ECE 590 – OOP and Machine Learning
Lecture 05 Organizing Programs and Data

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

September 7, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 3–4
- Next lecture: Project 2 Instruction

Outline

EasyNN Expressions

```
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    printf("program %p, expr_id %d, op_name %s, op_type %s, inputs %d (",
        prog, expr_id, op_name, op_type, num_inputs);
    for (int i = 0; i != num_inputs; ++i)
        printf("%d,", inputs[i]);
    printf(")\n");
}
```

- While the above code provides logging to troubleshoot issues later, we need to store the expressions somewhere for future evaluations.
- Indeed, since we need to implement a few functions for EasyNN, and they are called at different times by our Python code, we need to store data obtained from those functions and use it later so the functions could be conceptually "connected".

Class Type for EasyNN Expression

```
#include <string>

struct expression {
    int expr_id;
    std::string op_name;
    std::string op_type;
    ...
}; // struct expression
```

- Group everything about an EasyNN expression together by using a structure.
- To define and use a proper class type will improve productivity.
  - Convenience is usually a good indication.
  - Designing class types and associated operations is the most fundamental programming activity in C++.

Class Type for EasyNN Expression

```
#include <string>

struct expression {
    int expr_id;
    std::string op_name;
    std::string op_type;
    ...
}; // struct expression
```

- `std::string` is from the standard header `string`.
  - Type of a C++ string
- What about the operands?

The C++ Vector

```
#include <vector>
#include <iostream>
int main() {
    std::vector<int> integers;
    for (size_t i = 0; i < 10; ++i) {
        integers.push_back(i);
    }
    for (size_t i = 0; i < integers.size(); ++i) {
        std::cout << integers[i] << std::endl;
    }
    return 0;
}
```

- This program will display integers from 0 to 9, each on a line.
- `std::vector` is from the standard header `vector`.
- `std::vector` is a <u>container</u> holding objects as its elements.

Vector vs. Array

- `std::vector` is one of the many C++ counterparts of array.
  - Array is a built-in type of C++ inherited from C that is notoriously hard to work with.
- `std::vector` is much easier to use than array.
  - As function parameters.
  - Able to introduce new elements/remove old elements easily.
  - Without sacrificing any performance (if you know how).

Vector Objects

```
std::vector<int> integers;
```

- `std::vector` is a <u>template class</u>.
  - A core language feature that makes C++ powerful.
- We <u>instantiate</u> a template class by supplying additional types within `<>` as “parameters”.
  - Then variables/objects of that type can be constructed.
- `std::vector<int>` is the type of vectors containing `int` objects.
  - All the objects within a vector should be of the same type.
  - Different vectors can hold objects of different types.
- A vector object contains no element (being empty) by default.

Add Elements to Vector

```
for (size_t i = 0; i < 10; ++i) {
    integers.push_back(i);
}
```

- The member function `push_back` appends a new element to the vector object.
- Containers store values. The value of `i` at the time of `push_back` is copied to the new element.

Updated Class Types

```
#include <string>
#include <vector>

struct expression {
    int expr_id;
    std::string op_name;
    std::string op_type;
    std::vector<int> inputs;
}; // struct expression

struct program {
    std::vector<expression> exprs;
}; // struct program
```

- You may define a `std::vector` holding elements of user-defined class types.
    - Define a class type `program` to store the expressions from an EasyNN DAG.
- You should avoid to use C-style string (`char *` and `const char *`) and C-style vector (`int[]`, `expression[]`, etc) whenever possible.

Working with Vectors

```
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    ...
    expression expr;
    expr.expr_id = expr_id;
    expr.op_name = op_name; expr.op_type = op_type;
    expr.inputs.assign(inputs, inputs+num_inputs);
    prog->exprs.push_back(expr);
}
```

- C-style strings can be assigned to C++ strings directly.
- C array `inputs` can be turned into a range $[\text{inputs}, \text{inputs} + \text{num\_inputs})$.
  - Which is used by the `assign` member function of `std::vector` to make a copy of the C array.
- `prog` is a pointer to a `program` object.
  - As indicated by its type `program *`.
  - The members are accessed using the arrow operator `->`.

Where does the prog object comes from?

```
// libeasynn.cpp
...
#include "libeasynn.h"
#include "program.h"
#include "evaluation.h"

program *create_program() {
    program *prog = new program;
    printf("program %p\n", prog);
    return prog;
}

void append_expression(program *prog, ....) {
    ....
    prog->exprs.push_back(expr);
}
```

- Without diving into the details, we create a `program` object in `create_program`, and our Python code will pass it back every time we need it.

Outline

Object-Oriented Programming (OOP)

- A programming paradigm.
- Abstraction: to use a type, you don't need to know its details
- Encapsulation: the details of a type is hidden from you
  - You can only manipulate its objects in a predefined way.
  - The operations is therefore guaranteed to be valid.
- Modularity: things related to a type is required to be grouped together

Any issue?

```
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    ...
    expression expr;
    expr.expr_id = expr_id; // what if I forgot it?
    expr.op_name = op_name; expr.op_type = op_type;
    expr.inputs.assign(inputs, inputs+num_inputs);
    prog->exprs.push_back(expr);
}
```

- While the above code may work, it breaks many aspects of OOP.
    - Whoever need to modify this function need to know a lot about the class types `expression` and `program`.
- A small mistake like forgetting to assign `expr_id` to `expr.expr_id` may lead to undefined behaviors (UB).

Undefined Values and Undefined Behaviors

```
int i;
```

- Integer variables not initialized have an undefined value, will lead to undefined behaviors (UBs) if you try to use that value later.
- There are many possibilities for undefined behaviors.
  1. Program crashes immediately.
  2. Program runs and finishes as usual.
  3. Program runs as usual but will crash at some completely irrelevant places at a later time.
  4. Program runs as usual but generates wrong results for no reason.
  - The last two are the most difficult to debug, and that's why you should avoid UBs.

Typical Sources of Undefined Behaviors

- A variable that is not initialized properly.
    - Preventable by always initiating a variable when defining it.
    - Can be enforced for class types via constructor.
- Violations of invariants among variables.
    - `inputs` has `num_inputs` elements so accessing `inputs[-1]` and `inputs[num_inputs]` are UBs.
    - Preventable by always verifying the invariants.
    - Can be enforced for class types via encapsulation.
    - Performance is a concern as C++ needs to compete with C.
- Since many such behaviors happen inside a loop as elements are visited using a wrong index, you should always create the loop using the loop range pattern.

Constructor

```
struct expression {
    int expr_id;
    std::string op_name;
    std::string op_type;
    std::vector<int> inputs;

    expression(int expr_id,
        const char *op_name, const char *op_type,
        int *inputs, int num_inputs);
}; // struct expression
```

- A constructors is a special member function.
  - Has the same name as the class type.
  - Won't return a result (NEVER put `void` there)
  - Will be called automatically when constructing objects

Protection

- By using the constructor, one can create `expression` objects without knowing its details.
- However, one can still access the data members, and may make changes accidentally.
- We should allow users to manipulate objects <u>only</u> through constructors and other member functions.
  - In other words, we need to <u>protect</u> data members.

Public and Private Members

```
struct expression {
private:
    int expr_id_;
    std::string op_name_;
    std::string op_type_;
    std::vector<int> inputs_;
public:
    expression(int expr_id,
        const char *op_name, const char *op_type,
        int *inputs, int num_inputs);
}; // struct expression
```

- A protection label defines how the members thereafter should be protected, until the next label appears.
- Public members are defined using the label `public:`.
  - Accessible to all users of the type
- Private members are defined using the label `private:`.
  - Inaccessible to users of the type but accessible to member functions.
  - Notice the `_` I introduced at the end of each `private` data member as a usual convention.

Define Class Types Using Classes

```
class expression {
    int expr_id_;
    std::string op_name_;
    std::string op_type_;
    std::vector<int> inputs_;
public:
    expression(int expr_id,
        const char *op_name, const char *op_type,
        int *inputs, int num_inputs);
}; // class expression
```

- By default, all members in a `struct` are public.
- Since we need to keep data members private, the language construct like `struct` but has all members being private by default is `class`.
- Default protection is the only difference between `struct` and `class`.

Define Constructor

```
expression::expression(int expr_id,
    const char *op_name, const char *op_type,
    int *inputs, int num_inputs) :
    expr_id_(expr_id), op_name_(op_name), op_type_(op_type),
    inputs_(inputs, inputs+num_inputs) {
}
```

- Constructor initializers: the syntax between the parameter list and the function body starting by `:`.
  - Used by compiler to initialize data members before the function body get executed
  - The data members are initialized using the values that appear between `()`.
- Since we initialize all the data members using constructor initializers, the body of the constructor is empty.

Accessor Functions

```
class expression {
    ...
public:
    ...
    int get_id();
    std::string get_op_name();
    std::string get_op_type();
    ...
}; // class expression

int expression::get_id() {
    return expr_id;
}
...
```

- Member functions whose names follow some conventions.
- getter functions allow to read data members.
- setter functions allow to write data members.
- Introduce both only when necessary.

More on Member Function

```
class program
{
    std::vector<expression> exprs_;
public:
    void append_expression(int expr_id,
        const char *op_name, const char *op_type,
        int inputs[], int num_inputs);
}; // class program
```

- Similar to `expression`, we may improve `program` by introducing protection and member functions.
- `expr_` is already of a class type and will be initialized properly as an empty vector so there is no need to define a constructor for `program`.

Using Constructor and Member Functions

```
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    ...
    prog->append_expression(expr_id, op_name, op_type, inputs, num_inputs);
}

void program::append_expression(int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    exprs_.puch_back(expression(expr_id,
        op_name, op_type, inputs, num_inputs));
}
```

- `append_expression` now delegates most work to `program::append_expression`
- An `expression` object can be created directly without a name by using the constructor `expression(...)`.

Summary

- Introduce new class types for group data.
- Use protection, constructor and member functions to improve class types.