Programming languages for HPC
&
Basic concepts of C++

Programming languages for HPC

ELSEVIER

Science of Computer Programming

Volume 205, 1 May 2021, 102609

Ranking programming languages by energy efficiency

Rui Pereira [a b], Marco Couto [c b], Francisco Ribeiro [c b], Rui Rua [c b], Jácome Cunha [c b], João Paulo Fernandes [d], João Saraiva [c b]

Show more

+ Add to Mendeley   Share   Cite

https://doi.org/10.1016/j.scico.2021.102609

Get rights and content

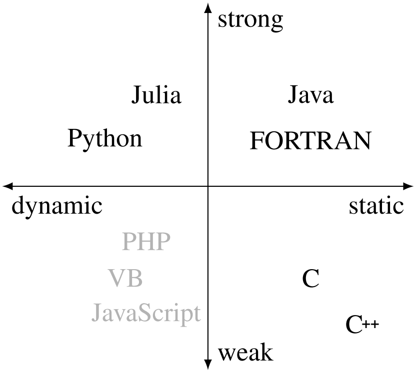
Benchmark of 27 programming languages using computer benchmark language game

binary-trees				
	Energy (J)	Time (ms)	Ratio (J/ms)	Mb
(c) C	39.80	1125	0.035	131
(c) C++	41.23	1129	0.037	132
(c) Rust $\Downarrow_2$	49.07	1263	0.039	180
(c) Fortran $\Uparrow_1$	69.82	2112	0.033	133
(c) Ada $\Downarrow_1$	95.02	2822	0.034	197
(c) Ocaml $\downarrow_1$ $\Uparrow_2$	100.74	3525	0.029	148
(v) Java $\uparrow_1$ $\Downarrow_{16}$	111.84	3306	0.034	1120
(v) Lisp $\downarrow_3$ $\Downarrow_3$	149.55	10570	0.014	373
(v) Racket $\downarrow_4$ $\Downarrow_6$	155.81	11261	0.014	467
(i) Hack $\uparrow_2$ $\Downarrow_9$	156.71	4497	0.035	502
(v) C# $\downarrow_1$ $\Downarrow_1$	189.74	10797	0.018	427
(v) F# $\downarrow_3$ $\Downarrow_1$	207.13	15637	0.013	432
(c) Pascal $\downarrow_3$ $\Uparrow_5$	214.64	16079	0.013	256
(c) Chapel $\uparrow_5$ $\Uparrow_4$	237.29	7265	0.033	335
(v) Erlang $\uparrow_5$ $\Uparrow_1$	266.14	7327	0.036	433
(c) Haskell $\uparrow_2$ $\Downarrow_2$	270.15	11582	0.023	494
(i) Dart $\downarrow_1$ $\Uparrow_1$	290.27	17197	0.017	475
(i) JavaScript $\downarrow_2$ $\Downarrow_4$	312.14	21349	0.015	916
(i) TypeScript $\downarrow_2$ $\Downarrow_2$	315.10	21686	0.015	915
(c) Go $\uparrow_3$ $\Uparrow_{13}$	636.71	16292	0.039	228
(i) Jruby $\uparrow_2$ $\Downarrow_3$	720.53	19276	0.037	1671
(i) Ruby $\Uparrow_5$	855.12	26634	0.032	482
(i) PHP $\Uparrow_3$	1,397.51	42316	0.033	786
(i) Python $\Uparrow_{15}$	1,793.46	45003	0.040	275
(i) Lua $\downarrow_1$	2,452.04	209217	0.012	1961
(i) Perl $\uparrow_1$	3,542.20	96097	0.037	2148
(c) Swift		n.e.		

Programming languages for HPC

From a developer perspective

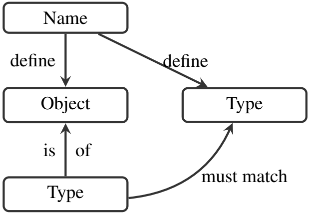
- Classify according to type checking
  - statically ↔ dynamically typed
    `int i = 3` ↔ `i = 3`
  - weakly ↔ strongly typed
    `int i = 3; string s = "a"; print i+s`
  - dynamic conversion possible if weakly typed



Programming languages for HPC

From a developer perspective

- Classify according to type checking
  - statically $\leftrightarrow$ dynamically typed
    `int i = 3` $\leftrightarrow$ `i = 3`
  - weakly $\leftrightarrow$ strongly typed
    `int i = 3; string s = "a"; print i+s`
  - dynamic conversion possible if weakly typed
- Static type checking
  - Protection from runtime errors
  - No runtime type deduction $\rightarrow$ faster computation
  - Example: Precomputed result-types in tensor calculus



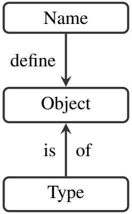
Stack/Memory view:

`00 00 00 01`	Address of `i`
`00 03 FF FF`	Value

Programming languages for HPC

From a developer perspective

- Classify according to type checking
  - statically $\leftrightarrow$ dynamically typed
    `int i = 3` $\leftrightarrow$ `i = 3`
  - weakly $\leftrightarrow$ strongly typed
    `int i = 3; string s = "a"; print i+s`
  - dynamic conversion possible if weakly typed
- Static type checking
  - Protection from runtime errors
  - No runtime type deduction $\rightarrow$ faster computation
  - Example: Precomputed result-types in tensor calculus
- Dynamic type checking requires RunTime Type Information (RTTI)
  - No compilation step, type deduction at runtime
  - Dynamic dispatchment, late binding, ...



Stack/Memory view:

`00 00 00 01` Address of `i`

`00 03 FF FF ···` Value?

Programming languages for HPC

Which language to learn? Let's formulate some criteria:

- General purpose language (no domain specific langauge)
- Need to produce highly efficient and portable programs
- Large software/library ecosystem
- Large supportive community maintaining language (so that it's unlikely it may vanish in the near future)
- Good starting point to learn further languages

Programming languages for HPC

Which language to learn? Let's formulate some criteria:

- General purpose language (no domain specific langauge)
- Need to produce highly efficient and portable programs
- Large software/library ecosystem
- Large supportive community maintaining language (so that it's unlikely it may vanish in the near future)
- Good starting point to learn further languages

> - Python or C++ are ideal candidates
> - C++ $\rightarrow$ Python easier than Python $\rightarrow$ C++
>
> $\Rightarrow$ Let's begin with C++!
>
> Examples: /project/cip/2023-SS-NQP/shared/example/cpp/lecture

C++: Basic concepts

Fundamental types

- `void`/`nullptr_t`: no valid type/invalid pointer type
- `bool`: 1 Bit representing boolean `True`/`False`
- `char` et al.: ASCII characters (or more for unicode support: `wchar_t`, `char16_t`, ...)
- `signed`/`unsigned int` et al.: Integer number with different ranges (`short`, `int`, `long`, `long long`), `signed` is default
- `float` et al.: Floating point number with single (`float`, 32 Bit), double (`double`, 64 Bit) or extended (`long double`, 80 Bit) precision

C++: Basic concepts

Fundamental types

- `void`/`nullptr_t`: no valid type/invalid pointer type
- `bool`: 1 Bit representing boolean `True`/`False`
- `char` et al.: ASCII characters (or more for unicode support: `wchar_t`, `char16_t`, ...)
- `signed`/`unsigned int` et al.: Integer number with different ranges (`short`, `int`, `long`, `long long`), `signed` is default
- `float` et al.: Floating point number with single (`float`, 32 Bit), double (`double`, 64 Bit) or extended (`long double`, 80 Bit) precision

Pointers/References

- For each type `T` there is a pointer type `T*` (can be `nullptr`)
- For each type `T` there is a reference type `T&` (must point to valid memory)
- For each type `T` there is a rvalue type `T&&` (only represents intermediate values or literals)

C++: Basic concepts

The holy trinity of `Const`'ness:

```
// value of i may change later
int i = 3;

// p is a constant pointer to an integer, the memory block p points to can't be
    changed via p
const int* p = &i;

// p is a constant pointer to a non-constant integer, the memory block p points to
    can be changed via p
int* const p = &i;

// p is a constant pointer to a constant memory block, neither p can be changed,
    nor the memory block via p
const int* const p = &i;
```

Note: Read `const`-definitions from right to left!

C++: Basic concepts

Operators: `Unary`, `Binary` and `Ternary`

- `Unary`, for instance
  - Arithmetic operation: `+=`, `-=`, `*=`, `/=`, `++`, `--`
  - Logical operations: `!`, `!=`
  - Bitwise operations: `~`, `~=`
- `Binary`, for instance
  - Arithmetic operation: `+`, `-`, `*`, `/`, `%`
  - Logical operations: `&`, `|`
  - Bitwise operations: `&`, `|`, `^`
  - Stream operations: `<<`, `>>`
- `Ternary`: `<condition>?<expr1>:<expr2>`
  - Execute `expr1` if `condition` evaluates to `True`
  - Execute `expr2` if `condition` evaluates to `False`

```
// int is signed 32bit integer!
// int i=0 then means i=0x0000FFFF
int i=0, j; j=(++i); // now j=1
int i=0, j; j~=i; // now j=-65536
```

```
int i=10, j; j=i%3; // now j=1
int i=7, j; j=i&2; // now j=2
```

```
int i=1, j;
j=(i>0)? 1: (i<0)? -1: 0; // implements
    sgn()
```

C++: Basic concepts

`Functions` and `Routines`

- General syntax of routines:
    - Return type or `void`
    - Routine identifier
    - Routine parameter

`main` function:

```
int main(int argc, char** argv[]);
```

- Must return `int`
- Must take one `int` parameter and one pointer to `char`-array

```
#include <iostream>

int arithmetic_sum(const int& _l, const int& _u) {
 int result = 0;
 for(int i = _l; i <= _u; i++) { result += i; }
 return result;
}

void output_arithmetic_sequence(const int& _l, const int& _u) {
 std::cout << "sum " << _l << " to " << _u << ": " << arithmetic_sum(_l, _u) << std::endl;
}
```

C++: Basic concepts

Flexible, static typing: `Templates`

- `function templates` provide automatic specializations of functions acting on different types `T`

```
#include <iostream>

template <typename T>
void print_sum( const T& _lhs , const T& _rhs) {
    std::cout << _lhs << "+" << _rhs << "=" << (_lhs + _rhs) << std::endl;
}
```

- `class templates` provide automatic specializations of different class types

```
template <typename T>
struct Complex { T real; T imag; };
```

C++: Basic concepts

`Template` specializations allow for compact type-dependent declarations

```
template <typename T> struct Complex; // forward declaration
template <typename T> struct TypeInfo { typedef T BaseType; };
template <> struct TypeInfo<Complex<float>> { typedef float BaseType; };
template <> struct TypeInfo<Complex<double>> { typedef double BaseType; };
```

Now we can define generalized norm function

```
#include <cmath>

template <typename T>
typename TypeInfo<Complex<T>>::BaseType norm(const Complex<T>& _value) {
    return std::sqrt((_value.real*_value.real)+(_value.imag*_value.imag));
}
```

This can be generalized even further introducing a function template

```
template <typename X>
typename TypeInfo<X>::BaseType norm(const X& _value);
```

C++: Basic concepts

Operator overloading for convenient arithmetics

- Unary operators:

```
template <typename T>
struct Complex {
    T real;
    T imag;
    Complex<T>& operator+=(const Complex<T>& _rhs) {
        this->real += _rhs.real; this->imag += _rhs.imag;
        return *this;
    } // implements z1 += z2; for Complex<T> z1,z2;
}
```

- Binary operators:

```
template <typename T>
Complex<T> operator+(const Complex<T>& _lhs, const Complex<T>& _rhs) {
    Complex<T> result(_lhs); result += _rhs;
    return result;
} // implements z3 = z1 + z2; for Complex<T> z1,z2,z3;
```

C++: Basic concepts

The `auto` keyword: Automatic type deduction

- Quite often the type of a variable can be inered from the interpreter, e.eg.:
  - In case of literals: `i = 10`, `i = 1.0`
  - In case of return types of functions: `z = foo()`

```
auto i = 1u; // defines i as unsigned int
auto j = -2; // defines j as signed int
auto f = 1.0/j; // defines f as double
```

- This is very helpful since in particular templates can render types rather confusing
- Also simplifies loops via ranged based accessors:

```
std::vector<T> list(10); // a 10-element vector of doubles
for(auto& el : list) {
 el = 2.0; // el is a reference so we fill vector with 2.0
}
```

C++: Basic concepts

Lambda expressions for in-place functor definitions

- In some situations objects representing functions (functors) are necessary
- Lambda expressions allow for compact definition of functors

```
auto cmp = [](const Complex<T> _lhs, const Complex<T>& _rhs)->bool {
    return norm(_lhs) < norm(_rhs);
};
```

- Functor `cmp` implements binary operator performing weak comparison and can be passed as argument to other functions

```
#include <algorithm>

template <typename T>
void weak_sort(std::vector<Complex<T>>& _list) {
    std::sort(_list.begin(), _list.end(), cmp);
};
```

C++: File types and compilation

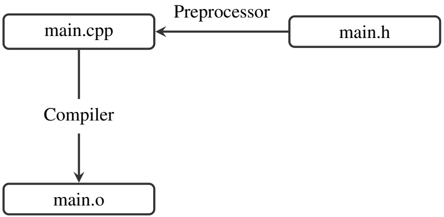
How do we convert source code into actual programs?

- Source code files
  - Source files with file ending ***.cpp** provide the implementation of our programm
  - Declarations can be outsourced into header files with file endings ***.h** or ***.hh**

C++: File types and compilation

How do we convert source code into actual programs?

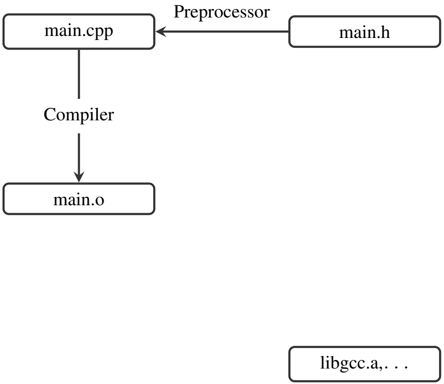
- Source code files
  - Source files with file ending ***.cpp** provide the implementation of our programm
  - Declarations can be outsourced into header files with file endings ***.h** or ***.hh**
- Object files with file endings ***.o** or ***.obj** contain compiled implementations in binary form



C++: File types and compilation

How do we convert source code into actual programs?

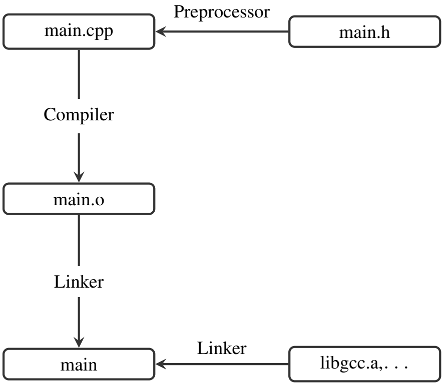
- Source code files
  - Source files with file ending ***.cpp** provide the implementation of our programm
  - Declarations can be outsourced into header files with file endings ***.h** or ***.hh**
- Object files with file endings ***.o** or ***.obj** contain compiled implementations in binary form
- Shared libraries files with file endings ***.so** or ***.a** are a collection of compiled objects (library)



C++: File types and compilation

How do we convert source code into actual programs?

- Source code files
  - Source files with file ending ***.cpp** provide the implementation of our programm
  - Declarations can be outsourced into header files with file endings ***.h** or ***.hh**
- Object files with file endings ***.o** or ***.obj** contain compiled implementations in binary form
- Shared libraries files with file endings ***.so** or ***.a** are a collection of compiled objects (library)
- Binary executables (typically no file-ending or ***.exe**) are programs that can be run by the operating system



C++: File types and compilation

- Preprocessor: Replace `#include <*>` statements with actual file contents
- Compiler: Create `*.o` file from preprocessed source file
- Both typically provided by compiler `g++` and executed in single call specifying `-c` option

```
sebastian.paeckel@cip-cl-compute2:~/2023-SS-NQP/shared/example/cpp/lecture/type_deduction$ /usr/bin/g++ -Wall -Wextra -Wpedantic -g3 -O0 -I./ -c main.cpp -o build/Debug/main
.o
sebastian.paeckel@cip-cl-compute2:~/2023-SS-NQP/shared/example/cpp/lecture/type_deduction$ ll build/Debug/
total 1
-rw-rw----+ 1 sebastian.paeckel ls-schollwoeck 254008 Apr 19 21:03 main.o
```

C++: File types and compilation

- Preprocessor: Replace `#include <*>` statements with actual file contents
- Compiler: Create `*.o` file from preprocessed source file
- Both typically provided by compiler `g++` and executed in single call specifying `-c` option

```
sebastian.paeckel@cip-cl-compute2:~/2023-SS-NQP/shared/example/cpp/lecture/type_deduction$ /usr/bin/g++ -Wall -Wextra -Wpedantic -g3 -O0 -I./ -c main.cpp -o build/Debug/main
.o
sebastian.paeckel@cip-cl-compute2:~/2023-SS-NQP/shared/example/cpp/lecture/type_deduction$ ll build/Debug/
total 1
-rw-rw----+ 1 sebastian.paeckel ls-schollwoeck 254008 Apr 19 21:03 main.o
```

- Linker: Link external libraries and object file into binary executable
- Typically provided by compiler `g++` and executed specifying `-o` option

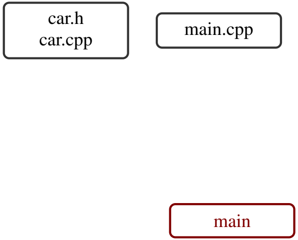
```
sebastian.paeckel@cip-cl-compute2:~/2023-SS-NQP/shared/example/cpp/lecture/type_deduction$ /usr/bin/g++ -Wall -Wextra -Wpedantic -g3 -O0 -I./ build/Debug/main.o -o build/Deb
ug/main
sebastian.paeckel@cip-cl-compute2:~/2023-SS-NQP/shared/example/cpp/lecture/type_deduction$ ll build/Debug/
total 1
-rwxrwx--x+ 1 sebastian.paeckel ls-schollwoeck 144864 Apr 19 21:06 main
-rw-rw----+ 1 sebastian.paeckel ls-schollwoeck 254008 Apr 19 21:05 main.o
```

- Use `-I` option to add directories to search path
- Use `-W` option to add directories to configure shown compiler warnings
- Use `-g`, `-O`, ... options to configure compiler optimization

C++: Project structure

- Preprocessor expands all `#include` directives recursively → large projects then generate large compiled code files
- Implemented functionality is often used in different contexts, independently

As a consequence, structure project by functionality

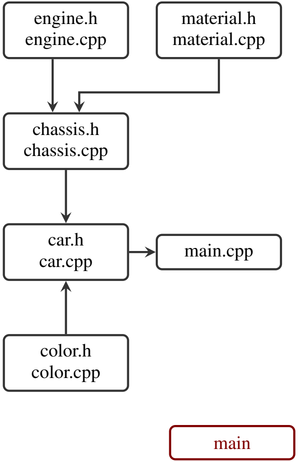


C++: Project structure

- Preprocessor expands all `#include` directives recursively $\rightarrow$ large projects then generate large compiled code files
- Implemented functionality is often used in different contexts, independently

As a consequence, structure project by functionality

- Avoid too many nested `#include` statements
- Implement independent functionalities in independent `*.cpp` files with associated headers `*.h` (always pairwise)
- Executables (`main`-functions) should only serve as user front end

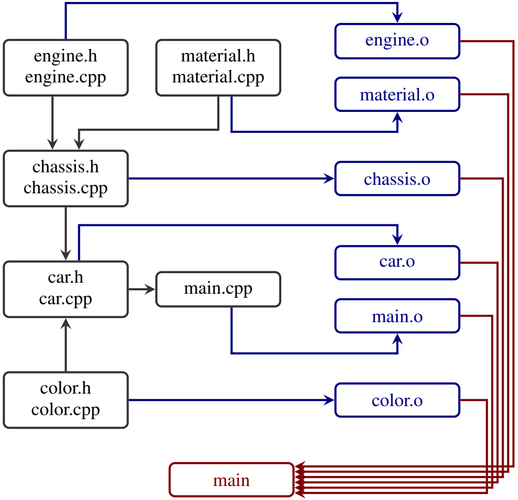


C++: Project structure

- Preprocessor expands all `#include` directives recursively $\rightarrow$ large projects then generate large compiled code files
- Implemented functionality is often used in different contexts, independently

As a consequence, structure project by functionality

- Avoid too many nested `#include` statements
- Implement independent functionalities in independent `*.cpp` files with associated headers `*.h` (always pairwise)
- Executables (`main`-functions) should only serve as user front end



C++: Compiling complex programs using `make`

- `Make` is a tool that executes (file-)operations based on dependencies
- `Make` establishes rules for targets (files that should be build) that need to fulfill certain dependencies
- If dependencies are missing or outdated, `Make` searches for rules to build them
- Compilation and linking chains are handled automatically

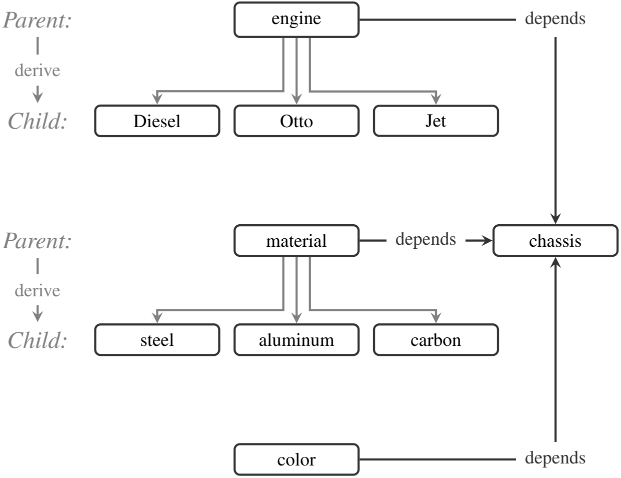
General syntax for a rule:

```
target: <dependency1> <dependency2> ...
   shell command that builds target from
        dependencies
```

```
# define compiler variable
CC = /usr/bin/g++

# define compiler flags
CPPFLAGS = -Wall -Wextra -Wpedantic -g3 -O0

# define depending objects
OBJS = color.o material.o engine.o chassis.o car.o

# define linker flags
LDFLAGS =

# include external definitions
include make.inc

# define rule for binary
main: main.o $(OBJS$
   $(CC) $(CPPFLAGS) $^ -o $@ $(LDFLAGS)

# define rule for object files
%.o: %.cpp
   $(CC) $(CPPFLAGS) -c $^ -o $@
```

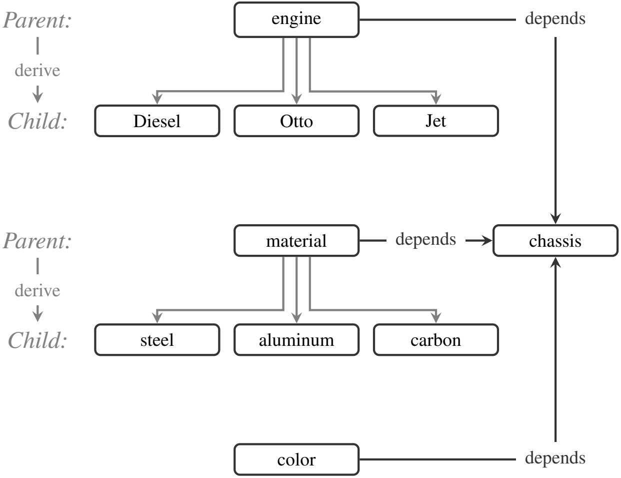
Object-Oriented Programming (OOP): Structuring complex code in C++

- Relationships between data structures:
  - Inclusive: Inheritance
  - Dependent: Attributes of certain types
- OOP: Organize code around contained data, not functionality



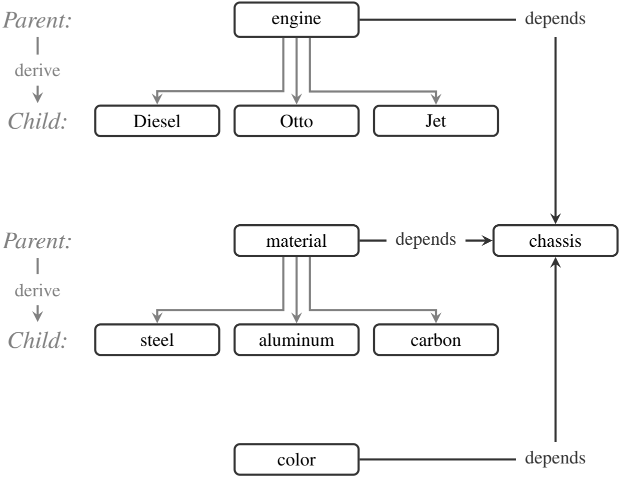
Object-Oriented Programming (OOP): Structuring complex code in C++

- Relationships between data structures:
  - Inclusive: Inheritance
  - Dependent: Attributes of certain types
- OOP: Organize code around contained data, not functionality
- Derived classes extend/specialize data



Object-Oriented Programming (OOP): Structuring complex code in C++

- Relationships between data structures:
  - Inclusive: Inheritance
  - Dependent: Attributes of certain types
- OOP: Organize code around contained data, not functionality
- Derived classes extend/specialize data
- Car: Inherit from chassis
  - Engine-type: Otto, Diesel
  - Material-type: Steel, aluminum, carbon
- Plane: Inherit from chassis
  - Engine-type: Diesel, Jet
  - Material-type: Aluminum, carbon



OOP in C++

```cpp
class Engine {
private: // not visible in derived classes, not accessible from instance
    unsigned int serial_id;

protected: // visible in derived classes, not accessible from instance
    std::string fuel;

public: // visible in derived classes, accessible from instance
    unsigned int next_maintenance;

    Engine(const unsigned int& _serial_id) // construtor
    : serial_id(_serial_id) {}; // init default values

    const std::string& get_fuel() const { return this->fuel; }
};

class Diesel : public Engine { // maintain visibility of parent class attributes
public:
    Diesel(const unsigned int& _serial_id) // override constructor
    : Engine(_serial_id) { this->fuel = "Diesel"; this->next_maintenance = 2*365 };
};
```