ECE 449/590 – OOP and Machine Learning
Lecture 10 Class Invariant

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 26, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 9
- Next lecture: Accelerated C++ 9

Outline

(Simple) Class Design Overview

- Class types group data and functions together.
- Data members
  - Holding data for the objects of the class type.
  - Usually `private` for encapsulation.
- Member functions
  - Define valid operations available to the class type.
  - Constructors are special member functions that initialize objects.
- What language features are available to help us define more complicated class types?

Class Design for Calendar Dates

```
class date {
    int year_, month_, day_;
public:
    date(int y, int m, int d);

    bool set(int y, int m, int d);

    int get_year();
    int get_month();
    int get_day();

    std::string to_string();
}; // class date
```

- An intuitive and typical class design.
  - A ctor(constructor) to initialize `date` objects so that data members won't have undefined values.
  - Setter and getters.
  - Helper functions like `to_string` for printing and troubleshooting so that one don't have to use getters extensively.

Using Constructor

```
date first(2021, 1, 1);
std::cout << first.to_string() << std::endl;
date someday; // compiling error
```

- You can provide year/month/day to construct a `date` object.
  - The ctor is called implicitly and automatically.
- You have to provide them to construct any `date` object.
  - It is guaranteed by compiler that there is no undefined behavior because of members not initialized.

Multiple Constructors

```
class date {
public:
    date(int y, int m, int d);
    date(std::string str);
    ...
}; // class date

date first(2021, 1, 1);
date first_from_str("2021/1/1");
```

- A class can have multiple ctors.
  - They should have different parameters types – so the compiler can decide which one to call given the arguments.

Default Constructor

```
class date {
public:
    date();
    ...
}; // class date

date::date() : year_(1970), month_(1), day_(1) {
}

date epoch; // epoch contains 1970/1/1 instead of undefined values
```

- Default constructor: a ctor takes no arguments.
- Default-initialization: constructing an object without providing any arguments
  - Don't put `()` after `epoch`.
- Default ctor is called automatically for default-initialization.

Constness

```cpp
date first(2021, 1, 1);

const date const_first = first;

std::cout << first.to_string() << std::endl;
std::cout << const_first.to_string() << std::endl; // compiling error
```

- We cannot change `const_first`.
- Although we won't change `const_first` in the member function `to_string`, the compiler doesn't know that and will complain if we call it.
  - We need to tell the compiler so.

Handle Constness in Member Functions

```
class date {
public:
    ...
    int get_year() const;
    int get_month() const;
    int get_day() const;
    std::string to_string() const;
}; // class date

std::string date::to_string() const {
    ...
}
```

- You create a `const` member function by adding `const` after its parameter list.
  - The compiler will complain for any modification to the object in `const` member functions.
- For `const` objects, you can only call `const` member functions.

Outline

Language Features vs. Design Decisions

- We start to see more language features for class design.
  - Why are there so many rules that seem restricting?
- Defining a nice class in C++ is a very challenging task.
  - Nice: easy to use, less chance to make mistakes.
  - Need to use many language features.
  - Need to make design decisions.
- Language features and design decisions are actually closely related.
  - C++ is designed to support established design practices.
- How to design a class?
  - Beyond the simple `date` class where we can rely on intuitions.
  - What should be the data members?
  - What should be the member functions?

State of Object

- Consider any object.
  - Consisting of data members and member functions specified by its type
- The values of the members and the objects referred to by members are collectively called the state of the object.
  - Or simply called its value
- For example,
  - State of `date`: values of `year_`, `month_`, `day_`
  - State of `expression`: values of `expr_id_`, `op_name_`, `op_type_`, `inputs_`
- To use an object, the major concern is to keep its state valid, or well defined.

Class Invariant

- An invariant is something that will always remain true during some progress.
- If we consider a `date` or an `expression` object during program execution,
  - We expect `year_`/`month_`/`day_` to be a valid calendar date.
  - We expect the expression with `expr_id_` to use an operation with `op_name_` and `op_type_` andn operands from `inputs_`.
- Class invariant: the condition for the state of an object to be valid
  - It is implied by the type of the object so we call it class invariant.
  - The class invariants of `date` and `expression` are shown above.

Roles of Class Interface

- Class interface: declarations of constructors and public member functions
- Constructors should establish the class invariant for the objects when they are constructed.
- Public member functions should maintain the class invariant.
- Therefore, one can safely assume all the objects of the class always satisfy the invariant as long as they are manipulated through the class interface.
  - As guaranteed by mathematical induction.

Precondition and Postcondition

- Precondition: the constraints that arguments of a function should satisfy.
  - E.g. when calling `date::set`, the provided year/month/day should be valid.
- Postcondition: the constraints that returned and modified values of a function should satisfy.
  - E.g. after calling `date::set`, the object should have the desired year/month/day while remaining valid.
- Garbage in, garbage out
  - A correct function may perform errorousnously, i.e. violating the postcondition, if the precondition is violated.
  - This is the most usual mistake made by programmers.
- How to ENFORCE preconditions?

Public Member Functions

- Class invariant should hold before and after a call to a public member function.
  - It serves as part of the precondition and the postcondition regarding data members for any public member function.
- Holding class invariant is easy for `const` member functions.
  - Nothing is changed: class invariant remains valid
- Non-`const` public member functions are expected to make some progress: change the state of the object from a valid one to another valid one.
  - When we refer to implementation, we mean how that change is computed.
  - The class invariant may be violated during the computation.

Private Member Functions

- Implementations could be very complicated.
  - Class designers need to organize the computations into functions.
  - Those functions should be private member functions.
- It is <u>not necessary</u> for the private member functions to have the class invariant as the pre- and the postcondition.
- In other words,
  - If a member function may violate class invariant, it need to be private.
  - Otherwise, it could be public.

Data Members

- If invariants exist among a set of variables, it is a good idea to form a class with them as data members.
  - `year_`, `month_`, `day_`.
  - `expr_id_`, `op_name_`, `op_type_`, `inputs_`
- Avoid to design a class that leads to a god object – the object that tries to do everything.
  - Variables where no invariant exist should not be bundled into a class directly.

Outline

Class Design

Class Invariant

More on Default Constructor

Implicitly-Declared Default Constructor

- The compiler will generate a public default ctor for <u>any non-reference type</u> if the type has no user-defined ctors.
- For built-in types, e.g. `int` and `bool`, it will do nothing.
- For class types, it will default-initialize their members.
  - There will be a compiling error if a member has no default constructor.

Why?

- Having a user-defined ctor is a hint of non-trivial class invariants.
  - The compiler expects the class designer to provide a default ctor if one tries to default-initialize an object.
- Otherwise, the compiler attempts to maintain the weakest class invariant – members should satisfy their individual class invariant, by default-initializing them.
- Why don't default ctors assign some value to built-in types to avoid undefined behaviors?
  - Again, this is a rule from the C language for performance reasons.

Default Constructor: Example I

```
class date {
public:
    date();
    date(int y, int m, int d);
    date(std::string str);
    ...
}; // class date

date epoch; // will not compile if date::date() is not provided
```

- Has user-defined ctor.
- So there will be no implicitly-declared default ctor.
- One must provide the default ctor for default-initialization to compile.

Default Constructor: Example II

```
class vec_ref {
    std::vector<int> &ref;
}; // class vec_ref

vec_ref vref; // compiling error
```

- No user-defined ctor
- So there will be an implicitly-declared default ctor.
- It will default-initialize `ref`.
    - It's a reference type and default-initialization makes no sense.
    - So there is a compiling error.

Default Constructor: Example III

```
// our very first definition of expression
struct expression {
    int expr_id;
    std::string op_name;
    std::string op_type;
    std::vector<int> inputs;
}; // struct expression

expression expr; // compile OK but not nice
```

- No user-defined ctor
- So there will be an implicitly-declared default ctor.
  - Default-initialize `op_name` and `op_type` to an empty string by the default ctor of `std::string`.
  - Default-initialize `inputs` to an empty vector by the default ctor of `std::vector`.
- `expr_id`, which is of built-in types, will be default-initializd as undefined values.
  - Not nice: compiler won't help if someone forgets to assign a value to `expr_id`.

Default Constructor: Example IV

```
// our better expression design
class expression {
   ...
public:
    expression(....);
    ...
}; // class expression

expression expr; // compiling error
```

- There is a user-defined ctor.
- So there is no implicitly-declared default ctor.
- There is a compiling error since the compiler fails to find the default ctor for default-initialization.
  - Nice design: compiler enforces that all arguments to the ctor should be provided.

How to initialize reference members?

```
class vec_ref {
    std::vector<int> &ref;
public:
    vec_ref(std::vector<int> &param);
}; // class vec_ref

vec_ref::vec_ref(std::vector<int> &param)
  : ref(param) {
}

std::vector<int> int_vec
vec_ref vref(int_vec);
```

- You have to initialize reference members using the constructor initializers.

Summary and Advice

- Each class type should have a class invariant.
  - Constructors establish the class invariant.
  - Public member functions maintain the class invariant.
- Implicitly-declared default ctor
  - Generated automatically for types w/o user-defined ctors
  - It will default-initialize the members recursively.
  - Do nothing for a member of a built-in type.
- Member functions can be `const` or non-`const`, depending on their semantics.