Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 7 – Compile-Time Computation

Thomas Corbat / Felix Morgner
Rapperswil, 18.04.2023
FS2023

OST
Ostschweizer
Fachhochschule

Agenda

- **Recap Week 6**
- **Compile-time Evaluation with constexpr**
- **Literal Types**
- **User-Defined Literal Operators**

Goals

- **Participants should…**
  - … be able to write C++ code that is evaluated by the compiler
  - … know the restrictions of constexpr functions
  - … be able to write their own literal types
  - … be able to write their own user defined literal operators

Recap Week 6

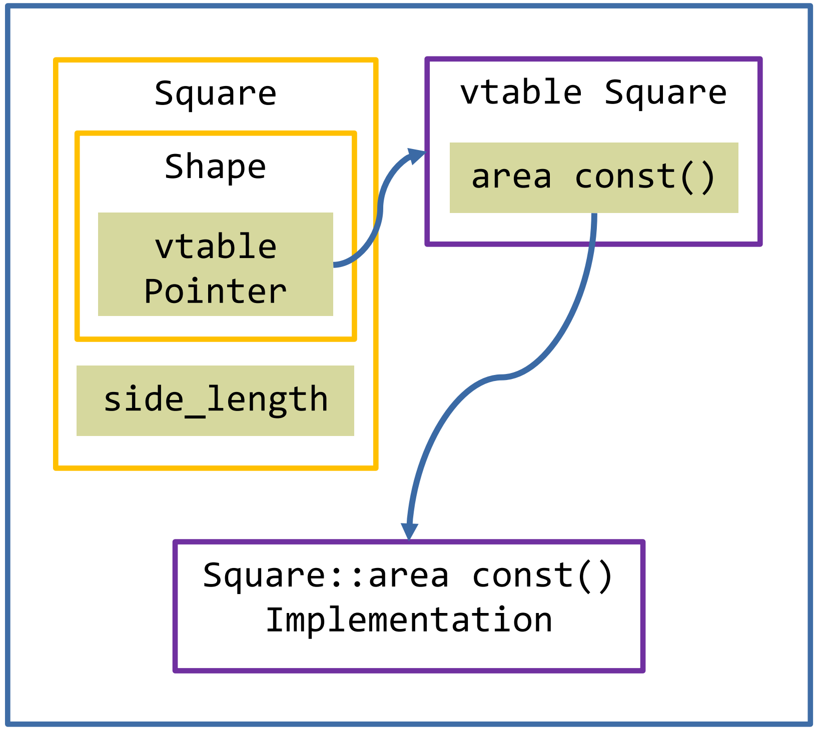
Ostschweizer
Fachhochschule

- **A polymorphic call of a virtual function requires lookup of the target function**

```
struct Shape {
  virtual unsigned area() const = 0;
  virtual ~Shape();
};

struct Square : Shape {
  Square(unsigned side_length)
    : side_length{side_length} {}
  unsigned area() const {
    return side_length * side_length;
  }
  unsigned side_length;
};
```

```
decltype(auto) amountOfSeeds(Shape const & shape) {
  auto area = shape.area();
  return area * seedsPerSquareMeter;
};
```



Overload Selection with SFINAE

- **Since there is a problem during substitution that overload is discarded**

Name increment is looked up

`increment(unsinged)`	`increment` Template

↓

Template arguments are deduced

`increment(unsigned)`	~~`increment<int>(int)`~~

↓

Best overload is selected

`increment(unsigned)`

- **Now the result is 42**

```
unsigned increment(unsigned i) {
  return i++;
}

template<typename T>
auto increment(T value) ->
    decltype(value.increment()) {
  return value.increment();
}

int main() {
  return increment(42);
}
```

- **This approach, using decltype(...) as trailing return type, is infeasible in general**
  - Function might have return type `void`
  - It is not elegant for complex bodies

```
template <typename T>
concept Incrementable = requires (T const v) {
  {v.increment()} -> std::same_as<T>;
};
```

```
auto increment(Incrementable auto value) -> T {
  return value.increment();
}
```

```
template <Incrementable T>
auto increment(T value) -> T {
  return value.increment();
}
```

Constexpr

Ostschweizer Fachhochschule

- **(static) const variables in namespace scope of built-in types initialized with constant expression are usually put into ROMable memory, if at all.**
- **Allowed in constant expression context**
- **No complicated computations (except with macros)**
- **No guarantee to be done at compile-time in all cases**

```
size_t const SZ = 6 * 7;
double x[SZ];
```

Constant expression context

Compile-time evaluation

Constant Expression Contexts

- **Non-type template arguments**

```
std::array<Element, 5> arr{};
```

- **Array bounds**

```
double matrix[ROWS][COLS]{};
```

- **Case expressions**

```
switch (value) {
case 42:
  //...
}
```

- **Enumerator initializers**

```
enum Light {
  Off = 0, On = 1
};
```

- **`static_assert`**

```
static_assert(order == 66);
```

- **`constexpr` variables**

```
constexpr unsigned pi = 3;
```

- **`constexpr if` statements**

```
if constexpr (size > 0) {
}
```

- **`noexcept`**

```
Blob(Blob &&) noexcept(true);
```

- **...**

```
static_assert(isGreaterThanZero(Capacity));

static_assert(sizeof(int) == 4, "unexpected size of int");
```

- **`static_assert` checked at compile-time**
  - Compilation fails if it evaluates to `false`
  - The compiler needs to be able to evaluate the expression
- **Syntax:**
  - `static_assert(condition);`
  - `static_assert(condition, message);`

```
constexpr unsigned pi = 3;
constinit unsigned pi = 3;
```

- **Evaluated at compile-time (mandatory)**
- **Initialized by a constant expression**
  - Literal value
  - Expression computable by the compiler
  - `constexpr` function calls
- **Require literal type**
- **Can be used in constant expression contexts**

- **Possible contexts**
  - Local scope
  - Namespace scope
  - `static` data members
- **`constexpr` variables are `const`**
- **`constinit` variables are non-`const`**

```cpp
constexpr auto factorial(unsigned n) {
  ...
}
```

- **constexpr functions can...**
  - ... have local variables of “literal” type. The variables must be initialized before used.

```cpp
int local;
```

```cpp
LiteralType local{};
```

```cpp
int local;
f(local);
```

```cpp
std::string local{};
```

Since C++20

  - ... use loops, recursion, arrays, references
  - ... even contain branches that rely on run-time features, if branch is not executed during compile-time computations, e.g., `throw`
  - ... but can only call `constexpr` functions

- **`constexpr` evaluation can now...**
  - ... allocate dynamic memory (`new`, `delete`) that is cleaned up by the end of the compilation

```
constexpr auto allocate() -> int* {
  return new int{};
} //requires corresponding delete somewhere
```

  - ... be virtual member functions

```
struct Base {
  constexpr virtual auto modify() -> void;
};
```

  - but still cannot use exception handling (`throw`, `try/catch`), on executed path

```
constexpr auto throwError() -> void {
  throw std::logic_error{""};
}
```

- **Constexpr functions are usable in constexpr and non-constexpr contexts**

```
constexpr auto factorial(unsigned n) {
  auto result = 1u;
  for (auto i = 2u; i <= n; i++) {
    result *= i;
  }
  return result;
}

constexpr auto factorialOf5 = factorial(5);

auto main() -> int {
  static_assert(factorialOf5 == 120);
  std::cout << factorial(5);
}
```

- **Consteval functions are usable in constexpr contexts only**

```
consteval auto factorial(unsigned n) {
  auto result = 1u;
  for (auto i = 2u; i <= n; i++) {
    result *= i;
  }
  return result;
}

constexpr auto factorialOf5 = factorial(5);

auto main() -> int {
  static_assert(factorialOf5 == 120);
  unsigned n;
  std::cin >> n;
  std::cout << factorial(n);
}
```

- **The compiler will prevent Undefined Behavior**
  - Leads to compilation error

```cpp
constexpr auto divide(int n, int d) -> int {
  return n / d;
}

constexpr auto surprise = divide(0, 0);

auto main() -> int {
  std::cout << surprise;
}
```

```
..\CTUB.cpp:7:33:   in 'constexpr' expansion of 'divide(0, 0)'
..\CTUB.cpp:4:12: error: '(0 / 0)' is not a constant expression
   return n / d;
          ~~^~~
```

Restrictions Only on Used Branch

- **If constexpr evaluation does not reach an invalid statement, the code is valid**

```
constexpr auto throwIfZero(int value) -> void {
  if (value == 0) {
    throw std::logic_error{""};
  }
}

constexpr auto divide(int n, int d) -> int {
  throwIfZero(d);
  return n / d;
}

constexpr auto five = divide(120, 24);
constexpr auto failure = divide(120, 0);
```

?

```
auto whatIsTheAnswer() -> int {
  return 42;
}

static_assert(whatIsTheAnswer() == 42);
```

```
constexpr int global = 42;
constexpr auto allocate(bool useGlobal) -> int const* {
  if (useGlobal) {
    return &global;
  } else {
    return new int{};
  }
}
constexpr int const* ptr = allocate(true);
```

?

`auto whatIsTheAnswer() -> int {`
`  return 42;`
`}`
 	
`static_assert(whatIsTheAnswer() == 42);`
	Incorrect
	
	The expression in static_assert requires a compile-time expression. `whatIsTheAnswer()` is not a constexpr function.

`constexpr int global = 42;`
`constexpr auto allocate(bool useGlobal) -> int const* {`
`  if (useGlobal) {`
`    return &global;`
`  } else {`
`    return new int{};`
`  }`
`}`
`constexpr int const* ptr = allocate(true);`
	Correct
	
	As long as the path with the new expression is not taken, it is valid to have the code in a `constexpr` function. `allocate(false);` could not be used to initialize a compile-time constant.

Literal Types

Ostschweizer
Fachhochschule

Literal Types

- **Built-in scalar types, like `int`, `double`, pointers, enumerations, etc.**
- **Structs with some restrictions[1]**
  - Trivial destructor (non-user-defined)
  - With a `constexpr`/`consteval` constructor
- **Lambdas**
- **References**
- **Arrays of literal types**
- **`void`**
- **Literal Types can be used in `constexpr` functions, but only `constexpr` member functions can be called on values of literal type**

[1] https://en.cppreference.com/w/cpp/named_req/LiteralType

Literal Class Type Example

- **Trivial Destructor**
- **Constexpr/consteval Constructor**
  - At least one
- **Constexpr/consteval Member Functions**
  - `const` & `non-const`
  - Only those are useable in `constexpr` context
- **Can be a template**
- **It can still contain non-constexpr constructors and member functions**

```
template <typename T>
class Vector {
  constexpr static size_t dimensions = 3;
  std::array<T, dimensions> values{};
public:
  constexpr Vector(T x, T y, T z)
    : values{x, y, z}{}
  constexpr auto length() const -> T {
    auto squares = x() * x() +
                   y() * y() +
                   z() * z();
    return std::sqrt(squares);
  }
  constexpr auto x() -> T& {
    return values[0];
  }
  constexpr auto x() const -> T const& {
    return values[0];
  }
  //...
};
```

- **Can be used in `constexpr` and non-`constexpr` contexts**
- **Non-`const` member functions can be used to modify the object**
- **`constexpr` variables are `const`**

```
Constexpr auto create() -> Vector<double> {
  Vector<double> v{1.0, 1.0, 1.0};
  v.x() = 2.0;
  return v;
}

constexpr auto v = create();
static_assert(doubleEqual(v.length(), 2.4495));

auto main() -> int {
  //v.x() = 1.0; //possible with constinit
  auto v2 = create();
  v2.x() = 2.0;
}
```

- **Note on Vector: Has hardcoded three dimensions (x/y/z).**

```
template <size_t n>
struct fact {
  static size_t const value{(n > 1)?  n * fact<n-1>::value : 1};
};

template <>
struct fact<0> { // recursion base case: template specialization
  static size_t const value = 1;
};

TEST(testFactorialCompiletime) {
  constexpr auto result = fact<5>::value;
  ASSERT_EQUAL(result, 2 * 3 * 4 * 5);
}
```

- **"Integer" only (almost) through non-type template parameters (Until C++20*)**

- **Capture types (the types returned by lambda expressions) are literal types as well**
  - They can be used as types of `constexpr` variables
  - They can be used in `constexpr` functions
  - Restrictions to `constexpr` functions and variables apply as well
- **Examples for demonstration purposes**

```
constexpr double pi = 3.14159;

constexpr auto area = [](double r) {
  return pi * r * r;
};

constexpr auto circleArea = area(2.0);
```

```
constexpr auto cubeVolume(double x) {
  auto area = [x] {return pi * x * x;};
  return area() * x;
}

constexpr auto cV = cubeVolume(5.0);
```

Compile-Time Factorial with Variable Templates

```
template <size_t N>
constexpr size_t factorial = factorial<N - 1> * N;

template <> //Base case
constexpr size_t factorial<0> = 1;
```

- **Variable templates...**
  - ... can be `constexpr/constinit`
  - ... can be defined recursively `->` specialization to define the base case
- **Useage**
  - Template-ID (Name and template arguments)

```
static_assert(factorial<0> == 1);
static_assert(factorial<5> == 120);
```

Right or Wrong? (Preview)

?

```
template <std::size_t N>
constexpr size_t McCarthy91 = [] {
    if constexpr (N <= 100) {
      return McCarthy91<McCarthy91<(N + 11)>>;
    }
    return N - 10;
  }();
```

```
struct Point {
  constexpr Point(double x, double y)
    : x{x}, y{y}{}
  Point() : Point{0.0, 0.0}{}
private:
  double x;
  double y;
};

constexpr Point origin{0.0, 0.0};
```

?

<pre>template <std::size_t N>
constexpr size_t McCarthy91 = [] {
    if constexpr (N <= 100) {
      return McCarthy91<McCarthy91<(N + 11)>>;
    }
    return N - 10;
  }();</pre>
	Correct
	
	Variable templates can be initialized by lambdas, which are literal types, as long as there is a base case. With `constexpr if` we reach that base case.

<pre>struct Point {
  constexpr Point(double x, double y)
    : x{x}, y{y}{}
  Point() : Point{0.0, 0.0}{}
private:
  double x;
  double y;
};
 	
constexpr Point origin{0.0, 0.0};</pre>
	Correct
	
	Point is a literal type. Not all constructors need to be `constexpr`. However, in compile-time constant expressions only the `constexpr` constructors can be used.

User-Defined Literals

Ostschweizer Fachhochschule

- **Type for velocity**

```
template <typename Unit>
struct Speed {
  constexpr explicit Speed(double value)
    : value{value}{};
  constexpr explicit operator double() const {
    return value;
  }
private:
  double value;
};
```

- **Can be used in**

Quite verbose

Example	Valid
`Speed<Unit::kmh> s{5.0};`	Yes
`Speed<Unit::kmh> s = 5.0;`	Non-explicit
`auto s = Speed<Unit::kmh>{5.0}`	Yes
`auto s = 5.0;`	Not a speed object

- **Repetitive occurrence of explicit conversion `Speed<Unit::XYZ>{x}`**

```
auto speed1 = Speed<Kph>{5.0};
auto speed2 = Speed<Mph>{5.0};
auto speed3 = Speed<Mps>{5.0};
```

- **What if we had the possibility to attach units to our literals?**
  - User-defined literals

```
auto speed1 = 5.0_kph;
auto speed2 = 5.0_mph;
auto speed3 = 5.0_mps;
```

- **Overloading**
  - `UDLSuffix` could lexically be any identifier, but must start with underscore _
    (other suffixes are reserved for the standard)
  - Allows to add dimension, conversion, etc.
  - If possible define UDL operator functions as `constexpr`

```
operator"" _UDLSuffix()
```

- **Add the suffix to integer, float and string literals**
  - Suffix belongs to literal (no whitespace between)

```
5.0_kph;  //correct
5.0 _mph; //wrong
```

- **Rule: put overloaded UDL operators that belong together in a separate namespace**
  - Only a `using namespace` can import them

```
using namespace velocity::literals;
```

```
namespace velocity::literals {

constexpr inline auto operator"" _kph(unsigned long long value) -> Speed<Kph> {
  return Speed<Kph>{safeToDouble(value)};
}

constexpr inline auto operator"" _kph(long double value) - Speed<Kph> {
  return Speed<Kph>{safeToDouble(value)};
}

//...

}
```

User-Defined Literals for **Speed**

```
TEST(overtakePedestrianAt10Kph) {
  ASSERT(isFasterThanWalking(10.0_kph));
}

TEST(testConversionFromKphToMph) {
  ASSERT_EQUAL(1.60934_kph, 1.0_mph);
}
```

- **Shorter and more expressive literals**
- **`ASSERT_EQUAL` for double already has a margin for its comparison**

- **For literal numbers the following signatures are useful**
  - Integral constants

```
auto operator"" _suffix(unsigned long long) -> TYPE
```

  - Example

```
constexpr inline auto operator"" _kph(unsigned long long value) -> Speed<Kph> {
  return Speed<Kph>{safeToDouble(value)};
}
constexpr auto speed = 5_kmh;
```

  - Floating point constants

```
auto operator "" _suffix(long double) -> TYPE
```

- **For string literals the following signature is useful**

```
auto operator"" _suffix(char const *, std::size_t len) -> TYPE
```

```
namespace mystring {
inline auto operator"" _s(char const *s, std::size_t len) -> std::string {
  return std::string { s, len };
}
}
```

  - Note: This implementation above cannot be `constexpr`. Why?
  - Example:

```
using namespace mystring;
auto s = "hello"_s;
s += " world\n";
std::cout << s;
```

- **“RAW” UDL Operator**

```
auto operator"" _suffix(char const *) -> TYPE
```

```
namespace mystring {
inline auto operator"" _s(char const *s) -> std::string  {
  return std::string { s };
}
}
```

  - Note: Works only for integral and floating literals, not for string literals!
  - Example:

```
using namespace mystring;
auto rs = 42_s;
rs += " raw\n";
std::cout << rs;
```

Raw UDL Operator Applied

- **Ternary suffix**
  - Base 3
- **Examples**

Ternary	Decimal
0	0
1	1
2	2
10	3
11	4
12	5
20	6
21	7
22	8
100	9

- **Problem: exception at run-time**

```
namespace ternary {
auto operator"" _3(char const *s) -> unsigned long long {
  size_t convertedupto{};
  auto res = std::stoull(s, &convertedupto, 3u);
  if (convertedupto != strlen(s))
    throw std::logic_error{"invalid ternary"};
  return res;
}
}
```

```
using namespace ternary;
int four = 11_3;
std::cout << "four is " << four << '\n';
try {
  four = 14_3; // throws
} catch (std::exception const &e) {
  std::cout << e.what() << '\n';
}
```

- **Template UDL Operator**

```
template <char...>
auto operator"" _suffix() -> TYPE
```

- **Empty parameter list**
- **Variadic template parameter**
- **Characters of the literal are template arguments**

```
120_ternary; // => operator "" _ternary()
             // with template arguments '1', '2' and '0'
```

- **Unfortunately, the template UDL operator does not work with string literals (Until C++20)**

Raw UDL Extreme: Ternary Number at Compile-Time (1/2)

- **Run-time errors for number conversion is bad**
- **There exists a variadic template version of UDL operators**
- **Interpretation of the characters (at compile-time) often requires a variadic class/variable template with specializations**

```
template <char ...Digits> requires (is_ternary_digit(Digits) && ...)
constexpr auto operator"" _ternary() -> unsigned long long {
  return ternary_value<Digits...>;
}
```

- **We will also need a helper function to get the value of the digit: 3^n**

```
constexpr auto three_to(std::size_t power) -> unsigned long long {
  return power ? 3ull * three_to(power - 1) : 1ull;
}
```

Raw UDL Extreme: Ternary Number at Compile-Time (2/2)


```
template <char ...Digits>
extern unsigned long long ternary_value;

template <char ...Digits>
constexpr unsigned long long ternary_value<'0', Digits...> {
  ternary_value<Digits...>
};

template <char ...Digits>
constexpr unsigned long long ternary_value<'1', Digits...> {
  1 * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};

template <char ...Digits>
constexpr unsigned long long ternary_value<'2', Digits...> {
  2 * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};

template<>
constexpr unsigned long long ternary_value<>{0};
```

Ternary Number Compile-Time Steps

- **Example: `120_ternary`**
- **`120_ternary` -> resolves to `ternary_value<'1', '2', '0'>`**

Partial specialization: `ternary_value<'1', Digits...>`

Value: `1 * 3`^{2} `+ ternary_value<'2', '0'>`

> Partial specialization: `ternary_value<'2', Digits...>`
>
> Value: `2 * 3`^{1} `+ ternary_value<'0'>`
>
> > Partial specialization: `ternary_value<'0', Digits...>`
> >
> > Value: `ternary_value<>`
> >
> > > Partial specialization: `ternary_value<>`
> > >
> > > Value: `0`
> >
> > Value: `0`
>
> Value: 6 `from 2 * 3`^{1} `+ 0`

Value: 15 `from 1 * 3`^{2} `+ 6`

- **Can we avoid the duplication of the specialization for `'0'`, `'1'` and `'2'`?**

```
constexpr auto is_ternary_digit(char c) -> bool {
  return c == '0' || c == '1' || c == '2';
}

constexpr auto value_of(char c) -> unsigned {
  return c - '0';
}

template <char D, char ...Digits>
constexpr ternary_value<D, Digits...> {
  value_of(D) * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};
```

- **Standard suffixes don't have a leading underscore**
- **Suffix for `std::string`: `s`**
- **Suffix for `std::complex` (imaginary): `i, il, if`**
- **Suffixes for `std::chrono::duration`: `ns, us, ms, s, min, h`**
- **More might be defined in the future**
- **Is the following example a problem?**

```
using namespace std::string_literals;
using namespace std::chrono_literals;
auto one_s = 1s;
auto one_point_zero_s = 1.0s;
auto fourty_two_s = "42"s;
```

Right or Wrong? (Preview)

?

```
auto operator"" _m(unsigned long long v) -> Distance {
  return Distance{v};
}

auto depth = -15_m;
```

```
auto operator"" _regex(std::string value) {
  return std::regex{value};
}

auto isIdentifier(std::string testee) -> bool {
  auto pattern = "[a-zA-Z_][a-zA-Z0-9_]*"_re;
  return std::regex_match(testee, pattern);
}
```

Right or Wrong?

?

`auto operator"" _m(unsigned long long v) -> Distance {`
`  return Distance{v};`
`}`
 	
`auto depth = -15_m;`
	Correct
	
	Even though the literal operator only takes an unsigned value, this works, if the `Distance` type has a unary minus operator. First the literal `15_m` is resolved then the unary minus is applied to the resulting object.

`auto operator"" _regex(std::string value) {`
`  return std::regex{value};`
`}`
 	
`auto isIdentifier(std::string testee) -> bool {`
`  auto pattern = "[a-zA-Z_][a-zA-Z0-9_]*"_re;`
`  return std::regex_match(testee, pattern);`
`}`
	Incorrect
	
	The possible overload of UDL operators are limited to a specific set of parameters. `std::string` is not a valid overload.

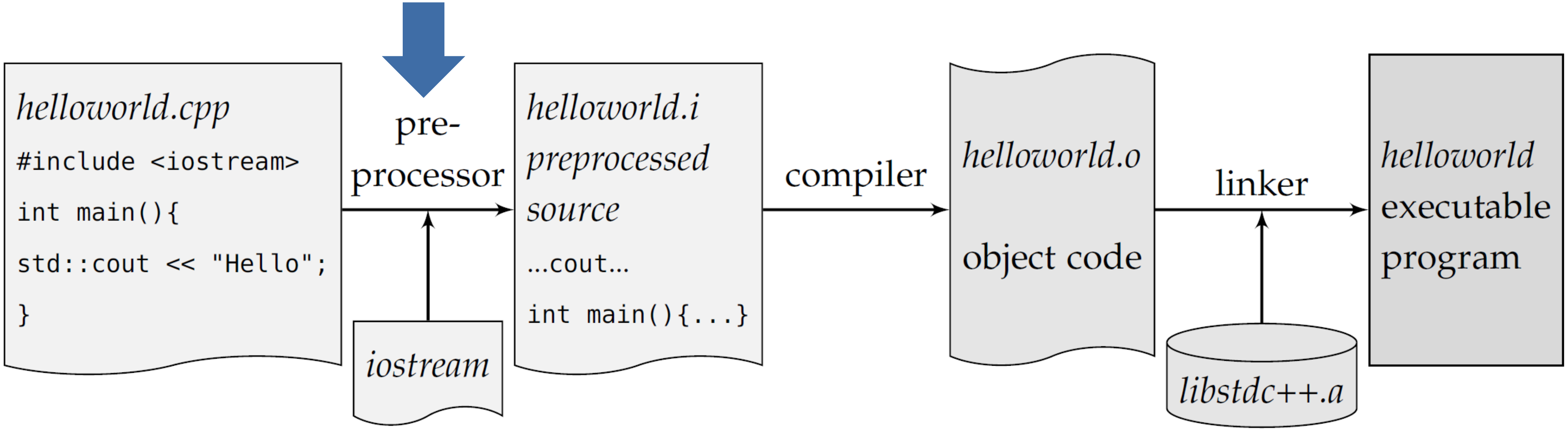
Summary

Summary

- **Many computations for which the arguments are known upfront can be computed at compile time**
- **Until C++20 dynamic memory allocation is not possible**
- **All literal types can be used in constexpr contexts**
- **User defined literals help giving meaning to simple values**
- **They can be used to compute numbers at compile-time**

Preprocessor

OST
Ostschweizer
Fachhochschule


helloworld.cpp
#include <iostream>
int main(){
std::cout << "Hello";
}
pre-
processor
iostream
helloworld.i
preprocessed
source
...cout...
int main(){...}
compiler
helloworld.o
object code
linker
libstdc++.a
helloworld
executable
program

```
# define identifier replacement-list new-line
```

- ***Identifier* is a unique name in the preprocessor's monolithic namespace**
  - By convention the preprocessor names are ALL_CAPS
  - Valid until `#undef NAME`
- **Replacement-list is a, possibly empty, sequence of preprocessor tokens, i.e. parts of of code**
- **New-line terminates the replacement-list**
  - Multi-line macros need to escape the line-breaks with a backslash
- **Example**

```
#define NUMBER_OF_ROWS 5
```

- **Object- and function-like macro names are replaced by the replacement-list they define**

```
#define NUMBER_OF_ROWS 5

int rows[NUMBER_OF_ROWS];
```

➡

```


int rows[5];
```

- **After the substitution replacement is applied again**
  - In this retry already replaced macros are ignored (to avoid endless recursion)

```
#define A B
#define B A

A //gets replaced by B and then by A again
```

```
# define identifier ( identifier-list? ,? ...? ) replacement-list new-line
```

- **Features an optional parameter list, containing only names (no types)**
- **Ellipsis (…) can be used for a variadic number of arguments**

```
#define ADD(A, B) A + B
```

```
#define N_TIMES(N, ACTION) \
  for (int i = 0; i < N; i++) { \
    ACTION; \
  }
```

- **# in the replacement list before a macro parameter converts it into a string-literal**

```
#define CUTE(name) cute::test(&name, #name)
```

```
CUTE(testFunction)
```

→

```
cute::test(&testFunction, "testFunction")
```

- **Can be helpful for log output as there is no reflection**

```
#define PRINT_RESULT(TERM) std::cout << #TERM " = " << TERM
```

```
PRINT_RESULT(1 + 2);
```

→

```
std::cout << "1 + 2 " " = " << 1 + 2;
```

- **## in the replacement list between tokens splices them to a single new token**

```
#define CALL(VERSION) Action ## _ ## VERSION ()
```

```
void Action_A();
void Action_B();

CALL_ACTION(B);
```

➡

```
void Action_A();
void Action_B();

Action_B();
```

- **Textual inclusion of another file**
  - Typically, for inclusion of header files but there is no restriction to include any file-type
- **#include "path"**
  - Typically, used for including a header file from the same project or workspace

```
#include "Subsystem/Component/Module.hpp"
```

- **#include <path>**
  - Typically, used for external includes, like the standard library or system libraries

```
#include <algorithm>
#include <boost/operators.hpp>
```

- **Enable a section depending on a condition**

```
#if constant-expression new-line

#elif constant-expression new-line

#else new-line

#endif new-line
```

- **Check for macro definitions**

```
#if defined identifier new-line
```

```
#ifdef identifier new-line
```

- **Inverted**

```
#ifndef identifier new-line
```

- **Enforcing semicolon after a macro reference can be achieved by putting the contained statements in a do-while loop**

```
#define SOME_STATEMENTS \
  do { \
    Statement1();
    Statement2();
  } while (false)
```

- **Use of function-like parameters are put in parentheses when operator precedence is important**

```
#define MUTIPLY(A, B) \
  (A) * (B)

Int result = MULTIPLY(1 + 1, 2)
```

Predefined Macros

Macro	
`__cplusplus`	202002L (current C++ standard int literal)
`__DATE__`	String literal "Mmm dd yyyy"
`__FILE__`	String literal containing the name of the source file
`__LINE__`	int literal of the current line number
`__TIME__`	String literal "hh:mm:ss"
…	

Don’ts

- **#pargma directive**
  - Implementation defined behavior

- **Relative include paths to current file**

```
#include "../../Module.hpp"
```

  - Use relative paths to a known include directory
  - Respect case-sensitive spelling (also on Windows)

- **Redefining keywords is undefined behavior**

```
#define private   public
#define protected public
#define class     struct
```

- **Using the preprocessor unless inevitable!**

Self Study

Compile time code

Ostschweizer Fachhochschule

Vanished Calls

- **Have a look at the following code on Godbolt and the binary that is generated**

```cpp
constexpr auto factorial(unsigned n) {
  auto result = 1u;
  for (auto i = 2u; i <= n; i++) {
    result *= i;
  }
  return result;
}

constexpr auto factorialOf5 = factorial(5);

auto main() -> int {
  static_assert(factorialOf5 == 120);
  return factorialOf5;
}
```

```asm
main:
  push rbp
  mov rbp, rsp
  mov eax, 120
  pop rbp
  ret
```

https://www.godbolt.org/z/suMDYp

- **No calls to the `factorial` function are present anymore**

- **Compile-time conditional inclusion statement**

```
auto printAll() -> void {
}

template <typename First, typename...Types>
auto printAll(First const& first, Types const&...rest) -> void {
  std::cout << first;
  if constexpr (sizeof...(Types)) {
    std::cout << ", ";
    printAll(rest...);
  }
}
```

- **Requires compile-time expression**

`if constexpr` Instantiated

- **Instance for `printAll("Hello"s);`**

```
auto printAll(std::string const& first) -> void {
  std::cout << first;
  if constexpr (0) {
    std::cout << ", ";
    printAll(); //rest... expansion
  }
}
```

```
auto printAll(std::string const& first) -> void {
  std::cout << first;





}
```

- **We don't need a base case anymore!**