# COMP 2012 Object-Oriented Programming and Data Structures

## Assignment 2 Digital Signatures

### Introduction

In this programming assignment, you are going to familiarize yourself with the concepts of inheritance and polymorphism. You will have the opportunity to implement a toy version of two digital signature algorithms ( DSA and Schnorr). If you have taken COMP 2711 (or and Discrete Structures course) before, you should be familiar with the RSA Public Key Encryption scheme. In fact, there is a RSA-based digital signature. In this assignment, we omit RSA signature for simplicity.

Before you work on this assignment, you can take a look at a short introduction of the digital signature below, and please also make sure you **read the description part** to get familiar with the relationship between different classes. Besides, you should read Digital Signatures protocol for the full specification of the functions you are to implement.

### Digital Signatures

Digital Signature Algorithms are being used to verify if a message originates from a certain person. Let's assume that you're Alice, and you're sending your friend Bob a message to tell how much you love him, and you wrote to him the following letter:

"Dear Bob, I love you. Best, Alice"

But we also know that there's a third person, Eve, that knows you love Bob, and is also planning to send a message to Bob with the exact same message as a joke. Eve doesn't have to wait until you are sure you love Bob and you're sure you want to send the message. How can you prove to Bob that it's actually you that sent the message?

For this, there are multiple digital signature algorithms option available, but all of them work in a similar way:

Step 0: You have a public key and a private key.
Step 1: You "sign" your message using your private key.
Step 2: You send your signature and your message to Bob.
Step 2.5: You announce to everyone in the class your public key.
Step 3: Bob, using your public key, verifies that the message originates from you.

### Menu



### Page maintained by

Zhuo CAI
Email: zcaiam@cse.ust.hk
Last Modified: 04/02/2022 01:47:22

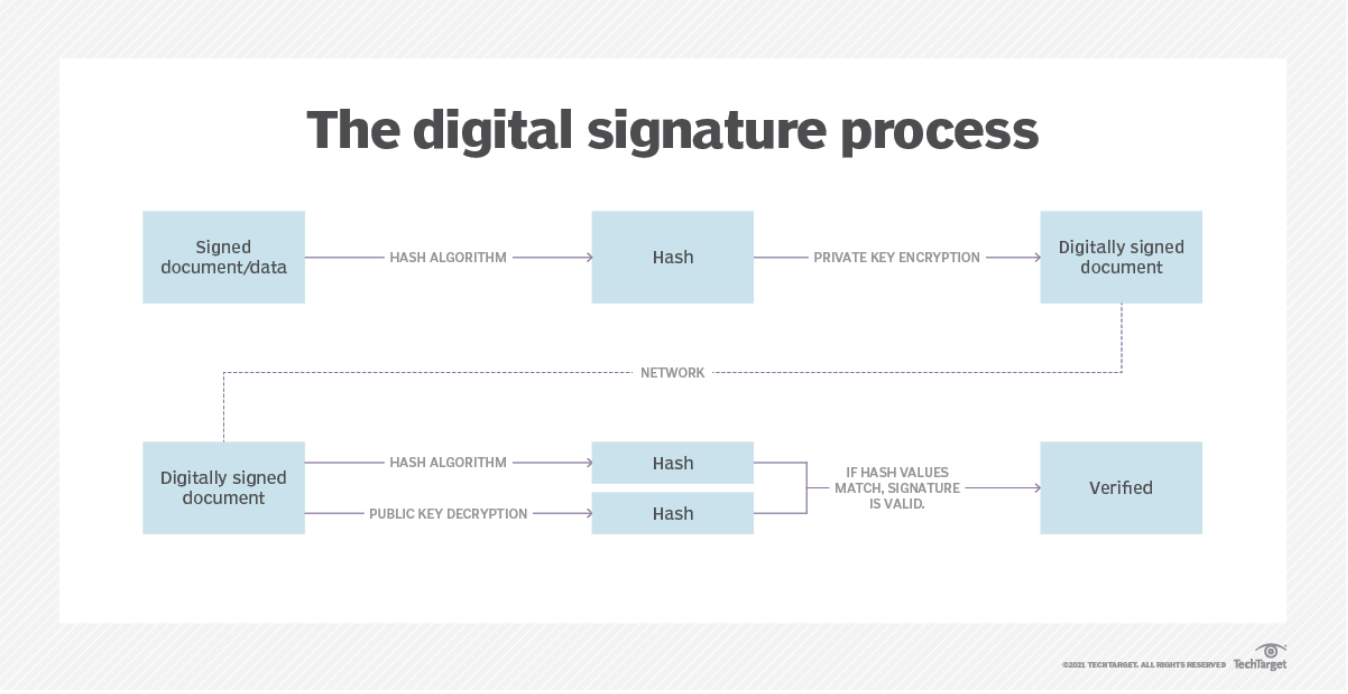
### Homepage

Course Homepage

Due to the cryptographic primitives, it is impossible for Eve to create a fake signature and imitate you, even if you release your public key to everyone, as long as you keep your private key to yourself.

## Digital Signature Process

The digram belows explains the digit signature process




In our example, the sender, Alice, creates a digital signature using a private key to encrypt the signature. At the same time, hash data is created and encrypted.

The recipient, Bob, uses the signer's public key to decrypt the signature to verify that the document's sender is indeed Alice and the message was not alter after it was signed.

In summary, the processes include:

1. Generate keys
2. Sign
3. Verify

Schnorr Algorithm and Digital Signature Algorithm are two algorithms for the digital signature process. You can find the details for the 3 processes of the 2 algorithms in this PDF file: Digital Signatures protocol. It gives the details of how the keys should be generated, how the signing and verification are done.

We value academic integrity very highly. Please read the Honor Code section on our course web page to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

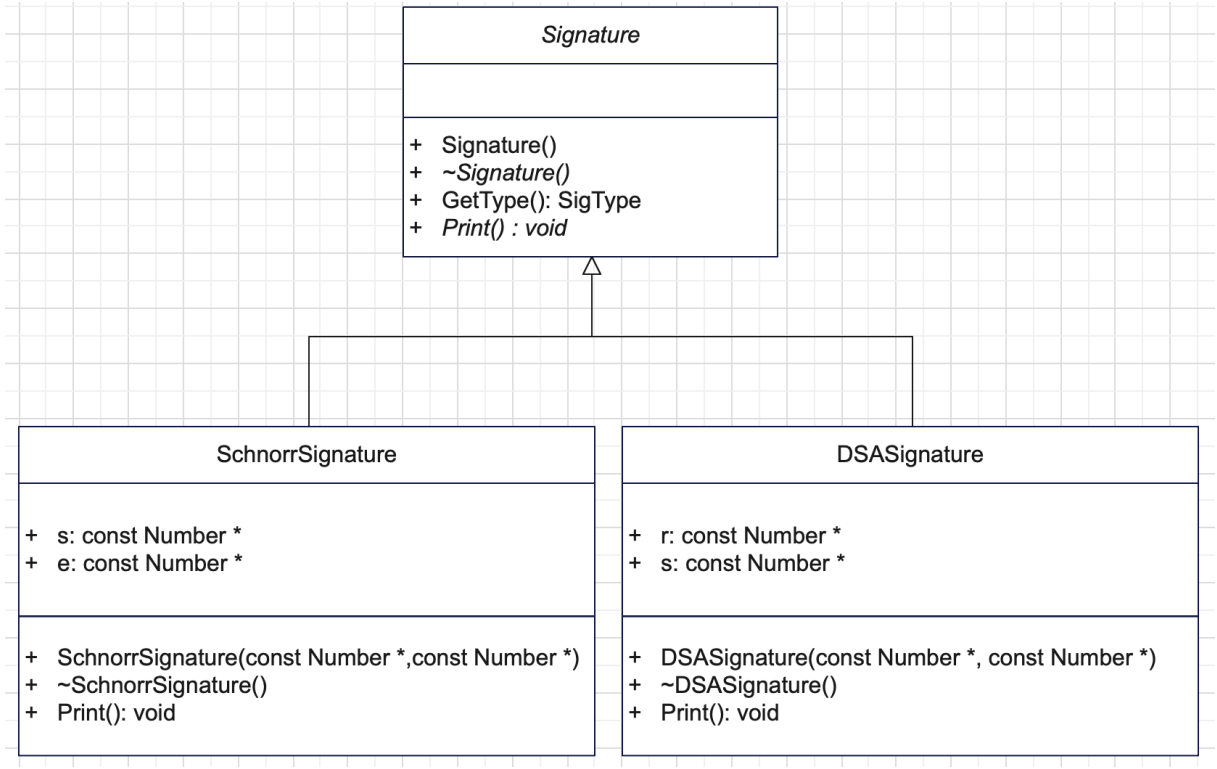
- Do NOT try your "luck" - we use some sophisticated plagiarism detection software to find cheaters. It is much better than most students think. It has been proven times and times again tricks didn't work. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment - it is much more than that. It is simply not worth to cheat at all. You would hurt your friend and yourself by doing that. It is obvious that a real friend won't ask you to get involved in a plagiarism act in any way due to the consequences. Read the Honor Code again before you even try to think about cheating.
- Serious offenders will fail the course immediately and there may be additional disciplinary actions from the department and university.

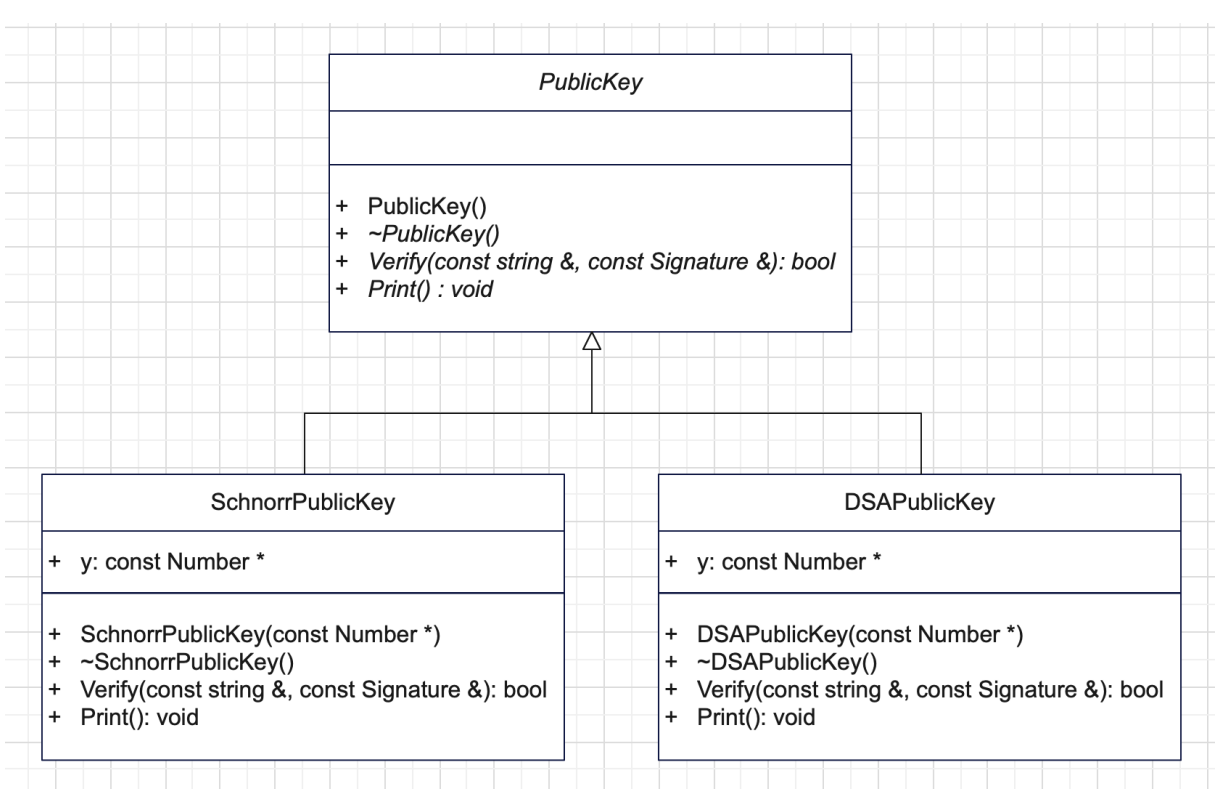
## Description

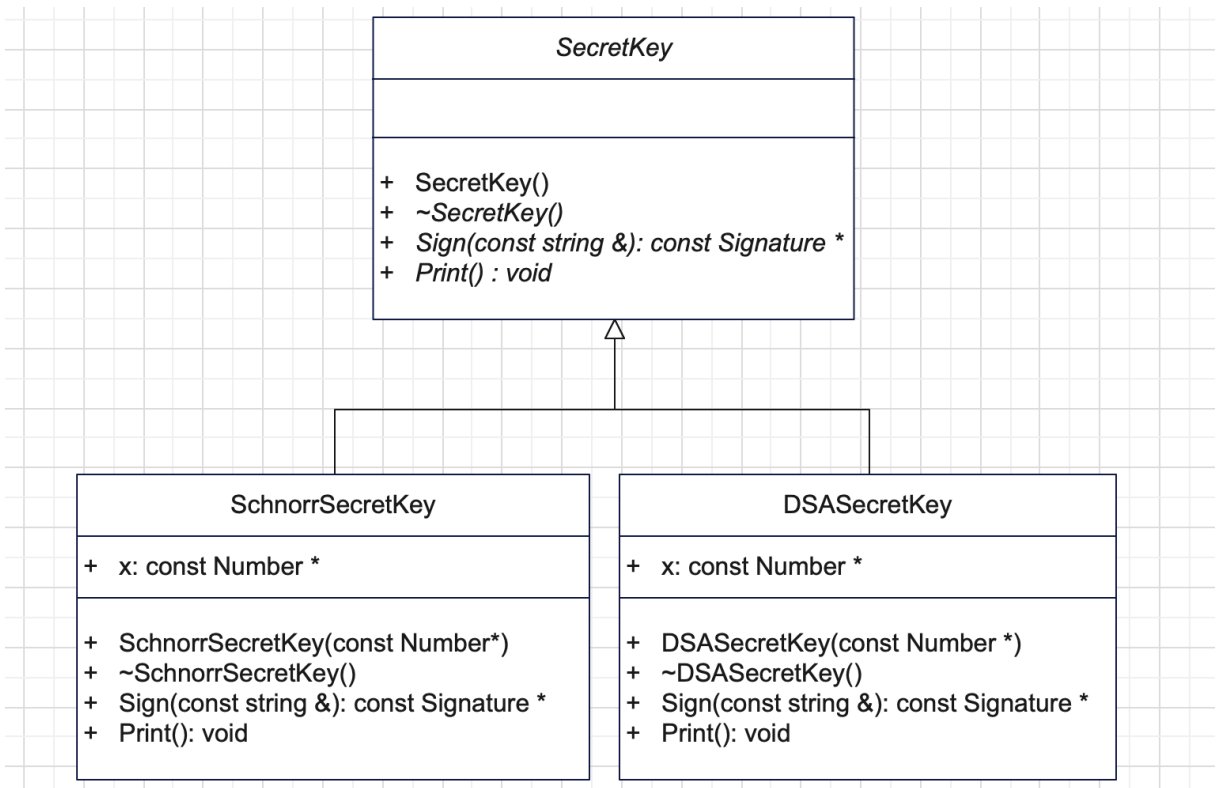
You must finish this assignment with the provided files. Download them in the Download section.

Your task is to complete all the missing parts in `signatures.cpp` and `signatures.h` according to the instructions below.

## Important Signature-related Classes




In this assignment, we implement toy versions of DSA digital signature and Schnorr digital signature.

There are 9 different classes given to you with the skeleton code:

The signed signature of a message is modelled under the `Signature` class, which has two derived classes: (in signatures.cpp/.h)

- Abstract base class `Signature`
- `DSASignature`
- `SchnorrSignature`

These objects hold the actual signature.

As discussed before, a user needs a secret signing key and a public key in order to be able to sign and verify messages respectively.

- Abstract base class `SecretKey`
- `DSASecretKey`
- `SchnorrSecretKey`

`SecretKey` class provides the `Sign()` function necessary to create a signature on a message.

- Abstract base class `PublicKey`
- `DSAPublicKey`
- `SchnorrPublicKey`

`PublicKey` class provides the `Verify()` function necessary to verify a signature of a message.

Note that some functions in the abstract base classes should be virtual.

## Number Class

In DSA and Schnorr, the public/secret keys and signatures consist of integers of many bits (1024 or 256). They cannot be represented using common integer types such as `int64` (signed integer type with width of exactly 64 bits). GMP Library offers arbitrary precision numbers, but is difficult to install on Windows. Therefore we use `int64` to represent all numbers and cut down the bits of public/secret keys and signatures.

We implement all necessary operations and functions of number in the class `Number`, in `number.h/number.cpp`. These functions and parameters of p, q, g are static functions/variables. Please check self-study notes in course website on static functions if you are not familiar with its usage.

- `Number::Add(n1, n2)`: integer addition. Return a `Number` that holds the sum of n1 and n2.
- `Number::Sub(n1, n2)`: integer subtraction. Return a `Number` that holds the difference of n1 and n2.
- `Number::Neg(n)`: integer negation. Return a `Number` that holds the negation of n.
- `Number::Mul(n1, n2)`: integer multiplication. Be cautious about integer overflow. Return a `Number` that holds the product of n1 and n2.
- `Number::Mod(n, p)`: integer modulo. Returns a `Number` that holds the integer $n \bmod p$. Almost every operation in this assignment is modular operation. Please use this to get correct results and avoid integer overflow. If you don't know what is integer overflow, please check integer overflow.
- `Number::MulMod(n1, n2, p)`: apply modulo operation after multiplication.
- `Number::Pow(base, exp, mod)`: Return a `Number` that holds the integer (base to the power of exp) modular mod. Notice that the exponent should be a non-negative number.
- `Number::Inv(n, mod)`: modulo inverse (mod should be a prime number). Return a `Number` that holds the unique integer $n^{-1}$ such that $n \times n^{-1} = 1 \bmod mod$.
- `Number::Rand(lower, upper)`: Return a `Number` that holds a random value in {lower, lower+1, ..., upper-1}. This used to generate keys and create signatures.
- `Number::NSign(n)`: If n > 0, return 1. If n=0, return 0. If n < 0, return -1. Use this function to compare two Numbers.
- `Number::Hash()`Return the first 16 bits of SHA256 cryptographic hash function applied on the input. The input can be `std::string`, or `Number`, or `(Number, std::string)`. In the last case, `Hash(Number, std::string)`, the input is the concatenation of the bits of `Number` and the bits of `std::string`.

In `gmp_number.h/gmp_number.cpp`, the Number class is implemented based on GMP library, thus providing complete DSA/Schnorr algorithms. You can install GMP yourself and change the codes/filenames to experiment with gmp number, but **do not use GMP in your submission**.

## Users and Admins

You don't need to read into `user.h/user.cpp/admin.h/admin.cpp` to complete the assignment. STL is used in class `User` and `Admin`, you are not required to write STL operations yourself. Knowing the usage of STL might help you understand the whole program. If might also help you debug. You can check the vector of STL in self-study notes on website. If you want to know more, you can search online for any tutorial or document.

Each `User` has a name, a pair of DSA public/secret keys and a pair of Schnorr public/secret keys. A `User` can create a signature for a message and send both the message and the signature to another `User` by calling `SendMessage()` function. The `Admin` delivers the message to the other `User`. The `ReceiveMessage()` function of the other `User` is called to verify the signature.

The `Admin` holds the public information of all users: username, pointer to `User` object, DSA/Schnorr public keys. The admin acts as both communication network and Certificate Authority (CA).

End of Description

# Tasks

Be sure to read Digital Signatures protocol for the full specification of the functions you are to implement.

## 1) Complete header file signatures.h

Specifically, you would have to complete the class headers for the following classes:

- `Signatures`
- `SchnorrSignature`
- `DSASignature`
- `PublicKey`
- `DSAPublicKey`
- `SecretKey`
- `SchnorrSecretKey`

You can refer to the class diagram above in adding the missing member functions. Please note that you will also need to decide if the functions have to be `virtual` or not.

## 2) Define and implement the neccessary constructors and destructors for the following classes in `signatures.cpp`

- `Signature`
- `SchnorrSignature`
- `SchnorrSecretKey`
- `SchnorrPublicKey`
- `DSASignature`
- `DSASecretKey`
- `DSAPublicKey`

For destructors, make sure you release dynamically allocated object properly and produce the correct cout output.

## 3) Complete `DSAPublicKey::Verify` and `SchnorrPublicKey::Verify` functions in `signatures.cpp`

## 4) Complete `SchnorrSecretKey::Sign` function in `signatures.cpp`

## 5) Complete Schnorr Key Generation under the global `GenerateKey` function in `signatures.cpp`

End of Tasks

# Compilation and Debug

After you complete your code, you should be able to run the program using the following commands:

```
> make
> ./bin/main
```

The expected output is :

```
P=669320e7
Q=48b9
G=414245a8

User information:
Username: Alice
DSA Public Key is: 5f2d63e5
Schnorr Public Key is: 57835ec
Username: Bob
DSA Public Key is: 2039f51d
Schnorr Public Key is: 48af7ba5

===> Alice send message to Bob using DSA
Bob: Receive Message from Alice
message=hello
signature: r=194b, s=14b
Verification passed

===> Bob send message to Alice using Schnorr
Alice: Receive Message from Bob
message=hi
signature: s=1a6e, e=4024
Verification passed

===> Alice send malicious message to Bob using DSA
Destruct DSASignature...
Bob: Receive Message from Alice
message=hello
signature: r=1b0b, s=16f5
Verification failed

===> Bob send malicious message to Alice using Schnorr
Destruct SchnorrSignature...
Alice: Receive Message from Bob
message=hi
signature: s=3db, e=3760
Verification failed

Destruction===
Destruct Admin...
Destruct User Alice...
Destruct SigPair...
Destruct DSAPublicKey...
Destruct DSASecretKey...
Destruct SigPair...
Destruct SchnorrPublicKey...
Destruct SchnorrSecretKey...
Destruct DSASignature...
Destruct DSASignature...
Destruct User Bob...
Destruct SigPair...
Destruct DSAPublicKey...
Destruct DSASecretKey...
Destruct SigPair...
Destruct SchnorrPublicKey...
Destruct SchnorrSecretKey...
Destruct SchnorrSignature...
Destruct SchnorrSignature...
```

We use hex (base 16) form of integers. If you don't know hex integers and want more information, you can check Hexadecimal.

Notice that `GenerateKey` and `Sign` functions require random number. If you use `Number::Rand` function differently, your implementation might generate different keys and signatures. Please make sure that you get the exact same results as the expected output, by using `Number::Rand` only in `Schnorr::Sign` and Schnorr GenerateKey, similar to the provided implementation of DSA counterparts.

- Please read the honor code and strictly follow it.
- You should start with the given skeleton code.
- Read the "Specification Clarification" and "Reported Bugs" section for some common clarifications. You should check that a day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work then.
- We will be grading your code on a Linux machine. Filenames are case sensitive in Linux.
- You are not allowed to use STL, global variables, static variables, or any additional libraries.
- You need to clean up your memory properly.
- Remove your cin/cout debug output in your submission.

End of Compile and Debug

## Download

Digital Signatures protocol
It gives the details of how the keys should be generated, how the signing and verification are done for the DSA and Schnorr Algorithm.

Skeleton Code for Mac/Linux - v2 or Skeleton Code for Windows - v2
The files and folders in the zipped file are:

- **Makefile** - It is the makefile for compilation.
- **src** - All the source files (.cpp) can be found in this folder.
- **include** - All the header files (.h) can be found in this folder.
- **bin** - The executable will be stored in this folder and the object files will be stored in the sub-folder, **obj**.

End of Download

## Memory Leak

Memory leak checking is done via the `-fsanitize=address,leak,undefined` option (related documentation here) of a recent g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to "Your Output" tab in the test case details popup) for errors such as memory leak. Other errors/bugs such as out-of-bounds, use-after-free bugs, and some undefined-behavior-related bugs may also be detected. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leak yourself using the same options, you may follow our Checking for memory leak yourself guide.

End of Memory Leak

## Grading

1. If your program can be compiled, you will receive 10%. This is testcase 1 (given).
2. If you correctly implement Schnorr `GenerateKey`, you will receive 10%. This is testcase 2 (given).

3. If your program can output the destruction messages correctly when there is no message between users, you will receive 10%. There are two test cases, 1 given and 1 hidden, each counting 5%.
4. If your program can output the destruction messages correctly without memory leakage when there is no message between users, you will receive 10%. There are two test cases, 1 given and 1 hidden, each counting 5%.
5. If you correctly implement `SchnorrSecretKey::Sign`, you will receive 10%. There are five test cases, 2 given and 3 hidden, each counting 2%.
6. If you correctly implement `SchnorrPublicKey::Verify`, you will receive 10%. This is a hidden test case. It uses GenerateKey and Sign implemented by TA to create a few signatures, and change the some of the signatures. Then your `SchnorrPublicKey::Verify` is called on these signatures. If your function can return the correct boolean results without runtime error, you will pass the test case. There are five test cases, 2 given and 3 hidden, each counting 2%.
7. If you correctly implement `DSAPublicKey::Verify`, you will receive 10%. There are five test cases, 2 given and 3 hidden, each counting 2%.
8. If your program can output the expected output shown on webpage, you will receive 10%. There are 2 testcases 80 and 81 (given). Testcase 81 checks memory leakage.
9. If your program can output correct results on hidden test case of multiple messages (similar to the provided case), you will receive 10%. This is testcase 9 (hidden).
10. If your program can output correct results on hidden test case of multiple messages without memory leakage, you will receive 10%. This is testcase 10 (hidden).

Currently, the given test cases on Zinc are for {1, 2, 30, 40, 50, 51, 60, 61, 70, 71, 8}.
After the deadline, there will be additional test cases for {31, 41, 52, 53, 54, 62, 63, 64, 72, 73, 74, 9, 10}.
As a hint, you can simulate the hidden test cases by writing test codes your self. For example, you can write codes to create an arbitrarily wrong signature and see whether your Verify can always return the correct results.

End of Grading

## Submission and Deadline

Deadline: 23:59:00 on 3 April 2022 (Sunday)

### Canvas Submission

Create a single zip file that contains only the **signature.h and signature.cpp files** with the folder structure and name it `pa2.zip`.

```
src/signatures.cpp
include/signatures.h
```

Submit the `pa2.zip` to ZINC. ZINC usage instructions can be found here.

**The filenames have to be exactly the same**. It must be a zip file, not rar, not 7z, not tar, not gz, etc. Inside the zip file, the cpp files need to have correct names. If your submissions cannot be uncompressed, they will not be graded.

Make sure your source files can be successfully compiled.If we cannot even compile your source file, your work will not be graded, and you will get zero mark for your PA2. Therefore, you should at least put in dummy implementations to the parts that you cannot finish so that there will be no compilation error.

**Make sure you actually upload the correct version of your source files - we only grade what you upload**. Some students in the past submitted an empty file or a wrong file or an exe file which is worth zero mark. So **you must download and double-check the file you have submitted**. You may refer to the illustration below.

You may submit your file multiple times, but only the last version will be graded.

Submit early to avoid any last-minute problem. Only zinc submissions will be accepted.

Note 1: If you have no idea how to create a zip file, you may see How to create a zip file in Windows 10 or How to create a zip file in Mac OS X.

End of Submission and Deadline

# FAQ

## Frequently Asked Questions

- Q: How to create a correct zip file for PA2?

  A: In an empty folder, create a "src" folder and an "include" folder. Copy the signatures.h to "include" folder and the signatures.cpp to "src" folder. Then select these two folders and compress.

- Q: How to access derived class members via a base class pointer?

  A: Refer to polymorphism.

- Q: How to write test cases ourselves to make sure we are correct in hidden test case.

  A:

  - Try to add more users, and send more messages. See whether the cout output is expected. Make sure the range of output is correct. The verification results are correct.
  - Test verify() function. Generate a invalid signature by giving two arbitrary numbers to signature constructor. It is very very likely (probability ~ 1-1/q) that Verify() should return false.

- Q: Should I consider xxx for hidden test cases?

  A: You don't need to consider

  - The sender or receiver is not valid (not exist, public keys invalid, public keys not registered).
  - P, Q, G parameters are invalid.
  -

## Specification Clarification

## Reported Bugs

- Number::Mod(): C/C++ modular operator can return negative value. -7 % 3 = -1. This is the reason why some implementation of verify() produces different results in case 50 and 51. Fixed in skeleton-v2. (2022-03-29)

End of FAQ