Department I - C Plus Plus

Modern and Lucid C++ Advanced
for Professional Programmers

Week 1 – C Plus Plus Recap

OST
Ostschweizer
Fachhochschule

Thomas Corbat / Felix Morgner

Rapperswil, 21.02.2023
FS2023

Basics Language Features

OST
Ostschweizer
Fachhochschule

Values vs. References

- **Semantics of Values and (lvalue) References**

```
auto foo(int p) -> void {
  p = 23;
}
...
int i = 5;
foo(i);
```

```
auto foo(int& p) -> void {
  p = 23;
}
...
int i = 5;
foo(i);
```

- **What is the value of `i` after `foo(i)`?**

- **We will look at rvalue References too**

```
auto foo(int&& p) -> void {
  ...
}
...

foo(5);
```

Constness

- **With the `const` keyword you have a powerful tool to improve your code**
- **Where can we place the `const` keyword?**
  - Types of Variables/Parameters/Return Types
  - Member Functions (`this` pointer)
  - Pointers

```
Spaceship vulture{};
vulture.load(Stuff{});

Spaceship const eagle{};
out << eagle.built();
```

```
struct Spaceship {
  auto load(Stuff const&) -> void;
  auto built() const -> Date;
};
```

- **`load()` modifies the Spaceship thus it must not be `const`**
- **`built()` only queries the date, does not modify Spaceship and therefore should be `const`**

Classes/Enums/Functions/Lambdas

- **Composite types (Classes)**

```
struct Telephone {
  auto dial(PhoneNumber) -> void;
private:
  Log<Calls> call_log;
};
```

- **Enums**

```
enum class Gender {
  female, male, apache
  //50 more
};
```

- **Lambdas**

```
auto catchMeIfYouCan() -> void {
  Criminal abagnale{"Frank"s};
  auto hanratty = [abagnale] {
    offer_deal(abagnale);
  };
}
```

- **Functions**

```
auto cookBreakfast(Kitchen& kitchen) -> Meal {
  auto frying_pan = kitchen.getPan();
  kitchen.sink().wash(frying_pan);
  auto oven = kitchen.oven();
  oven.put(frying_pan);
  oven.turnOn(Oven::Temperature::hot);
  frying_pan.add(Egg{});
  frying_pan.add(Bacon{});
  return frying_pan.slightlyBurntFood();
}
```

Namespaces

- **Namespaces**
  - Named

```
namespace Labyrinth {
  Minotaur asterion{};
}
```

  - Global

```
//Global namespace
auto main() -> int {}
```

  - Anonymous

```
namespace {
}
```

  - Inline

```
namespace MyLib {
  inline namespace V1 {
    struct WillImprove {
    };
  }
}
```

- **Variables**
  - Local

```
auto tourDeRappi() -> void {
  Restaurant baeren{};
}
```

  - Global

```
Climate warming{};
```

  - Member

```
class Ship {
  int numberOfLeaks{};
};
```

  - Can all be `static`, what would each mean?

Exceptions

- **You can throw everything in C++**
- **`try/catch`**
  - but no `finally` – is that a problem?
- **Catch clauses tried from top to bottom**
- **Exception wildcard ellipsis (`...`)**
- **Good style to**
  - Throw by value
  - Catch by `const &`

```cpp
auto goHomeFromLecture() -> void try {
  try {
    waitForBell();
  } catch (FellAsleepException const & e) {
    wakeUp();
    wonderWhyTheRoomIsDarkAndEverybodyIsGone();
  }
  packYourStuff();
  try {
    getUp();
  } catch (LegGotPinsAndNeedlesException const & e) {
    dieOfPain() || stayMotionless(TIME_TO_RECOVER);
  }
  leaveHSR();
  gotoTheStation();
  //...
} catch(...) {
  //Did not expect that. I don't know what it is.
  wonderAboutException();
  //Let somebody else care...
  throw;
}
```

(Integral) Type Promotion

- **Operators for primitive types are specified in the language (E.g +, -, /,...)**
  - Caveat: Only for expressions with operands of the same type

```
int intValue1 = 15;
int intValue2 = 24;
auto intIntSum = intValue1 + intValue2;

long longValue1 = 11l;
auto longIntSum = longValue1 + intValue1;

double doubleValue = 128.0;
auto doubleIntSum = doubleValue + intValue1;

unsigned unsignedValue = 99u;
auto unsignedIntSum = unsignedValue + intValue1;
```

Unsigned Overflow

- **Negative/positive overflow of unsigned integers is defined**
  - However, it might feature unexpected behavior

```
int zeroIndex = 0;
for (unsigned size = 5; size <= 10; size--) {
  if (zeroIndex <= size - 1) {
    std::cout << "access with 0 is ok for size " << size << '\n';
  } else {
    std::cout << "access with 0 is not ok for size " << size << '\n';
  }
}
```

- **Program output:**

```
access with 0 is ok for size 5
access with 0 is ok for size 4
access with 0 is ok for size 3
access with 0 is ok for size 2
access with 0 is ok for size 1
access with 0 is ok for size 0
```

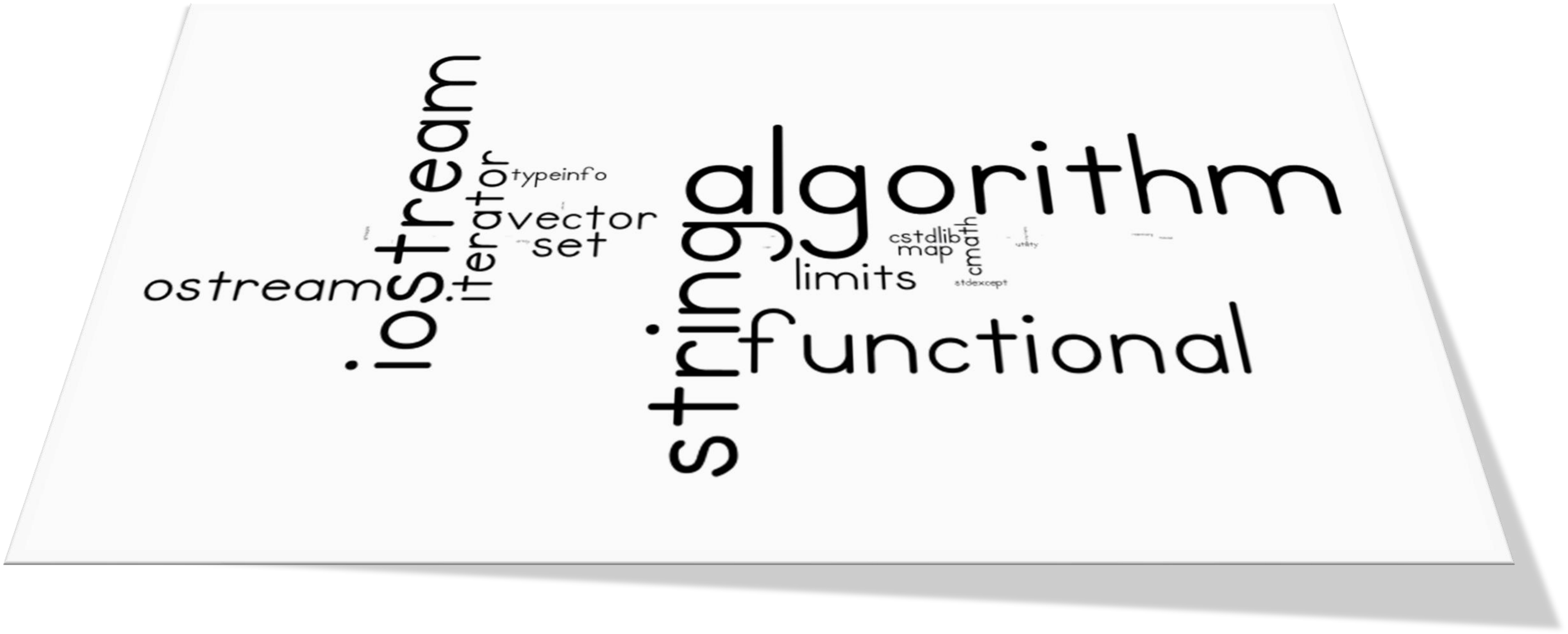
Keyword Trivia

- `explicit`
- `inline`
- `using`
- `virtual`
- `mutable`
- `friend`
- `override`
- `final`

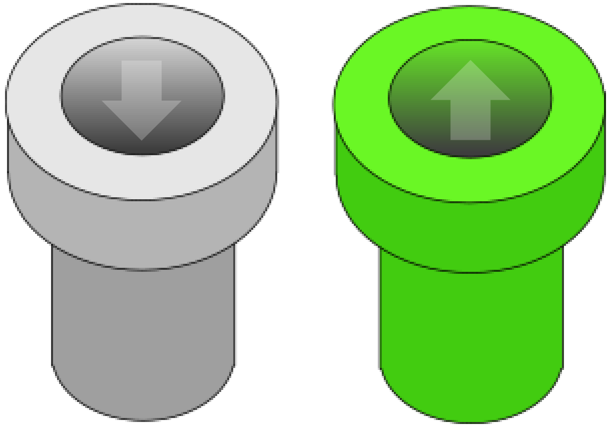
- **In which context can they be used?**
- **What do they mean?**
- **Do you know any other keywords?**

Standard Library

Standard Library



Streams



- **Input and output for programs**
  - `std::cin` and `std::cout` (only in `main` function)
- **Using and overloading input and output operators**
  - `operator >>`
  - `operator <<`

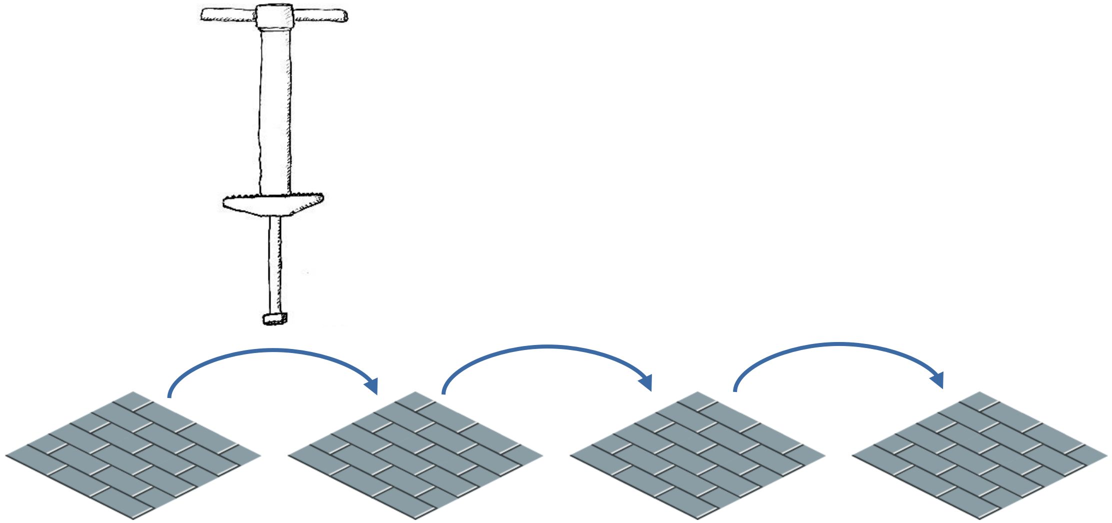
```
struct Order {
  unsigned amount;
  Product product;
};

auto operator<<(std::ostream& out, Order const& o) -> std::ostream& {
  out << o.amount << "x" << o.product;
  return out;
}
```

Iterators

- **Iterators specify ranges**

- **Jump from element to element**

```
auto iteration() -> void {
  std::vector<Tile> tiles{...};
  auto pogo_stick = std::begin(tiles);
  pogo_stick++;
  auto tile = *pogo_stick;
}
```



- **Capabilities depend on the specific iterator type**

- **Categories:**
  Input iterator
  Forward iterator
  Bidirectional iterator
  Random access iterator
  Output iterator

- **We will implement our own iterators in this module**

Algorithms

- **Benefits of using standard algorithms over hand-written loops**
  - Correctness
  - Easier to read and understand
  - Performance

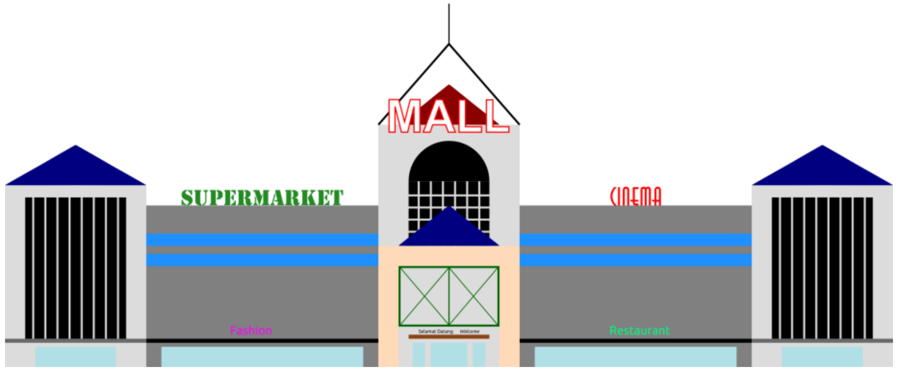
```
auto findWithLoop(std::vector<int> const& values, int const v) ->  {
  auto const end = std::end(values);
  for(auto it = std::begin(values); it != end; ++it) {
    if (*it == v) {
      return true;
    }
  }
  return false;
}
```

```
auto findWithAlgorithm(std::vector<int> const& values, int const v) -> bool {
  auto const pos = std::find(std::begin(values), std::end(values), v);
  return pos != std::end(values);
}
```

Smart Pointers

- **Heap memory management in C++ should be handled with `std::shared_ptr` and `std::unique_ptr`**

```
auto mall() -> void {
  auto moreDoor = std::make_shared<Mall>();
  auto sideDoor = moreDoor;
  auto moreDoor = sideDoor;
  auto hodor = moreDoor;
}
```



```
auto toolbox() -> void {
  auto handle = std::make_unique<Hammer>();
  //You cannot have a hammer with two handles
  auto handle_too = handle;
}
```

- **We will look at legacy alternatives using `new` and `delete` in this module though**

Advanced Topics

OST
Ostschweizer
Fachhochschule

Inheritance

- **The most capable means of introducing coupling into your software**

```
class Parent {
  //Members of Parent
};

class Child : public Parent {
  //Members of Child
};
```

- **Can be used to "mix-in" functionality into your classes**
  - E.g. `boost::operators`
- **Multiple inheritance will be discussed in more detail**
  - Especially diamond hierarchies

Templates

- **To write compile-time polymorphic classes and functions**
  - With template argument deduction for template functions

```
template<typename T>
class Box {
  T content;
public:
  auto peek() const -> T const& {
    return content;
  }
 auto open() -> T& {
    return content;
  }
};
```

```
template<typename T>
auto wrap(T const& t) -> Box<T> {
  return Box<T>{t};
}


...
Present doll{};
Box<Present> gift = wrap(doll);
```

- **Variable templates will be introduced**
- **SFINAE**

Testing

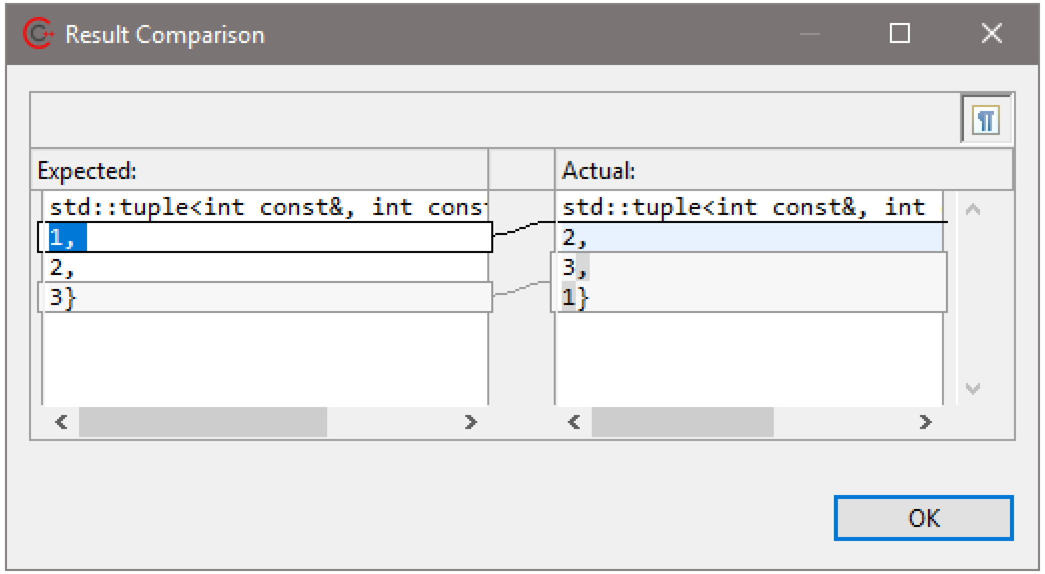
- **Creating unit tests using CUTE**

```
TEST(thisIsATest) {
  ASSERTM("start writing tests", false);
}
```

- **The CUTE plug-in has a simplified wizard, regarding libraries**

- **Also new the visualization of tuples**

```
TEST(testTuplesAreDifferent) {
  int const a{1};
  int const b{2};
  int const c{3};
  ASSERT_EQUAL(std::tie(a, b, c), std::tie(b, c, a));
}
```



```
std::tuple<int const&, int cons
1,
2,
3}
```

```
std::tuple<int const&, int
2,
3,
1}
```

- **We might have a look at a different test approach later in this course for our container example**

Undefined Behavior

DANGER

Undefined Behavior

- **Signed overflow**

```
auto evilInside() -> void {
  int i = 1;
  while (i++ > 0);
}
```

- **Dangling references**

```
auto evilOutside() -> int& {
  int i = 0;
  return i;
}
```

- **Multiple side effects**

```
auto whatsThat() -> void {
  int i{};
  i = i++ + ++i;
}
```

- **Accessing (possibly) unallocated memory**

```
auto wishMeLuck() -> void {
  std::vector<int> values{1, 2};
  int second = values[2];
  ...
}
```

- **Not UB but the emojis of programming**

```
bool statement = ...;
if (statement == true) {
  return false;
} else if (false == statement) {
  return true;
} else {
  return true;
}
```