ECE 449/590 – OOP and Machine Learning
Lecture 04 C++ Overview

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

August 31, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 0–2
- Next lecture: Accelerated C++ 3–4

Outline

The Hello World Program

```
// a small C++ program
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

- Print “Hello, world!” on your display.
- Simple enough to test your C++ installation.

Comments

```
// a small C++ program
```

- The `//` characters begin a comment
- Explain the program to a human reader
  - Ignored by the compiler
  - Explain your intention instead of obvious fact

#include

- C++ is consisting of
  - Core language, which is always readily available
  - Standard library, which is not
- You must ask explicitly for standard library facilities.
  - By using `#include` directives
  - Almost always at the beginning of a program
- We need facilities for stream input/output. So we write:

```
#include <iostream>
```

The main Function

```
int main() {
    ...
}
```

- Every C++ program must contain a function named `main`.
  - The program starts at the `main` function.
  - Also known as the entry point of your program.
- The `main` function is not required for C++ shared libraries, as we will introduce later for our course projects.

Return Type and Parameters

`int main()`

- `main` is required to return an integer as its result.
  - 0 for success; otherwise indicates a problem
  - `int` is the <u>type</u> for integers defined by the core language.

Curly Braces (or simply Braces)

```
int main()
{                    // left brace
    ...              // the statements go here
}                    // right brace
```

- After `()` for parameters, we continue the `main` function with a sequence of statements enclosed in curly braces `{}`.
- Statements define the functionality of a function.
  - Executed sequentially in the order in which they appear

Using Standard Library for Output

```
std::cout << "Hello, world!" << std::endl;
```

- This is a statement – a statement always ends with `;` .
- First, “Hello, world!” is written to `std::cout`.
  - `std::cout` refers to the standard console output.
  - The standard library's output operator `<<` is used.
- Then, `std::endl` is written.
  - `std::endl` refers to a end-of-line symbol.
  - Additional output will appear on a new line.

String Literals

- `"Hello, world!"`: a string literal
  - Begin and end with `"`
  - Must appear entirely on one line of the program
- Use backslash `\` to include special characters
  - `\n`: newline
  - `\t`: tab
  - `\"`: `"`
  - `\\`: `\`
- Adjacent string literals are concatenated automatically.
  - `"Hello" "," "world!"` is the same as `"Hello, world!"`.
  - That's the way to specify long strings across multiple lines.

```
"This string literal"
" spans two lines."
```

The return Statement

```
return 0;
```

- Recall `main` must return an integer as its result.
- Use a `return` statement for such purpose.
  - End execution of the function
  - Pass the value between `return` and `;` back

Spaces

- Spaces are required only when they keep adjacent symbols from running together.
  - Both newlines and tabs are spaces.
  - Except `//` comments, `#include`, and string literals
- Use spaces wisely to make your code much easier to read.
  - Programs are indented for readability.
  - Use newlines to break lines longer than 80 characters – better to refactor your program as we will discuss in later lectures.

C++ as Better C

```
/*
   a small C++ program
   that uses many C features
*/
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

- Most C features can still be used in C++. Feel free to use them

Outline

Expression Statements

```
std::cout << "Hello, world!" << std::endl;
```

- This is an expression statement.
  - An expression followed by `;`
- An expression asks to compute something.
  - The computation returns a result,
  - and may also have side effects.
- We discard the result in our example since we are only interested in its side effects,
  - which change the display.

Side Effects

- Side effects are critical for imperative languages like C/C++ and Java, since they allow to change the program state.
- `3+4` returns `7` and has no side effects.
- `a=7` returns `a` and has the side effect that changes `a` to `7`.
- `std::cout << "Hello, world!" << std::endl` has side effects that change the display.
  - What is returned?

Operators and Operands

- An expression is consisting of operators and operands.
- For `std::cout << "Hello, world!" << std::endl`,
  - Operands: `std::cout`, `"Hello, world!"`, `std::endl`
  - Operators: the two `<<` symbols
- Every operand has a type.
- The effect of an operator depends on the types of its operands.
  - Type of `std::cout` is `std::ostream`.
- The `<<` operator takes two operands: `L << R`
  - When `L` is of the type `std::ostream`, it will write `R` to `L`.
- So how does the above expression work with 2 `<<`'s and 3 operands?

Operator Associativity

- `<<` is left-associative.
- `std::cout << "Hello, world!" << std::endl`
  is equivalent to
  `(std::cout << "Hello, world!") << std::endl`
- The first `<<` (after `std::cout`) is first evaluated.
- Then the result is used as the left operand of the second `<<`.
- The result returned by the first `<<` is actually `std::cout`.
  - So after `"Hello, world!"` is written, `std::endl` is written.
  - It therefore enables chained output operations.

Operator Precedences

- For expressions that contain many operands and operators, the order the operators are evaluated depends on their precedences.
  - E.g. `1+2*3` returns `7` instead of `9` since `*` has higher precedence than `+`.
- Except obvious precedences like those among arithmetic operators, use `()` to highlight your intention
  - So you don't need to memorize them (at least for this course).

Outline

Hello World!

Expression

Function and for Loop

Expression from EasyNN

- The concept of expression extends to other programming languages.
- A language feature named operator overloading further allows languages like C++ and Python to define what an operator is supposed to do for user-defined operand types.
  - Enable our EasyNN library to capture the DFG of a computation in Python in the SSA form.
- We will need to process those SSA statements in C++ for our projects.
- A few problems to solve
  - How to pass information between Python and C++ code?
  - How to represent the DFG as the SSA form in C++?
  - How to evaluate the SSA form in C++?

An EasyNN Example

```
a = nn.Input("a")
b = a+a
print(b.statements())
```

```
t0 = a.Input()
t1 = .Add(t0,t0)
```

- The SSA form of the DAG consists of two EasyNN expressions.
- Each expression contains a single operator and generates a single output.
  - The output is stored to a variable.
  - There is either no operand for input operators, or
  - Operands are variables generated by other expressions that appear earlier.
- The variables are named as `t`*id* where we may use *id* to refer to the expression generating the variable.

Working with Existing Code

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

- With existing code, new features are usually added by implementing various functions.
- The EaysNN python code will call the `append_expression` function to pass one EasyNN expression to C++.
- We'll need to implement this and a few other functions to complete Project 2.
  - We'll need to understand what information are available from their parameters first.

EasyNN Expressions

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

- Assume the `append_expression` function will pass one EasyNN expression from Python to C++.
    - This is indeed a C function as we will discuss later.
    - Ignore `prog`.
- Intuitively, the parameters provide:
    - *id* of the expression.
    - Information regarding the operator.
    - Information regarding the operands.

Working with C Types

```
const char *op_name,
const char *op_type,
int inputs[],
int num_inputs
```

- `op_name` and `op_type`: name and type of the operator.
  - `char *` is the type of a C-style string.
  - `const char *` indicates that you are not supposed to modify the content of the string.
- `inputs`: the array (C-style vector) of the operands.
  - In function parameters, `[]` stands for array.
  - Each operand is an integer, i.e. the *id* of the expression generating it.
  - How many operands are there?
- `num_inputs`: the number of the operands.
  - Put index between `[]` to visit each element
  - You should use only the valid indices: `0`, `1`, . . ., `num_inputs-1`

Logging

```
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");
}
```

- Logging helps you to understand the parameters now, and to troubleshoot issues later.
  - I prefer the `printf` format string over `cout` because it is more readable, and most languages support it.
- Use a for loop since we don't know the number of operands.
  - `++i` increases the integer `i` by `1`.
  - Operator `!=` stands for not-equal.

Example Output

- EasyNN expressions

```
t0 = a.Input()
t1 = .Add(t0,t0)
```

- C++ output

```
program 0x1e32cd0, expr_id 0, op_name a, op_type Input, inputs 0 ()
program 0x1e32cd0, expr_id 1, op_name , op_type Add, inputs 2 (0,0,)
```

- This is the start point of Project 2.
  - Run the programs and read the logging messages before modifying any code to understand how existing code works.

The for Statement

```
for (int i = 0;       // for (init-statement
    i != num_inputs; //     condition expression;
    ++i)             //     increment expression)
{                    // {
    ...              //     for body enclosed by {}
}                    // }
```

- `for` statement: for header followed by a for body
- for header: allow to present up-front the three most essential elements for any loop to improve readability
  - init-statement: initialize the loop variable
  - condition expression: the condition to execute the for body, should return a boolean value – `true` or `false`.
  - increment expression: the way to change the loop variable at the end of the iteration
- Within the for body
  - Use `break;` to exit the loop immediately
  - Use `continue;` to terminate the current iteration immediately

for Loop and while Loop

```
for (;;)      // loop forever
{
    ...       // you can still use break; to exit the loop
}

for (; cond;) // same as while (cond)
{
    ...
}
```

- The init-statement can be an empty statement `;`.
- Both condition and increment expressions are also optional.
- So you can use a for loop when you need a while loop.
    - It's easier to add and highlight init-statement and increment expressions at a later time.

Pattern of Range and for Loop

- Patterns: established/recurring professional practices
- In C++ (and Python), ranges are used to describe a set of elements and are always asymmetric (half-closed half-open)
  - When there are `num_inputs` elements in `inputs`, the valid indices must belong to the range $[0, \text{num\_inputs})$.
  - A sub-string is specified by $[\text{begin\_index}, \text{end\_index})$.
- To iterate through every element in a range $[\text{begin}, \text{end})$, you should use the for header below:

```
for (index = begin; index != end; ++index)
```

  - The loop invariants guarantee `index` is within the range.
  - Every professional reading this line will immediately identify the pattern and know your intention.

Summary

- Basic C++ syntax is similar to C and Java.
- At times, you may need to work with C functions and types in C++ to interface with OS and other languages.
- The concepts of expression and loop range extend to languages beyond C++, making them good investment toward software development.
- Run programs and read the logging messages to understand how existing code works.