C++ PROGRAMMING

Lecture 3

Secure Software Engineering Group
Philipp Dominik Schubert

Fraunhofer IEM
HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
SECURE SOFTWARE ENGINEERING GROUP

CONTENTS

Note on this and the next lecture

- C++ can get complicated very quickly (and in this and the next lecture it will!)
  - Do not be frustrated
  - Understanding takes some time
  - “Complicated” mechanisms are the price for C++’s power
    - All those mechanism are cleverly designed
- Steps of learning new things

1. This is awesome!
2. This is tricky!
3. This is crap!
4. I am crap!
5. This might be okay!
6. This is awesome!

Union

- Store information that …
  - share the same memory
  - are alternatives to each other
  - has the size of its largest data member
- Only one member can be used at a time
  - You better know which one!
- Useful if memory is very limited

Union

- Example

```
union CID {
    char c;
    int i;
    double d;
};
```

- What size would CID be?
  - Size is 8 bytes (on most modern machines)
  - Check with `sizeof(CID)` when in doubt
- I never used a union in my life!
- **If possible, use `std::variant` instead**
  - `#include <variant>`
  - `std::variant` also stores what alternative is currently valid

- Usage

```
int main() {
    CID x;
    x.c = 'A';
    std::cout << x.c << '\n';
    x.i = 100;
    std::cout << x.i << '\n';
    x.d = 3.14;
    std::cout << x.d << '\n';
    // don't do that
    x.i = 123456789;
    std::cout << x.c << '\n'; // non-sense
    return 0;
}
```

Union – a better alternative

- Use `std::variant` instead

```
#include <iostream>
#include <variant>

int main() {
  std::variant<int, double> v = 42;
  v = 123.456;
  if (std::holds_alternative<double>(v)) {
    std::cout << "'v' stores a double with value: " << std::get<double>(v)
              << '\n';
  } else {
    std::cout << "'v' stores an int with value: " << std::get<int>(v) << '\n';
  }
  return 0;
}
```

Enum

- Used to store a bunch of states
  - A machine might be ´on´ or ´off´
  - A traffic light has colors ´green´, ´yellow´ and ´red´
  - How to store this in an understandable manner?

  - Example
    - How to model a machine that can be in state ´on´ or ´off´

      ```
      bool machine_state = true;
      ```
    - And if there are many states?

      ```
      int current_state = 21;
      ```
      - Not very meaningful nor readable

Enum

- `Enum` - enumerations allow for introducing meaningful states
- Machine state

```
enum MachineState { ON, OFF };
MachineState ms = ON;
MachineState other_machine = OFF;
```

- Meaningful and efficient
  - Compiler internally stores states as `int`
  - Compiler keeps track of `enum` members and corresponding `int` values
  - Compiler starts enumerating at 0, unless you tell it otherwise

Enum

```
enum MachineState { ON, OFF };
MachineState ms = ON;
MachineState other_machine = OFF;
```

- Compiler starts enumerating at 0, unless you tell otherwise

```
std::cout << ON << '\n';  // prints 0
std::cout << OFF << '\n';        // prints 1
```

- A traffic light might look like

```
enum TrafficLight { GREEN, YELLOW, RED };
```

Enum

- If compiler should use another enumeration
  - Use

    ```
    enum TrafficLight { GREEN=42, YELLOW, RED };
    ```

    - `GREEN` is `42` internally, `YELLOW` is `43` and `RED` is `44`

  - This is possible as well

    ```
    enum TrafficLight { GREEN=100, YELLOW=12, RED=4 };
    ```

  - Stick to the default unless you have reason to do otherwise

Enum

- Enumerations have one problem
  - Namespace pollution
  - Example

```
#include <vector>
using namespace std;
enum Types { vector, other };

int main() {
  vector<int> v(10);
  return 0;
}
```

  - Refrain from `using namespace`

- Error message (using g++)

```
pollution.cpp: In function 'int main()':
pollution.cpp:7:2: error: reference to 'vector' is ambiguous
  vector<int> v(10);
  ^
pollution.cpp:4:18: note: candidates are: Types vector
 enum Types { vector, other };
                  ^
In file included from /usr/include/c++/5/vector:64:0,
                 from pollution.cpp:1:
/usr/include/c++/5/bits/stl_vector.h:214:11: note: template<class _Tp, class _Alloc> class std::vector
     class vector : protected _Vector_base<_Tp, _Alloc>
           ^
pollution.cpp:7:9: error: expected primary-expression before 'int'
  vector<int> v(10);
         ^
```


HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

Enum

- There is a solution
  - Use `enum class` aka scoped enums
  - These enums are only visible in a certain scope
  - Provides type safety
  - Introduced in C++11

```
#include <iostream>
#include <vector>
using namespace std;

enum class Types { vector, other };

int main() {
  vector<int> v(10);
  // this vector lives in the
  // scope Types
  Types type = Types::vector;
  return 0;
}
```

Enum

- “Problem”
  - Due to type safety there is no implicit conversion to `int`

    ```
    std::cout << Types::vector << '\n';
    ```

  - You cannot print the states that easily
  - If you want to print a scoped enum use
    - C++11

      ```
      static_cast<typename underlying_type<Types>::type>(type)
      ```

    - C++14

      ```
      static_cast<underlying_type_t<Types>>(type)
      ```

Enum – insider information

```cpp
#include <iostream>
#include <string>
#include "llvm/ADT/StringSwitch.h"

enum class State {
#define STATE_DEF(NAME, TYPE) TYPE,
#include "enum-definition.def"
};

std::string toString(const State &S) {
  switch (S) {
  default:
#define STATE_DEF(NAME, TYPE) \
  case State::TYPE:           \
    return NAME;              \
    break;
#include "enum-definition.def"
  }
}
```

```cpp
State toState(const std::string &Str) {
  State S = llvm::StringSwitch<State>(Str)
#define STATE_DEF(NAME, TYPE) .Case(NAME, State::TYPE)
#include "enum-definition.def"
                .Default(State::Error);
  return S;
}
std::ostream &operator<<(std::ostream &OS, const State &D) {
  return OS << toString(D);
}
int main() {
  State S = State::A;
  State T = State::B;
  std::cout << "S's state is: " << S << '\n';
  State U = toState("C");
  State V = toState("Blah!");
  std::cout << "V's state is: " << V << '\n';
  if (S == T) {
    std::cout << "S and T carry the same state!\n";
  }
  return 0;
}
```

- Generate code at compile time based on a definition file

enum-definition.def

```cpp
#ifndef STATE_DEF
#define STATE_DEF(NAME, TYPE)
#endif
STATE_DEF("A", A)
STATE_DEF("B", B)
STATE_DEF("C", C)
STATE_DEF("D", D)
STATE_DEF("ERROR", Error)
#undef STATE_DEF
```

User defined / non-built-in types with `struct`

- `struct` lets you define your own data type
- Define a `struct` that stores information about a person

```
struct Person {
    std::string name;
    std::string surname;
    unsigned age;
};
```

1. A variable inside a `struct` is also called a data member, member variable or field
2. A function inside a `struct` is called a function member or member function
3. Data or functions inside a `struct` can be accessed with `.` (point operator)

```
Person peter;
peter.name = "Peter";
peter.surname = "Griffin";
peter.age = 41;
std::cout << peter.age << '\n';
```

User defined types with `struct`

```
struct Person {
    std::string name;
    std::string surname;
    unsigned age;
};
```

- Data inside a `struct` can be accessed with `.` (point operator)
  - This is tedious!
  - Users of Person might forget to initialize one of the data members
- Is there a more clever way to create a variable and initialize it?
  - Use constructors

- Create a variable of type Person and store some data in that variable

```
Person peter;
peter.name = "Peter";
peter.surname = "Griffin";
peter.age = 41;
```

Special member functions

```
struct Person {
    std::string name;
    std::string surname;
    unsigned age;
};
```

- Is there a more clever way of getting data into a variable of type person?
  - Again take a deep breath
  - Person already contains special member functions that you cannot see
  - If not defined by the user, the compiler generates them for you as required
    - This only works here because we are using built-in and STL data types (`std::string`/`unsigned`)

Special member functions

```
struct Person {
    std::string name;
    std::string surname;
    unsigned age;
};
```

- The special member functions are:
  - Constructor(s) // is executed when creating a variable, **there may be more than one ctor**
  - Destructor // is executed when object is no longer in use/out-of-scope (is destroyed)
  - Copy-constructor // is executed when object is copied (remember parameter passing)
  - Move-constructor // is executed when object is moved (remember returning data from function)
  - Copy-assignment-operator // is executed when object is copied via **=** (see copy constructor)
  - Move-assignment-operator // is executed when object is moved via **=** (see move constructor)

User defined types with `struct`

```
struct Person {
    std::string name;
    std::string surname;
    unsigned age;
};
```

- Why does it have to be so complicated?
  - Goal: make user-defined-types feel like built-in types to developers (e.g. default parameter passing: copy)
  - It will become clear in time!
  - C++ uses the RAII concept
    - “Resource acquisition is initialization”
    - When a variable of user-defined type is introduced, C++ has to ensure that …
      A. A concrete instance of that type will be created (acquire resources, e.g. memory)
      B. It will be initialized correctly

Constructor

- Writing a constructor

```
struct Person {
    Person(std::string n, std::string sn, unsigned a)
        : name(n), surname(sn), age(a) {
      std::cout << "ctor\n";
    }
    std::string name;
    std::string surname;
    unsigned age;
};
```

- A constructor’s name is the type’s name
- The following code fails now

```
    Person peter;          // there is no such constructor
```

- A variable of type person can now **only** be created via: `Person peter("Peter", "Griffin", 41);`

Constructor

```
Person peter("Peter", "Griffin", 41);
```

- This calls the constructor which does its job and initializes the data members
  - name
  - surname
  - age
- It also prints `"ctor"`
- Users of type Person cannot fail to initialize variables of that type correctly
  - That is exactly what we wanted!

Destructor

- Writing a destructor

```
struct Person {
    Person(std::string n, std::string sn, unsigned a)
          : name(n), surname(sn), age(a) {
          std::cout << "ctor\n";
    }
    ~Person() { std::cout << "dtor\n"; }
    std::string name;
    std::string surname;
    unsigned age;
};
```

- A destructor’s name is the `struct` name but starts with ~ “anti-constructor”
- The destructor does the clean up when the variable is no longer needed
  - Users of type Person cannot fail to clean up the data!

Ctor and dtor

- Now assume this program

```
int main() {
    Person peter("Peter", "Griffin", 41); // ctor called
    // do some stuff with peter
    return 0; // dtor is called here, because the variable goes out of scope!
}
```

- Constructor and destructor act as a universal “do” and “undo” mechanism!

Copy constructor

- Writing a copy constructor

```
struct Person {
  Person(std::string n, std::string sn, unsigned a)
      : name(n), surname(sn), age(a){
    std::cout << "ctor\n";
  }
  ~Person() { std::cout << "dtor\n"; }
  Person(const Person& p) = default;
  std::string name;
  std::string surname;
  unsigned age;
};
```

- Again: same name as the `struct` and receives one argument as shown on the left-hand side
- Because Person only contains value and STL data types, we don´t need to write a copy ourselves
  - Compiler knows how to copy such types
  - Omit a definition or better: mark as default
  - **This will change when we use dynamic memory allocation (next lecture)**

Copy constructor

- Peter can be copied!

```
void someFunction(Person p) {
  // do useful stuff;
  // dtor called for p
}
int main() {
  Person peter("Peter", "Griffin", 41); // ctor called
  Person clone(peter); // copy called
  someFunction(peter); // copy called
  // do some stuff with peter and clone
  return 0; // dtor is called for peter and for clone
}
```

Copy assignment operator

- Writing a copy-assignment operator

```
struct Person {
  Person(std::string n, std::string sn, unsigned a)
      : name(n), surname(sn), age(a) {
    std::cout << "ctor\n";
  }
  ~Person() { std::cout << "dtor\n"; }
  Person(const Person& p) = default;
  Person& operator= (const Person& p) = default;
  std::string name;
  std::string surname;
  unsigned age;
};
```

- The copy assignment operator receives one argument as shown on the left-hand side
- Because Person only contains value and STL data types, we don´t need to write a copy ourselves
  - Compiler knows how to copy such types
  - Just set it to default
  - **This will change when we work with dynamic memory allocation (next lecture)**

Copy assignment operator

- Now a Person can be copied via **=**

```
int main() {
  Person peter("Peter", "Griffin", 41); // ctor called
  Person chris("Chris", "Griffin", 15); // ctor called
  chris = peter;                         // copy assign called
  // chris now contains the same data as peter
  // do some other stuff
  return 0; // dtor is called for peter and for chris
}
```

Move constructor

- Writing a move constructor

```
struct Person {
  Person(std::string n, std::string sn, unsigned a)
      : name(n), surname(sn), age(a) {
    std::cout << "ctor\n";
  }
  ~Person() { std::cout << "dtor\n"; }
  Person(const Person& p) = default;
  Person& operator= (const Person& p) =default;
  Person(Person&& p) = default;
  std::string name;
  std::string surname;
  unsigned age;
};
```

- Move constructor’s name is `struct` name, receives one argument as shown on left-hand side
  - It receives a so-called rvalue reference!
    - A temporary value that has “no address”
    - `unsigned age = 42;`
    - 42 has no address, it is a temporary
- Because Person only contains value and STL data types, we don´t need to write a move ourselves
  - Compiler knows how to move such types
  - Just set it to default
  - **This will change when we work with dynamic memory allocation (next lecture**

Move constructor

- Now a Person can be move constructed

```
Person someFunction() { Person p("Some", "Guy", 30); return p; }
int main() {
  Person peter("Peter", "Griffin", 41); // ctor called
  Person chris(std::move(peter));       // move called
  // peter can't be used at this point any more!
  std::cout << chris.name << '\n';
  Person guy(someFunction()); // move called
  return 0; // dtor is called for peter, chris, and guy
}
```

- A person can now be moved
  - We steal it’s data!
  - Sometimes move can replace copy (e.g. when returning a value from a function)
    - This is will become important when user-defined-types use dynamic memory allocation
  - Almost no overhead (or even no overhead at all, if the compiler is smart enough)

Move assignment operator

- Writing a move assignment operator

```
struct Person {
  Person(string n, string sn, unsigned a)
      : name(n), surname(sn), age(a) {
    std::cout << "ctor\n";
  }
  ~Person() { std::cout << "dtor\n"; }
  Person(const Person& p) = default;
  Person& operator= (const Person& p) = default;
  Person(Person&& p) = default;
  Person& operator= (Person&& p) = default;
  std::string name;
  std::string surname;
  unsigned age;
};
```

- Just set it to default

Move assignment operator

- Now a Person can be moved using the assignment operator

```
int main() {
  Person peter("Peter", "Griffin", 41); // ctor called
  Person chris ("Chris", "Griffin", 14); // ctor called
  chris = std::move(peter); // move assignment called
  // peter can't be used at this point any more!
  std::cout << chris << '\n';
  return 0; // dtor is called for peter and chris
}
```

- A person can now be moved using the assignment operator

User defined types with `struct`

- Does one really have to bother with all those special member functions for such a simple `struct`?
  - **No!**
- We started with

```
struct Person {
  std::string name;
  std::string surname;
  unsigned age;
};
```

- Note: the compiler can generate all this constructor madness for POD (“plain old data”) types automatically
  - A POD is a `struct` or `class` that only contains built-in data types
    - Compiler knows how built-in (and STL types) have to be constructed, destructed, copied and moved!
  - **But:** All this will become necessary for types that use dynamic memory allocation

Our final types

- Make your wish for compiler generated constructors and assignments explicit!
  - You get an error message if the compiler can't do it

```
struct Person {
  std::string name;
  std::string surname;
  unsigned age;
  Person(std::string n, std::string sn, unsigned a) : name(n), surname(sn), age(a) {}
  ~Person() = default;
  Person(const Person& p) = default;
  Person& operator= (const Person&p) = default;
  Person(Person&& p) = default;
  Person& operator= (Person&& p) = default;
};
```

- Note: one can also delete certain special member functions!
- Use keyword `delete`

```
Person(const Person& p) = delete; // copy not allowed
```

- Note: since C++11 you can initialize built-in types like non-built-in types (constructor-like)!

```
Person p("Peter", "Griffin", 45);      int i(42);     double d(1.234);
Person p{"Peter", "Griffin", 45};      int i{42};     double d{1.234};
```

Class

- Remember `struct`
  - Structs store a bunch of data
    - Data members
    - Have special member functions
    - Can have further member functions
    - Members (data and functions) can be accessed via `.` (point operator)
    - **Important**
      - Users can access **all** members from the outside
      - Everything is public: data is interface

- Example

```
struct Vec3 {
  double x;
  double y;
  double z;
};

Vec3 v;
v.x = 1;
v.y = 2;
v.z = 3;
```

Class

- Remember `struct`
  - All members are public be default
  - But you can make them private nevertheless
    - Usually you **don't** want to that for structs!
- Example

```
struct Vec3 {
  double x;
  double y;
 private:
  double z;
};

Vec3 v;
v.x = 1;
v.y = 2;
v.z = 3; // error: x is private
```

Class

- **Classes allow separation of data and interface**
- Consider

```
struct Vec3 {
  double x;
  double y;
  double z;
};
```

and

```
class Vec3 {
 public:
  double x;
  double y;
  double z;
};
```

- Here there is no difference
- Exact same behavior
- Notice keyword `public`
- What other keyword might exist?
  - `private`
  - `protected // later on`

HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

Class

- **Classes allow separation of data and interface**
- Example

```
class Vec3 {
private:
  double x;
  double y;
  double z;
};
```

- Usage

```
Vec3 v;
v.x = 1; // error: x is declared
            private member
v.y = 1; // error: x is declared
            private member
v.z = 1; // error: x is declared
            private member
```

- How useful is that?
  - We locked ourselves out!

Class

- **Classes allow separation of data and interface**
- But wait, let's provide some functionality

```
class Vec3 {
 private:
  double x;
  double y;
  double z;
 public:
  constexpr Vec3(double x, double y, double z) : x(x), y(y), z(z) {}
  constexpr size_t size() { return 3; }
};
```

- Usage

```
Vec3 v(1.1, 2.2, 3.3)
size_t vssize = v.size();
```

- Now we can access Vec3's constructor
- And the member function `size()`
- Let's add some more functionality!

Class

- Provide some more functionality

```
class Vec3 {
 private:
  double x;
  double y;
  double z;
 public:
  constexpr Vec3() : x(0), y(0), z(0) {}
  constexpr Vec3(double x, double y, double z) : x(x), y(y), z(z) {}
  constexpr size_t size() { return 3; }
  constexpr double euclidean_length() { return std::sqrt(x * x + y * y + z * z); }
  friend std::ostream& operator<< (std::ostream& os, const Vec3& v) {
    return os << v.x << " " << v.y << " " << v.z;
  }
};
```

Class

```
class Vec3 {
 private:
  double x;
  double y;
  double z;
 public:
  Vec3() : x(0), y(0), z(0) {}
  Vec3(double x, double y, double z) : x(x), y(y), z(z) {}
  size_t size() { return 3; }
  double euclidean_length() {
    return sqrt(x * x + y * y + z * z);
  }
  friend std::ostream& operator<< (std::ostream& os, const Vec3& v) {
    return os << v.x << " " << v.y << " " << v.z;
  }
};
```

- Example usage of Vec3

```
int main() {
  Vec3 v(1,2,3);
  // print its data
  std::cout << v << '\n';
   // print its length
  std::cout << "euclidean_len: "
            << v.euclidean_length() << '\n';
  // print its size
  std::cout << "size: " << v.size() << '\n';
  return 0;
}
```

Class

- Struct
  - Data is interface
- Class
  - Distinction between data and interface
  - Data can only be manipulated through well defined interface!
  - Make user-defined types easy and safe to use
- **Only difference between struct and class is the default visibility**
  - struct is public by default
  - class is private by default

`Class` vs `Struct`

- If there is no difference, when to use what?
- Structs
  - Use structs for PODs (“plain old data”)
  - Use member functions as shorthands
  - For simple data types
  - E.g. modelling a point comprising two coordinates
    - There are not many ways how to misuse a simple point
- Classes
  - Use classes for non-PODs
  - More sophisticated data types
    - Modelling a mathematical vector with more complex operations
    - Graph types, etc.

How to organize a C++ project?

- C++ allows for separation of code into header and implementation files (unlike Java)
- For logical related code …
  - that is …
    1. a collection of functions designed for a specific purpose
    2. a user defined type (that may contains member functions) (`struct` or `class`)
  - put function declarations and / or type declarations in a header file (ending “`.h`”)
    - Do not forget the include guards
  - put the (member) function / global variable definitions in an implementation file (ending “`.cpp`”)
- This allows separate compilation of implementation files / modules!
  - A compiled implementation file / module results in an object file (ending “`.o`”)
    - Object files contain machine code, but may contain unresolved references (e.g. function calls)
- The linker links all object files, resolves all references and produces an executable program

How to organize a C++ project?

File: Vec3.h

```
#ifndef VEC3_H_
#define VEC3_H_
#include <iostream>

void freeFunction();

extern int value;

class Vec3 {
private:
  double x;
  double y;
  double z;
public:
  Vec3();
  Vec3(double,double,double);
  size_t size();
  double euclidean_length();
  friend std::ostream&
    operator<< (
      std::ostream& os,
      const Vec3& v);
};
#endif
```

File: main.cpp

```
#include <iostream>
#include "Vec3.h"

int main() {
  freeFunction();
  std::cout << value << '\n';
  Vec3 v(10, 20, 30);
  std::cout << v << '\n';
  std::cout << v.size() << '\n';
  std::cout <<
    v.veuclidean_length() << '\n';
  return 0;
}
```

- Each `.cpp` file can be compiled separately into an `.o` file
- Once all sources have been compiled, linker links all `.o` files (and external libraries) into an executable program

File: Vec3.cpp

```
#include "Vec3.h"
#include <iostream>

void freeFunction() {/* def */}

int value = 42;

Vec3::Vec3() : x(0), y(0), z(0) {};
Vec3::Vec3(double x,
           double y,
           double z)
         : x(x), y(y), z(z) {};

size_t Vec3::size() { return 3; }

double Vec3::euclidean_length() {
 return sqrt(x * x + y * y + z * z);
}

std::ostream& operator<< (
    std::ostream& os,
    const Vec3& v) {
 os << v.x <<" "<< v.y <<" "<< v.z;
}
```

How to organize a C++ project?

- Header files are only included but never compiled
- Implementation files are typically compiled and linked separately
- Option A (your homework)

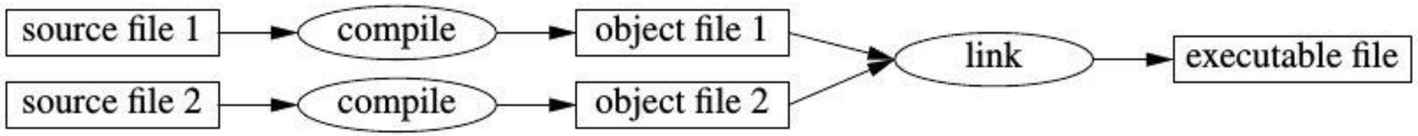
```
$ clang++ -std=c++17 –Wall –Wextra Vec3.cpp main.cpp –o main
```

- Option B (real C/C++ projects)

```
$ clang++ -std=c++17 –Wall –Wextra –c Vec3.cpp
$ clang++ -std=c++17 –Wall –Wextra –c main.cpp
$ clang++ -std=c++17 –Wall –Wextra Vec3.o main.o –o main
```

- Realistic projects would use build systems such as Makefile, CMake, etc.
  - Build systems allow developers to encode how a project has to be build

- **`Project/`**
  - **`main.cpp`**
  - **`Vec3.h`**
  - **`Vec3.cpp`**



Language-processing system revisited

- A few programs from this language-processing system (Linux)
  - `cpp` – the c preprocessor
  - `g++` or `clang++` – a C++ compiler
  - `as` – a assembler
  - `nm` – a tool to list symbols defined in object files
  - `ld` – a linker
- Usually a C++ compiler calls all those programs for you

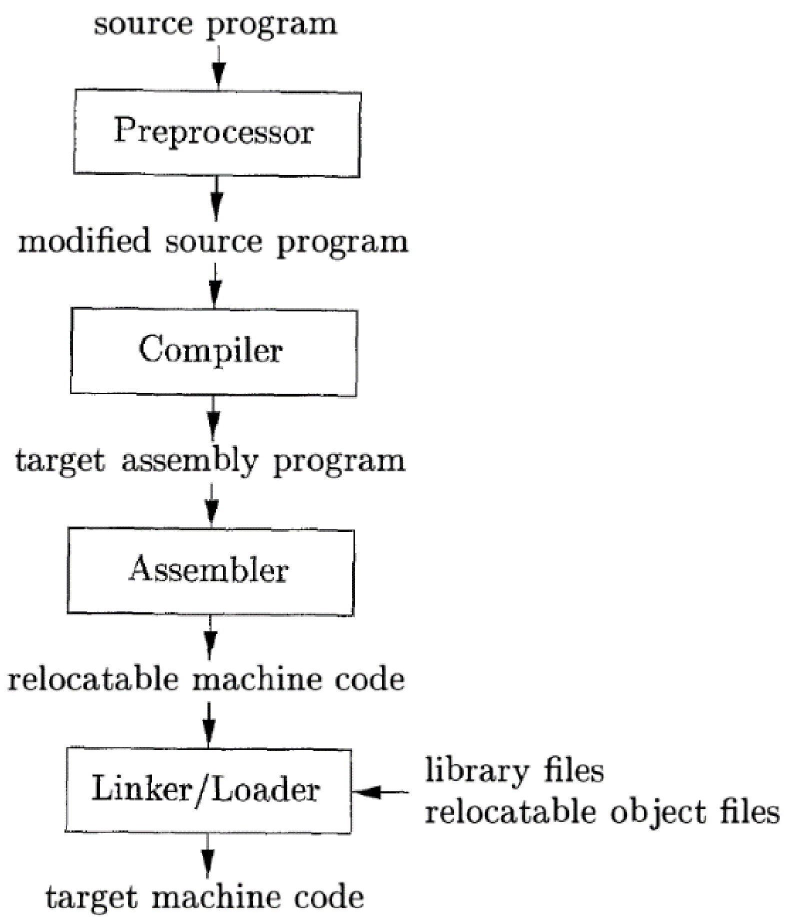


Figure 1.5: A language-processing system

More on C/C++ compiler toolchains

- Use tools to automatically improve your code
- “Everything in C++ is hard”
  - Even simple code formatting is hard (e.g. preprocessor macros → later on)
  - Powerful and clever tools are required
  - Clang/LLVM provides (AST-based) tools for managing large code bases
    - clang-format
      - formats code
      - format can be specified by a configuration file
    - clang-tidy
      - analysis and transformation tool
      - automatically improves and modernizes code
      - parameterized by a configuration file
    - and many more …

- I uploaded some exemplary project and configurations files and give some examples on how to use them in the exercise class(es)
- You are welcome to use those tools!

Namespaces

File: Vec3.h

```
#ifndef VEC3_H_
#define VEC3_H_
namespace first {
void foo();
} // namespace first

namespace second {
void foo();

class Vec3 {
private:
  double x;
  double y;
  double z;
public:
  Vec3();
  Vec3(double,double,double);
  constexpr size_t size();
  double euclidean_length();
};
} // namespace second
#endif
```

- Avoid name clashes
- Please refrain from using `using namespace std;`
- A namespace can be defined in several parts of a project
- Namespaces can be nested

File: main.cpp

```
#include <iostream>
#include "Vec3.h"

int main() {
  frist::foo();
  second::foo();
  second::Vec3 v(10, 20, 30);
  std::cout << v.size() << '\n';
  return 0;
}
```

File: Vec3.cpp

```
#include <iostream>
#include "Vec3.h"

namespace first {
void foo(){ std::cout << "first"; }
} // namespace first

namespace second {
void foo(){ std::cout << "second";}

Vec3::Vec3() {};
Vec3::Vec3(double x,
           double y,
           double z)
          : x(x), y(y), z(z) {};

size_t Vec3::size() { return 3; }

double Vec3::euclidean_length() {
 return sqrt(x * x + y * y + z * z);
}
} // namespace second
```

Recap

- Union
- Enum and enum class
- Struct
- Special member functions
- Class
- Struct versus Class
- How to organize a C++ project
- Language-processing system revisited
- Namespaces

Thank you for your attention
Questions?