Contents

1. Unions
2. Enumerations
3. Structures
4. Classes
5. Organizing a C++ project
Note on this and next lecture

- C++ can get complicated very quickly (and in this and next lecture 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!

Developers at the beginning of a project. vs. Developers at the end of a project.
Union

- Store information that
  - share the same memory
  - are alternatives to each other
  - have the size of the biggest data member
  - only one member can be used at a time
    - you better know which one!
  - is interpreted in different ways
- Useful if memory is very limited
Union

- Example

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

- What size would CID be?
  - Size is 8 bytes
  - Check with `sizeof(CID)` if in doubt

- I never used a single union!

- If possible, use `std::variant`
  - `#include <variant>`
  - `std::variant` stores what alternative is currently valid

- Usage

```c
int main() {
    CID x;
    x.c = 'A';
    cout << x.c << '\n';
    x.i = 100;
    cout << x.i << '\n';
    x.d = 3.14;
    cout << x.d << '\n';
    // don't do that
    x.i = 123456789;
    cout << x.c << '\n'; // this is non-sense
    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 a 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 meaningful!
Enum

- Enum - enumerations allow introduction of meaningful states

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

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

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

- **Compiler start enumerating at 0, unless you tell otherwise**
  ```cpp
  cout << ON << '\n'; // prints 0
  cout << OFF << '\n'; // prints 1
  ```

- **A traffic light might look like**
  ```cpp
  enum TrafficLight { GREEN, YELLOW, RED };
  ```
**Enum**

- If compiler should use another enumeration
  - Use
    ```java
    enum TrafficLight { GREEN=42, YELLOW, RED }
    ```
  - **GREEN is 42 internally, YELLOW is 43 and RED is 44**

- This is possible as well
  ```java
  enum TrafficLight { GREEN=100, YELLOW=12, RED=4 }
  ```

- Unless you have reason to, stick to the default
## Enum

- Enums have one problem
  - Namespace pollution
- Example

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

enum Types { vector, other };

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

- Error message (using g++)

```cpp
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: SomeTypes vector
    enum SomeTypes { 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);
    ^
```
# 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

```cpp
#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

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

- You cannot print the states that easy

- If you want to print a scoped enum use

  - C++11
    ```cpp
    static_cast<
        typename underlying_type<Types>::type>
    (type)
    ```

  - C++14
    ```cpp
    static_cast<
        underlying_type_t<Types>>(type)
    ```
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

```c
struct Person {
    string name;
    string surname;
    unsigned age;
};
```

A. A variable inside a **struct** is also called a data member, member variable or field

B. A function inside a **struct** is called a function member or member function

C. Data or functions inside a **struct** can be accessed with . (point operator)

```c
Person peter;
peter.name = "Peter";
peter.surname = "Griffin";
peter.age = 41;
cout << peter.age << endl;
```
User defined types with **struct**

```cpp
struct Person {
    string name;
    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 members
- Isn’t there a more clever way create a variable and get data into it?
  - There is! Constructors

- Create a variable of type `Person` & put data in it
  ```cpp
  Person peter;
peter.name = "Peter";
peter.surname = "Griffin";
peter.age = 41;
  ```
Special member functions

```c++
struct Person {
    string name;
    string surname;
    unsigned age;
};
```

- Isn´t there a more clever way to get data into a person?
  - Again take a deep breath
  - Person already contains special member functions that you cannot see
  - If not defined by the user, the compiler defines them for you as required
    - This only works here because we are using built-in & STL data types (string / unsigned)
Special member functions

```cpp
struct Person {
    string name;
    string surname;
    unsigned age;
};
```

- The special member functions are:
  - Constructor(s) // is executed when creating a variable, **there might be more than one ctor**
  - Destructor // is executed when object is no longer in use (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`

```c
struct Person {
    string name;
    string surname;
    unsigned age;
};
```

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

- Writing a constructor
  ```cpp
  struct Person {
      Person(string n, string sn, unsigned a)
          : name(n), surname(sn), age(a) {
          cout << "ctor\n";
      }
      string name;
      string surname;
      unsigned age;
  };
  ```
- A constructor’s name is the `struct` name
- The following code fails now
  ```cpp
  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 his job and initialized the data members
  - name
  - surname
  - age
- It also prints "ctor"
- Users of type Person cannot fail to initialize correctly
  - That is what we wanted!
Destructor

- Writing a destructor

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

- A destructor’s name is the `struct` name but starts with ~ (anti-constructors)
- 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 & dtor

- Now assume this program

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

- Constructor and destructor is a universal “do” and “undo” mechanism!
Copy constructor

- Writing a copy constructor

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

    ~Person() { cout << "dtor" << endl; }

    Person(const Person& p) = default;

    string name;
    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
  - Compiler knows how to copy such types
  - Omit a definition or better: set it to default
  - This will change when we work with dynamic memory allocation (next lecture)
Copy constructor

Peter can be copied!

```cpp
void someFunction(Person p) { /* do useful stuff */ }

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

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

~Person() { cout << "dtor" << endl; }

Person(const Person& p) = default;

Person& operator=(const Person& p) = default;

string name;
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
  - 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 =

```c
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 does
    // do some other stuff
    return 0; // dtor is called for peter and for chris
}
```
Move constructor

Writing a move constructor

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

Person(Person&& p) = default;

string name;
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

- 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
  
  ```cpp
  Person someFunction() { Person p("Some", "Guy", 30); return p; }
  int main() {
    Person peter("Peter", "Griffin", 41); // ctor called
    Person chris(move(peter)); // move called
    // peter can't be used at this point any more!
    cout << chris << endl;
    Person guy(someFunction()); // move called
    return 0; // dtor is called for peter and for chris
  }
  
  - A person can now be moved
    - We steal it´s data!
    - Sometimes move can replace copy (e.g. return a value from a function)
      - This is important when user-defined-types use dynamic memory allocation
    - Almost no overhead (or even no overhead at all if the compiler is smart)
Move assignment operator

- Writing a move assignment operator
  - Just set it to default

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

    ~Person() { cout << "dtor" << endl; }

    Person(const Person& p) = default;
    Person& operator=(const Person& p) = default;
    Person(Person&& p) = default;
    Person& operator=(const Person&& p) = default;

    string name;
    string surname;
    unsigned age;
};
```
Move assignment operator

- Now a Person can be moved using the assignment operator

```cpp
int main() {
    Person peter("Peter", "Griffin", 41); // ctor called
    Person chris = move(peter); // move called
    // peter can't be used at this point any more!
    cout << chris << endl;
    return 0; // dtor is called
}
```

- A person can now be moved via assignment operator
User defined types with `struct`

- Does one really have to bother with all those special member function madness for such a simple `struct`?
  - NO!
- We started with
  ```
  struct Person {
    string name;
    string surname;
    unsigned age;
  };
  ```
- Note the compiler can generate all this constructor madness for POD ("plain old data") objects automatically
  - A POD is a `struct` or `class` that only contains built-in or STL data types
    - Compiler knows how STL types have to be constructed, destructed, copied and moved!
  - **BUT:** All this will be necessary for types that use dynamic memory allocation
    - You have to know for non-POD types
Recommendation

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

- Finally Person would look like

```cpp
struct Person {
    string name;
    string surname;
    unsigned age;
    Person(string n, 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: since C++11 you can initialize built-in types like non-built-in types (constructor-like)!

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

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

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

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

Example

```c
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 for structs!

Example

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

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

- Classes allow separation of data and interface
- Consider

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

  and

  ```c
  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` // which we ignore for the moment
**Class**

- **Classes allow separation of data and interface**
- **Example**
  ```cpp
  class Vec3 {
    private:
      double x;
      double y;
      double z;
  };
  ```
- **Usage**
  ```cpp
  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

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

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

Usage

```cpp
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

```cpp
class Vec3 {
private:
    double x;
    double y;
    double z;
public:
    Vec3() : x(0), y(0), z(0) {} // provide a default ctor
    Vec3(double x, double y, double z) : x(x), y(y), z(z) {} // provide ctor
    constexpr size_t size() { return 3; } // provide a function that tells us the size
    double euclidean_length() { return sqrt(x * x + y * y + z * z); } // Vec3 length
    friend ostream& operator<<(ostream& os, const Vec3& v) { // Overload shift op
        return os << v.x << " " << v.y << " " << v.z;
    }
};
```
Class

- Provide some more functionality

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) {}
    constexpr size_t size() { return 3; }
    double euclidean_length() {
        return sqrt(x*x+y*y+z*z);
    }
    friend ostream& operator<<(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
    cout << v << '\n';
    // print its length
    cout << "euclidean_len: " << v.euclidean_length() << '\n';
    // print its size
    cout << "size: " << v.size() << '\n';
    return 0;
}
Class

- Struct
  - Data is interface

- Class
  - Distinction between data and interface
  - Data can only be manipulated through a well defined interface!
  - Make user-defined types easy and safe to use

**Only difference between struct and class is the default visibility**
- struct’s default is public
- class’s default is private
Class VS Struct

- If there is no difference, when to use what?

- Structs
  - Use structs for PODs (“plain old data”)
  - Use struct’s member functions as shorthands
  - For simple data types
  - E.g. modelling a point containing two coordinates
    - There are not many ways how to misuse a simple point

- Classes
  - Use classes for non-PODs
  - More sophisticated data types
  - E.g. modelling a mathematical vector with more complex operations defined on it
How to organize a C++ project?

- C++ allows for separation of code into header and implementation files (unlike Java)
- For logically 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 definitions into 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 file
How to organize a C++ project?

Each .cpp file can be compiled separately into .o.

Once all sources have been compiled, linker links all .o files (and external libraries) into the executable program.
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 from object files
- **ld** – a linker

Usually a C++ compiler calls all those programs for you

![Diagram of a language-processing system](image_url)

**Figure 1.5: A language-processing system**
Recap

- Union
- Enum & enum classes
- Structs
- Special member functions
- Classes
- Struct versus Class
- How to organize a C++ project
- Language-processing system revisited
Thank you for your attention

Questions?