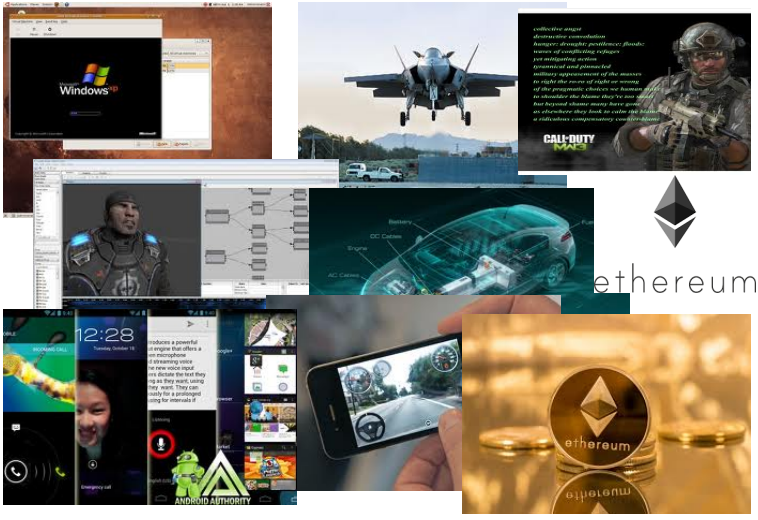
Introduction to Software Verification

Yulei Sui

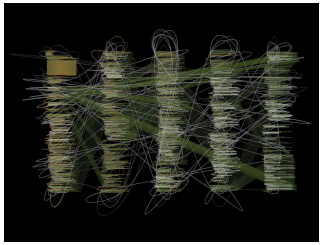
University of Technology Sydney, Australia

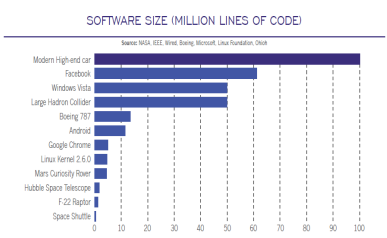
Software Is Everywhere

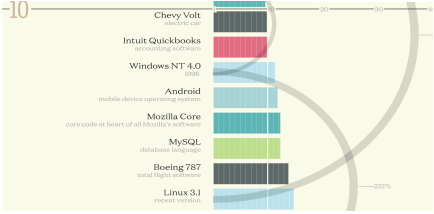


Modern System Software

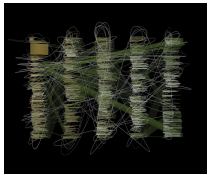
– Extremely Large and Complex

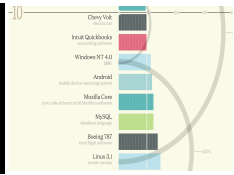






Software Becomes More Buggy





More Complex!

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

Memory Leaks

Buffer Overflows

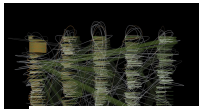
Null Pointers

Use-After-Frees

Data-races

More Buggy!

Software Becomes More Buggy



More Complex!

Vulnerabilities (security defects)

Design apps to run in cloud

The risks

Quality issue: many more "underwater" than those reported "above the water"

The National Vulnerability Database (DHS/US-CERT)

- Lists >47,000 documented vulnerabilities

Undiscovered/unreported (0-day) vulnerabilities are huge

- 20X[1] multiplier
- 47,000 x 20 = estimated **940,000 vulnerabilities** replicated in many products

Greater than 80% of attacks happen at the application layer

Public vulnerabilities are tip of the iceberg !

More Buggy!

https://www.slideshare.net/innotech_conference/hp-cloud-security-inno-tech-20140501

Let us take a look at some real-world vulnerability examples ...

Memory Leak

- A dynamically allocated object is not freed along some execution path of a program
- A major concern of long running server applications due to gradual loss of available memory.

```
1  /* CVE-2012-0817 allows remote attackers to cause a denial of service through adversarial connection requests.*/
2  /* Samba --libads/ldap.c:ads_leave_realm */.
3
4  host = memAlloc(hostname);
5   ...
6  if (...)   {...; return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to release host on error.
7
```

```
1  /* A memory leak in Php-5.5.11 */
2  for (...) {
3      char* buf = readBuffer();
4      if (condition)
5          printf (buf);
6      else
7          continue;   // buf is leaked in else branch
8      freeBuf(buf);
9  }
```

Buffer Overflow

- Attempt to put more data in a buffer than it can hold.
- Program crashes, undefined behavior or zero-day exploit[1].

```
1  /* A simplified example from "Young and Mchugh, IEEE S&P 1987", exploited by attackers to bypass verfication*/
2
3    void verfiyPassword(){
4      char buff[15]; int pass = 0;
5       printf ("\n Enter the password : \n");
6      gets(buff);
7
8      if (strcmp(buff, "thegeekstuff")){     // return non-zero if the two strings do not match
9           printf ("\n Wrong Password \n");
10     }
11     else{                                  // return zero if two strings matched or a buffer overrun
12          printf ("\n Correct Password \n");
13          pass = 1;
14     }
15     if (pass)
16          printf ("\n Root privileges given to the user \n");
17   }
18
```

[1] Heartbleed, a well-known vulnerability in OpenSSL is also caused by buffer overflow (It took more than 2 years to discover and fix it since first patch, and over 500,000 websites were affected). Vulnerability is exploited when more data can be read than should be allowed.

Uninitialized Variable

- Stack variables in C and C++ are not initialized by default.
- Undefined behavior or denial of service via memory corruption

```
1  /* An uninitialized variable vulnerability simplified from gnuplot (CVE-2017-9670) */
2
3  void load(){
4          switch ( ctl ) {
5                  case -1:
6                          xN = 0; yN = 0;
7                          break;
8                  case 0:
9                          xN = i; yN = -i;
10                         break;
11                 case 1:
12                         xN = i + NEXT_SZ; yN = i - NEXT_SZ;
13                         break;
14                 default:
15                         xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
16                         break;
17         }
18                 plot(xN, yN);
19 }
20
21
```

Use-After-Free

- Attempt to access memory after it has been freed.
- Program crashes, undefined behavior or zero-day exploit.

```
1  /*  CVE-2015-6125 and CVE-2018-12377 with similar heap use after free patterns*/
2
3  char* msg = memAlloc(...);
4  ...
5  if (err) {
6          abrt = 1;
7           ....
8          free(msg); // the memory is released when an error occurs at server
9  }
10 ...
11 if (abrt) {
12          ...
13          logError("operation aborted before commit", msg); // try to access released heap variable,
14                                                            // causing either crash or writing confidential data
15 }
```

Code Review by Developers

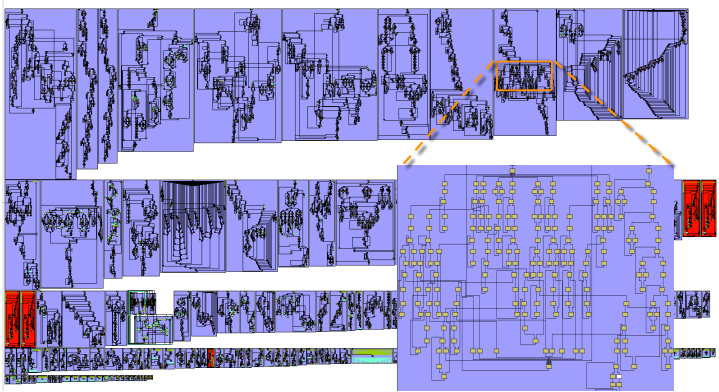
However ...

A large project (e.g., consists of millions of lines of code) is almost impossible to be manually checked by human :

-- intractable due to potentially unbounded number of paths that must be analyze
-- undecidable in the presence of dynamically allocated memory and recursive data structures

How about real-world large programs?

Whole-Program CFG of 300.twolf (20.5K lines of code)



#functions: 194

#pointers: 20773 #loads/stores: 8657 Costly to reason about flow of values on CFGs!

Software Verification

Software verification is a discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements.

Software Verification

- Software Verification is useful for proving the correctness, safety and security of a program, and a key aspect of testing can execute as expected.
  - ”Have we made what we were trying to make?”
    - Are we building the system right?
    - Does our design meet the user expectations?
    - Does the implementation conform to the specifications?

Why Software Verification?

- Better quality in terms of more secure and reliable software
  - Help reduce the chances of system failures and crashes
  - Cut down the number of defects found during the later stages of development
  - Rule out the existence of any backdoor vulnerability to bypass a program's authentication
- Reduce time to market
  - Less time for debugging.
  - Less time for later phase testing and bug fixing
- Consistent with user expectations/specifications
  - Assist the team in developing a software product that conforms to the specified requirements
  - Help get a better understanding of (legacy) parts of a software product

What Types of Software Verification We Have?

Code verification vs design verification

- **Design verification**: verifying design of a software system.
  - Design specs: specification languages for components of a system. For example,
    - Z language for business requirements,
    - Promela for Communicating Sequential Processes
    - B method based on Abstract Machine Notation.
    - Specification Language (VDM-SL)
    - ...

What Types of Software Verification We Have?

Code verification vs design verification

- **Design verification**: verifying design of a software system.
  - Design specs: specification languages for components of a system. For example,
    - Z language for business requirements,
    - Promela for Communicating Sequential Processes
    - B method based on Abstract Machine Notation.
    - Specification Language (VDM-SL)
    - ...
- **Code verification**: verifying correctness of source code (**This subject**)
  - Code specs (e.g., return a sorted list):
    - Assertions and pre/postconditions in Hoare logic (design by contract)
    - Type systems
    - Well-formed comments or annotations
    - ...

How to Perform Software Verification?

Verify or prove the correctness of your code against the specifications via

- **Dynamic verification** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...

How to Perform Software Verification?

Verify or prove the correctness of your code against the specifications via

- **Dynamic verification** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...
- **Static verification** (inspecting the code before it runs) (**This subject**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Model checking** (exhaustive exploration of state space by modeling programs as state transition systems)
    - **Abstract interpretation** (a general theory of sound approximation of a program through program abstractions or abstract values)
    - **Symbolic execution** (a practical way to use symbolic expressions instead of concrete values to explore the possible program paths)

Assertions as Specifications in Software Verification

Assertions are program statements used to **test assumptions** made by software designers/programmers/testers. An assertion is a predicate or an expression that **always should evaluate to true** at that point during code execution.

```
assert(expr);
```

unfold →

```
if(expr is true){
    // continue normal execution
}
else{
    __assert_fail();
    // program failure and terminate the program
}
```

Assertions as Specifications in Software Verification (Example)

- Assertions are program statements used to **test assumptions** made by software designers/programmers/testers.
- An assertion is a expression/predicate that **always should evaluate to true** at that point during code execution.

Assertions as Specifications in Software Verification (Example)

- Assertions are program statements used to **test assumptions** made by software designers/programmers/testers.
- An assertion is a expression/predicate that **always should evaluate to true** at that point during code execution.

```
1  #include <assert.h>
2  void main() {
3      int x_axis = 5;
4      int y_axis = 1;
5      // assertion succeeds
6      assert(x_axis > 0 && y_axis > 0);
7      plot(x_axis, y_axis);
8      int y_axis = x - 5;
9      // assertion fails and program crashes
10     assert(x_axis > 0 && y_axis > 0);
11 }
```

Assertions as Specifications in Software Verification (Example)

- Assertions are program statements used to **test assumptions** made by software designers/programmers/testers.
- An assertion is a expression/predicate that **always should evaluate to true** at that point during code execution.

```c
#include <assert.h>
void main() {
    int x_axis = 5;
    int y_axis = 1;
    // assertion succeeds
    assert(x_axis > 0 && y_axis > 0);
    plot(x_axis, y_axis);
    int y_axis = x - 5;
    // assertion fails and program crashes
    assert(x_axis > 0 && y_axis > 0);
}
```

```c
void display_number(int* myInt) {
  assert(myInt != nullptr);
  printf("%d", *myInt);
}
int main () {
  int myptr = 5;
  int* first_ptr = &myptr;
  int* second_ptr = nullptr;
  // assertion succeeds
  display_number(first_ptr);
  // assertion fails and program crashes
  display_number(second_ptr);
}
```

Assertion-Based Software Verification

Assertions can be seen as partial software specifications that

- help a programmer **read the code**
- help the program **detect its own defects**
- help catch errors earlier and **pinpoint sources of errors**

Assertion-Based Software Verification

Assertions can be seen as partial software specifications that

- help a programmer **read the code**
- help the program **detect its own defects**
- help catch errors earlier and **pinpoint sources of errors**

Assertions are typically used in the following scenarios.

- Software **developers** can add assertions during their programming or implementation to verify their expected results.
- Software **testers** can add assertions as a part of the unit testing process.
- Project **managers** or third parties can add assertions in the middle or end of a program execution to verify and understand code bases.

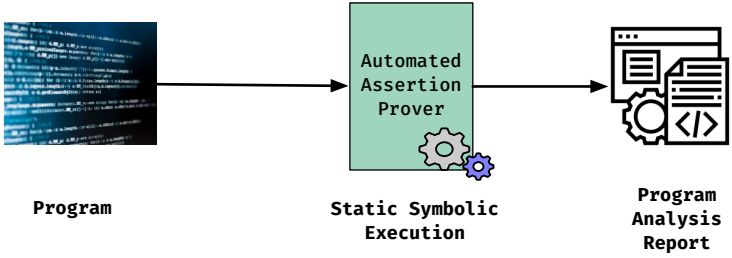
Software Verification vs Software Analysis?

- **Software Analysis** Course (`https://github.com/SVF-tools/Teaching-Software-Analysis`)
  - Aim to ***find existence of bugs***, e.g., If there exist a path, a bug can/may be triggered
- **Software Verification** Course (`https://github.com/SVF-tools/Teaching-Software-Verification`)
  - Aim to ***prove absence of bugs***, e.g., For all paths, user specification should be satisfied and no bug should be triggered.

The Project of This Subject

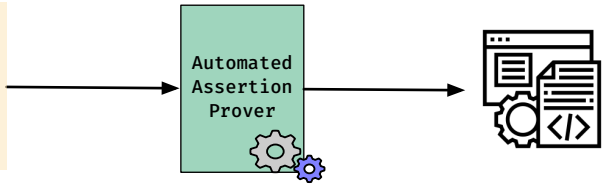
Goal of this subject: develop your own software verification tool in 12 weeks.

More concretely: develop a static symbolic execution engine in C++ to verify C programs with assertions at compile time.



The Project of This Subject

```
1 void Overflow_case()
2 {
3     char buff[3] = {'\0','\0','\0'};
4     char input[] = "Overflow";
5     int  input_length = 8;
6     assert(input_length ⩽ 3);
7     strcpy(buff, input);
8 }
```

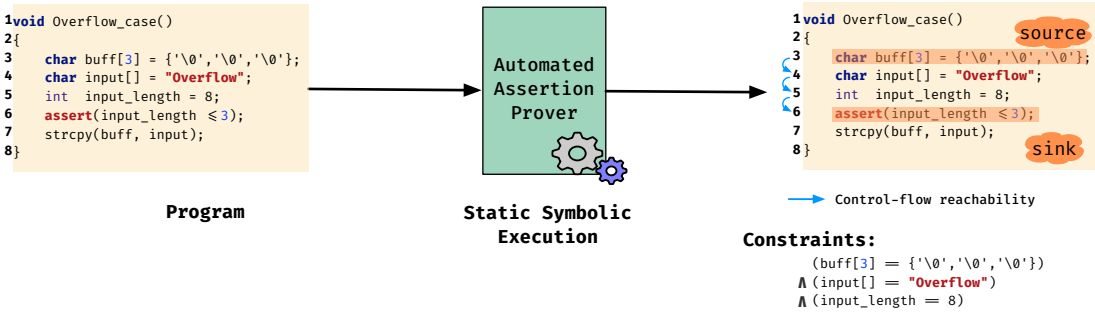


Program

Static Symbolic Execution

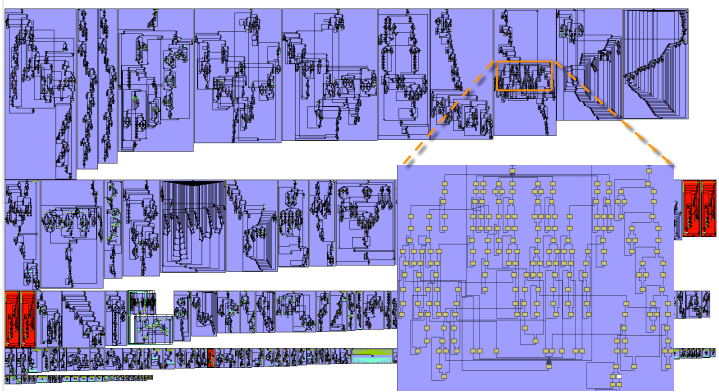
Program Analysis Report

The Project of This Subject



How about real-world large programs?

Whole-Program CFG of 300.twolf (20.5K lines of code)

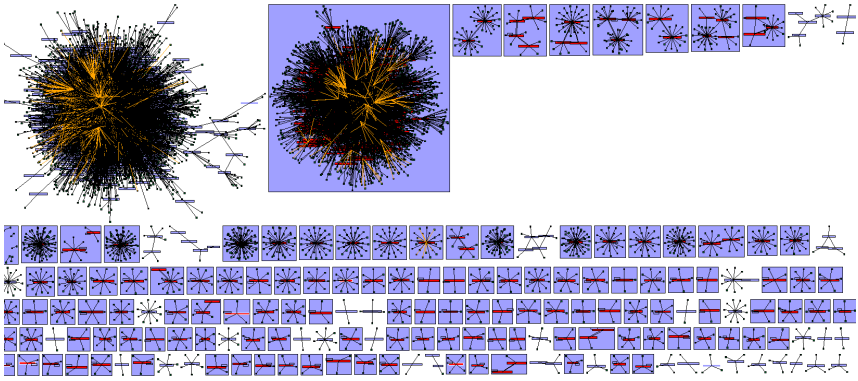


#functions: 194

#pointers: 20773 #loads/stores: 8657 Costly to reason about flow of values on CFGs!

How about real-world large programs?

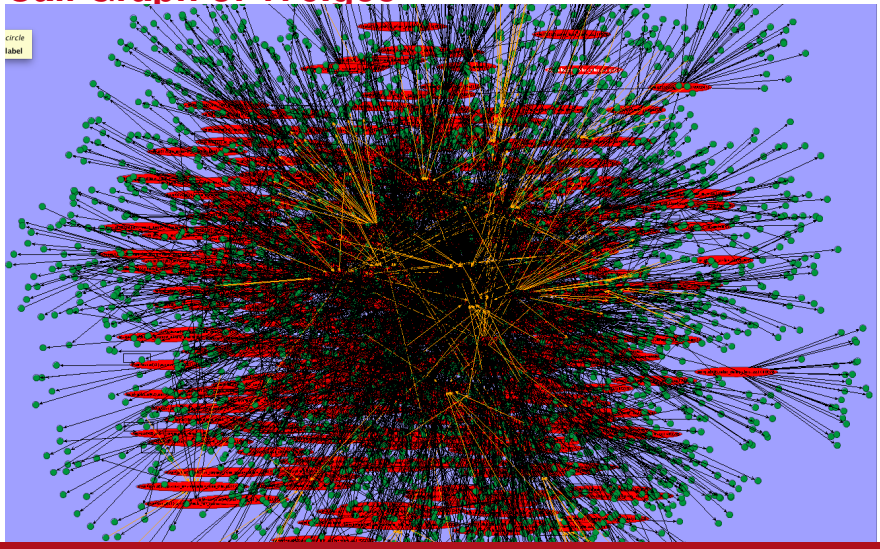
Call Graph of 176.gcc (230.5K lines of code)



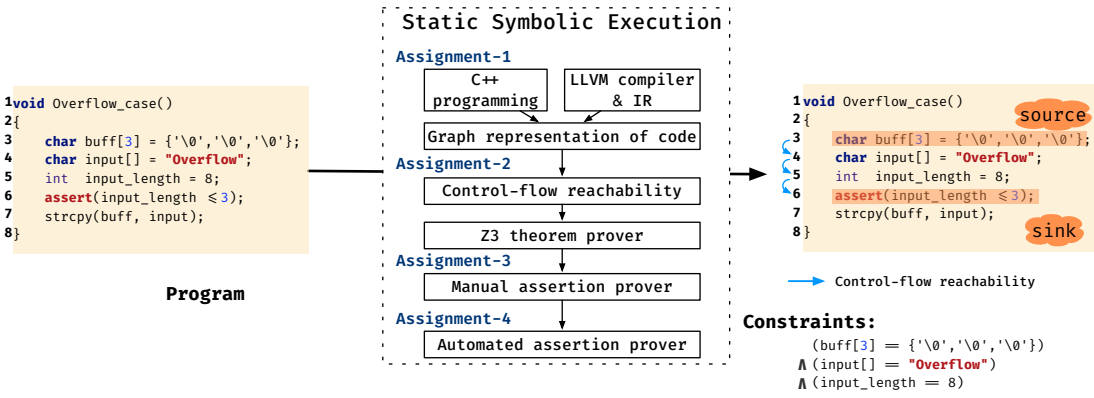
#functions: 2256 #pointers: 134380 #loads/stores: 51543

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc

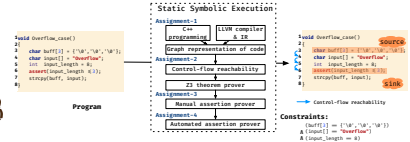


The Project of This Subject

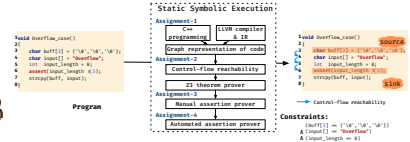


The Project of This Subject

The project sounds complicated?



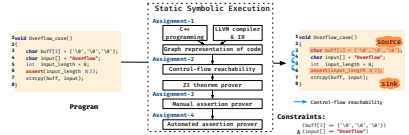
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?

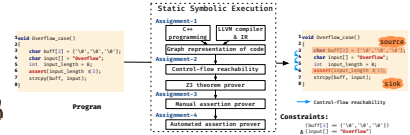
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
- How many lines of code do I need to write?

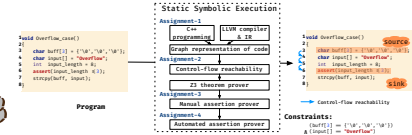
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
- How many lines of code do I need to write?
  - **500-800 lines** of core code **in total** for all the four assignments
- Really? What are the challenges then?

The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
- How many lines of code do I need to write?
  - **500-800 lines** of core code **in total** for all the four assignments
- Really? What are the challenges then?
  - Good C++ programming and debugging skills
  - Understanding basic principles of compilers and symbolic execution
  - Understanding assertion-based software verification and apply it in practice
  - **Please do attend each class** to make sure you can keep up!

What's Next?

- (1) Self-enrol groups on canvas.
  - Though you will join a group, it is used to discuss assignment tasks and solve programming issues. You will still need to submit your code implementation individually for each assignment.
- (2) Configure Programming Environment
  `https://github.com/SVF-tools/Teaching-Software-Verification/wiki/Installation-of-Docker,-VSCode-and-its-extensions`
  - Write and run your program in a docker container (virtual machine) on top of any operating system.
- (3) Write a hello world C++ program.
- (4) Revisit and practice C++ programming (more about C++ programming will be coming next week)