C++

Daniel Zint, Rafael Ravedutti, Harald Köstler
23.09.2019

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lecture 2

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Schedule 24.09.2019

- Pointer and references
- Const, auto, decltype
- Practical session
- Using declarations
- Strings
- Vectors
- Iterators basics
- STL algorithms: sort, find, copy
- Practical session
- Expressions
- Statements
- Dynamic Memory
- Practical session

References

- & is an alias/reference

```
int a = 5;
int &b = a;
b = 7;
std::cout << a << std::endl;
```

- Important in functions

```
void incr(int& a) {
    ++a;
}

int main()
{
    int a = 5;
    incr(a);
    std::cout << a << std::endl;
}
```

- In general, & returns the address of a variable.

```
int a = 5;
std::cout << a << std::endl;
std::cout << &a << std::endl;
```

Pointer

- C pointer

```
int ival = 1024;
int *pi = &ival;   // pi points to an int
int **ppi = &pi;   // ppi points to a pointer to an int
```

- Smart pointer (will be covered next week in the lecture)

Const

- Constant variables

```
const int a = 5;
```

- Constant references

```
int a = 5;
const int &b = a;
```

- Pointers to constant variables

```
const int a = 5;
const int *b = &a;
```

- Constant pointers

```
int a = 5;
int * const b = &a;
```

- There is more about const, i.e. in classes. Member functions can be declared constant, to state that they do not change member variables.
- const does not change the behavior of a program. It is just a tool for you, to avoid bugs. Use it!

Constexpr

- Just like const but evaluated at compile time

```
constexpr int foo() {
    return 3;
}

int main()
{
    constexpr int a = 5;
    constexpr int b = foo();
}
```

- Whenever possible constexpr should be preferred over const

Auto

- Get type of a variable automatically

```
auto a = 5;     // int
auto b = 5.;    // double
auto c = 5.f;   // float
```

- Even works as a return type of a function

```
auto square(int a) {
    return a * a;
}

int main()
{
    int a = 5;
    int b = square(a);
}
```

- Using auto is recommended but don’t overdo it.
  Beware of implicit conversion when working with auto!
- I recommend using it only to shorten down notation, not in general.

Decltype

- Declare variable with type of some expression (i.e. of a variable or a function call) without evaluating it.

```
int a = 0;
decltype(a) c = a;   // c is an int
decltype((a)) d = a; // d is a reference to a
decltype(f()) e;
```

- Annoying syntax (welcome to C++) but more safe than auto. Implicit conversion is not such a big deal here.

Using Declarations

- The keyword using allows using functions without writing their namespace

```cpp
#include <iostream>

using std::cout; using std::endl;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

- Do not use using in header files (.h-files)! This might lead to ambiguity when this header file is included somewhere.

Strings

- A string in C

```
const char* chArr = "I am a char array";
```

  please don’t do that

- A string in C++

```
std::string str = "I am a string";
```

- The std::string is what you want. You can concatenate strings, get their lengths and easily modify them.

- Beware that writing a string with “…” does not create a string but a char array! Explicit conversion is required sometimes

```
std::string("not a string yet");
```

Strings

- You can also implement a string using

```
std::string str(10, "x");   // xxxxxxxxxx
```

- Some more useful operations

```
a > b;          // case-sensitive comparison using dictionary ordering
a == b;         // equality check
a + b;          // concatenate a and b
a.empty();      // check if a is empty
a.size();       // returns the number of chars in a
a.length();     // same as a.size();
a[n];           // return char at position n
a.c_str();      // return c-style string (useful for old libraries)
```

Practical Session (4)

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Vectors

- In C++ you can almost forget about the classical C-array

```
int a[] = {1,2,3,4};
```

- Instead you should use the std::vector whenever possible

```
#include <vector>
std::vector<int> a = {1,2,3,4};
```

- Other ways of initializing a vector

```
std::vector<int> a;           // empty vector
std::vector<int> b(10);       // vector with 10 elements, each initialized to 0
std::vector<int> c(10, -1);   // vector with 10 elements, each initialized to -1
std::vector<int> d(c);        // copy vector c into d
std::vector<int> e = c;       // copy vector c into e
```

Vectors

- Useful operations

```cpp
std::vector<int> a;      // empty vector
a.push_back(1);          // a = {1}
a.push_back(2);          // a = {1,2}
a.push_back(3);          // a = {1,2,3}
std::cout << a[0] << std::endl;
std::cout << a.size() << std::endl;
std::cout << (a == std::vector<int>{1,2}) << std::endl;

for(auto& v : a){
    std::cout << v << " ";
}
std::cout << std::endl;
```

There are many more. You find them here:
https://de.cppreference.com/w/cpp/container/vector

Iterators

- Iterators are fancy pointers
- Initialization with auto is reasonable

```
std::vector<int> a {1,2,3};
std::vector<int>::iterator b = a.begin();
auto e = a.end();                          // same type as b
```

- They can be used to iterate through a container, i.e. std::vector

```
std::vector<int> a {1,2,3,4,5,6,7};

for(auto it = a.begin(); it != a.end(); ++it){
    std::cout << *it << " ";
}
std::cout << std::endl;

for(auto it = a.rbegin(); it != a.rend(); ++it){
    std::cout << *it << " ";
}
std::cout << std::endl;

for(auto it = a.begin(); it != a.begin() + a.size() / 2; ++it){
    std::cout << *it << " ";
}
std::cout << std::endl;
```

STL algorithms

- Iterators are especially useful in combination with the STL algorithms
- Some examples are
  - sort
  - find
  - copy

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int> a {1,2,3,4,5,6,7};

    auto it = std::find(a.begin(), a.end(), 3);

    if(it != a.end())
        std::cout << "Number found" << std::endl;
    else
        std::cout << "Number not found" << std::endl;
}
```

Typedef

- Variable declarations become very lengthy sometimes
- We can shorten down names using typedefs (C-style)

```
typedef std::vector<float> vecf;

int main()
{
    vecf a {1.f,2.f,3.f};
}
```

- The C++11 standard added another way to do exactly the same and actually even more!

```
using vecf = std::vector<float>;
```

- Use using instead of typedef
  - More intuitive (right to left assignment, not left to right)
  - Compatible to templates
- Do not overuse typedefs!

Practical Session (5)

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Expressions

Expression: Lowest level of computation in C++ program.

- Expressions generally apply an operator to one or more operands.
- Each expression yields a result.
- Expressions can be used as operands, so we can write compound expressions requiring the evaluation of multiple operators.
- Example: i+j

Operands: Values on which an expression operates

Result: Value or object obtained by evaluating an expression.

Operators

Operator: Symbol that determines what action an expression performs.

- C++ defines a set of operators and specifies how many operands each operator takes.
- C++ also defines the precedence and associativity of each operator
- Operators may be overloaded and applied to values of class type.

Meaning of an operator (what operation is performed and the type of the result) depends on the types of its operands.

Operators

- sizeof is actually an operator, telling you the size required to store a variable of this type

```
double a;
std::cout << sizeof a << std::endl;
std::cout << sizeof(int) << std::endl;
```

- The comma operator (,) is always evaluated left to right! It secures execution of the left operand before the right operand. The result of the left operand is discarded and the result of the right operand is passed on.

```
#include <iostream>
int main()
{
    int n = 1;
    int m = (++n, std::cout << "n = " << n << '\n', ++n, 2*n);
    std::cout << "m = " << (++m, m) << '\n';
}
```

I would recommend to avoid the comma operator. It confuses people.

Explicit Conversions

Cast: An explicit conversion.

static_cast: An explicit request for a type conversion that the compiler would do implicitly. Often used to override an implicit conversion that the compiler would otherwise perform.

Example: int i; double d = static_cast<double>(i) / 2;

const_cast: A cast that converts a const object to the corresponding nonconst type. Please, do not use that!

dynamic_cast: Used in combination with inheritance and run-time type identification.

Statements

- **expression statement**: An expression followed by a semicolon.
  - An expression statement causes the expression to be evaluated.
- Null statement: ;
- Declaration statement: int i;
- Compound statements (Blocks): { … }
  - A block is not terminated by a semicolon, except e.g. classes!
- Statement scope
  - Variables defined in a condition must be initialized
  - Example: for(int i; ;) {}

If Statement


```
if (val <= a)
  if (val == a)
    ++count;
else
  count = 1;
```

- **dangling else**: how to process nested if statements with more ifs than elses?
  - In C++, an else is always paired with the closest preceding unmatched if
  - Curly braces can be used to hide an inner if

switch Statement

- **switch statement**: Conditional execution statement that starts by evaluating the expression that follows the switch keyword.
  - Control passes to the labeled statement with a case label that matches the value of the expression.
  - Case labels must be constant integral expressions
  - If there is no matching label, execution either branches to the default label, if there is one, or falls out of the switch if there is no default label.
  - Execution continues across case boundaries until the end of the switch statement or a break is encountered
  - Comment if no break at the end!
  - Be careful with variable definitions inside a switch statement

If and switch Statements

break statement: Terminates the nearest enclosing loop or switch statement.

Execution transfers to the first statement following the terminated loop or switch.

case label: Integral constant value that follows the keyword case in a switch statement.

- No two case labels in the same switch statement may have the same value.
- If the value in the switch condition is equal to that in one of the case labels, control transfers to the first statement following the matched label.
- Execution continues from that point until a break is encountered or it flows off the end of the switch statement.

default label: The switch case label that matches any otherwise unmatched value computed in the switch condition.

While and for Statements

- **While Statement**
- **Do While Statement**
  - Always ends with a semicolon
- **For loop statement**
  - Parts of for header are init-statement, condition, expression: for (; ;)
  - Multiple definitions in for header: for (int i = 0, int j = 1; ;)
  - Range based for

Range-based for (C++11)

Range-based for: Control statement that iterates through a collection of values.

Form: *for (declaration : expression) statement*

Other Statements

continue statement: Terminates the current iteration of the nearest enclosing loop.

Execution transfers to the loop condition in a while or do or to the expression in the for header.

goto statement: Do not use!

Practical Session (6)

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Thank you for your Attention!

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT