Technical Slide

In this lesson

- Useful language features

In this lesson

- Useful language features
- Specific features and pitfalls of C++, Java and Python

In this lesson

- Useful language features
- Specific features and pitfalls of C++, Java and Python
- Pros and cons of languages

Arrays

Array

- Size is fixed

Arrays

Array

- Size is fixed
- Could take/set an element by index

Arrays

Array

- Size is fixed
- Could take/set an element by index
- This operation is really fast

Arrays

Array

- Size is fixed
- Could take/set an element by index
- This operation is really fast

Dynamic array (vector/list)

- Same as a usual array

Arrays

Array

- Size is fixed
- Could take/set an element by index
- This operation is really fast

Dynamic array (vector/list)

- Same as a usual array
- Size could be changed

Arrays

Array

- Size is fixed
- Could take/set an element by index
- This operation is really fast

Dynamic array (vector/list)

- Same as a usual array
- Size could be changed
- Could take twice as much space, as the total size of elements

String

- Array of characters + useful tools

String

- Array of characters + useful tools
- Concatenate, extract/find substring

String

- Array of characters + useful tools
- Concatenate, extract/find substring
- Split, trim (strip)

String

- Array of characters + useful tools
- Concatenate, extract/find substring
- Split, trim (strip)
- Convert to/from numbers

String

- Array of characters + useful tools
- Concatenate, extract/find substring
- Split, trim (strip)
- Convert to/from numbers
- Regular expressions

Array-like structures

- Bitset — an array of bits

Array-like structures

- Bitset — an array of bits
    - Each bit takes a bit in memory, not a byte as in an array of booleans

Array-like structures

- Bitset — an array of bits
  - Each bit takes a bit in memory, not a byte as in an array of booleans
  - Bits are addressed in an array of integers, as if they are concatenated

Array-like structures

- Bitset — an array of bits
  - Each bit takes a bit in memory, not a byte as in an array of booleans
  - Bits are addressed in an array of integers, as if they are concatenated
  - Could count ones, do bitwise and, or, xor, etc — in about $n/32$ int operations

Array-like structures

- Bitset — an array of bits
  - Each bit takes a bit in memory, not a byte as in an array of booleans
  - Bits are addressed in an array of integers, as if they are concatenated
  - Could count ones, do bitwise and, or, xor, etc — in about $n/32$ int operations
- Big integers — arbitrary-size integer numbers

Array-like structures

- Bitset — an array of bits
  - Each bit takes a bit in memory, not a byte as in an array of booleans
  - Bits are addressed in an array of integers, as if they are concatenated
  - Could count ones, do bitwise and, or, xor, etc — in about $n/32$ int operations
- Big integers — arbitrary-size integer numbers
- Big decimals — arbitrary-precision floating point numbers

  Big integer shifted by a power of 2

Queues

Queue

- Push to the back
- Take from the front

Queues

Queue

- Push to the back
- Take from the front

Stack

- Push to the front
- Take from the front

Queues

Queue

- Push to the back
- Take from the front

Stack

- Push to the front
- Take from the front

Deque

- Push to the front/back
- Take from the front/back
- Could be used as a queue/stack

Technical Slide

Set

Set

- Insert an element

Set

- Insert an element
- Check if some value is contained

Set

- Insert an element
- Check if some value is contained
- Ordered set — could binary search for a value
  "find the greatest value less than 10^9 in the set"

Set

- Insert an element
- Check if some value is contained
- Ordered set — could binary search for a value
  "find the greatest value less than 10^9 in the set"
- Unordered set (hash set): $O(1)$ per operation — hash table

Set

- Insert an element
- Check if some value is contained
- Ordered set — could binary search for a value
  "find the greatest value less than 10^9 in the set"
- Unordered set (hash set): $O(1)$ per operation — hash table
- Ordered set (tree set): $O(\log n)$ per operation — binary search tree

Set

- Insert an element
- Check if some value is contained
- Ordered set — could binary search for a value
  "find the greatest value less than 10^9 in the set"
- Unordered set (hash set): $O(1)$ per operation — hash table
- Ordered set (tree set): $O(\log n)$ per operation — binary search tree
- On practice, ordered set is only slightly slower

Set

- Insert an element
- Check if some value is contained
- Ordered set — could binary search for a value
  "find the greatest value less than 10^9 in the set"
- Unordered set (hash set): $O(1)$ per operation — hash table
- Ordered set (tree set): $O(\log n)$ per operation — binary search tree
- On practice, ordered set is only slightly slower
- In both, much slower operations than in an array
  And more space per element

Map (dict)

Map (dict)

- Associative array: any type could be used for keys

Map (dict)

- Associative array: any type could be used for keys
- Space proportional to the number of elements

Map (dict)

- Associative array: any type could be used for keys
- Space proportional to the number of elements
- Just a set of key-value pairs

Map (dict)

- Associative array: any type could be used for keys
- Space proportional to the number of elements
- Just a set of key-value pairs
- Unordered (hash map) and ordered (tree map)

Algorithms

Algorithms

- Sort — $O(n \log n)$
  Stability — order on equals

Algorithms

- Sort — $O(n \log n)$
  Stability — order on equals
- Binary search — $O(\log n)$

Algorithms

- Sort — $O(n \log n)$
  Stability — order on equals
- Binary search — $O(\log n)$
- Random numbers generator
  Start with some seed
  Generate next "random" number
  The sequence depends only on the seed
  Random integer in $[l, r)$
  Shuffle a sequence

Input/output

Input/output

- Read until the end of file

Input/output

- Read until the end of file
- Read whole lines

Input/output

- Read until the end of file
- Read whole lines
- Formatted output

Input/output

- Read until the end of file
- Read whole lines
- Formatted output
- Printing floating point numbers

Fast input/output

Fast input/output

- Individual operations on files are very slow

Fast input/output

- Individual operations on files are very slow
- Buffered: instead of writing characters as they come, accumulate them in a temporary array — buffer
  Write it to the file only when it's large enough

Fast input/output

- Individual operations on files are very slow
- Buffered: instead of writing characters as they come, accumulate them in a temporary array — buffer

  Write it to the file only when it's large enough
- Could also force to write the buffer — "flush"
  - Interactive problems
  - Debug output — confusing when it prints not where it's in the code

Technical Slide

Strings

Strings

- C way: arrays of `char`

Strings

- C way: arrays of `char`
- C++ way: `string`

Strings

- C way: arrays of `char`
- C++ way: `string`
- Fixed-size vs dynamic

Strings

- C way: arrays of `char`
- C++ way: `string`
- Fixed-size vs dynamic
- Functions (like `strcmp`, `strcat`) vs members

Strings

- C way: arrays of `char`
- C++ way: `string`
- Fixed-size vs dynamic
- Functions (like `strcmp`, `strcat`) vs members
- Slightly faster vs convenient

Strings

- C way: arrays of `char`
- C++ way: `string`
- Fixed-size vs dynamic
- Functions (like `strcmp`, `strcat`) vs members
- Slightly faster vs convenient
- `string` is used more often

Input/output

Input/output

- C way:

```
int a;
scanf("%d", &a);
print("%d", a);
```

Input/output

- C way:

```
int a;
scanf("%d", &a);
print("%d", a);
```

- Fast

Input/output

- C way:

```
int a;
scanf("%d", &a);
print("%d", a);
```

- Fast
- Powerful templates

Input/output

- C way:

```
int a;
scanf("%d", &a);
print("%d", a);
```

- Fast
- Powerful templates
- Dangerous: no type checks, just writes to the memory

Input/output

- C way:

```
int a;
scanf("%d", &a);
print("%d", a);
```

- Fast
- Powerful templates
- Dangerous: no type checks, just writes to the memory
- Templates may differ in different compilers

Input/output

- C++ way:

```
int a;
cin >> a;
cout << a;
```

Input/output

- C++ way:

```
int a;
cin >> a;
cout << a;
```

- More convenient on simple input/output

Input/output

- C++ way:

```
int a;
cin >> a;
cout << a;
```

- More convenient on simple input/output
- Slow by default:

Input/output

- C++ way:

```
int a;
cin >> a;
cout << a;
```

- More convenient on simple input/output
- Slow by default:
- Synchronises with `stdio`

  — turn off with

  `ios_base::sync_with_stdio(false);`

Input/output

- C++ way:

```
int a;
cin >> a;
cout << a;
```

- More convenient on simple input/output
- Slow by default:
- Synchronises with `stdio`
  — turn off with
  `ios_base::sync_with_stdio(false);`
- `cout` buffer flushes on each `cin`
  — turn off with `cin.tie(0);`

Input/output

- C++ way:

```
int a;
cin >> a;
cout << a;
```

- More convenient on simple input/output
- Slow by default:
- Synchronises with `stdio`
  — turn off with
  `ios_base::sync_with_stdio(false);`
- `cout` buffer flushes on each `cin`
  — turn off with `cin.tie(0);`
- `cout << endl;` flushes — use `cout << '\n';`

Input/output

- C++ way:

  ```
  int a;
  cin >> a;
  cout << a;
  ```
- More convenient on simple input/output
- Slow by default:
- Synchronises with `stdio`
  — turn off with
  `ios_base::sync_with_stdio(false);`
- `cout` buffer flushes on each `cin`
  — turn off with `cin.tie(0);`
- `cout << endl;` flushes — use `cout << '\n';`
- Just as fast as `printf`/`scanf`, if tuned correctly

Undefined behavior

Undefined behavior

- Incorrect array indices: negative or too large

```
int a[10];
a[-1] = a[10] = 0;
```

Undefined behavior

- Incorrect array indices: negative or too large

```
int a[10];
a[-1] = a[10] = 0;
```

- Using local variables before assignment

```
int a;
a++;
cout << a;
```

Undefined behavior

- Incorrect array indices: negative or too large

```
int a[10];
a[-1] = a[10] = 0;
```

- Using local variables before assignment

```
int a;
a++;
cout << a;
```

- Non-void functions without return

Undefined behavior

- Incorrect array indices: negative or too large

```
int a[10];
a[-1] = a[10] = 0;
```

- Using local variables before assignment

```
int a;
a++;
cout << a;
```

- Non-void functions without return
- Signed integer overflow

Undefined behavior

- Compiler may do anything: your program could crash, or work incorrectly, or even correctly
  It could depend on compiler version, flags, memory before running

Undefined behavior

- Compiler may do anything: your program could crash, or work incorrectly, or even correctly
  It could depend on compiler version, flags, memory before running
- Know common UBs

Undefined behavior

- Compiler may do anything: your program could crash, or work incorrectly, or even correctly
  It could depend on compiler version, flags, memory before running
- Know common UBs
- Some may be detected by compiler warnings: turn on as much as possible
  `g++ -Wall -Wextra ...`

Undefined behavior

- Compiler may do anything: your program could crash, or work incorrectly, or even correctly
  It could depend on compiler version, flags, memory before running
- Know common UBs
- Some may be detected by compiler warnings: turn on as much as possible
  `g++ -Wall -Wextra ...`
- Platform-dependent flags
  Sanitizing: memory issues
  Linking libs with pedantic implementations: e.g. `std::vector` which always checks indices

Other remarks

- `Segmentation fault` — use a debugger to find the exact place

Other remarks

- `Segmentation fault` — use a debugger to find the exact place
- Compilation errors — always start from the first

Other remarks

- `Segmentation fault` — use a debugger to find the exact place
- Compilation errors — always start from the first
- `using namespace std;` — shortens code, but occupies variable names

Other remarks

- `Segmentation fault` — use a debugger to find the exact place
- Compilation errors — always start from the first
- `using namespace std;` — shortens code, but occupies variable names
- Compiler differences — perfomance, variable size (`long double`), `scanf/printf` templates

Other remarks

- `Segmentation fault` — use a debugger to find the exact place
- Compilation errors — always start from the first
- `using namespace std;` — shortens code, but occupies variable names
- Compiler differences — perfomance, variable size (`long double`), `scanf`/`printf` templates
- Assigning/passing structures — $O(n)$ time! Use pointers or references where needed

Other remarks

- `Segmentation fault` — use a debugger to find the exact place
- Compilation errors — always start from the first
- `using namespace std;` — shortens code, but occupies variable names
- Compiler differences — perfomance, variable size (`long double`), `scanf`/`printf` templates
- Assigning/passing structures — $O(n)$ time! Use pointers or references where needed
- C++11 features

```
unordered_set
vector<int> a = {1, 2, 3};
for (auto x : a)
```

Technical Slide

1. Lesson 1: Language specifics
   - Video 1.1: Basic data structures
   - Video 1.2: Advanced data structures and I/O
   - Video 1.3: C++
   - Video 1.4: Java
   - Video 1.5: Python
   - Video 1.6: Comparing languages

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers
- `BufferedReader` is fast, but it only reads whole lines

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers
- `BufferedReader` is fast, but it only reads whole lines
- Pass lines to a `StringTokenizer`

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers
- `BufferedReader` is fast, but it only reads whole lines
- Pass lines to a `StringTokenizer`
- Parse numbers from tokens by e.g. `Integer.parseInt`

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers
- `BufferedReader` is fast, but it only reads whole lines
- Pass lines to a `StringTokenizer`
- Parse numbers from tokens by e.g. `Integer.parseInt`
- Put it in a separate class with `Scanner`-like methods, e.g. `nextInt()`

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers
- `BufferedReader` is fast, but it only reads whole lines
- Pass lines to a `StringTokenizer`
- Parse numbers from tokens by e.g. `Integer.parseInt`
- Put it in a separate class with `Scanner`-like methods, e.g. `nextInt()`
- Include in your template code, to not write every time

Input/Output

- `Scanner` is convenient, but very slow
  Only small inputs, less than 10 000 integers
- `BufferedReader` is fast, but it only reads whole lines
- Pass lines to a `StringTokenizer`
- Parse numbers from tokens by e.g. `Integer.parseInt`
- Put it in a separate class with `Scanner`-like methods, e.g. `nextInt()`
- Include in your template code, to not write every time
- For output, `PrintWriter` is fine

Collections

- Collections always store objects, not primitives

Collections

- Collections always store objects, not primitives
- Could use a primitive wrapper like `Integer`
  But `Integer` takes 16 bytes, not 4!

Collections

- Collections always store objects, not primitives
- Could use a primitive wrapper like `Integer`
  But `Integer` takes 16 bytes, not 4!
- Object overhead with collections
  `ArrayList<Integer>` — much worse than `int[]`

Collections

- Collections always store objects, not primitives
- Could use a primitive wrapper like `Integer`
  But `Integer` takes 16 bytes, not 4!
- Object overhead with collections
  `ArrayList<Integer>` — much worse than `int[]`
- Collections: unsynchronised and synchronised
  `ArrayList` vs `Vector`
  Use unsynchronised — optimised for a single thread

Strings

- `String` — immutable

Strings

- `String` — immutable
- Every operation produces a new object, so most are linear

Strings

- `String` — immutable
- Every operation produces a new object, so most are linear
- `s += 'a'` is also $O(n)$!

Strings

- `String` — immutable
- Every operation produces a new object, so most are linear
- `s += 'a'` is also $O(n)$!
- `StringBuilder` — special class for growing strings
  `append` method — $O(1)$
  Elements are `char` — no object overhead

Other remarks

- Size of any object — at least 8 bytes more than the size of fields

Other remarks

- Size of any object — at least 8 bytes more than the size of fields
- `Collections.sort` — merge sort: stable, always $O(n \log n)$

Other remarks

- Size of any object — at least 8 bytes more than the size of fields
- `Collections.sort` — merge sort: stable, always $O(n \log n)$
- `Arrays.sort` — a version of quick sort: unstable, faster on average, but could take n^2 on specific tests!
  Shuffle the array before sorting

Other remarks

- Size of any object — at least 8 bytes more than the size of fields
- `Collections.sort` — merge sort: stable, always $O(n \log n)$
- `Arrays.sort` — a version of quick sort: unstable, faster on average, but could take n^2 on specific tests!
  Shuffle the array before sorting
- Do not forget to clone

Technical Slide

1. Lesson 1: Language specifics
   - Video 1.1: Basic data structures
   - Video 1.2: Advanced data structures and I/O
   - Video 1.3: C++
   - Video 1.4: Java
   - Video 1.5: Python
   - Video 1.6: Comparing languages

Speed up

- Local variables are faster than global
  Local — `list`, global — `dict`

Speed up

- Local variables are faster than global
  Local — `list`, global — `dict`
- Put global code in a separate function, to not use global variables

```
def main():
    # write global code here

main()
```

Speed up

- Local variables are faster than global
  Local — `list`, global — `dict`
- Put global code in a separate function, to not use global variables

```
def main():
    # write global code here

main()
```

- Appends with + create new object, so linear time

```
s = s + 'a' + 'b'
a = a + [0]
```

  Use += or `append`

Speed up I/O

Speed up I/O

- Instead of `input` and `print` use file I/O — like `read` or `write`

Speed up I/O

- Instead of `input` and `print` use file I/O — like `read` or `write`
- Read and write all at once

```
sys.stdin.read()
sys.stdin.readlines()
sys.stdout.write(' '.join(map(str, a)))
```

Lists

- A lot of useful tools for lists: standard functions like `sum`, `min`, `join` and the module `itertools`

Lists

- A lot of useful tools for lists: standard functions like `sum`, `min`, `join` and the module `itertools`
- Not only shorten the code, but are also faster than `for`:

```
s = sum(a)

s = 0
for x in a:
    s += x
```

Additions are performed inside the C code of `sum`!

Other remarks

- Different versions of Python: Python 2 and Python 3
  In Python 2, `range(n)` creates a list, and `xrange(n)` — a generator, which is much faster
  In Python 3, `range(n)` — a generator

Other remarks

- Different versions of Python: Python 2 and Python 3
  In Python 2, `range(n)` creates a list, and `xrange(n)` — a generator, which is much faster
  In Python 3, `range(n)` — a generator
- On average, Python 2 is slightly faster

Other remarks

- Different versions of Python: Python 2 and Python 3
  In Python 2, `range(n)` creates a list, and `xrange(n)` — a generator, which is much faster
  In Python 3, `range(n)` — a generator
- On average, Python 2 is slightly faster
- PyPy — another interpretator — could be faster, especially PyPy 2

Other remarks

- Different versions of Python: Python 2 and Python 3
  In Python 2, `range(n)` creates a list, and `xrange(n)` — a generator, which is much faster
  In Python 3, `range(n)` — a generator
- On average, Python 2 is slightly faster
- PyPy — another interpretator — could be faster, especially PyPy 2
- Max depth of recursion is 1000 by default
  Use `sys.setrecursionlimit` to increase

Other remarks

- `eval` and `exec` help in some implementation problems

Other remarks

- `eval` and `exec` help in some implementation problems
- No compiler — no prior checks
  Test solutions even more carefully

Other remarks

- `eval` and `exec` help in some implementation problems
- No compiler — no prior checks
  Test solutions even more carefully
- No compile errors with compiler's message
  Everything — a runtime error

Other remarks

- `eval` and `exec` help in some implementation problems
- No compiler — no prior checks
  Test solutions even more carefully
- No compile errors with compiler's message
  Everything — a runtime error
- Do not forget to clone
  `b = a[:]` for lists
  `[[]] * n` — all sublists are the same one!
  `[[] for i in range(n)]` — correct

Technical Slide

1. Lesson 1: Language specifics
    - Video 1.1: Basic data structures
    - Video 1.2: Advanced data structures and I/O
    - Video 1.3: C++
    - Video 1.4: Java
    - Video 1.5: Python
    - Video 1.6: Comparing languages

C++

- Most popular language on competitions

C++

- Most popular language on competitions
- Very fast, decent standard library

C++

- Most popular language on competitions
- Very fast, decent standard library
- Undefined behavior situations and uninformative crashes may be hard to debug

Java

Java

- Still fast enough — only 1.5-2 times slower than C++, on average

Java

- Still fast enough — only 1.5-2 times slower than C++, on average
- Standard library in some cases overpowers that of C++, e.g. `BigInteger`

Java

- Still fast enough — only 1.5-2 times slower than C++, on average
- Standard library in some cases overpowers that of C++, e.g. `BigInteger`
- Need to implement fast reading

Java

- Still fast enough — only 1.5-2 times slower than C++, on average
- Standard library in some cases overpowers that of C++, e.g. `BigInteger`
- Need to implement fast reading
- More checks and limitations than in C++
  Informative `RuntimeException`

Java

- Still fast enough — only 1.5-2 times slower than C++, on average
- Standard library in some cases overpowers that of C++, e.g. `BigInteger`
- Need to implement fast reading
- More checks and limitations than in C++
  Informative `RuntimeException`
- Codes are longer

Python

- 10–100 times slower than C++ — some problems could not be solved at all

Python

- 10–100 times slower than C++ — some problems could not be solved at all
- Standard library lacks sorted set and bitset

Python

- 10–100 times slower than C++ — some problems could not be solved at all
- Standard library lacks sorted set and bitset
- More high-level, programs are shorter

Python

- 10–100 times slower than C++ — some problems could not be solved at all
- Standard library lacks sorted set and bitset
- More high-level, programs are shorter
- Useful where C++ is too cumbersome, or big integers are needed
  Implementation/math problems

Summary

- Learn to use standard library tools

Summary

- Learn to use standard library tools
- Know common pitfalls

Summary

- Learn to use standard library tools
- Know common pitfalls
- Choose language wisely