VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

## Cấu Trúc Dữ Liệu và Giải Thuật (DSA)

**DSA1 - HK241**

# Cách Chạy Test Case

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 6/2024

# Mục lục

# 1 Unit Test

## 1.1 Lý Thuyết

### 1.1.1 Khái Niệm Cơ Bản

**Unit Test (Kiểm Thử Đơn Vị)**: Là việc kiểm tra các đơn vị nhỏ nhất của phần mềm, như hàm, phương thức hoặc lớp, để đảm bảo chúng thực hiện đúng chức năng được thiết kế.

**Unit Test Framework (Khung Kiểm Thử Đơn Vị)**: Các công cụ hoặc thư viện giúp viết, tổ chức và chạy các unit test. Ví dụ: JUnit (Java), NUnit (.NET), Catch2 (C++), pytest (Python).

**Test Case (Trường Hợp Kiểm Thử)**: Một tập hợp các đầu vào, các điều kiện, và các kết quả dự kiến để kiểm tra một đơn vị cụ thể. Mỗi test case nên kiểm tra một khía cạnh cụ thể của chức năng.

### 1.1.2 Các Đặc Điểm Chính

- **Isolation (Cô Lập)**: Các unit test phải được thực hiện trong một môi trường tách biệt để không bị ảnh hưởng bởi các yếu tố bên ngoài hoặc các phần của hệ thống chưa được kiểm tra.
- **Repeatability (Có Thể Lặp Lại)**: Các unit test nên cho kết quả nhất quán khi chạy nhiều lần trong cùng một môi trường.
- **Automation (Tự Động Hóa)**: Unit tests thường được tự động chạy bởi các công cụ hoặc hệ thống tích hợp liên tục (CI) để đảm bảo rằng mã nguồn mới không phá vỡ các chức năng hiện có.

### 1.1.3 Lợi Ích

- **Phát Hiện Sớm Lỗi**: Giúp phát hiện lỗi sớm trong quá trình phát triển, khi lỗi còn dễ sửa chữa hơn.
- **Tài Liệu Mã Nguồn**: Test cases cung cấp tài liệu cho các chức năng của mã nguồn, giúp các nhà phát triển khác hiểu cách sử dụng và các yêu cầu của mã.
- **Refactoring An Toàn**: Khi bạn thay đổi mã, các unit test đảm bảo rằng các thay đổi không phá vỡ các phần khác của hệ thống.
- **Xác Minh Mã Mới**: Khi thêm các tính năng mới, unit tests giúp xác minh rằng các tính năng mới hoạt động đúng và không gây lỗi cho các phần còn lại của hệ thống.

## 1.2 Phương Pháp Thiết Kế Unit Test

### 1.2.1 Phương Pháp Equivalence Partitioning (Phân Vùng Tương Đương)

**Mục Đích**: Phương pháp này nhằm chia đầu vào của một hàm hoặc hệ thống thành các nhóm (hoặc vùng) mà các đầu vào trong cùng một nhóm có cùng tính chất. Mục tiêu là để giảm số lượng các test case cần thiết bằng cách kiểm tra một đại diện cho mỗi nhóm. Điều này giúp đảm bảo rằng tất cả các loại đầu vào được kiểm tra mà không cần phải thử nghiệm tất cả các khả năng.

**Cách Thực Hiện**:

1. **Xác Định Các Vùng Đầu Vào**: Đầu tiên, phân loại các đầu vào thành các nhóm tương đương. Một nhóm tương đương là một tập hợp các giá trị đầu vào mà hệ thống xử lý theo cách tương tự.
2. **Chọn Đại Diện Cho Mỗi Vùng**: Lựa chọn một hoặc vài giá trị đại diện cho mỗi nhóm để kiểm tra. Những giá trị này được sử dụng để đại diện cho toàn bộ nhóm, vì vậy việc thử nghiệm các giá trị này là đủ để kiểm tra tính chính xác của phần mềm cho cả nhóm.

**Ví Dụ**: Xem xét một hàm kiểm tra số nguyên từ 1 đến 10. Các vùng đầu vào có thể bao gồm:

- **Vùng Hợp Lệ**: Số trong khoảng từ 1 đến 10 (ví dụ: 5).
- **Vùng Không Hợp Lệ**: Số nhỏ hơn 1 (ví dụ: 0) và số lớn hơn 10 (ví dụ: 11).

Thay vì kiểm tra tất cả các giá trị trong khoảng từ 1 đến 10, bạn chỉ cần chọn một vài đại diện từ mỗi vùng để kiểm tra.

### 1.2.2 Phương Pháp Boundary Value Analysis (Phân Tích Giá Trị Biên)

**Mục Đích**: Phương pháp này tập trung vào việc kiểm tra các giá trị nằm ở ranh giới của các vùng đầu vào hợp lệ hoặc không hợp lệ. Lỗi thường xảy ra ở các điểm biên của các vùng đầu vào hơn là các giá trị nằm giữa các điểm biên. Phân tích giá trị biên giúp đảm bảo rằng hệ thống xử lý đúng các giá trị ở các điểm này.

**Cách Thực Hiện**:

1. **Xác Định Các Giá Trị Biên**: Xác định các giá trị ở biên của vùng đầu vào hợp lệ và không hợp lệ.
2. **Kiểm Tra Giá Trị Tại Biên và Xung Quanh**: Kiểm tra các giá trị tại điểm biên, điểm biên +1, điểm biên -1, và các giá trị bên trong phạm vi. Điều này giúp đảm bảo rằng các điểm biên và các giá trị xung quanh chúng được xử lý chính xác.

**Ví Dụ**: Xem xét hàm nhận số nguyên từ 1 đến 10. Các giá trị biên cần kiểm tra bao gồm:

- **Biên Thấp**: 1 (biên hợp lệ) và 0 (biên không hợp lệ).
- **Biên Cao**: 10 (biên hợp lệ) và 11 (biên không hợp lệ).
- **Giá Trị Xung Quanh Biên**: 2 (gần biên thấp) và 9 (gần biên cao).

Điều này giúp phát hiện các lỗi có thể xảy ra khi xử lý các giá trị ở các điểm biên và xung quanh chúng.

### 1.2.3 Phương Pháp Thiết Kế Unit Test Dựa Trên White-Box Testing

#### 1.2.3.a Path Testing (Kiểm Thử Đường Đi)

**Mục Đích**: Đảm bảo rằng tất cả các đường đi có thể có trong mã nguồn đều được kiểm tra.

**Cách Thực Hiện**:

1. Xác định tất cả các đường đi khả dĩ trong chương trình.
2. Thiết kế các test case để kiểm tra từng đường đi hoặc nhóm các đường đi có liên quan.

**Ví Dụ**: Nếu bạn có một hàm với cấu trúc điều kiện phức tạp, hãy viết test case để kiểm tra từng nhánh của cấu trúc điều kiện đó.

#### 1.2.3.b Branch Testing (Kiểm Thử Nhánh)

**Mục Đích**: Đảm bảo rằng tất cả các nhánh trong cấu trúc điều kiện của mã đều được thực thi.

**Cách Thực Hiện**:

1. Xác định tất cả các điều kiện trong mã nguồn.
2. Thiết kế các test case để đảm bảo rằng tất cả các nhánh đều được kiểm tra.

**Ví Dụ**: Đối với câu lệnh điều kiện 'if-else', tạo test case để kiểm tra cả phần 'if' và phần 'else'.

#### 1.2.3.c Condition Testing (Kiểm Thử Điều Kiện)

**Mục Đích**: Đảm bảo rằng tất cả các điều kiện trong các câu lệnh điều kiện đều được kiểm tra.

**Cách Thực Hiện**:

1. Xác định tất cả các điều kiện trong mã nguồn.
2. Viết test case để kiểm tra các giá trị khác nhau của từng điều kiện.

**Ví Dụ**: Đối với điều kiện 'x $>$ 5', tạo test case với giá trị 'x' nhỏ hơn, bằng và lớn hơn 5.

#### 1.2.3.d Loop Testing (Kiểm Thử Vòng Lặp)

**Mục Đích**: Đảm bảo rằng các vòng lặp hoạt động đúng với các điều kiện khác nhau.

**Cách Thực Hiện**:

1. Xác định các vòng lặp trong mã nguồn.
2. Viết test case để kiểm tra các tình huống vòng lặp khác nhau.

**Ví Dụ**: Đối với vòng lặp 'for' từ 1 đến 'n', kiểm tra với 'n $= 0$', 'n $= 1$', và 'n $= 10$'.

#### 1.2.3.e Statement Testing (Kiểm Thử Câu Lệnh)

**Mục Đích**: Đảm bảo rằng tất cả các câu lệnh trong mã nguồn đều được thực thi ít nhất một lần.

**Cách Thực Hiện**:

1. Xác định tất cả các câu lệnh trong mã nguồn.
2. Thiết kế các test case để đảm bảo mọi câu lệnh đều được thực thi.

**Ví Dụ**: Đảm bảo rằng mọi câu lệnh trong một hàm đều được kiểm tra ít nhất một lần.

## 1.3 Cách Chạy Unit Test

1. **Bước 1:** #inlcude các file sẽ test vào file unit_test.hpp

```cpp
// file unit_test.hpp
#include "../memory_layout/memory_layout.hpp"
#include "../memory_layout/text/declaration.hpp"

class UnitTest
{
}
```

2. **Bước 2:** Tạo ra các hàm và registerTest trong file unit_test.hpp

```cpp
// file unit_test.hpp
class UnitTest
{
public:
  UnitTest()
  {
    UnitTest::registerTest("function1", &UnitTest::function1);
    UnitTest::registerTest("function2", &UnitTest::function2);
    UnitTest::registerTest("function3", &UnitTest::function3);
    // TODO unit new
  }

private:
  bool function1();
  bool function2();
  bool function3();
  // TODO unit new
}
```

- Nếu Tên bị trùm lặp thì sẽ lỗi

```cpp
UnitTest::registerTest("function1", &UnitTest::function1);
UnitTest::registerTest("function1", &UnitTest::function2);
```

The terminal is currently in the memory_layout folder

```
-> ./main test_unit function1
terminate called after throwing an instance of 'std::
    runtime_error'
  what():  Test with name 'function1' already exists.
[1]    83331 IOT instruction (core dumped)  ./main
    test_unit function1
```

- function tạo ra luôn có cấu trúc là **bool nameFunction();**

3. **Bước 2:** Triển Khai hàm vừa tạo bên unit_test.cpp

```cpp
// file unit_test.cpp
bool UnitTest::function1()
{
  // data
  ListDelarations list(1);
  list.push("Call VoTien");

  // expect
  string expect = "[\n\tCallDeclaration(VoTien)\n]";

  // output
  string output = list.str();

  // remove data

  // prit result
  if (expect == output)
  {
    cout << green << "test function 1 --------------- PASS" << reset << "\n";
    return true;
  }
  else
  {
    cout << red << "test function 1 --------------- FAIL" << reset << "\n";
    cout << "\texpect : " << expect << endl;
    cout << "\toutput : " << output << endl;
    return false;
  }
}
```

- **Tạo Dữ Liệu Đầu Vào**: Khởi tạo đối tượng hoặc cấu trúc dữ liệu cần thiết cho việc kiểm thử. Cung cấp đầu vào cụ thể để kiểm tra chức năng.
- **Xác Định Kết Quả Mong Đợi**: Đặt giá trị kết quả mong đợi mà bạn dự đoán được từ chức năng đang được kiểm thử. Kết quả mong đợi nên phản ánh chính xác kết quả mà bạn kỳ vọng.
- **Thực Hiện Chức Năng**: Gọi chức năng hoặc phương thức cần kiểm tra và lấy kết quả đầu ra.
- **So Sánh Kết Quả và In Kết Quả Kiểm Thử**: So sánh kết quả đầu ra với kết quả mong đợi và in kết quả kiểm thử để xem chức năng có hoạt động đúng như mong đợi hay không.

4. **Bước 4:** biên dịch chương trình dùng lệnh sau sẽ tạo ra file **main**

The terminal is currently in the memory_layout folder

```
-> g++ -o main \
    main.cpp \
    memory_layout/memory_layout.cpp \
    memory_layout/text/declaration.cpp \
    unit_test/unit_test.cpp \
    random_test/random_test.cpp
```

**Trên Ubuntu/mac/linux có thể chạy lệnh ./run.sh**

The terminal is currently in the memory_layout/run.sh file

```
#!/bin/bash

# Check if the executable 'main' already exists and remove it
if [ -f main ]; then
    echo "Removing existing executable 'main'."
    rm main
fi

# Compile the C++ program
g++ -o main \
    main.cpp \
    memory_layout/memory_layout.cpp \
    memory_layout/text/declaration.cpp \
    unit_test/unit_test.cpp \
    random_test/random_test.cpp

# Check if the compilation was successful
if [ $? -eq 0 ]; then
    echo -e "\033[36m\033[1mCompilation successful. Run ./main
       to execute.\033[0m"
    echo -e "\033[32m\033[1mterminal unit test\033[0m"
    echo -e "\033[31m./main test_unit\033[0m"
    echo -e "\033[31m./main test_unit all\033[0m"
    echo -e "\033[31m./main test_unit nameFunctionUnitTest
       \033[0m"
    echo
    echo -e "\033[32m\033[1mterminal auto test\033[0m"
    echo -e "\033[31m./main test_random number_1 number_2\033[0
       m"
    echo -e "\033[31m./main test_random number\033[0m"
else
    echo -e "\033[31m\033[1mCompilation failed.\033[0m"
fi
```

5. **Bước 5:** Thực thi trương trình

The terminal is currently in the memory_layout folder

```
Compilation successful. Run ./main to execute.
terminal unit test
./main test_unit
./main test_unit all
./main test_unit nameFunctionUnitTest
```

- **./main test_unit and ./main test_unit all** sẽ chạy tất cả các test case được registerTest
- **./main test_unit nameFunctionUnitTest** chạy 1 test case bạn chọn

## 1.4 Cách Fixbug Unit Test

1. **Đúng tất cả test case khi chạy ./main test_unit**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test_unit
Running all unit tests: ----------
test function 1 --------------- PASS
test function 2 --------------- PASS
test function 3 --------------- PASS

Result -------------------------
All tests passed!
```

2. **Sai một số test case khi chạy ./main test_unit**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test_unit
Running all unit tests: ----------
test function 1 --------------- PASS
test function 2 --------------- FAIL
- expect : [
        InitDeclaration(VoTien)
]
- output : [
        CallDeclaration(VoTien)
]
test function 3 --------------- FAIL
- expect : [
        DeleteDeclaration(VoTien)
]
- output : [
        CallDeclaration(VoTien)
]

Result -------------------------
Some tests failed:  function2  function3
Pass rate: 33.3333%
```

- **Expect:** Kết quả mong đợi từ chương trình để kiểm tra tính đúng đắn.
- **Output:** Kết quả thực tế mà chương trình trả về sau khi chạy test.
- **Tests Failed:** Danh sách các bài kiểm tra không thành công vì kết quả không khớp với mong đợi.
- **Pass Rate:** Tỷ lệ phần trăm số bài kiểm tra thành công trên tổng số bài kiểm tra.

3. **đúng test đã chọn khi chạy ./main test_unit nameFunctionUnitTest**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test_unit function1

Running unit test: function1 ----------
test function 1 --------------- PASS
```

4. **Fail test đã chọn khi chạy ./main test_unit nameFunctionUnitTest**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test_unit function1

Running unit test: function2 ----------
test function 2 --------------- FAIL
- expect : [
        InitDeclaration(VoTien)
]
- output : [
        CallDeclaration(VoTien)
]
```

5. **Fail name test đã chọn khi chạy ./main test_unit nameFunctionUnitTest**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test_unit function1

Running unit test: function4 ----------
terminate called after throwing an instance of 'std::
    runtime_error'
  what():  Test with name 'function4' does not exist.
[1]    10087 IOT instruction (core dumped)  ./main test_unit
    function4
```

## 1.5 Những lưu ý khi sử dụng Unit Test

Việc sử dụng Unit Test là một phương pháp quan trọng trong phát triển phần mềm, giúp đảm bảo tính đúng đắn của các đoạn mã trước khi tích hợp chúng vào hệ thống lớn hơn. Khi viết và sử dụng Unit Test, cần chú ý một số điểm sau để tăng tính hiệu quả và dễ dàng phát hiện lỗi:

1. **Sử dụng `friend class` để kiểm tra các hàm private:**
   Trong một số trường hợp, bạn cần kiểm tra các hàm hoặc thành phần dữ liệu được khai báo dưới dạng `private` trong lớp (class). Một giải pháp phổ biến là sử dụng từ khóa `friend` để cho phép các lớp test có thể truy cập và kiểm tra trực tiếp các hàm này. Điều này giúp đảm bảo tính toàn vẹn của các thành phần private mà không cần thay đổi quyền truy cập của chúng chỉ vì mục đích test.

2. **Test các hàm nhỏ trước khi test các hàm lớn:**
   Để đảm bảo quá trình kiểm thử diễn ra thuận lợi, nên test các hàm nhỏ và độc lập trước. Sau đó, bạn có thể chuyển sang test các hàm lớn hơn có sử dụng hoặc gọi đến các hàm nhỏ này. Điều này giúp bạn dễ dàng cô lập và phát hiện lỗi nếu có sự cố xảy ra, đồng thời làm cho việc sửa lỗi đơn giản hơn vì lỗi thường xuất phát từ các hàm nhỏ.

3. **Thêm thông tin debug bằng cách in kết quả trong quá trình test:**
   Để dễ dàng theo dõi và sửa lỗi (debug) trong quá trình kiểm thử, bạn nên thêm các câu lệnh `print` để hiển thị kết quả trung gian. Những câu lệnh này sẽ giúp bạn so sánh kết quả mong đợi và kết quả thực tế ngay lập tức. Đặc biệt khi các lỗi logic hoặc các vấn đề không rõ ràng xảy ra, việc in ra những thông tin này có thể cung cấp dữ liệu quý giá để tìm ra nguyên nhân gốc rễ.

4. **Sử dụng đối tượng làm `output` thay vì chuỗi:**
   Trong nhiều trường hợp, kết quả của một unit test không cần thiết phải là một chuỗi ký tự (string). Bạn có thể sử dụng một đối tượng (object) làm đầu ra (output) và sau đó gọi hàm `str()` hoặc các phương thức tương tự để lấy ra chuỗi khi cần so sánh. Điều này giúp bạn có thể dễ dàng kiểm tra các thuộc tính phức tạp của đối tượng mà chỉ cần trả về `true/false` khi so sánh, nhằm đơn giản hóa việc kiểm tra tính đúng đắn của kết quả.

# 2 Random Test

## 2.1 Lý thuyết về Random Testing

Random Testing, hay còn gọi là kiểm thử ngẫu nhiên, là một phương pháp kiểm thử phần mềm trong đó các đầu vào được tạo ra ngẫu nhiên để kiểm tra tính đúng đắn của phần mềm. Đây là một kỹ thuật quan trọng trong việc phát hiện lỗi và đánh giá độ ổn định của hệ thống mà không cần phải thiết kế các trường hợp kiểm thử cụ thể.

### 2.1.1 Khái niệm cơ bản

- **Nguyên tắc hoạt động:** Random Testing dựa trên nguyên tắc tạo ra các dữ liệu đầu vào một cách ngẫu nhiên và kiểm tra hệ thống với các dữ liệu này. Mục tiêu là để phát hiện lỗi trong các tình huống mà các phương pháp kiểm thử truyền thống có thể bỏ qua.
- **Không cần biết trước:** Phương pháp này không yêu cầu hiểu rõ cấu trúc nội bộ của phần mềm. Thay vào đó, nó tạo ra các đầu vào ngẫu nhiên và kiểm tra đầu ra để đảm bảo rằng phần mềm hoạt động như mong đợi.
- **Tính ngẫu nhiên:** Các đầu vào được tạo ra ngẫu nhiên có thể giúp phát hiện các lỗi tiềm ẩn mà các phương pháp kiểm thử khác không thử nghiệm. Điều này giúp đảm bảo rằng phần mềm có thể xử lý nhiều tình huống khác nhau mà không bị lỗi.

### 2.1.2 Quy trình Random Testing

1. **Tạo đầu vào ngẫu nhiên:** Sử dụng các phương pháp tạo dữ liệu ngẫu nhiên để sinh ra các đầu vào cho hệ thống. Các công cụ và thư viện có thể được sử dụng để đảm bảo rằng các đầu vào là đủ ngẫu nhiên và đại diện cho các trường hợp sử dụng khác nhau.
2. **Chạy kiểm thử:** Thực hiện kiểm thử hệ thống với các đầu vào ngẫu nhiên đã được tạo ra. Điều này có thể bao gồm việc kiểm tra các chức năng khác nhau của phần mềm để xem nó xử lý các đầu vào như thế nào.
3. **Ghi lại kết quả:** Ghi lại kết quả của quá trình kiểm thử, bao gồm cả các lỗi hoặc vấn đề được phát hiện. Điều này có thể bao gồm việc lưu lại đầu vào và đầu ra để phân tích thêm.
4. **Phân tích kết quả:** Xem xét các kết quả kiểm thử để xác định các lỗi hoặc vấn đề. Dựa trên các vấn đề phát hiện, có thể cần thực hiện các bước sửa lỗi và kiểm thử lại.

### 2.1.3 Ưu điểm và nhược điểm

- **Ưu điểm:**
  - *Đơn giản và dễ triển khai:* Không yêu cầu thiết kế các trường hợp kiểm thử phức tạp. Dễ dàng tích hợp vào quy trình phát triển phần mềm.
  - *Khám phá lỗi tiềm ẩn:* Có thể phát hiện các lỗi mà các phương pháp kiểm thử khác không thử nghiệm, đặc biệt là các lỗi không được dự đoán trước.
  - *Đảm bảo đa dạng:* Đầu vào ngẫu nhiên giúp đảm bảo rằng hệ thống được kiểm thử trong nhiều tình huống khác nhau.
- **Nhược điểm:**
  - *Khả năng phát hiện lỗi không đồng đều:* Các lỗi có thể không được phát hiện nếu đầu vào ngẫu nhiên không đủ đại diện cho các trường hợp sử dụng phổ biến.
  - *Không có khả năng bao quát tất cả các tình huống:* Không đảm bảo rằng tất cả các tình huống hoặc tất cả các phần của hệ thống đều được kiểm tra.
  - *Có thể cần số lượng lớn đầu vào:* Để đạt được độ tin cậy cao, có thể cần chạy rất nhiều kiểm thử với đầu vào ngẫu nhiên.

### 2.1.4 Ví dụ về Random Testing

Giả sử bạn đang kiểm thử một hàm tính toán tổng của hai số nguyên. Bạn có thể sử dụng Random Testing để tạo ra các số nguyên ngẫu nhiên và kiểm tra xem hàm tính toán tổng có chính xác không. Ví dụ:

```
import random

def add(a, b):
    return a + b

# Random Testing
def random_test():
    for _ in range(1000):  # Thực hiện 1000 lần kiểm thử
        a = random.randint(-1000, 1000)
        b = random.randint(-1000, 1000)
        expected = a + b
        result = add(a, b)
        assert result == expected, f"Failed for {a} + {b}: expected {expected}, got {result}"

random_test()
```

Trong ví dụ trên, hàm ‘random_test()‘ tạo ra các cặp số nguyên ngẫu nhiên và so sánh kết quả của hàm ‘add()‘ với kết quả mong đợi. Nếu có sự khác biệt, lỗi sẽ được phát hiện và báo cáo.

## 2.2 Cách Chạy Random Test

1. **Bước 1:** #inlcude các file sẽ test vào file random_test.hpp

```cpp
// file random_test.hpp
#include "../memory_layout/memory_layout.hpp"
#include "../memory_layout/text/declaration.hpp"

class UnitTest
{
}
```

2. **Bước 2: Tạo input** theo cấu trúc input/input_number

input/input_1

```
1
Call VoTien
```

3. **Bước 3: Tạo expected** theo cấu trúc expected/expected_number

expected/expected_1

```
[
        CallDeclaration(VoTien)
]
```

4. **Bước 4:** biên dịch chương trình dùng lệnh sau sẽ tạo ra file **main**

The terminal is currently in the memory_layout folder

```
-> g++ -o main \
    main.cpp \
    memory_layout/memory_layout.cpp \
    memory_layout/text/declaration.cpp \
    unit_test/unit_test.cpp \
    random_test/random_test.cpp
```

**Trên Ubuntu/mac/linux có thể chạy lệnh ./run.sh**

The terminal is currently in the memory_layout/run.sh file

```
#!/bin/bash

# Check if the executable 'main' already exists and remove it
if [ -f main ]; then
    echo "Removing existing executable 'main'."
    rm main
fi

# Compile the C++ program
g++ -o main \
    main.cpp \
    memory_layout/memory_layout.cpp \
    memory_layout/text/declaration.cpp \
    unit_test/unit_test.cpp \
    random_test/random_test.cpp

# Check if the compilation was successful
if [ $? -eq 0 ]; then
    echo -e "\033[36m\033[1mCompilation successful. Run ./main
       to execute.\033[0m"
    echo -e "\033[32m\033[1mterminal random test\033[0m"
    echo -e "\033[31m./main test\_random\033[0m"
    echo -e "\033[31m./main test\_random all\033[0m"
    echo -e "\033[31m./main test\_random nameFunctionUnitTest
       \033[0m"
    echo
    echo -e "\033[32m\033[1mterminal auto test\033[0m"
    echo -e "\033[31m./main test_random number_1 number_2\033[0
       m"
    echo -e "\033[31m./main test_random number\033[0m"
else
    echo -e "\033[31m\033[1mCompilation failed.\033[0m"
fi
```

5. **Bước 5:** Thực thi trương trình

The terminal is currently in the memory_layout folder

```
Compilation successful. Run ./main to execute.
terminal auto test
./main test_random number_1 number_2
./main test_random number
```

- **./main test_random number_1 number_2**: Lệnh này sẽ thực hiện kiểm thử ngẫu nhiên từ số `number_1` đến số `number_2`.
- **./main test_unit number**: Lệnh này sẽ chạy một test case cụ thể mà bạn đã chọn, xác định bởi `number`.

## 2.3 Cách Fixbug Random Test

1. **Pass tất cả test case khi chạy ./main test_random number_1 number_2**

The terminal is currently in the memory_layout folder

```
memory_layout -> ../main test\_random 1 2
Running RandomTest with numbers: 1 to 2 ----------
Test input_1 --------------- PASS
Test input_2 --------------- PASS

Result -------------------------
All tests passed!
```

2. **Fail một số test case khi chạy /main test_random number_1 number_2**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test\_random
Running RandomTest with numbers: 1 to 2 ----------
Test input_1 --------------- PASS
Test input_2 --------------- FAIL

Result -------------------------
Some tests failed:  input_2
Pass rate: 50%
```

- **File expected_number in random_test/expected:** Kết quả mong đợi từ chương trình để kiểm tra tính đúng đắn.
- **File output_number in random_test/output:** Kết quả thực tế mà chương trình trả về sau khi chạy test.
- **Tests Failed:** Danh sách các bài kiểm tra không thành công vì kết quả không khớp với mong đợi.
- **Pass Rate:** Tỷ lệ phần trăm số bài kiểm tra thành công trên tổng số bài kiểm tra.

3. **Pass test đã chọn khi chạy ./main test_unit nameFunctionUnitTest**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test\_random 1

Running RandomTest with single number: 1 ----------
Test input_1 --------------- PASS

Result -------------------------
All tests passed!
```

4. **Fail test đã chọn khi chạy ./main test_random number**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test\_random 2

Running RandomTest with single number: 2 ----------
Test input_2 --------------- FAIL

Result -------------------------
Some tests failed:  input_2
Pass rate: 0%
```

5. **Fail name test đã chọn khi chạy ./main test_unit nameFunctionUnitTest**

The terminal is currently in the memory_layout folder

```
memory_layout -> ./main test\_random 5 6

Running RandomTest with numbers: 5 to 6 ----------
Error: Could not open file random_test/input/input_5.txt for
   reading.
Invalid number argument: stoi
```