C++ Programming

Lecture 1
Software Engineering Group

Philipp D. Schubert
Contents

1. More on data types
2. Expressions
3. Const & Constexpr
4. Statements
5. Control flow
6. Recap
More on data types: built-in arrays

- A variable can hold a value of a certain type
  - Example
    ```
    int i = 42;
    ```
- What if I need 10 integers for solving a task?
  ```
  int one = 1;
  int two = 2;
  ...
  ```
  and if I need 1000 integers or more?
- Use arrays
  - Built-in **static** arrays can store N objects of the same type
  - They are stored in one contiguous block of memory (one after another)

- **Static**: at compile time
- **Dynamic**: at runtime
More on data types: built-in arrays

- Create an array of 4 integers
  ```cpp
  int array[4];
  array[0] = 10;
  array[1] = 20;
  array[2] = 30;
  array[3] = 40;
  cout << array[0] << 'n';
  cout << array[3] << 'n';
  int number = array[2];
  ```
  - What does this print?

- Caution
  ```cpp
  Never ever try something like
  array[-3] = 12;
  or
  array[5] = 13;
  ```
  - If indices are “out-of-bounds” we have undefined behavior
  - At best
    - Program crashes
  - At worst
    - Program continues
    - Results are nonsense and you are not even aware of that

- Problems
  - An array does not know its size!
  - Increases probability for out of bounds!

- You wish to use `std::array` or `std::vector` instead! // next time!
Multi-dimensional arrays

- Arrays can have multiple dimensions
- Example a 2D array (which is a matrix)
  ```c
  int matrix[2][2];
  matrix[0][0] = 1;
  matrix[0][1] = 2;
  matrix[1][0] = 3
  matrix[1][1] = 4;
  int n = matrix[1][0]; // What is n's content?
  ```
- You can create arrays of arbitrary dimensions

- Analog to
  - \( \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \) , \( \text{matrix} \in \mathbb{Z}^{2 \times 2} \)
  - \( n \) is 3
Expressions

- “An expression is a sequence of operators and their operands, that specifies a computation. …”
- “… Expression evaluation may produce a result and may generate side-effects.” [en.cppreference.com]
- Operands can be variables or literals
- Operators

<table>
<thead>
<tr>
<th>Common operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>assignment</strong></td>
</tr>
<tr>
<td><strong>increment</strong></td>
</tr>
<tr>
<td><strong>decrement</strong></td>
</tr>
<tr>
<td><strong>arithmetic</strong></td>
</tr>
<tr>
<td><strong>logical</strong></td>
</tr>
<tr>
<td><strong>comparison</strong></td>
</tr>
<tr>
<td><strong>member access</strong></td>
</tr>
<tr>
<td><strong>other</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>symbol</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>assignment</td>
</tr>
<tr>
<td>a += b</td>
<td>increment</td>
</tr>
<tr>
<td>a -= b</td>
<td>decrement</td>
</tr>
<tr>
<td>a *= b</td>
<td>arithmetic</td>
</tr>
<tr>
<td>a /= b</td>
<td>logical</td>
</tr>
<tr>
<td>a %= b</td>
<td>comparison</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>member access</td>
</tr>
<tr>
<td>a ^ b</td>
<td>other</td>
</tr>
</tbody>
</table>

**Table taken from http://en.cppreference.com**
Expressions

- Examples: arithmetic, consider `int i = 5;`
  - `-i`
  - `i + 10`
  - `i - 5 * 2 * 2`
  - `6 * 6`
  - `--i`
  - `11 % i`

- Evaluates to
  - `-5`
  - `15`
  - `-15`
  - `36`
  - `4`
  - `1`

<table>
<thead>
<tr>
<th>Common operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
</tr>
<tr>
<td><code>a = b</code></td>
</tr>
<tr>
<td><code>a += b</code></td>
</tr>
<tr>
<td><code>a -= b</code></td>
</tr>
<tr>
<td><code>a /= b</code></td>
</tr>
<tr>
<td><code>a &amp;= b</code></td>
</tr>
<tr>
<td>`a</td>
</tr>
<tr>
<td><code>a &lt;&lt;= b</code></td>
</tr>
<tr>
<td><code>a &gt;&gt;= b</code></td>
</tr>
</tbody>
</table>

[Table taken from http://en.cppreference.com]
Expressions

- Examples: comparison, consider `int i = 5;`
  - `i == 5`
  - `i > 100`
  - `i <= 5`
  - `100 >= 99`

- Evaluates to
  - 1 or `true`
  - 0 or `false`
  - 1 or `true`
Expressions

- Examples: comparison & logic, consider `int i = 5;
  - !(i == 5)
  - (i > 100) || (i == 5)
  - (i <= 5) && (-10 <= 1)
  - false || true

- Evaluates to
  - `0` or `false`
  - `1` or `true`
  - `1` or `true`
  - `1` or `true`

[Table taken from http://en.cppreference.com]
Expressions

- Keep operators precedence in mind
- In doubt always use parentheses: (expr)
- expr then gets evaluated first

[Table taken from http://en.cppreference.com]
Operator = (assign) revisited

- = is the assignment operator
  - Not the mathematical equals (check for equality would be ==)

- Example
  ```c
  int value = 10;
  ```
  - In words: evaluate the expression on the right side and shove the result into the variable specified on the left hand side!
  ```c
  int other = 2 * 2 + 3; // after this assignment other stores the value 7
  ```

- An assignment has a “left-hand side” and a “right-hand side”
  - Lvalue and Rvalue
    - An lvalue is an address (variable, reference, or pointer)
    - An rvalue is an expression that can be evaluated (to a value)
Variables revisited: `const` qualifier

- Variables can be qualified with `const`.
- Do qualify constant variables with `const`!
- Examples

  ```
  const double PI = 3.1415926535; // ok: initialized at compile time
  const int fortytwo = 21 + 21;  // ok: initialized at compile time
  const double value = calculateValue(); // ok: initialized at run time
  const int i; // error: i is uninitialized const
  PI = 3; // error: PI is const
  fortytwo = 12; // error: fortytwo is const
  double a = PI * 2; // ok: PI is const
  cout << fortytwo << '\n'; // ok: fortytwo is const
  ```

- Constant variables can be read, but “never” written to after initialization.
- Use `const` as much as possible.
  - It will prevent you from making mistakes.
Calculating ahead of time: `constexpr` (at compile time)

- Use `constexpr` for constant expressions
- Variables can be `constexpr`
  ```cpp
  constexpr double d = 2.5*6.8+120;
  constexpr int i = 12*12;
  ```
- Functions can be `constexpr` as well // next lecture
- Note: `constexpr` produces constant values (\(d\) and \(i\) cannot be changed, \(d\) and \(i\) are `const`)
- Workflow when programming in C++
  1. compile source code to executable program
  2. run the executable
- Constant expressions are (may be) evaluated at compile time!
  - Effectively pre-computing
  - Leads to increased performance (but slows down compilation time)
- `constexpr` similar to `const` but may be evaluated at compile time
Statements

- "Statements are fragments of the C++ program that are executed in sequence. The body of any function is a sequence of statements." [en.cpp.reference.com]

- Example

  ```cpp
  int i = 2*3+10;   // this is a statement
  int j = 10;       // j is 10
  i = j;            // content of i is overwritten with j’s content
  cout << i << '\n'; // prints 10
  ```

- Note that `i = j;` overrides i’s content with whatever j’s content is
Mathematical formulas and functions

- Use `#include <cmath>` to include mathematical functions
  - `pow()`, `sqrt()`, `abs()`, `sin()`, `cos()`, ...
  - We will talk about functions in detail next time
  - For now just use them
    - What is the C++ equivalent to $x = \sqrt{2}$, $x \in \mathbb{R}$
      ```cpp
double x = sqrt(2);
```
    - What is the C++ equivalent to $y = \frac{1}{4}e^3$, $y \in \mathbb{R}$
      ```cpp
double y = 1/4*exp(3);
```
C++ includes the following types of statements

1. Expression statements  // e.g. n = n + 1;
2. Compound statements (blocks)  // next
3. Selection statements  // today
4. Iteration statements  // today
5. Jump statements  // e.g. return 0; in our main(), later on
6. Declaration statements  // e.g. int i;
7. Try blocks  // later on
8. Atomic and synchronized blocks  // later on
Compound Statements

- Compound statements or blocks are brace-enclosed sequences of statements

Example

```cpp
{  
    int i = 42;
    int j = i + 10;
}
```

Note something like this is possible

```cpp
int i = 1;
{  
    cout << i << '\n';
    int i = 2;
    cout << i << '\n';
    {  
        int i = 3;
        cout << i << '\n';
    }
}  
cout << i << '\n';
```
Scopes: { and }

- A variable can be defined multiple times with the same name (usually don’t do it)
- Each name that appears in a C++ program is only valid in some portion of the source code called its scope!

```cpp
{  
    int i = 42;  
    int j = i + 10;  
}
```

- If a variable goes out of scope it can no longer be accessed
- Example

```cpp
{  
    int i = 42;  
    // i can be used in this block (its scope)  
} // i goes out of scope at this point
i = 13;  // error: i can no longer be used
```
Statements

- C++ includes the following types of statements

1. Expression statements // e.g. n = n + 1;
2. Compound statements (blocks) // done!
3. Selection statements // next!
4. Iteration statements // today
5. Jump statements // e.g. `return 0;` in our main(), later on
6. Declaration statements // e.g. int i = 10;
7. Try blocks // later on
8. Atomic and synchronized blocks // later on
Selection statements aka control flow

- Just a bunch of statements in sequence is not powerful
  - How to express: “You pass if you got more than 50% in the exercises, otherwise you fail.”
  - We need conditional code execution
  - Three kinds of selection statements exist

- Selection statements or control flow constructs in C++ are
  - if ( condition ) statement
  - if ( condition ) statement else statement
  - switch ( condition ) statement
  - Note a statement can be a compound statement / block
  - A condition is an expression that can be evaluated to true or false
If statement

- if ( condition ) statement
  - Example
    ```cpp
    int i = 10;
    if (i < 100) {
        cout << "i is smaller than 100\n";
    }
    ```
- If statements allow to execute specific code depending on a condition!
- If only a single statement should be executed one can omit the braces { and }
  ```cpp
  int i = 10;
  if (i < 100)
      cout << "i is smaller than 100\n";
  ```
If statement with else branch

- if ( condition ) statement else statement

  Example

```cpp
int i = 10;
if (i < 100) {
    cout << "i is smaller than 100\n";
} else {
    cout << "i is bigger than 100\n";
}
```

- Braces not needed here: only one statement should be executed in each branch

```cpp
int i = 10;
if (i < 100)
    cout << "i is smaller than 100\n"; // the IF branch
else
    cout << "i is bigger than 100\n"; // the ELSE branch
```
If statement

- There might be more than two branches
  - Example

```cpp
int i = 3;
if (i == 1) {
    cout << "i is 1\n";
} else if (i == 2) {
    cout << "i is 2\n";
} else if (i == 3) {
    cout << "i is 3\n";
} else {
    cout << "i is something else\n";
}
```
Switch statement

- switch ( condition ) statement
- Similar to the if statement
- More convenient if many conditions to check
  - switch is optimized for this purpose
    switch ( expression ) {
      case expression:
        // branch
        break;
      
      ...  
      default:
        // default branch
        break;
    }
Switch statement

- Switch in action
- Example on the right
- What number will be printed?
- What is printed if c is 'X'

```cpp
char c = 'D';
switch (c) {
    case 'A':
        cout << 1 << '
';
        break;
    case 'B':
        cout << 2 << '
';
        break;
    case 'C':
        cout << 3 << '
';
        break;
    case 'D':
        cout << 4 << '
';
        break;
    default:
        cout << -1 << '
';
        break;
}
```
C++ includes the following types of statements:

1. Expression statements // e.g. n = n + 1;
2. Compound statements (blocks) // done!
3. Selection statements // done!
4. Iteration statements // next!
5. Jump statements // e.g. ´return 0;´ in our main(), later on
6. Declaration statements // e.g. int i = 10;
7. Try blocks // later on
8. Atomic and synchronized blocks // later on
The previous statements on their own are not powerful enough

Example calculate sum from 1 to 100!

\[ \text{int } i = 1 + 2 + 3 + \ldots + 100; \]

But if we want to sum from 1 to 10 or from 1 to 1000000?

What if your user can choose the upper end?

You cannot write an infinite number of programs up-frond!

Iteration statements or loop constructs in C++

while ( condition ) statement

do statement while ( expression);

for ( init-statement (optional); condition ("optional") ; expression (optional) ) statement

for ( for-range-decl : for-range-init) statement

Note a statement can be a compound statement / block
for loop

- Problem: sum up the numbers from 1 to 100!

```cpp
int sum = 1 + 2 + 3 + ... + 100;
```
```cpp
cout << "result: " << sum << \n';
```

- Better use a loop

- Structure of a for loop
  - for ( init-statement (optional);
    condition (optional) ;
    expression (optional) ) statement

```cpp
int sum = 0;
for (size_t i = 1; i <= 100; ++i) {
    sum += i; // means: sum = sum + i;
}
```

- Note: variables from outside the loop can be used inside! Same
  holds for loop counter!

- What is going on?
  1. i is initialized (only once)
  2. condition is checked
     1. If true
        1. execute loop body
        2. execute expression (usually increases
           loop counter), go to 2.
     2. If false
        1. skip the loop
**for loops**

- **Problem:** sum up the numbers from $1^2$ to $100^2$!
  ```cpp
  int sum = 0;
  for (size_t i = 1; i <= 100; ++i) {
    sum += i * i;
  }
  ```

- **Observe:** we can use the counter variable inside the loop!

- **Loops can have arbitrary steps**
  ```cpp
  int sum = 0;
  for (int i = 10; i < 4; i += 10) {
    sum += i;
  }
  cout << sum << ' \n';
  ```
Caution

- “Stupid is as stupid does.”
- What does this print?

```cpp
int sum = 0;
for (int i = 1; i < 3; ++i) {
    sum += i;
    --i;
}
cout << sum << 'n';
```
Another kind of for loop

- for loop
- for (init-statement (optional);
  Condition (optional);
  expression (optional)) statement

Example

```cpp
int sum = 0;
for (size_t i = 1; i <= 100; ++i) {
    sum += i;
}
```

- Ubiquitous

- range for loop (or range for)
- for (for-range-decl : for-range-init) statement

Example

```cpp
int sum = 0;
vector<int> vec = {1, 2, 3, 4, 5};
for (int i : vec) {
    sum += i;
}
```

- Useful when working on containers // later on!
- Detail: container has to implement
  - begin() and end() // later on!
While loops

- while loop
- while ( condition ) statement
- Example

```c
int sum = 0;
int i = 1;
while (i <= 100) {
    sum += i;
    i++;
}
```
- Rejecting while loop
While loops

- Same as for-loop: "Stupid is as stupid does."

```c
int i = 1;
while (i < 2) {
    cout << "not wise" << endl;
}
```

- One has to leave the loop at some point
- Condition has (usually) to be evaluated to false at some point
  - Sometimes a infinite loop is what you want
    - Infinite for loop
      ```c
      for (;;) { // do stuff }
      ```
    - Infinite while loop
      ```c
      while (true) { // do more stuff }
      ```
Another kind of while loop

- while loop
- while ( condition ) statement
- Example

```c
int sum = 0;
int i = 1;
while (i <= 100) {
    sum += i;
    i++;
}
```

- Rejecting while loop
- Body might not be executed

- do while loop
- do statement while ( expression);
- Example

```c
int sum = 0;
int i = 300;
do {
    sum += i;
    i++;
} while (i <= 100);
```

- Non-rejecting while loop!
- Body is executed at least once
4 basic loops

- Types of loops
  - For
  - Range for
  - While
  - Do while

- All loops are equivalent
  - Can be transformed into each other
  - Example on the right

- Use the most natural one for each situation!

```c
int i = 1;
int counter = 0;
do {
    i = i + 1;
    counter++;
} while (counter < 3);

int i2 = 1;
int counter2 = 0;
i2 = i2 + 1;
counter2++;
while (counter2 < 3) {
    i2 = i2 + 1;
    counter2++;
}
```
Breaking loops

- Loops can be broken
  - Use `break` keyword
  - Break leaves the loop it is used in
  - Example
    ```
    int i = 1;
    while (i > 0) {
        i += 1;
        break;
    }
    ```
Breaking loops

- Loops can be broken
  - Use `break` keyword
  - Break leaves the loop it is used in
  - Very useful when combined with an if statement
    - Example
      ```c
      int sensor_value;
      while (true) {
        // do measurements
        sensor_value = getSensorValue();
        if (sensor_value == 0) {
          break;
        }
      }
      // do other stuff
      ```
Jumping loop iterations

- Loop iterations can be jumped over
- Use `continue` keyword
  - Causes a jump to the end of loop body
  - Very useful when combined with an `if` statement
- Example
  ```cpp
  for (int i = 0; i < 10; i++)
  {
      if (i != 5) {
          continue;
      }
      cout << i << " ";
  }
  ```
- What will be printed?
  ```cpp
  0 1 2 3 4 6 7 8 9
  ```
- `break` would have landed on the other roof
A note on nesting

- You can nest loops and if statements

- Example nested loops

```c++
for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 5; ++j) {
        cout << "#";
    }
    cout << "\n";
}
```

- What does this code print?

```
#####
#####
#####
#####
#####
```
A note on nesting

- You can nest loops and if statements
- Example nested if statements

```cpp
int i = 15;
if (i >= 10) {
    if ((i % 5) == 0) {
        cout << "i is greater than 9 and dividable by 5\n";
    } else {
        cout << "i is greater than 9\n";
    }
} else {
    cout << "i is smaller than 10\n";
}
```

- What does this code print?
Algorithm and program

- With today's lecture you have a Turing-Complete language
  - That is, you can compute everything that a Turing-Machine can compute
    - That is, you can compute “everything” that is intuitively computable!
- [https://de.wikipedia.org/wiki/Alan_Turing](https://de.wikipedia.org/wiki/Alan_Turing)

- Algorithm versus program
  - An algorithm is a description of how to solve a problem
  - A program is an algorithm formulated for the computer
  - C++ programs are algorithms described in a bunch of statements

- You can now have the first tools to formulate algorithms in C++
Algorithms, Maths & C++

- You almost always can translate mathematics to C++
- How to obtain a solution for a given task?
- Usually:
  1. Start with a problem
  2. Abstract the problem & find an algorithm (a way of solving the problem)
  3. Formulate algorithm in mathematics
  4. Formulate mathematic algorithm in a programming language (e.g. C++)
  5. The resulting program then solves the problem
- I will try to make links between mathematics and C++ as often as possible
- Mathematics and computer science/programming are very similar
  - “Computer science is mathematics with electricity!”, Dirk Frettlöh
A fun example: calculating an integral

- Calculate $\int_{0}^{1} \frac{4}{1+x^2} \, dx$

- Assumption:
  - We don’t know how to calculate an antiderivative of $f(x) = \frac{4}{1+x^2}$

- Solution: use numerical integration 13 lines
  - Uses simple arithmetic
  - A computer is very fast at arithmetic

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    long double integral_val = 0.0;
    long double x = 0.0;
    const size_t N = 1000000;
    long double step_width =
        abs(0-1) / static_cast<long double>(N);
    for (size_t n = 0; n < N; ++n) {
        // evaluate function a point x
        integral_val += 4 / (1 + x * x);
        x += step_width;
    }
    integral_val /= N;
    cout << integral_val << '
';
    return 0;
}
```
Recap

- Built-in arrays
- Expressions
- Assignments
- Qualifiers
- Simple statements
- Mathematical formulas
- Scopes
- Statements
  - Selection: if and switch
  - Iteration: for and while
  - Nesting
- Algorithms, mathematics and computer science
Thank you for your attention

Questions?