is a vector with 1 element with value "hello"
std::vector<std::string> v2("hello");
   // ERROR: no constructor of vector accepts a string literal
```

This does not happen often, but it pays off to be aware.

Initializer lists

- As the name implies, `std::initializer_list<T>` can accept more than one value:

```cpp
struct S { int x; std::string s; };
S s{1, "hello"};                     // struct initializer

std::complex<double> z{0, 1};        // uses constructor
std::vector<double> v{0.0, 1.1, 2.2}; // uses list constructor
```

- Again, subtleties apply:

```cpp
std::complex<double> z1(0, 1); // uses constructor
std::complex<double> z2{};     // uses constructor,
                               // default value {0,0}
std::complex<double> z3();     // function declaration!

std::vector<double> v1{10, 3.3}; // list constructor
std::vector<double> v2(10, 3.3); // constructor, v has 10 elements set to 3.3
```

Default values

Initializing with empty `{}` yields the default value

- for number types, the default is a suitable representation of 0
- for user-defined types, the default value (if there is any) is determined by the constructor (see later)

```
int x{};      // x == 0
double d{};   // d == 0.0

std::vector<int> v{}; // v is the empty vector
std::string s{};      // s == ""
```

Missing initializers

Unfortunately the initializer is optional, leading to potential undefined behavior

- this is due to performance considerations
- objects with static storage duration are default initialized
- all other objects are not default initialized

```
int a;      // global, i.e. equivalent to int a{}; so a == 0

void f() {
    int x;     // local, not initialized!
    int y = x; // value of y is undefined!
}
```

- This can end up in very hard to find bugs!
  - use compiler warnings (-Wall) and heed them

Auto

- since the type of the initializer is known to the compiler, it can save you some work
- you can replace the type in a declaration by the keyword auto

```
auto a{123};    // a is of type int
auto b{'c'};    // b is of type char
```

- this is most useful for complicated types, such as

```
std::unique_ptr<int> u1{std::make_unique<int>()};
auto u2{std::make_unique<int>()}; // same as u1

std::vector<double> v{};
std::vector<double>::iterator i1 = v.begin();
auto i2 = v.begin(); // same as i1
```

Auto: how to know what happens

- use `type_index` from library boost:

```cpp
#include <print>
#include <memory>
#include <boost/type_index.hpp>
using namespace boost::typeindex;

int main() {
        auto a = {13, 14};
        auto u{std::make_unique<int>()};

        std::println("Type a: {}", type_id_with_cvr<decltype(a)>().pretty_name());
        std::println("Type u: {}", type_id_with_cvr<decltype(u)>().pretty_name());
}
```

- easiest to do in online tools such as `https://wandbox.org`:

```
Type a: std::initializer_list<int>
Type u: std::unique_ptr<int, std::default_delete<int> >
```

Contents

Arrays

For arrays with a fixed number of elements, use `std::array<T, num>`

- `num` elements of type `T` that lie contiguously in memory
- can be default initialized using `{}`, or list initialized
- size is accessible via member function `size()`
- access to elements using
  - `operator[]()` (not bounds-checked)
  - `at()` (bounds-checked)
- has iterators for iterating over all elements

Array: example

```cpp
#include <array>
#include <print>

int main() {
    std::array<int, 10> a{}; // array of 10 int, default initialized
    std::array<float, 3> b{0.0, 1.1, 2.2}; // array of 3 float

    for (unsigned i = 0; i < a.size(); ++i)
        a[i] = i + 1; // no bounds checking

    for (auto i : b) // loop over all elements of b
        std::println("{}", i);

    a.at(11) = 5; // run-time error: out_of_range exception
}
```

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at: __n (which is 11) >= _Nm (which is 10)
```

C-style arrays: DO NOT USE

There are old C-style arrays, for example

```cpp
int a[10] = {};
float b[3] = {0.0, 1.1, 2.2};
```

DO NOT USE THEM! THEY ARE EVIL!

- they only exist for C-style compatibility
- they don't know their own size, they have no bounds checking
- they are the source for many crashes, buffer overflows etc.

`std::array` can do everything C-style arrays can

- in a safe way with no overhead
- while providing additional functionality (such as bounds checking, iterators)

Non-fixed size arrays: vectors

If you need arrays that are not fixed size (like `std::array`), use `std::vector<T>`

- dynamically sized array, storage is automatically expanded and contracted as needed
- still guaranteed to be contiguously in memory
- same interface as `std::array`
- and additional functions, such as
  - `push_back` to insert elements at end
  - `clear` to clear the contents
  - `resize` to resize the vector

Vector: example

```cpp
#include <print>
#include <vector>

int main() {
    std::vector<int> a;  // default initialized to be empty
    for (unsigned i = 0; i < 10; ++i)
        a.push_back(i + 1);

    std::println("Size: {}", a.size());
    a.clear();
    std::println("Size: {}", a.size());

    a.resize(5); // a now contains 5 zeros
    std::println("Size: {}", a.size());

    for (unsigned i = 0; i < a.size(); ++i)
        a[i] = i + 1; // no bounds checking
}
```

Output:

```
Size: 10
Size: 0
Size: 5
```

Range for loops

Ranges (e.g. containers that support iterators, like `std::array` or `std::vector`) support special range for loops:

```
for (init-statement; range-declaration : range-expression)
    loop-statement
```

- `init-statement` is executed once (may be omitted)
- then `loop-statement` is executed once for each element in the range defined by `range-expression`
- `range-expression` should represent a sequence (e.g. an array, or object which defines iterators (i.e. functions `begin`, `end`) such as `std::vector`)
- `range-declaration` should declare a named variable of the element type of the sequence, or a reference to that type

Range for loop: example

```cpp
#include <print>
#include <format>
#include <vector>

int main() {
    std::vector<int> a{1, 2, 3, 4, 5}; // initializer list

    for (unsigned i = 0; i < 5; ++i)   // regular for loop
        a.push_back(i + 10);

    for (const auto& e : a)     // range for loop
        std::print("{}, ", e);
    std::println("");

    for (auto i : {47, 11, 3}) // range for loop
        std::print("{}, ", i);
}
```

Output:

```
1, 2, 3, 4, 5, 10, 11, 12, 13, 14,
47, 11, 3,
```

Contents

Expressions

An expression is a sequence of operators and operands

- evaluation of the expression can produce a result
  - for example, evaluation of 2+3 produces the result 5
- evaluation of the expression may produce side effects
  - for example, evaluation of `std::print("4")` prints 4 on the standard output

lvalues and rvalues

Each expression is characterized by two independent properties

- its type (e.g. `int`, `float`)
- its value category

There are several value categories, extremely simplified there is:

- lvalues that refer to the identity of an object
  - modifiable lvalues can be used on left-hand side of assignment
- rvalues that refer to the value of an object
  - lvalues and rvalues can be used on right-hand side of assignment

More on this later.

Operators

Operators act on a number of operands

- unary operators
  - such as: negation (`-`), address-of (`&`), dereference (`*`)
- binary operators
  - such as: equality (`==`), multiplication (`*`)
- ternary operator: `a ? b : c`

Most operators can be overloaded for user-defined types (see later).

Operands

The operands of any operator can be

- other expressions
- or primary expressions

Primary expressions are

- literals
- variable names
- and others (lambdas, fold expressions, see later)

Arithmetic operators

Operator	Explanation
`+a`	unary plus
`-a`	unary minus
`a + b`	addition
`a - b`	subtraction
`a * b`	multiplication
`a / b`	division
`a % b`	modulo
`~a`	bit-wise NOT
`a & b`	bit-wise AND
`a | b`	bit-wise OR
`a ^ b`	bit-wise XOR
`a << b`	bit-wise left shift
`a >> b`	bit-wise right shift

Undefined behavior can occur, e.g. on

- signed overflow
- division by zero
- shift by negative offset
- shift by offset larger than the width of the type

Logical and relational operators

Operator	Explanation
`!a`	logical NOT
`a && b`	logical AND (short-circuiting)
`a || b`	logical OR (short-circuiting)
`a == b`	equal to
`a != b`	not equal to
`a < b`	less than
`a > b`	greater than
`a <= b`	less than or equal to
`a >= b`	greater than or equal to

Assignment operators

Operator	Explanation
`a = b`	simple assignment
`a += b`	addition assignment
`a -= b`	subtraction assignment
`a *= b`	multiplication assignment
`a /= b`	division assignment
`a %= b`	modulo assignment
`a &= b`	bit-wise AND assignment
`a |= b`	bit-wise OR assignment
`a ^= b`	bit-wise XOR assignment
`a <<= b`	bit-wise left shift assignment
`a >>= b`	bit-wise right shift assignment

- left-hand side of assignment operator must be modifiable lvalue
- for built-in types, `a OP= b` is equivalent to `a = a OP b`, except that `a` is only evaluated once

Assignment operators (cont.)

Assignment operators return a reference to the left-hand side:

```
int a, b, c;
a = b = c = 42;    // a, b, and c have value 42
```

Very rarely used, except in these situations:

```
if (int d = computeValue()) {
    // executed if d is not zero
}
else {
    // executed if d is zero
}
```

Increment and decrement operators

Operator	Explanation
`++a`	prefix increment
`--a`	prefix decrement
`a++`	postfix increment
`a--`	postfix decrement

Logic differs between prefix and postfix variants:

- prefix variants increment/decrement the value of an object and return a reference to the result
- postfix variants create a copy of an object, increment/decrement the value of the original object, and return the unchanged copy

Ternary conditional operator

Operator	Explanation
`a ? b : c`	conditional operator

Semantics:

- a is evaluated and converted to `bool`
- if result was `true`, b is evaluated
- if result was `false`, c is evaluated

```
int n = (1 > 2) ? 21 : 42;      // 1 > 2 is false, i.e. n == 42
int m = 42;
((n == m) ? m : n) = 21;        // n == m is true, i.e. m == 21

int k{(n == m) ? 5.0 : 21};     // ERROR: narrowing conversion
((n == m) ? 5 : n) = 21;        // ERROR: assigning to rvalue
```

Precedence and associativity

- operators with higher precedence bind tighter than operators with lower precedence
- operators with equal precedence are bound in the direction of their associativity
  - left-to-right
  - right-to-left
- grouping is often not immediately obvious
  - use parentheses judiciously!

Precedence and associativity do not specify evaluation order

- evaluation order is mostly unspecified
- in general, it is undefined behavior to refer to and change the same object within one expression

Precedence and associativity (cont.)

Precedence can be obvious:

```
int a = 1 + 2 * 3;  // 1 + (2 * 3), i.e. a == 7
```

But it can get confusing very fast:

```
int b = 50 - 6 - 2;     // (50 - 6) - 2, i.e. b == 42
int c = b & 1 << 4 - 1; // b & (1 << (4 - 1)), i.e. c == 8
```

Bugs like to hide in expressions without parentheses:

```
if (0 <= x <= 99) // not what you might expect!
    std::print("I am always true!");

// shift should be 4 if sizeof(long) == 4, 6 otherwise
unsigned shift = 2 + sizeof(long) == 4 ? 2 : 4; // buggy!!
```

Operator precedence table

Prec.	Operator	Description	Associativity
1	::	scope resolution	left-to-right
	a++ a--	postfix increment/decrement	
	<type>() <type>{}	functional cast	
2	a()	function call	left-to-right
	a[]	subscript	
	. ->	member access	
	++a --a	prefix increment/decrement	
	+a -a	unary plus/minus	
	! ~	logical/bit-wise NOT	
	(<type>)	C-style cast	
3	*a	dereference	right-to-left
	&a	address-of	
	sizeof	size-of	
	new new[]	dynamic memory allocation	
	delete delete[]	dynamic memory deallocation	

Operator precedence table (cont.)

Prec.	Operator	Description	Associativity
4	`.*` `->*`	pointer to member	left-to-right
5	`a*b` `a/b` `a % b`	multiplication / division / modulo	left-to-right
6	`a+b` `a-b`	addition / subtraction	left-to-right
7	`<<` `>>`	bit-wise shift	left-to-right
8	`<=>`	three-way comparison (C++20)	left-to-right
9	`<` `<=` `>` `>=`	relational $<$ and $\leq$ relational $>$ and $\geq$	left-to-right
10	`==` `!=`	relational $=$ and $\neq$	left-to-right

Operator precedence table (cont.)

<table>
<tr><th>Prec.</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>11</td><td>&</td><td>bit-wise AND</td><td>left-to-right</td></tr>
<tr><td>12</td><td>^</td><td>bit-wise XOR</td><td>left-to-right</td></tr>
<tr><td>13</td><td>|</td><td>bit-wise OR</td><td>left-to-right</td></tr>
<tr><td>14</td><td>&&</td><td>logical AND</td><td>left-to-right</td></tr>
<tr><td>15</td><td>||</td><td>logical OR</td><td>left-to-right</td></tr>
<tr><td rowspan="7">16</td><td>a ? b : c</td><td>ternary conditional</td><td rowspan="7">right-to-left</td></tr>
<tr><td>throw</td><td>throw operator</td></tr>
<tr><td>=</td><td>direct assignment</td></tr>
<tr><td>+= -=</td><td>compound assignment</td></tr>
<tr><td>*= /= %=</td><td>compound assignment</td></tr>
<tr><td><<= >>=</td><td>compound assignment</td></tr>
<tr><td>&= ^= |=</td><td>compound assignment</td></tr>
<tr><td>17</td><td>,</td><td>comma</td><td>left-to-right</td></tr>
</table>

Contents

Simple statements

- declaration statement: declaration followed by semicolon

```
int i{0};
```

- expression statement: any expression followed by a semicolon

```
i + 3;     // valid, but rather useless expression statement
foo();     // valid and possibly useful expression statement
```

- compound statement: brace-enclosed sequence of statements

```
{                   // start of block and scope
    int j{1};   // declaration statement
    int k{2};   // declaration statement
}                   // end of block and scope
```

If statement

Conditionally execute another statement:

```
if (init_statement; condition)
    then_statement
else
    else_statement
```

- if `condition` evaluates to `true` after conversion to `bool`, `then_statement` is executed, otherwise `else_statement` is executed
- both `init_statement` and `else` branch can be omitted
- if present, `init_statement` must be an expression or declaration statement
- `condition` must be an expression statement or a single declaration
- `then_statement` and `else_statement` can be arbitrary (compound) statements

If statement (cont.)

`init_statement` is useful for local variables used only inside `if` :

```
if (auto value{computeValue()}; value < 42) {
    // do something
}
else {
    // do something else
}
```

This is equivalent to:

```
{
    auto value{computeValue()};
    if (value < 42) {
        // do something
    }
    else {
        // do something else
    }
}
```

If statements (cont.)

For nested `if` statements, the `else` is associated with the closest `if` that does not have an `else`:

```
// INTENTIONALLY MISLEADING!
if (condition0)
    if (condition1)
        // do something if (condition0 && condition1) == true
else
    // do something if condition0 == false (...not!)
```

If in doubt, use curly braces to make scopes explicit!

```
// working as intended
if (condition0) {
    if (condition1)
        // do something if (condition0 && condition1) == true
}
else {
    // do something if condition0 == false
}
```

Switch statement

Conditionally transfer control to one of several statements:

```
switch (init_statement; condition)
    statement
```

- `condition` is an expression or single declaration that is convertible to an enumeration or integral type
- body of `switch` statement may contain arbitrary number of `case constant:` labels and up to one `default:` label
- the `constant` values for all `case:` labels must be unique
- if `condition` evaluates to a value for which a `case:` label is present, control is passed to the labelled statement
- otherwise, control is passed to the statement labelled with `default:`
- the `break;` statement can be used to exit the `switch`

Switch statement (cont.)

Regular example:

```
switch (computeValue()) {
    case 21:
        // do something if computeValue() was 21
        break;
    case 42:
        // do something if computeValue() was 42
        break;
    default:
        // do something if computeValue() was != 21 and != 42
        break;
}
```

Switch statement (cont.)

Less regular example:

```
switch (computeValue()) {
    case 21:
    case 42:
        // do something if computeValue() was 21 or 42
        break;
    default:
        // do something if computeValue() was != 21 and != 42
        break;
}
```

- the body is executes sequentially until a `break;` statement is encountered, i.e. it can "fall through"
- compilers may generate warnings when encountering fall-through behavior
  - use special `[[fallthrough]];` statement to mark intentional fall through

While loop

Repeatedly execute a statement:

```
while (condition)
    statement
```

- executes `statement` repeatedly until the value of `condition` becomes `false`
  - `condition` is evaluated before each iteration
- `condition` is an expression that can be converted to `bool` or a single declaration
- `statement` is an arbitrary statement
- the `break;` statement can be used to exit the loop
- the `continue;` statement can be used to skip the remainder of the body

Do-while loop

Repeatedly execute a statement:

```
do
    statement
while (condition);
```

- executes `statement` repeatedly until the value of `condition` becomes `false`
    - `condition` is evaluated after each iteration
- `condition` is an expression that can be converted to `bool` or a single declaration
- `statement` is an arbitrary statement
- the `break;` statement can be used to exit the loop
- the `continue;` statement can be used to skip the remainder of the body

While vs. do-while

The body of a do-while loop is executed at least once:

```cpp
int i{42};

do {
    // executed once
} while (i < 42);

while (i < 42) {
    // never executed
}
```

For loop

Repeatedly executes a statement:

```
for (init_statement; condition; iteration_expression)
    statement
```

- executes `init_statement` once, then, if `condition` is `true`, executes `statement` and `iteration_expression` until `condition` becomes `false`
- `init_statement` is an expression or declaration
- `condition` is an expression that can be converted to `bool` or a single declaration
- `iteration_expression` is an arbitrary expression
- all three statements in the parentheses can be omitted
- the `break;` statement can be used to exit the loop
- the `continue;` statement can be used to skip the remainder of the body

For loop (cont.)

Example:

```
for (int i{0}; i < 10; ++i) {
    // do something
}

for (unsigned i{0}, limit{10}; i != limit; ++i) {
    // do something
}
```

Careful of integral overflows (signed overflows are undefined behavior)

```
for (uint8_t i{0}; i < 256; ++i) {
    // infinite loop
}

for (unsigned i = 42; i >= 0; --i) {
    // infinite loop
}
```

Contents

Functions in C++

- functions associate a sequence of statements (function body) with a name
- functions can have zero or more function parameters

```
return_type name ( parameter_list ) {
    statements (body)
}
```

- functions are invoked via the function-call expression
- parameters are initialized from the provided arguments

```
name ( argument_list );
```

Function return types

- no return type is marked using `void`:

```
void foo(int parameter0, float parameter1) {
    // do something with parameter0 and parameter1
}
```

- functions with a return type (non-void) must contain a `return` statement:

```
int meaningOfLife() {
    // extremely complex computation
    return 42;
}
```

- the main-function of a program returns an `int`, as an exception the `return` statement may be omitted (resulting in an implicit `return 0;`)

```
int main() {
    // run the program
}
```

Argument passing

- arguments are passed by value:

```
int square(int v) {
    v = v * v;
    return v;
}

int main() {
    int v{8};
    int w{square(v)};  // w == 64, v == 8
}
```

- arguments can also explicitly be passed by reference, see next section
- this distinction is essential for user-defined types! (see later)

Unnamed arguments

- function parameters can be unnamed, which means they cannot be used:

```
int meaningOfLife(int /* unused */) {
    return 42;
}
```

- the argument still has to be supplied on function invocation:

```
int v{meaningOfLife()};    // ERROR: expected argument
int w{meaningOfLife(123)}; // OK
```

Default arguments

- the last parameters of a function can have default values
  - after a parameter with a default value, all following parameters must have default values as well
- parameters with default values may be omitted when invoking the function

```cpp
int foo(int a, int b = 2, int c = 3) {
    return a + b + c;
}

int main() {
    int x{foo(1)};          // x == 6
    int y{foo(1, 1)};       // y == 5
    int z{foo(1, 1, 1)};    // z == 3
}
```

Contents

Reference declaration

a reference declaration declares an alias to an existing object

- Lvalue reference: `&declarator`
- Rvalue reference: `&&declarator`
- `declarator` can be any other declarator, except another reference declarator
  - most of the time, `declarator` is just a name

References are special:

- there are no references to `void`
- references are immutable (although the referenced object can be mutable)
- references are not objects, i.e. they do not necessarily occupy storage
  - hence there are no references (or pointers) to references
  - and there are no arrays of references

Reference declaration (cont.)

- the `&` or `&&` qualifiers are part of the declarator, not the type:

```
int i{10};
int &j{i}, k{i}; // j is reference to int, k is int
```

- we can insert or omit whitespaces before and after the `&` or `&&` qualifiers

```
int &m{i}; // valid
int& n{i}; // also valid
```

- by convention we use `int& n{i};`
  - to avoid confusion, statements should only declare one identifier at a time
  - very rarely, exceptions to this rule are necessary (e.g. in the init statements of `if`, `switch`, `for`)

Reference initialization

- a reference to type `T` must be initialized to refer to a valid object
    - an object of type `T`
    - a function of type `T`
    - an object implicitly convertible to `T`
- there are exceptions: (as always...)
    - function parameter declarations
    - function return type declarations
    - class member declarations (see later)
    - when using `extern` modifier

Lvalue references: examples

Alias for existing objects:

```
int i{10};
int j{42};
int& r{i};    // r is an alias for i

r = 21;       // modifies i to be 21
r = j;        // modifies i to be 42

i = 123;
j = r;        // modifies j to be 123
```

Lvalue references: examples (cont.)

Pass by reference for function calls:

```
void foo(int& value) {
    value += 42;
}

int main() {
    int i{10};
    foo(i);      // i == 52
    foo(i);      // i == 94
}
```

Lvalue references: examples (cont.)

Turning a function call into an lvalue expression:

```
int global0{0};
int global1{0};

int& foo(unsigned which) {
    if (!which)
        return global0;
    else
        return global1;
}

int main() {
    foo(0) = 42;   // global0 == 42
    foo(1) = 14;   // global1 == 14
}
```

Rvalue references: examples

Rvalue references cannot (directly) bind to lvalues:

```
int i{10};
int&& j{i};  // ERROR: cannot bind rvalue ref to lvalue
int&& k{42}; // OK
```

Rvalue references can extend the lifetime of temporary objects:

```
int i{10};
int j{32};

int&& k{i + j};  // k == 42
k += 42;         // k == 84
```

Rvalue references: examples (cont.)

Overload resolution (see later) allows to distinguish between lvalues and rvalues:

```
void foo(int& x);
void foo(const int& x);
void foo(int&& x);

int& bar();
int baz();

int main() {
    int i{42};
    const int j{84};

    foo(i);      // calls foo(int&)
    foo(j);      // calls foo(const int&)
    foo(123);    // calls foo(int&&)

    foo(bar());  // calls foo(int&)
    foo(baz());  // calls foo(int&&)
}
```

Const references

- references themselves cannot be `const`
- however, the reference type can be `const`
- a reference to `T` can be initialized from a type that is not `const`
  - e.g. `const int&` can be initialized from `int`

```
int i{10};
const int& j{i};
int& k{j};  // ERROR: binding reference of type int& to
            //        const int discards qualifiers
j = 42;     // ERROR: assignment of read-only reference
```

- lvalue references to `const` also extend lifetime of temporaries:

```
int i{10};
int j{32};
const int& k{i + j}; // OK, but k is immutable
```

Dangling references

- Caution: you can write programs where the lifetime of the referenced object ends while references to it still exist!
  - happens mostly when referencing objects with automatic storage duration
  - results in dangling reference and undefined behavior
- example:

```
int& foo() {
    int i{42};
    return i;    // MISTAKE: returns dangling reference!
}
```

- good compilers warn you about it
  - so make sure to use -Wall or /Wall
  - and heed all compiler warnings!

Contents

Converting between types

Sometimes, the automatic implicit conversion is not enough

- explicit type conversion can be forced to cast between related types

```
static_cast< new_type > ( expression )
```

  - converts the value of `expression` to a value of `new_type`
  - `new_type` must have same `const`-ness as the type of `expression`
  - many use cases (but never use it lightly!)

- several more casting operators exist (`const_cast`, `reinterpret_cast`, C-style cast: `(new_type) expression`)
  - do not use them! they are evil!
- `dynamic_cast` is sometimes used for polymorphic class hierarchies, see later

static_cast example

```
int sum(int a, int b);
double sum(double a, double b);

int main() {
    int a{42};
    double b{3.14};

    double x{sum(a, b)};                       // ERROR: ambiguous
    double y{sum(static_cast<double>(a), b)}; // OK
    int z{sum(a, static_cast<int>(b))};        // OK
}
```

Summary