Lập trình nâng cao
(tài liệu bổ sung cho nội dung online)

Bài 0. Cưỡi ngựa xem hoa
- Vector<>
- Struct - Sử dụng và tạo kiểu dữ liệu mới
- Địa chỉ bộ nhớ
- Con trỏ
- Hàm - Truyền tham số (tham trị, tham biến, địa chỉ)
- char* và const char *

Bài 1. Chương trình SDL đầu tiên
- Cài thư viện SDL2
- Chuẩn bị CodeBlocks
  - Kiểm tra setting trình biên dịch g++
- Setup project dùng SDL2
- Setup SDL cho project:

Bài 2. Sử dụng thư viện hiển thị ảnh SDL2_image

Bài 3. Project có nhiều file
- Bài 3a. Tách tiếp graphics thành graphics.h và graphics.cpp (optional)

Bài 4. Event chuột

Bài 5. Game Tictactoe

Bài 6. Event bàn phím

Bài 7a. Hoạt hình - Vật di chuyển

Bài 7. Hoạt hình - Background trôi

Bài 8. Hoạt hình - Nhân vật hoạt động - Sprite

Bài 9. Âm thanh và nhạc nền

Bài 10. Hiển thị text

Bài 11. Xử lý va chạm

Phụ lục 0. Simplified Flappy Bird

Phụ lục 1. GitHub và GitHub Desktop

Bài 0. Cưỡi ngựa xem hoa

Các bạn sẽ bắt đầu viết chương trình lớn từ … hôm nay, sẽ học nhiều thứ mới cũng từ hôm nay. Tuy nhiên, có rất nhiều cấu trúc ngôn ngữ mà bạn có thể chưa học hoặc chưa thạo.
Bài học này giúp bạn ôn tập và/hoặc làm quen với một số khái niệm ngôn ngữ quan trọng, như thể vốn từ cơ bản để bạn có thể đọc được một tiểu thuyết văn học bằng tiếng Anh. Các chủ đề sau của môn học sẽ đi vào chi tiết nền tảng các khái niệm đó, như thể khi bạn có hiểu biết về lịch sử và văn hóa của một quốc gia thì sẽ hiểu được sâu sắc hơn các tầng ngữ nghĩa của câu chuyện văn học lấy ngữ cảnh là quốc gia đó.

Trong khi môn học của chúng ta chưa đi vào cụ thể từng chủ đề này, các bạn hãy tạm "ngó qua" cho biết, và khi cần hãy quay lại "tra".

Vector<>

https://www.geeksforgeeks.org/vector-in-cpp-stl/

Vector là cấu trúc dữ liệu dạng mảng dễ sử dụng và rất thông dụng. Các bạn có thể xem và học theo chương trình mẫu tại link trên.

Struct - Sử dụng và tạo kiểu dữ liệu mới

Struct là cơ chế cho phép chúng ta tạo ra kiểu dữ liệu tích hợp từ nhiều mẩu dữ liệu thuộc các kiểu dữ liệu cơ bản. Ta có thể "gói" các mẩu dữ liệu có liên quan đến nhau vào trong một biến, để mô hình hóa một khái niệm nào đó.

Bạn có string là một kiểu dữ liệu tích hợp có sẵn trong thư viện chuẩn. Bạn có thể tạo ra các kiểu dữ liệu mới theo nhu cầu sử dụng. Ví dụ để mô hình hóa một nhân vật trong game cần đến nhiều hơn 1 biến, ta có thể định nghĩa kiểu dữ liệu dạng struct để "gói" các biến lưu dữ liệu về mỗi nhân vật game vào một chỗ cho dễ quản lý.

```
#include <iostream>

using namespace std;

struct Monster {   // định nghĩa kiểu dữ liệu Monster
    string name;   // mỗi biến kiểu Monster có 03 mẩu dữ liệu name, x và y.
    int x;         // mỗi mẩu dữ liệu này gọi là một trường (field)
    int y;

    void moveNorth() {     //có hàm (hàm thành viên) để sửa dữ liệu của biến
        x -= 10;              // hàm này sửa x
    }
};                  //nhớ kết thúc bằng dấu chấm phảy

int main(int argc, char* argv[])
{
    Monster alex;       // khai báo một biến alex thuộc kiểu Monster
    alex.name = "Alex";     // gán giá trị cho các trường của biến alex
    alex.x = 100;
    alex.y = 101;

    alex.moveNorth();       //gọi hàm của alex
                            // đọc giá trị các trường của alex
    cout << alex.name << " is at (" << alex.x << "," << alex.y << ")";

    return 0;
}
```

—----------

Gợi ý thử nghiệm:
Hãy sửa chương trình trên để thêm một biến `ben` thuộc kiểu Monster. Thử gán trị "Ben" cho trường `name` của `ben`, gán tọa độ x y nào đó cho `ben`, và in ra màn hình. Hãy dùng các lệnh cout để thấy `ben.name` và `alex.name` chẳng hạn là hai biến độc lập. Thử gán `ben = alex` xem chuyện gì xảy ra với name, x, y của hai biến đó.
Có ai thắc mắc vì sao không dùng **class** không? Câu trả lời là: như nhau cả, chỉ là do mới học thì struct đơn giản hơn thôi. Các bạn đọc code ở đâu đó thấy **class Monster** thay vì **struct Monster** thì cứ xem như hai thứ tương đương nhé.

Địa chỉ bộ nhớ

Địa chỉ Chúng ta định danh các biến trong chương trình bằng tên biến. Nhưng ở bên dưới, mỗi biến được lưu trong bộ nhớ tại một địa chỉ nào đó (hình dung phòng chiếu phim với các ghế được đánh số. Số ghế như là địa chỉ bộ nhớ, còn người ngồi tại ghế nào thì như là giá trị của ô nhớ đó trong bộ nhớ).
Bạn có thể thử xem một biến có địa chỉ gì bằng cách in địa chỉ của biến đó ra màn hình. Ví dụ:

```
Monster alex;
cout << "Memory address of alex is " << &alex << endl;
cout << "Memory address of alex.x is " << &alex.x << endl;
int x = 1;
cout << "Memory address of x is " << &x << endl;
```

Phép toán & cho chúng địa chỉ của biến. Khi in ra như trên, chúng ta được biểu diễn của địa chỉ theo hệ cơ số 16. Chạy đoạn chương trình trên ở các máy khác nhau có thể ra kết quả hơi khác một chút. Kết quả in ra màn hình đại loại có dạng:

```
Memory address of alex is 0x61fde0
Memory address of alex.x is 0x61fe00
Memory address of x is 0x61fddc
```

Bạn có thấy ba biến có địa chỉ khác nhau và gần gần nhau?
Hãy thử in địa chỉ của các biến khác thuộc loại dữ liệu khác. Nhớ rằng biến mới có địa chỉ, giá trị hằng số thì không. Thử xem.

Con trỏ

Đôi khi, ta cần lưu địa chỉ của các biến và thỉnh thoảng cần tính toán bằng địa chỉ bộ nhớ. Con trỏ là kiểu dữ liệu dành cho việc đó.

```
Monster alex;
Monster* pMonster = &alex;   // con trỏ pMonster trỏ tới biến alex
cout << "Memory address of alex " << pMonster << endl;

int x = 1;
int* p1 = &x;             // con trỏ p1 trỏ tới biến x
```

```
    int* p2; p2 = &alex.x;        // con trỏ p2 trỏ tới biến alex.x
    cout << "Memory address of x is " << p1 << endl;
    cout << "Memory address of alex.x is " << p2 << endl;
```

pMonster là biến con trỏ đang có giá trị là địa chỉ của alex. p1 là biến con trỏ đang có giá trị là địa chỉ của x. p2 là biến con trỏ đang có giá trị là địa chỉ của alex.x.
Kết quả in ra cũng tương tự nhưng hơi khác một chút

```
Memory address of alex 0x61fdd0
Memory address of x is 0x61fdcc
Memory address of alex.x is 0x61fdf0
```

Thử nghiệm: Bạn hãy tìm cách in địa chỉ của các biến con trỏ trong đoạn code trên ra xem sao.

Làm gì với con trỏ? Ta dùng nó để đọc/ghi dữ liệu mà nó trỏ tới. Ví dụ:

```
*p1 = *p1 + 1;   // tương đương với x = x + 1 khi p1 đang trỏ tới x
*p2 = *p2 + 1;   // giống alex.x = alex.x + 1 khi p2 đang trỏ tới alex.x
```

Chú ý nhé, **p1** là biểu thức con trỏ chứa địa chỉ (hiện là của x); còn ***p1** là biểu thức tương đương biến x (mà p1 đang trỏ tới).

pMonster trỏ tới alex thì sao? Hoàn toàn tương tự. **pMonster** là biểu thức con trỏ chứa địa chỉ (hiện là của alex); còn ***pMonster** là biểu thức tương đương biến alex (mà pMonster đang trỏ tới).
Bạn thử dùng biểu thức **(*pMonster).x** thay thế cho **alex.x** xem.

Hơi nhiều cú pháp, nhưng C++ nó thế, biết làm sao! Thêm một cú pháp nữa:
Thay vì **`(*pMonster).x`**, bạn có thể dùng cách viết tương đương là **`pMonster->x`**

Hàm - Truyền tham số (tham trị, tham biến, địa chỉ)

Ta có hai dạng nhu cầu khi truyền tham số vào một hàm:
(1) truyền dữ liệu vào để dùng cho các tính toán bên trong hàm
(2) truyền biến vào để lấy kết quả tính toán của hàm vào biến đó

Với nhu cầu chỉ truyền dữ liệu vào hàm, ta có cách truyền tham số là giá trị (truyền tham trị).
Với nhu cầu lấy kết quả tính toán từ bên trong hàm, nói cách khác là truyền biến vào và muốn sửa biến từ bên trong hàm, ta có hai cách làm: truyền tham biến, truyền địa chỉ.

Truyền bằng giá trị - pass by value

Khi đó, có thể truyền biến (thực ra chỉ là giá trị của biến), hoặc giá trị hằng số hay biểu thức (thực chất là giá trị của biểu thức) vào tham số đó. Nếu truyền biến vào làm tham số, nó sẽ không bị ảnh hưởng bởi hoạt động của hàm. Ví dụ, hãy thử đoạn code sau

```
int triple(int x) {
  x = x + x + x;
  return x;
}
…
int a = 10;
cout << triple(a) << “ “ << triple(1) <<  endl;
cout << triple(1+a) <<  “ “ << triple(1+1) << endl;
cout << “a = “ << a;
```

Để ý cú pháp của tham số được truyền bằng giá trị, triple(**int x**), nó gồm tên kiểu dữ liệu và tên tham số.
Để ý xem hàm triple được gọi với các kiểu tham số nào, và giá trị của a như thế nào sau khi nó được truyền vào làm tham số hàm?
Giá trị của a đã được truyền vào hàm và biến a không bị hàm sửa giá trị.

Truyền bằng tham chiếu - pass by reference
Khi ta khai báo tham số hàm là tham chiếu, hàm sẽ sửa được biến truyền vào. Ví dụ, hãy thử sửa hàm triple để thêm dấu tham chiếu (&) vào khai báo tham số x. Sau khi thêm dấu tham chiếu, chuyện gì xảy ra với đoạn code gọi hàm triple?

```
int triple(int& x) {
  x = x + x + x;
  return x;
}
…
int a = 10;
cout << triple(a) << “ “ << triple(1) <<  endl;
cout << triple(1+a) <<  “ “ << triple(1+1) << endl;
cout << “a = “ << a;
```

Các lời gọi hàm dùng đối số là giá trị hay biểu thức (tóm lại là đối số không phải biến) đều bị lỗi cú pháp. Sau khi sửa đi cho hết lỗi cú pháp, bạn sẽ thấy triple(a) cho kết quả là 30, và sau lời gọi hàm đó thì giá trị của a in ra cũng là 30 (đã bị thay đổi). Biến a đã được truyền vào hàm, và khi hàm sửa tham số x thì cũng là sửa giá trị của a. Biến x chẳng qua là một tham chiếu của a.

Truyền bằng địa chỉ (hay truyền bằng con trỏ)
Ta viết lại hàm triple như sau:

```
int triple(int* px) {
  *px = *px + *px + *px;
  return *px;
}
```

Lần này tham số là một con trỏ tới một biến. Và ở trong hàm ta dùng con trỏ đó để đọc và ghi trực tiếp biến mà con trỏ trỏ tới.

```
cout << triple(&a) << endl;
cout << "a = " << a;
```

Để gọi hàm triple mới này cho biến a, ta phải truyền địa chỉ của a vào cho hàm. Với lời gọi triple(&a), con trỏ px của hàm này sẽ nhận giá trị là địa chỉ của a. Và tất nhiên, kết quả hàm triple sửa trực tiếp biến a từ 10 thành 30.

Chú ý cẩn thận không nhầm:
Khi viết **int& x** thì x là tham chiếu kiểu int, không liên quan gì đến địa chỉ hay con trỏ.
Khi viết **p1 = &x** thì x là biến thường, &x là địa chỉ của x, còn p1 là con trỏ, không có cái gì là tham chiếu ở đây.

char* và const char *

Cái này liên quan đến con trỏ tới mảng char. Nói chung là rắc rối. Nhưng tạm thời chỉ có ngữ cảnh này hay gặp:
Nếu một hàm có tham số kiểu const char*, ví dụ:
`void greeting(const char* message) {...}`
thì bạn có thể gọi hàm đó với đối số là các kiểu xâu ký tự, trong đó có hằng xâu ký tự, ví dụ
`greeting("Hello world");` // thấy "Hello world" là hằng chứ không phải biến không?

Còn nếu bỏ từ khóa const đi, thành ra
`void greeting(char* message) {...}`
Thì lời gọi hàm `greeting("Hello world")` sẽ bị lỗi biên dịch.
Vì sao? "Hello World" là một thứ không được sửa. Nó đòi cam kết không bị sửa. Với từ khóa const, hàm greeting thứ nhất "cam kết" không sửa nội dung message. Hàm greeting thứ hai thì không cam kết.

Kết bài:

Đừng có đọc xuông nhé, tất cả các chủ đề trên bạn đều phải viết code để thử cho biết. Chủ đề nào chưa viết chưa chạy thì quay lại làm bây giờ. Làm xong hãy quay lại đây đọc tiếp.

Đến đây, nếu bạn nghĩ "*Hại não quá! Nhớ thế quái nào được!*", thì hoàn toàn không sao nhé. Nếu bạn học kỹ và hiểu sâu ngay lập tức từ mấy ví dụ này thì chúng ta sẽ không biết làm gì ở giờ lý thuyết trong 15 tuần tới ;). Just kidding.
Nội dung này chỉ là ***cưỡi ngựa xem hoa***. Từng chủ đề còn nhiều chi tiết, và còn nhiều chủ đề khác mà bạn phải học cẩn thận (thi cuối kỳ và kiểm tra hàng tuần nhé), thì mới đoán ra và hiểu được vì sao game của bạn chạy lung tung, vì sao đơ, lag, treo….. Bài này chỉ là để trong khi bạn đọc code mẫu trong những bài tiếp theo mà gặp phải những đoạn cú pháp không hiểu gì,

thì quay lại đây và biết phải đọc lại ở chỗ nào. Còn nếu bạn không muốn đọc lại ở đây mà đi hỏi luôn, ở Piazza hoặc trên lớp chẳng hạn, thì càng tốt nhé!

Bài 1. Chương trình SDL đầu tiên

***Lưu ý**: Các đường dẫn, tên file, cách cấu hình … trong hướng dẫn này dành cho project CodeBlocks, không áp dụng được cho project ở các môi trường phát triển khác, chẳng hạn Visual Code*

Cài thư viện SDL2

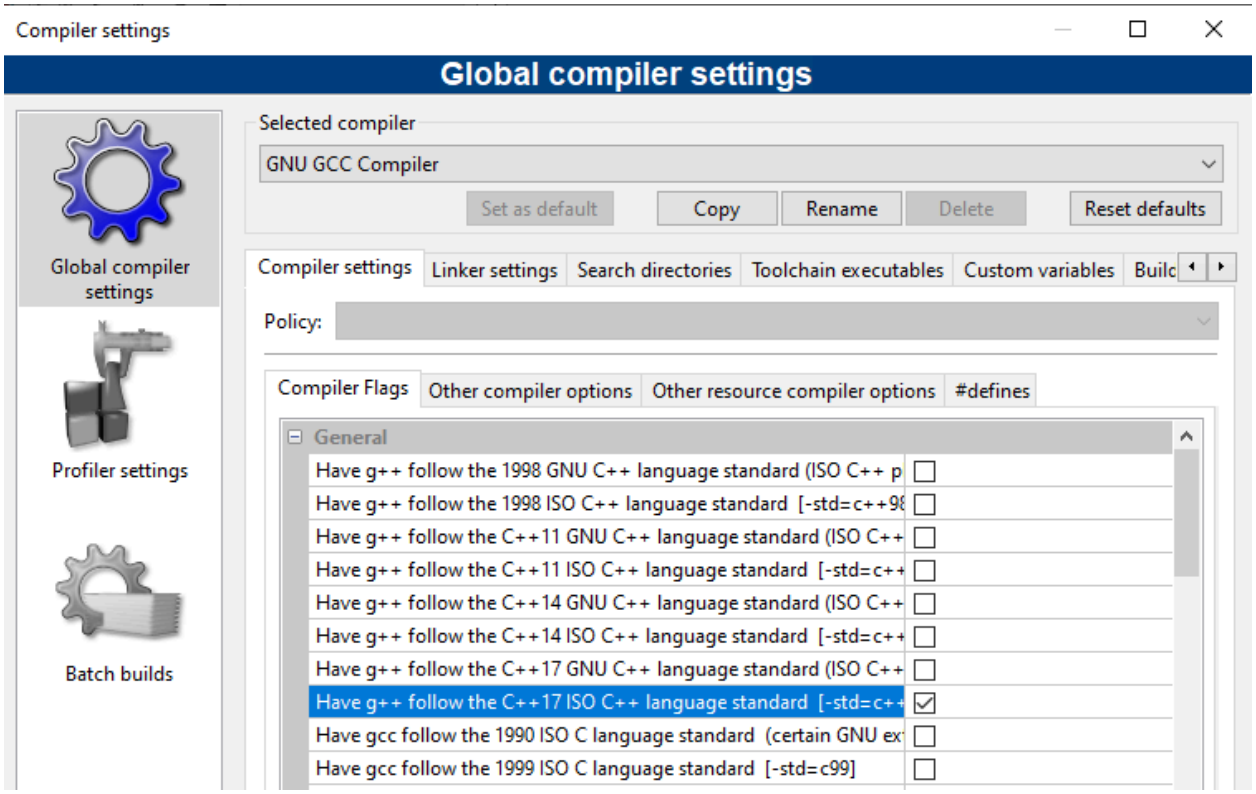
1. Download **SDL2-devel-2.28.5-mingw.zip** từ https://github.com/libsdl-org/SDL/releases/tag/release-2.28.5 hoặc một release stable cũ hơn. Điểm quan trọng là file download cần có dạng **SDL2-devel**-x.xx.x-**mingw**.xxxx thì mới có đủ nội dung cần thiết.
2. Gỡ nén, ví dụ vào F:\SDL2-2.28.5.
   Lưu ý: 1. việc double click vào file .zip hay tar.gz trong Folder view và nhìn thấy nội dung bên trong không phải là gỡ nén. Đó mới chỉ là xem trước (preview) nội dung. Bạn cần dùng Winrar hoặc chọn Extract here… Extract Files… từ context menu khi click chuột phải.
   2. Không dùng dấu cách, dấu tiếng Việt trong các thư mục nơi bạn gỡ nén thư viện SDL.
3. Để từ nay không bị nhầm phiên bản 32 và 64 bit, xóa thư mục F:\SDL2-2.28.5**i686-w64-mingw32**

Chuẩn bị CodeBlocks

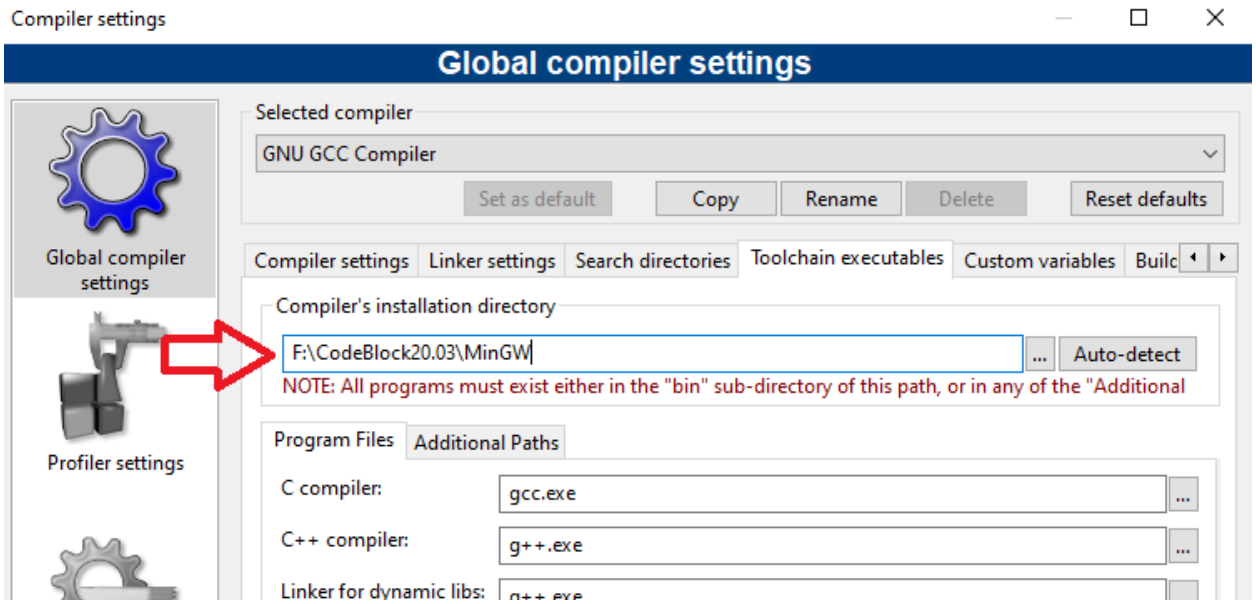
Máy có sẵn CodeBlock 20.30 hoặc cài lại (codeblocks-20.03mingw-setup.exe hoặc codeblocks-20.03mingw-nosetup.zip) từ https://www.codeblocks.org/downloads/binaries/
(Nếu không phải phiên bản trên thì uninstall bản cũ và cài lại bản mới)

Kiểm tra setting trình biên dịch g++

Bước này chỉ phải làm một lần để đảm bảo từ nay bạn sẽ dùng chuẩn C++ phù hợp, tránh các lỗi biên dịch linh tinh khó hiểu.
Dùng menu Settings | Compiler. Chọn tab Compiler Settings, Compiler Flags như hình dưới.
Uncheck tất cả các box, check duy nhất 1 box …**C++ ISO C++17…**.
(thực ra C++ 11, 14, 17 đều ổn, ISO hay GNU cũng đều được, một số hàm mới như remove_if của vector đòi chuẩn mới hơn, nhưng cũng không cần thiết cho môn học này)
Nếu bạn làm được như hình bên dưới thì bạn có thể bỏ qua phần chỉnh lỗi sau đó và bắt đầu setup project với SDL2

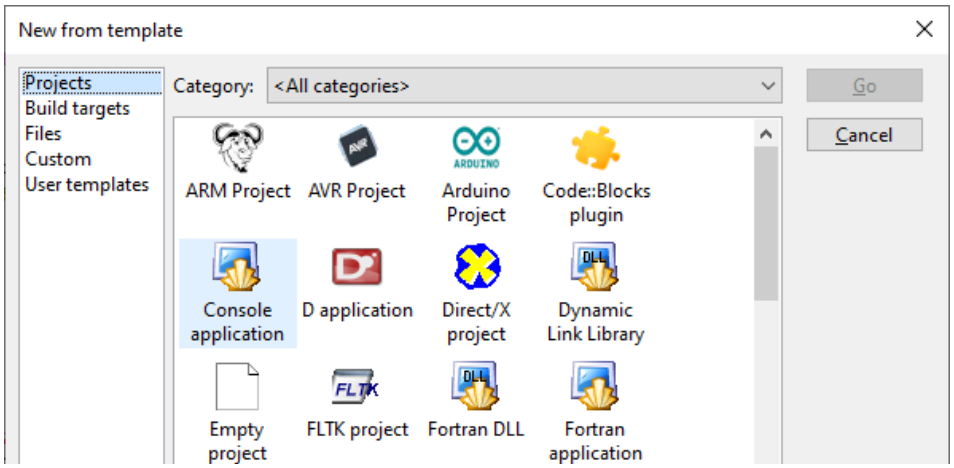


Nếu bạn không nhìn thấy các dòng C++ 11, 14, 17 như trên thì nghĩa là CodeBlock của bạn đang nối với một bộ mingw quá cũ, không phải bộ kèm theo CodeBlock 20.03. Bạn nên xóa bản mingw cũ kia đi, nó không còn hữu ích nữa. Nếu bạn đang dùng CodeBlock cổ hơn 20.03, hãy uninstall đi và cài lại bản mới. Sau đó kiểm tra để đảm bảo là nó được nối với MinGW đi kèm CodeBlock. Giả sử bạn cài CodeBlock vào thư mục F:\CodeBlock20.03\ thì Compiler's Installation directory cần trỏ tới thư mục MinGW bên trong thư mục đó. Xem hình bên dưới.

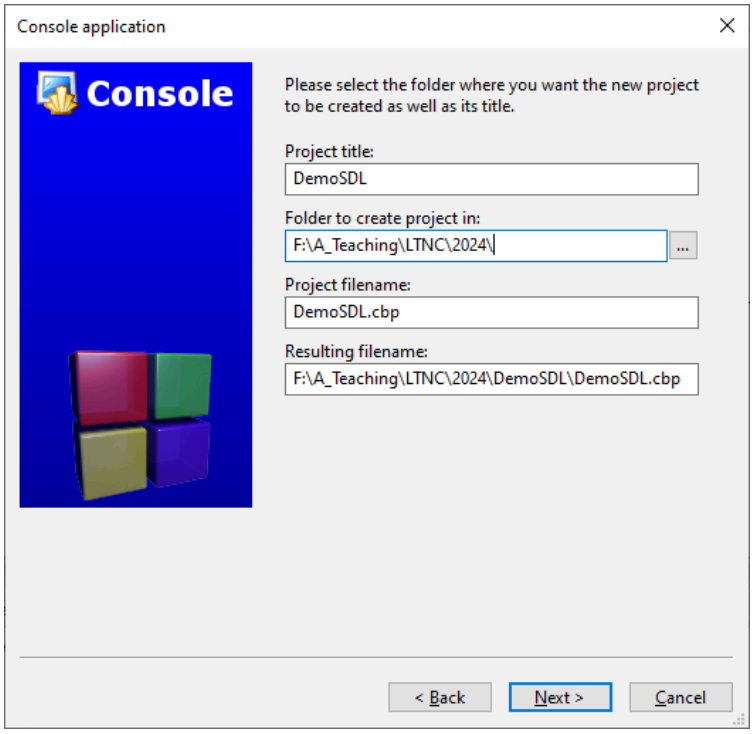


Setup project dùng SDL2

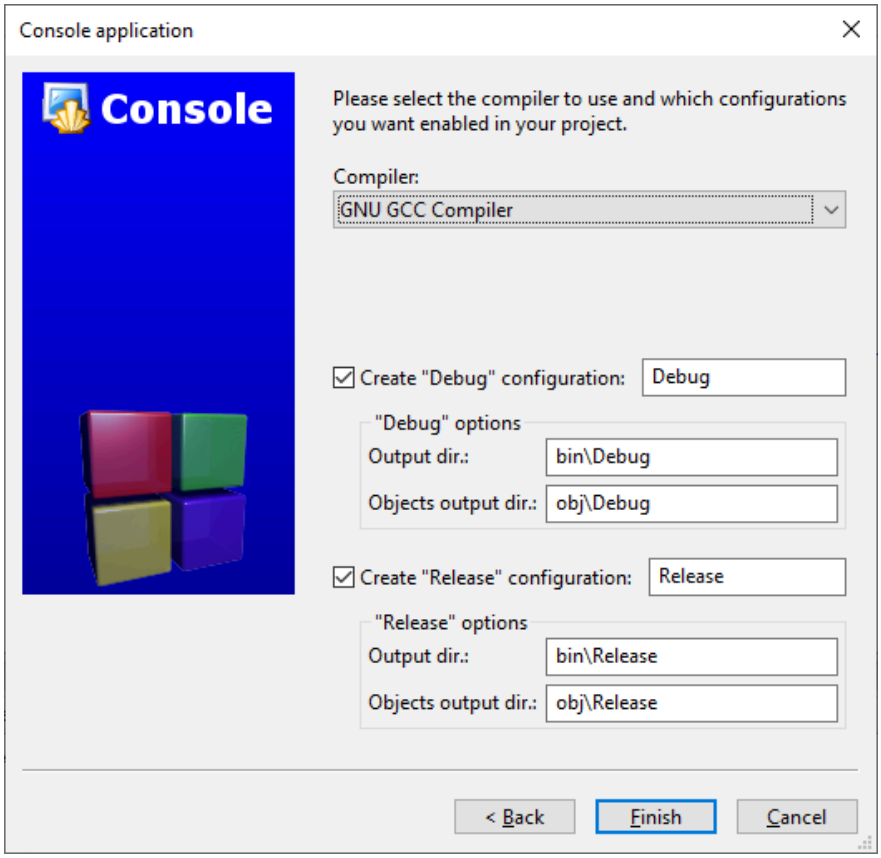
Tạo project mới **File | New | Project…**, chọn loại Console Application.



Chọn tên project và thư mục đặt project.



Chú ý để Compiler là GNU GCC Compiler

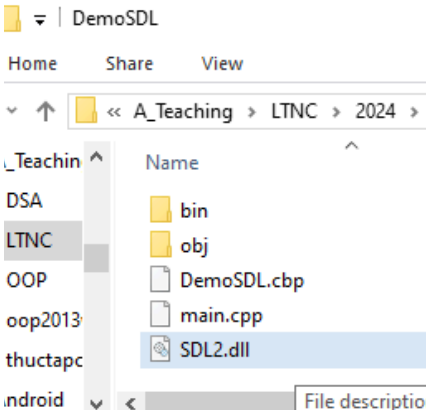


Setup SDL cho project:
Nếu bạn chưa xóa thư mục \i686-w64-mingw32 thì hãy cẩn thận: dùng đường dẫn….\x86_64-w64-mingw32, không dùng ..\i686-w64-mingw32
Nếu không thành công, cần làm lại 4 bước dưới đây để đảm bảo không nhầm đường dẫn.

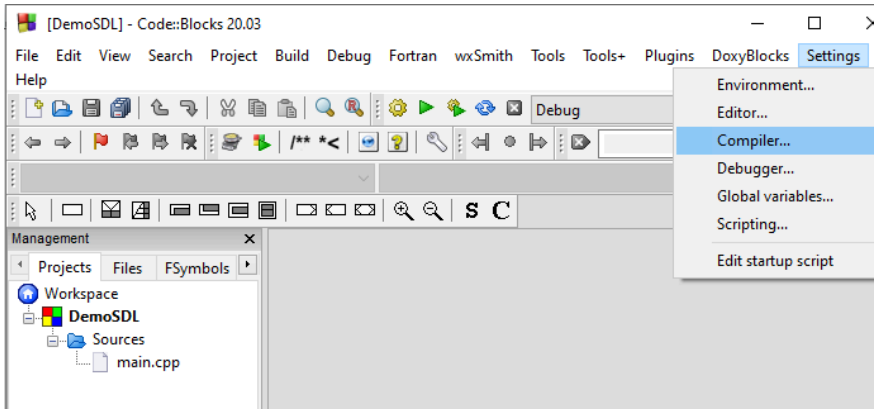
Ngắn gọn:

1. Copy file dll vào thư mục mã nguồn project (nơi có các file .cpp)
   …..\x86_64-w64-mingw32\bin\SDL2.dll
2. Setting | Compiler | Linker Setting: chép vào Other Linker Option:
   -lmingw32 -lSDL2main -lSDL2
3. Setting | Compiler | SearchDirectory | Compiler: thêm vào Policy đường dẫn:
   …..\x86_64-w64-mingw32\include\SDL2
4. Setting | Compiler | SearchDirectory | Linker: thêm vào Policy đường dẫn:
   …..\x86_64-w64-mingw32\lib

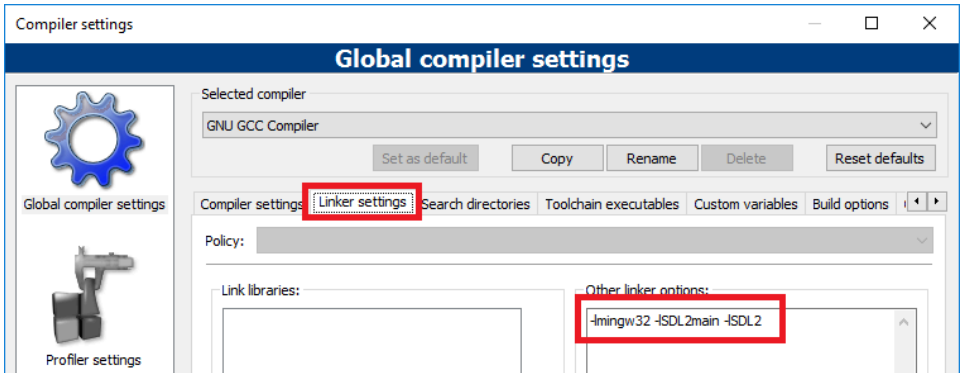
Chi tiết:
Bước 1. Copy file dll F:\SDL2-2.28.5\x86_64-w64-mingw32\bin**SDL2.dll**
vào thư mục mã nguồn project (nơi có các file .cpp)



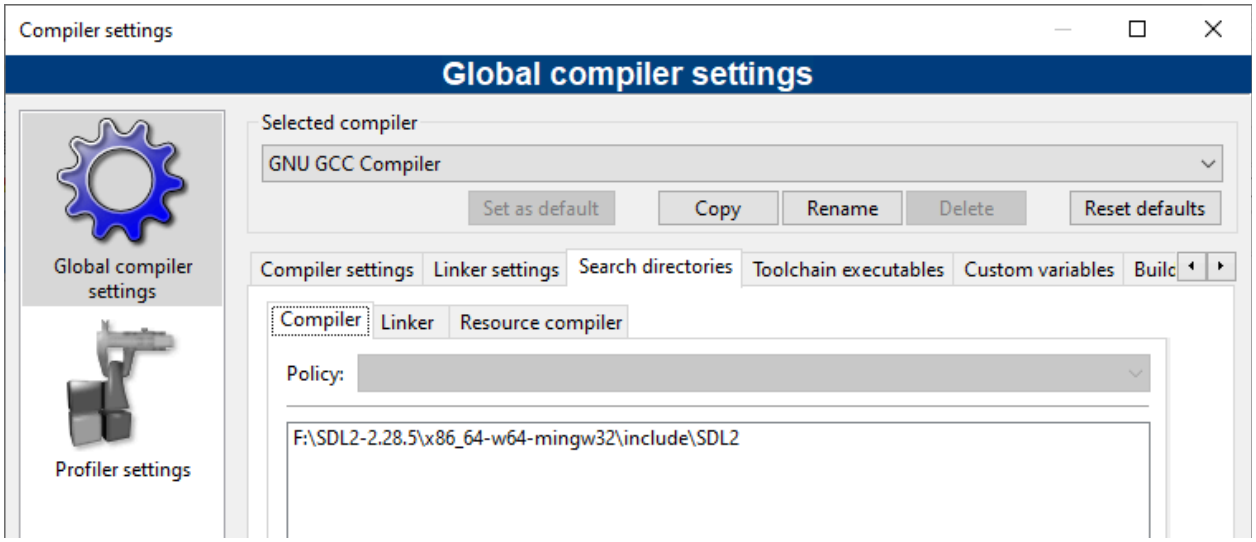
Bước 2. Option cho linker Setting | Compiler | Linker Setting:



chép vào Other Linker Option: **`-lmingw32 -lSDL2main -lSDL2`**

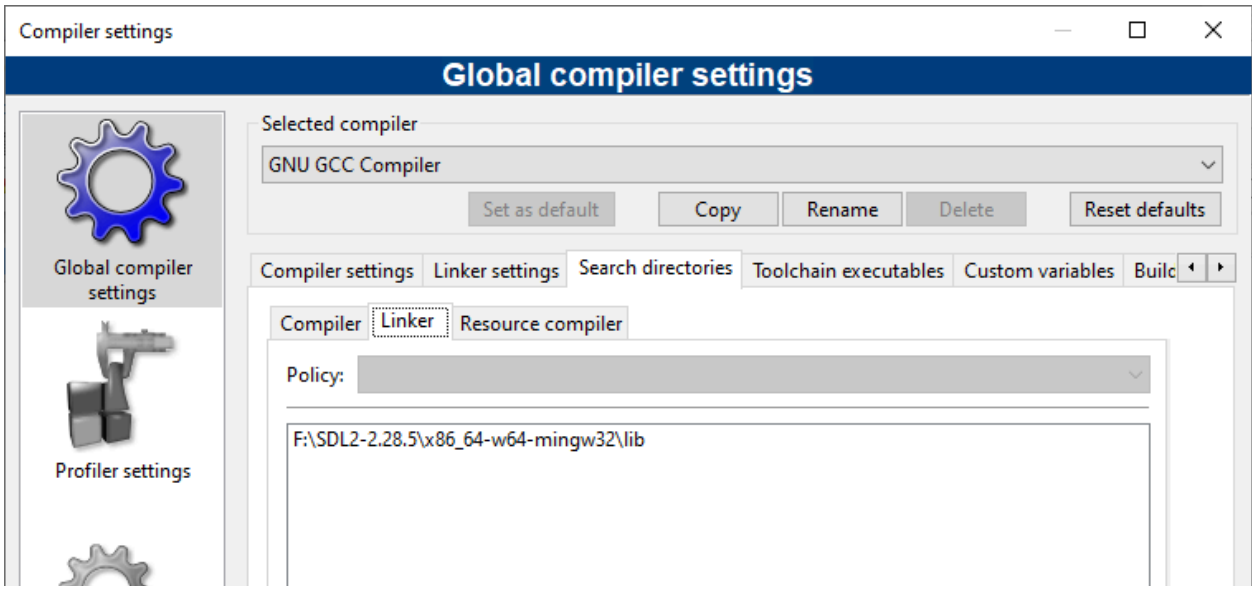


Bước 3. Đặt đường dẫn cho các file header
Setting | Compiler | SearchDirectory | **Compiler**: thêm đường dẫn:
…..\x86_64-w64-mingw32**\include\SDL2**



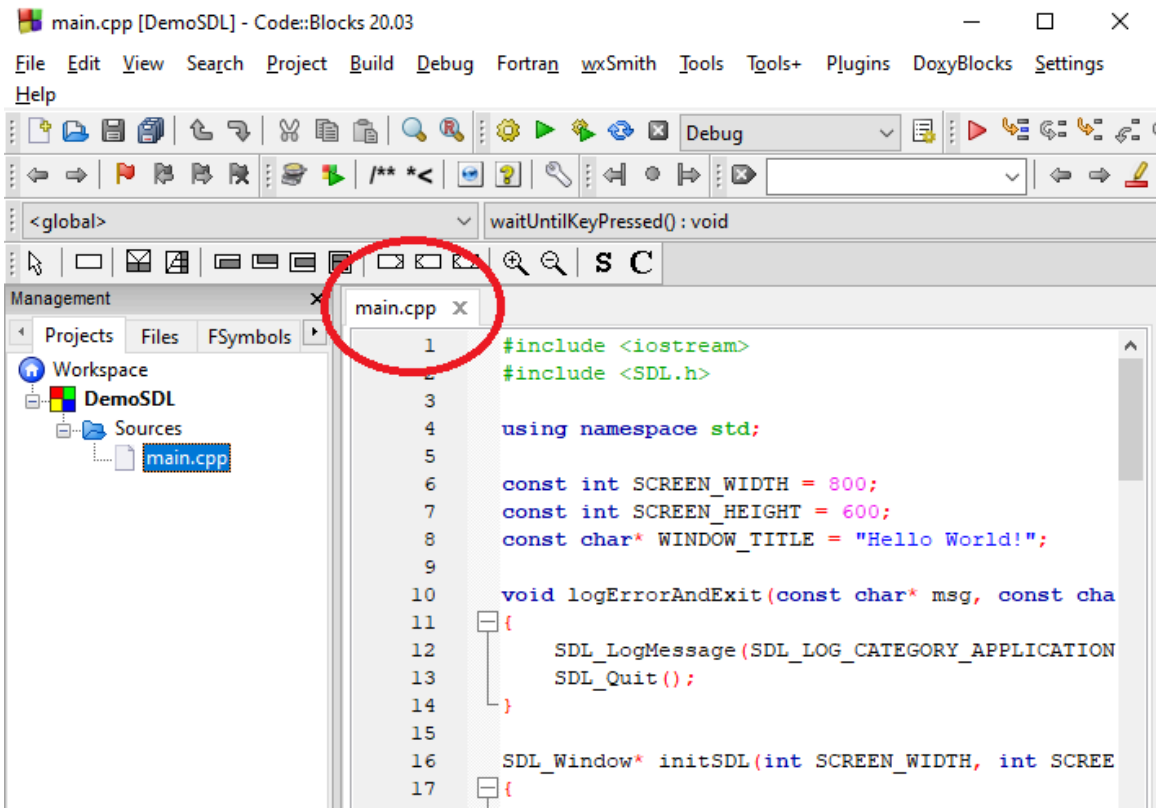
Bước 4. Setting | Compiler | SearchDirectory | **Linker**: thêm đường dẫn:

…..\x86_64-w64-mingw32**\lib**

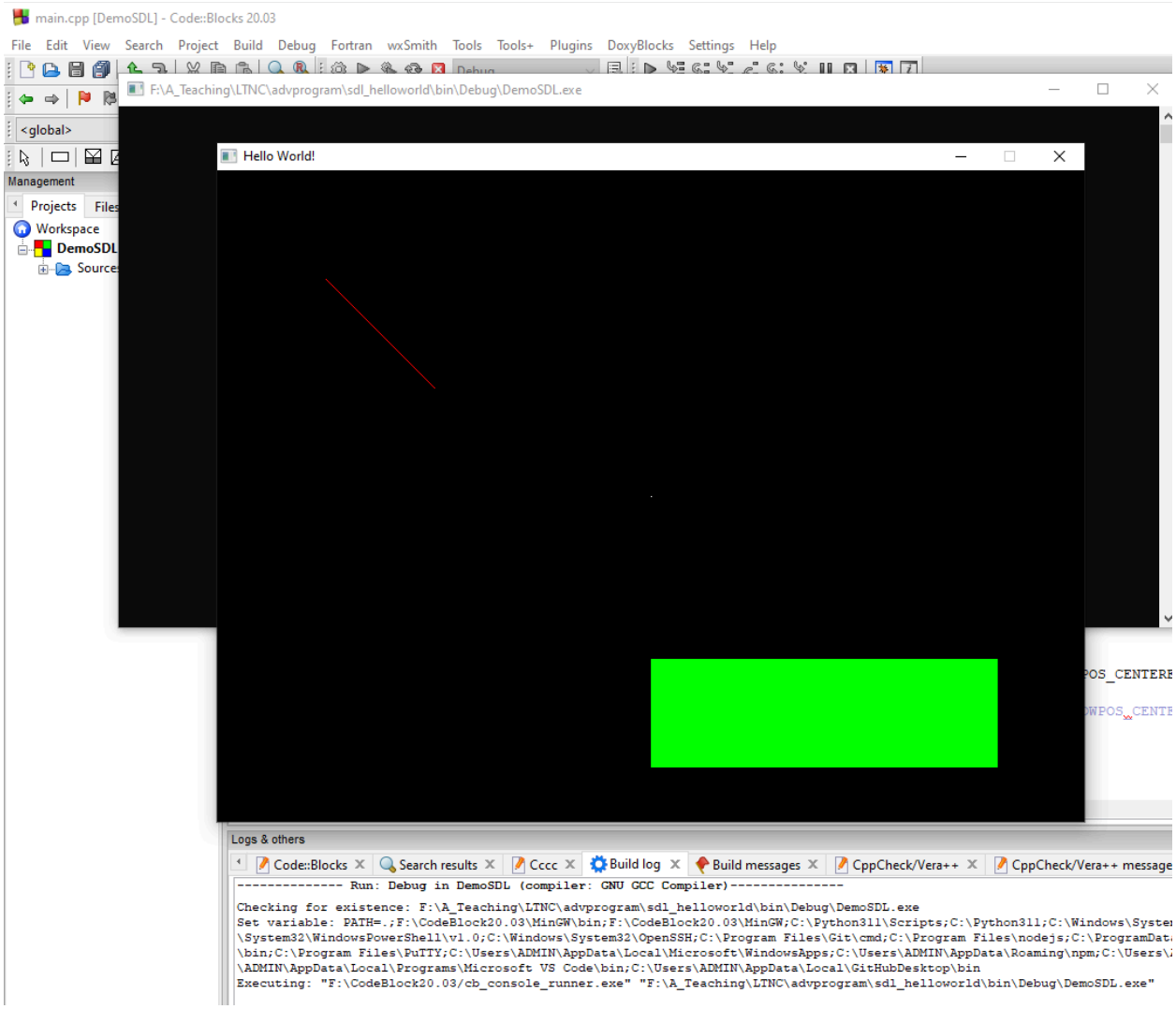


Chương trình SDL đầu tiên:

Lấy code tại https://github.com/chauttm/advprogram/blob/master/sdl_helloworld/main.cpp, chép nội dung vào file main.cpp của project. Bạn sẽ thấy đại loại như dưới đây. Chú ý nội dung hàm main.cpp nằm bên trong Sources của project DemoSDL

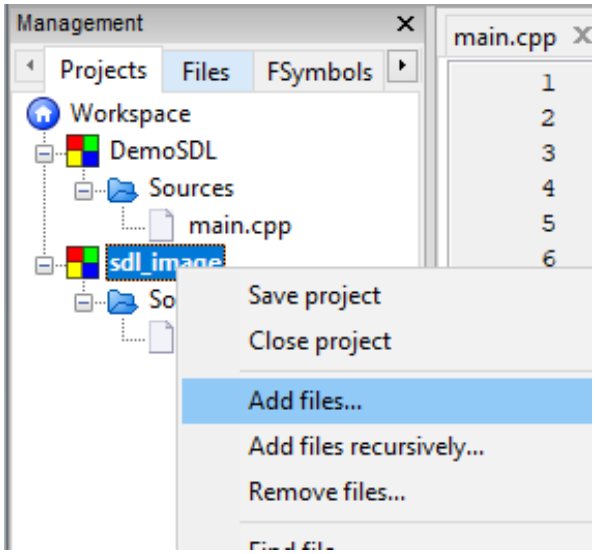


Dịch và chạy.
Nếu các bước trên thành công, bạn sẽ thấy kết quả tương tự như sau:



Xử lý lỗi:

- ***Lỗi liên quan SDL:*** xem danh sách lỗi và cách xử lý tại Piazza.
- ***Chương trình in ra Hello World:*** bạn đang chạy file mặc định của project chứ không phải code mẫu SDL ở trên. Bạn cần copy paste nội dung code vào main.cpp (file có sẵn) chứ không đơn giản chỉ mở file vừa download ra chạy.
  Lý do: bạn đang ở trong môi trường project, nên khi bấm build và run nó sẽ run file của project. Nếu bạn chưa add một file vào project mà đã chạy thì nó không chạy file đó đâu. Bạn cần copy paste code vào 1 file có sẵn trong project, hoặc add file muốn chạy vào project. Nếu bạn muốn add một file có sẵn vào project thì cần (1) copy file đó vào thư mục chứa mã nguồn của project (chỗ đang có main.cpp ấy) và (2) Click chuột phải vào Project rồi Add Files (xem hình)



Trong hình là 2 project cùng đang mở, hiện đang chọn add files vào project thứ hai sdl_image

Làm gì tiếp theo?

Đọc hiểu code. Để ý khung chương trình tại hàm main():

- các bước khởi tạo và kết thúc mà giống nhau ở hầu hết các chương trình
- quy trình thông dụng cần thực hiện cho mỗi khung hình: xóa màn hình → vẽ → hiện bản vẽ

Các hàm tiện ích như initSDL, createRenderer, waitUntilKeyPressed, quitSDL… đóng gói các thủ tục kỹ thuật để thực hiện những công việc cơ bản mà bạn sẽ còn dùng nhiều ở các chương trình khác. Sau này ta sẽ tách thành file tiện ích để tiện dùng cho các project khác nữa. Tại thời điểm này, nhiều bạn chưa hiểu nội dung của các hàm trên, nhưng hãy cứ dùng đã, sẽ tìm hiểu sau.

```
80  int main(int argc, char* argv[])
81  {
82      //Khởi tạo môi trường đồ họa
83      SDL_Window* window = initSDL(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_TITLE);
84      SDL_Renderer* renderer = createRenderer(window);
85
86      //Xóa màn hình
87      SDL_RenderClear(renderer);
88
89      //Vẽ gì đó
90      drawSomething(window, renderer);
91
92      //Hiện bản vẽ ra màn hình
93      //Khi chạy tại môi trường bình thường
94      SDL_RenderPresent(renderer);
95      //Khi chạy trong máy ảo (ví dụ phòng máy ở trường)
96      //SDL_UpdateWindowSurface(window);
97
98      //Đợi phím bất kỳ trước khi đóng môi trường đồ họa và kết thúc chương trình
99      waitUntilKeyPressed();
100     quitSDL(window, renderer);
101     return 0;
102 }
```

Thử sửa code, thay đổi mầu sắc tọa độ, dùng các hàm vẽ khác.

Tài liệu:
Tra cứu các hàm của SDL2 tại đây: https://wiki.libsdl.org/SDL2/CategoryAPI

Bài 2. Sử dụng thư viện hiển thị ảnh SDL2_image

***Lưu ý**: Các đường dẫn, tên file, cách cấu hình … trong hướng dẫn này dành cho project CodeBlocks, không áp dụng được cho project ở các môi trường phát triển khác, chẳng hạn Visual Code*

Cài thư viện SDL2_image

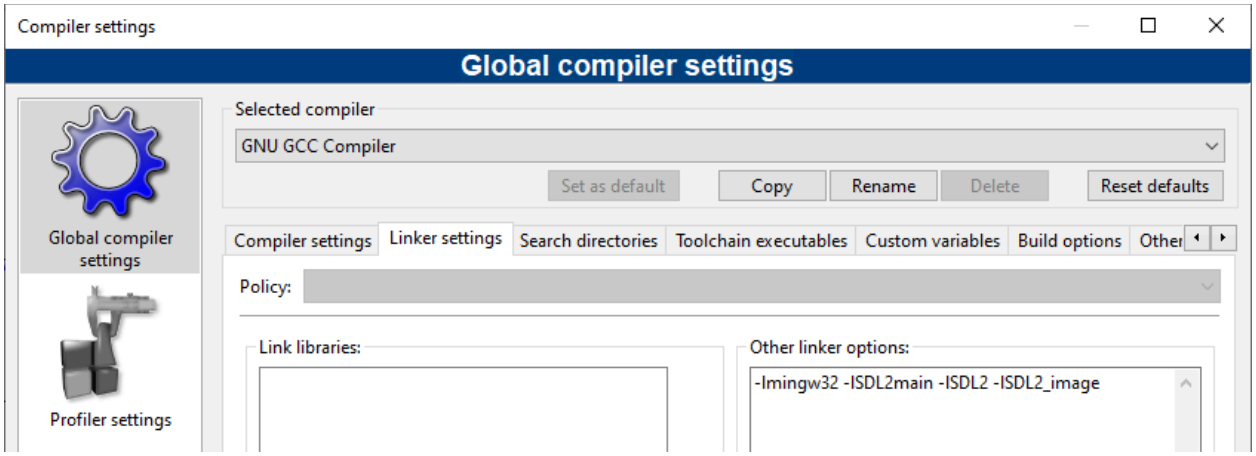
1. Download **SDL2_image-devel-2.8.2-mingw.zip** từ https://github.com/libsdl-org/SDL_image/releases hoặc một release stable cũ hơn. Điểm quan trọng là file download cần có dạng **SDL2_image-devel**-x.xx.x-**mingw**.xxxx thì mới có đủ nội dung cần thiết.
2. Gỡ nén, ví dụ vào F:\SDL\SDL2_image-2.8.2
3. Để từ nay không bị nhầm phiên bản 32 và 64 bit, xóa thư mục F:\SDL\SDL2_image-2.8.2**i686-w64-mingw32**

Setup project dùng SDL2_image

Nếu bạn chưa xóa thư mục \i686-w64-mingw32 thì hãy cẩn thận: dùng đường dẫn…. \x86_64-w64-mingw32, không dùng ..\i686-w64-mingw32
Nếu không thành công, cần làm lại 4 bước dưới đây để đảm bảo không nhầm đường dẫn.

Từ project đã sẵn cấu hình SDL*:*

1. Copy vào thư mục mã nguồn project (nơi có các file .cpp và SDL2.dll lần trước) F:\SDL\SDL2_image-2.8.2**\x86_64-w64-mingw32\bin\SDL2_image.dll**
2. Setting | Compiler | Linker Setting: chép thêm vào Other Linker Option: **-lSDL2_image**
   Kết quả sẽ như hình bên dưới
3. Setting | Compiler | SearchDirectory | Compiler: thêm vào Policy đường dẫn: F:\SDL\SDL2_image-2.8.2**\x86_64-w64-mingw32\include\SDL2**
4. Setting | Compiler | SearchDirectory | Linker: thêm vào Policy đường dẫn: F:\SDL\SDL2_image-2.8.2**\x86_64-w64-mingw32\lib**



Bạn có thể thấy là quy trình setup SDL2_image giống hệt với thư viện chính SDL2, chỉ khác nhau ở chi tiết. Other linker option -lSDL2_image bản chất là tham số dòng lệnh -l (lib) và tên file thư viện, ở đây là bộ file có tên libSDL2_image trong thư mục …\SDL2_image-2.8.2\...\lib. Nếu bạn mở thư mục ….\lib của thư viện SDL2, bạn sẽ thấy các file libSDL2 ứng với option -lSDL2, các file libSDL2main ứng với option -lSDL2main. Từ nay, bạn có thể tự luận ra cách setup các thư viện con khác của SDL

Chuẩn bị ảnh

Chương trình ví dụ trong bài này sẽ dùng đến 2 ảnh. Hãy download các file ảnh vào thư mục chứa mã nguồn của project (chỗ các file.cpp). Ảnh trong thư mục chứa mã nguồn có thể được gọi đến từ trong code bằng đường dẫn gián tiếp chỉ gồm tên file.
https://github.com/chauttm/advprogram/blob/master/sdl_image/Spongebob.png
https://github.com/chauttm/advprogram/blob/master/sdl_image/bikiniBottom.jpg

Viết code

Bắt đầu từ mã nguồn của Bài 1.

- **Include**. Để sử dụng thư viện, cần dòng include file header

```
#include <iostream>
#include <SDL.h>
#include <SDL_image.h>
```

- **Khởi tạo và giải phóng thư viện**: Để thư viện SDL2_image hoạt động hiệu quả, cần gọi hàm khởi tạo IMG_init() ở đầu chương trình và hàm IMG_Quit() ở cuối chương trình. Bạn hãy sửa các hàm initSDL() và quitSDL() để thêm các lời gọi hàm đó.

```
SDL_Window* initSDL(...
{
...
```

```
    if (!IMG_Init(IMG_INIT_PNG | IMG_INIT_JPG))
        logErrorAndExit( "SDL_image error:", IMG_GetError());

    return window;
}
...
void quitSDL(SDL_Window* window, SDL_Renderer* renderer)
{
    IMG_Quit();

    ...
}
```

Trong lệnh IMG_Init trên, ta khởi tạo thư viện cho hai loại ảnh PNG và JPG, bạn có thể thêm hoặc thay bằng các loại ảnh khác.

- **Nạp file ảnh vào texture**: Để nạp một file ảnh vào một đối tượng texture, gần gọi hàm IMG_LoadTexture với hai tham số là con trỏ tới renderer và đường dẫn đến file ảnh. Hàm sẽ trả về con trỏ tới texture chứa ảnh để từ nay có thể dùng để vẽ. Nếu không thể nạp ảnh, chẳng hạn do không tìm thấy file, hàm sẽ trả về con trỏ NULL, ta cần kiểm tra kết quả trả về vào log lỗi ra màn hình để khi debug chương trình có thể hiểu lỗi gì đang xảy ra.

```
SDL_Texture *texture = IMG_LoadTexture(renderer, filename);
if (texture == NULL) {
        SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION, SDL_LOG_PRIORITY_ERROR,
                       "Load texture %s", IMG_GetError());
}
```

Đoạn code trên sẽ còn dùng nhiều mỗi khi ta cần nạp ảnh, do đó bạn nên đưa nó vào một hàm tiện ích để có thể tái sử dụng và gọn code.

```
SDL_Texture *loadTexture(const char *filename, SDL_Renderer* renderer)
{
      SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION, SDL_LOG_PRIORITY_INFO,
                     "Loading %s", filename);

      SDL_Texture *texture = IMG_LoadTexture(renderer, filename);
      if (texture == NULL) {
        SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION, SDL_LOG_PRIORITY_ERROR,
                       "Load texture %s", IMG_GetError());
      }

      return texture;
```

```
}
```

Có thể thêm dòng log thông tin “Loading….” vào đầu hàm để dễ dàng hơn khi debug chương trình.

- **Hiện ảnh trên toàn cửa sổ**: Đây là cách đơn giản nhất để vẽ một ảnh. Chỉ cần đúng 1 lệnh. Trong đó ta truyền texture vừa nạp ảnh vào làm tham số thứ hai của hàm SDL_RenderCopy

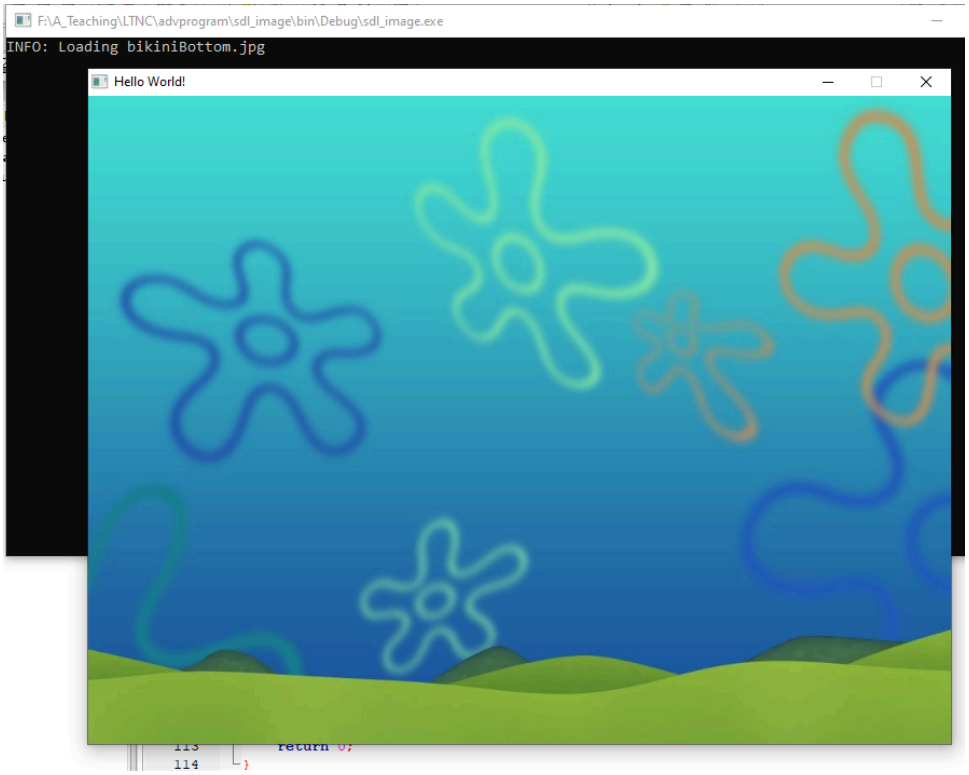
  ```
  SDL_RenderCopy( renderer, texture, NULL, NULL);
  ```

Giờ bạn có thể sửa hàm main để nạp và vẽ cái ảnh đầu tiên

```
int main(int argc, char *argv[])
{
    SDL_Window* window = initSDL(SCREEN_WIDTH, SCREEN_HEIGHT, WINDOW_TITLE);
    SDL_Renderer* renderer = createRenderer(window);

    SDL_Texture* background = loadTexture("bikiniBottom.jpg", renderer);
    SDL_RenderCopy( renderer, background, NULL, NULL);

    SDL_RenderPresent( renderer );
    waitUntilKeyPressed();
    SDL_DestroyTexture( background );
    background = NULL;

    quitSDL(window, renderer);
    return 0;
}
```

Để ý hai dòng đậm thứ nhất, ta nạp file ảnh bikiniBottom.jpg vào texture background rồi dùng renderer để vẽ nó. Ảnh đặt trực tiếp trong thư mục chứa mã nguồn có thể được gọi đến từ trong code bằng đường dẫn gián tiếp chỉ gồm tên file, cho nên đường dẫn ở đây không có thư mục.
Sau đó là lệnh SDL_RenderPresent với nhiệm vụ hiển thị bản vẽ ra màn hình. Nhắc lại quy trình là các lệnh vẽ chỉ vẽ lên bản vẽ và chỉ để khi chạy lệnh này thì bản vẽ mới được hiển thị ra màn hình.
Hai dòng code đậm ở sau đó có nhiệm vụ giải phóng bộ nhớ của texture mà ta không dùng đến nữa, và gán con trỏ texture về null để tránh lỗi chẳng may dùng nhầm về sau. Công việc này ở chương trình ví dụ nhỏ này không có ý nghĩa gì do chương trình kết thúc ngay sau đó. Tuy nhiên, ở các chương trình lớn hơn, khi mà ta cần dùng đến nhiều bộ nhớ và chương trình chạy lâu, game là điển hình, thì ta cần giải phóng các texture ngay khi không dùng đến chúng nữa để tiết kiệm bộ nhớ cho các nhu cầu về sau.

Chạy chương trình, bạn sẽ thấy kết quả như hình dưới.



- **Hiện ảnh ở một vị trí cụ thể**: Để ý lúc trước ta vẽ ảnh bằng cách gọi hàm SDL_RenderCopy với hai tham số cuối bằng NULL. Đó là vẽ mà không quan tâm vị trí vẽ và kích thước ảnh. Tham số cuối chính là để xác định vị trí vẽ ảnh trên màn hình. Để quy định vị trí vẽ ảnh, tham số cuối cần có là một hình chữ nhật SDL_Rect xác định vị trí vẽ trên màn hình. Một lần nữa, đây là công việc lặp đi lặp lại, nên ta cũng làm thành một hàm tiện ích

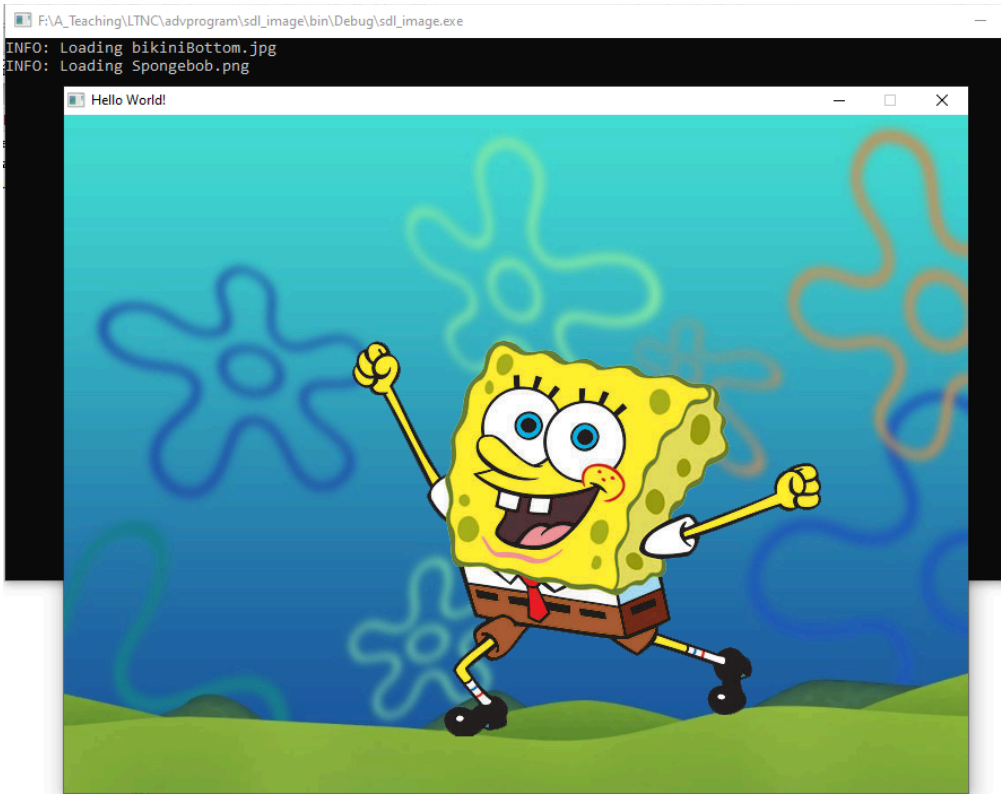
```
void renderTexture(SDL_Texture *texture, int x, int y,
                   SDL_Renderer* renderer)
      {
            SDL_Rect dest;
            dest.x = x;
            dest.y = y;
            SDL_QueryTexture(texture, NULL, NULL, &dest.w, &dest.h);

            SDL_RenderCopy(renderer, texture, NULL, &dest);
      }
```

dest là hình chữ nhật mà ta sẽ vẽ ảnh bên trong nó. nó có tọa độ x,y là tọa độ màn hình nơi ta muốn đặt góc trên trái của ảnh. Hàm SDL_QueryTexture tính kích thước của hình chữ nhật đó theo kích thước của ảnh đã nạp vào texture. Do đó, chiều cao h và chiều rộng w được truyền

vào lời gọi hàm SDL_QueryTexture theo địa chỉ để chúng có thể được gán giá trị từ bên trong hàm đó.
Và sửa hàm main để dùng hàm trên hiển thị ảnh thứ hai - ảnh SpongeBob trên hình nền là ảnh đã vẽ lúc trước.

```
  …
    SDL_RenderPresent( renderer );
    waitUntilKeyPressed();

    SDL_Texture* spongeBob = loadTexture("Spongebob.png", renderer);
    renderTexture(spongeBob, 200, 200, renderer);

    SDL_RenderPresent( renderer );
    waitUntilKeyPressed();

    SDL_DestroyTexture( spongeBob );
    spongeBob = NULL;
    SDL_DestroyTexture( background );
    background = NULL;
…
```

Ta thấy quy trình tương tự như đối với ảnh trước, nạp → vẽ → giải phóng, chỉ khác chút ở phần vẽ. Đoạn gọi `SDL_RenderPresent` và đợi phím nằm giữa chỉ có mục đích để bạn có thể chứng kiến thời điểm ảnh thứ hai được nạp và vẽ, bạn cần bấm phím trước khi việc đó xảy ra.
Kết quả sẽ như hình bên dưới.



Gợi ý thử nghiệm với code:

- Chuyển ảnh vào thư mục con images của thư mục chứa mã nguồn rồi sửa đường dẫn nạp ảnh từ “bikiniBottom.jpg” thành “images\\bikiniBottom.jpg”
- thay đổi các tọa độ và kích thước, xem ảnh méo, ảnh có tọa độ ngoài màn hình,
- đổi các ảnh mẫu sang dùng ảnh của bạn, thêm ảnh,
- đổi thứ tự vẽ ảnh để thấy ảnh chồng lên nhau ….
- thử hiệu ứng của việc sửa vị trí các lệnh `SDL_RenderPresent( renderer )` và `waitUntilKeyPressed()`
- **Hiện chỉ một phần ảnh**: Để ý rằng hai lần vẽ ảnh trước đều vẽ toàn bộ ảnh. Nếu chỉ muốn vẽ một phần của ảnh, tham số thứ ba của SDL_RenderCopy là để cho việc này. Bạn cần dùng một SDL_Rect định nghĩa khu vực cần vẽ của ảnh và truyền vào vị trí của tham số đó. Đoạn code dưới đây là ví dụ.

```
SDL_Rect spongeBobSrc;
spongeBobSrc.x = 10;
spongeBobSrc.y = 10;
spongeBobSrc.w = 100;
spongeBobSrc.h = 100;
SDL_RenderCopy( renderer, spongeBob, &spongeBobSrc, &spongeBobDest );
```

Lỗi thường gặp:
undefined reference to `IMG_Load' : Kiểm tra other linker option xem có -lSDL2_image chưa

và có lỗi chính tả nào không (nhầm I và l chẳng hạn)

Tài liệu

Tra cứu các hàm của SDL2_image tại đây: https://wiki.libsdl.org/SDL2_image/CategoryAPI

Code: https://github.com/chauttm/gameProject/blob/main/02_image/main.cpp

Bài 3. Project có nhiều file

Bài trước là ví dụ mini về việc dùng ảnh. Hàm main hơi nhiều chi tiết kỹ thuật, chỉ phù hợp với việc demo dùng ảnh chứ không tiện cho việc thêm các kỹ thuật mới. Ở bài này, ta bắt đầu dùng project nhiều file. Ta sẽ tách các tiện ích vẽ ra cho đỡ rối code trước khi thêm các kỹ thuật mới.

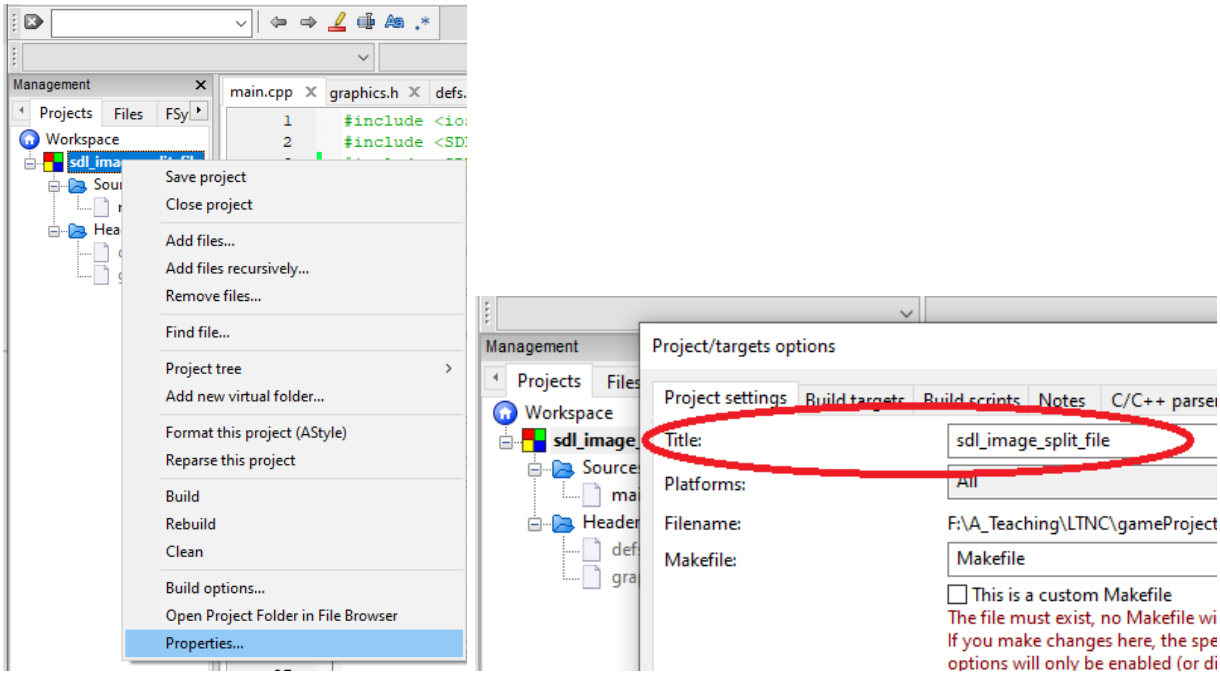
Trong bài này, ta sẽ tách từ main.cpp các mô-đun, mỗi mô-đun có một nhiệm vụ riêng, ví dụ mô-đun graphics dành riêng cho các công việc đồ họa, mô-đun defs chứa các thông tin cấu hình. Mỗi mô-đun có thể hiểu như là một thư viện mà bạn có thể tách ra mang dùng cho project khác. Về cấu trúc tổng quát, mỗi mô đun thường có hai phần:

- header (file .h) chứa các định nghĩa hằng, kiểu dữ liệu, các khai báo hàm và cài đặt một số hàm đơn giản
- phần cài đặt (file .cpp thường là cùng tên) chứa định nghĩa (cài đặt) các hàm

Bài này chỉ tách mô-đun kiểu đơn giản, mỗi mô-đun chỉ gồm phần header chứa toàn bộ code, kể cả cài đặt của tất cả các hàm. Ta sẽ học tách thàng cả header lẫn phần cài đặt trong một bài sau.

1. Chuẩn bị project.

Nên bắt đầu từ sản phẩm của bài trước, bạn có thể mở project mới, setup từ đầu, rồi copy code sang. Hoặc bạn có thể copy lấy bản sao toàn bộ thư mục project cũ, mở ra bản sao ở CodeBlock và đổi tên project mới bằng cách click chuột phải vào tên project ở cửa sổ Management và chọn Properties. Sửa Title.



Nhớ khớp tên thư mục và title project để tránh nhầm lẫn với project cũ.

2. Tạo file header chứa các giá trị dùng chung toàn cục.

Có những giá trị không thay đổi trong quá trình chạy chương trình mà sẽ được dùng đến ở nhiều logic chương trình, ví dụ như hiện giờ là kích thước cửa sổ, sau này có thể là tên file chứa bản đồ game…. Ta muốn để nó ở chế độ dùng chung thay vì phải truyền các giá trị này qua lại giữa các hàm. Những giá trị không đổi này nếu để ở mức toàn cục không vi phạm nguyên tắc “không dùng biến toàn cục”.

Chuyển các hằng giá trị dùng chung cho toàn project ra khỏi main.cpp, đưa vào một file header, tên def.h chẳng hạn. Ở CodeBlock bạn có thể New file trong khi đang mở project hoặc Add files có sẵn (chuột phải tại tên project trong cửa sổ Management)

Và bọc đầu cuối file def.h bằng bộ khóa ifndef….define…. endif.

—---------- defs.h ----------

```
#ifndef _DEFS__H
#define _DEFS__H


const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;
const char* WINDOW_TITLE = "Hello World!";


#endif
```

—----------

Bộ khóa trên có tác dụng tránh lỗi redefinition - một biến/hàm/hằng bị định nghĩa nhiều hơn một lần trong code - lỗi này xảy ra khi ta include một file header tại nhiều hơn một file khác.

Định danh `_DEFS__H` như trong đoạn code trên cần là một định danh không trùng với bất kỳ định danh nào khác. Do đó bạn có thể theo truyền thống là đặt tên theo tên file `def.h` → `_DEFS__H`.

File header defs.h đã sẵn sàng, ta include nó vào main.cpp

—--------- main.cpp —----------

```
#include <iostream>
#include <SDL.h>
#include <SDL_image.h>
#include "defs.h"


using namespace std;
…
```

Để ý là tại dòng include, ta dùng dấu nháy kép cho defs.h thay vì ngoặc nhọn như với các file thư việc khác. Lý do là các file được include dùng dấu nháy kép được tìm tại thư mục mã nguồn của project, còn các file được include bằng dấu ngoặc nhọn được tìm tại các đường dẫn thư viện setup trong cấu hình project (nhớ lại bạn đã setup đường dẫn đến SDL.h ở project như thế nào). Từ nay, tất các file header được tách ra sẽ được include dùng dấu nháy kép.

Biên dịch và chạy thử project để đảm bảo việc tách file defs.h đã làm đúng.

3. Tách phần điều khiển đồ họa

Ta muốn các chi tiết kỹ thuật SDL không làm rối logic chương trình, nên ta sẽ tách nó ra file khác. Ta muốn hàm main() sẽ có nội dung như sau:

—------ main.cpp ----
```
int main(int argc, char *argv[])
{
    Graphics graphics;
    graphics.init();

    SDL_Texture* background = graphics.loadTexture("bikiniBottom.jpg");
    graphics.prepareScene(background);

    graphics.presentScene();
    waitUntilKeyPressed();

    SDL_Texture* spongeBob = graphics.loadTexture("Spongebob.png");
    graphics.renderTexture(spongeBob, 200, 200);

    graphics.presentScene();
    waitUntilKeyPressed();

    SDL_DestroyTexture( spongeBob );
    spongeBob = NULL;
    SDL_DestroyTexture( background );
    background = NULL;

    graphics.quit();
    return 0;
}
```
—----------

Bạn có thể thấy là renderer, window không còn xuất hiện trong hàm main. (Ở bài trước ta có hai biến đó trong hàm main và mỗi khi vẽ lại truyền nó vào các hàm vẽ). Ta muốn các chi tiết kỹ thuật, các biến điều khiển kỹ thuật liên quan đến đồ họa được đóng gói hết trong graphics, cái mà ta khởi tạo ra ở đầu chương trình và đóng ở cuối chương trình.

Để được như trên, ta sẽ tạo struct Graphics và bê các biến/hàm đồ họa vào trong đó.

Tạo file graphics.h theo quy trình tương tự def.h ở trên. Nhớ include defs để ít nữa dùng kích thước cửa sổ.

—-------- graphics.h -----

```
#ifndef _GRAPHICS__H
#define _GRAPHICS__H

#include <SDL.h>
#include <SDL_image.h>
#include "defs.h"

struct Graphics {
    SDL_Renderer *renderer;
    SDL_Window *window;

    // các hàm khác

};

#endif // _GRAPHICS__H
```

—————————————

Tại main.cpp, thêm #include “graphics.h” (tương tự include defs.h). Sau đó, bê toàn bộ các hàm con ngoại trừ `waitUntilKeyPressed()` (cái này liên quan đến input hơn là vẽ, ta tạm để yên đó) sang graphics.h, đặt vào bên trong struct Graphics, và sửa prototype của chúng cho khớp nhu cầu gọi hàm từ trong main(). Chủ yếu là tháo renderer và window ra khỏi danh sách tham số của các hàm.

—————————————

```
struct Graphics {
    SDL_Renderer *renderer;
    SDL_Window *window;

    void logErrorAndExit(const char* msg, const char* error)
    {
        SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION, SDL_LOG_PRIORITY_ERROR,
                       "%s: %s", msg, error);
        SDL_Quit();
    }

    void init() {
        if (SDL_Init(SDL_INIT_EVERYTHING) != 0)
```

```
        logErrorAndExit("SDL_Init", SDL_GetError());

    window = SDL_CreateWindow(WINDOW_TITLE, SDL_WINDOWPOS_CENTERED,
                SDL_WINDOWPOS_CENTERED, SCREEN_WIDTH, SCREEN_HEIGHT,
                SDL_WINDOW_SHOWN);
    if (window == nullptr)
                logErrorAndExit("CreateWindow", SDL_GetError());

    if (!IMG_Init(IMG_INIT_PNG | IMG_INIT_JPG))
        logErrorAndExit( "SDL_image error:", IMG_GetError());

    renderer = SDL_CreateRenderer(window, -1,
                  SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);

    //renderer =
              SDL_CreateSoftwareRenderer(SDL_GetWindowSurface(window));

    if (renderer == nullptr)
         logErrorAndExit("CreateRenderer", SDL_GetError());

    SDL_SetHint(SDL_HINT_RENDER_SCALE_QUALITY, "linear");
    SDL_RenderSetLogicalSize(renderer, SCREEN_WIDTH, SCREEN_HEIGHT);
}

void prepareScene(SDL_Texture * background)
{
    SDL_RenderClear(renderer);
    SDL_RenderCopy( renderer, background, NULL, NULL);
}

void presentScene()
{
    SDL_RenderPresent(renderer);
}

SDL_Texture *loadTexture(const char *filename)
{
    SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION, SDL_LOG_PRIORITY_INFO,
                   "Loading %s", filename);
    SDL_Texture *texture = IMG_LoadTexture(renderer, filename);
    if (texture == NULL)
        SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION,
              SDL_LOG_PRIORITY_ERROR, "Load texture %s", IMG_GetError());
```

```
        return texture;
    }

    void renderTexture(SDL_Texture *texture, int x, int y)
    {
        SDL_Rect dest;

        dest.x = x;
        dest.y = y;
        SDL_QueryTexture(texture, NULL, NULL, &dest.w, &dest.h);

        SDL_RenderCopy(renderer, texture, NULL, &dest);
    }

    void quit()
    {
        IMG_Quit();

        SDL_DestroyRenderer(renderer);
        SDL_DestroyWindow(window);
        SDL_Quit();
    }
};
```

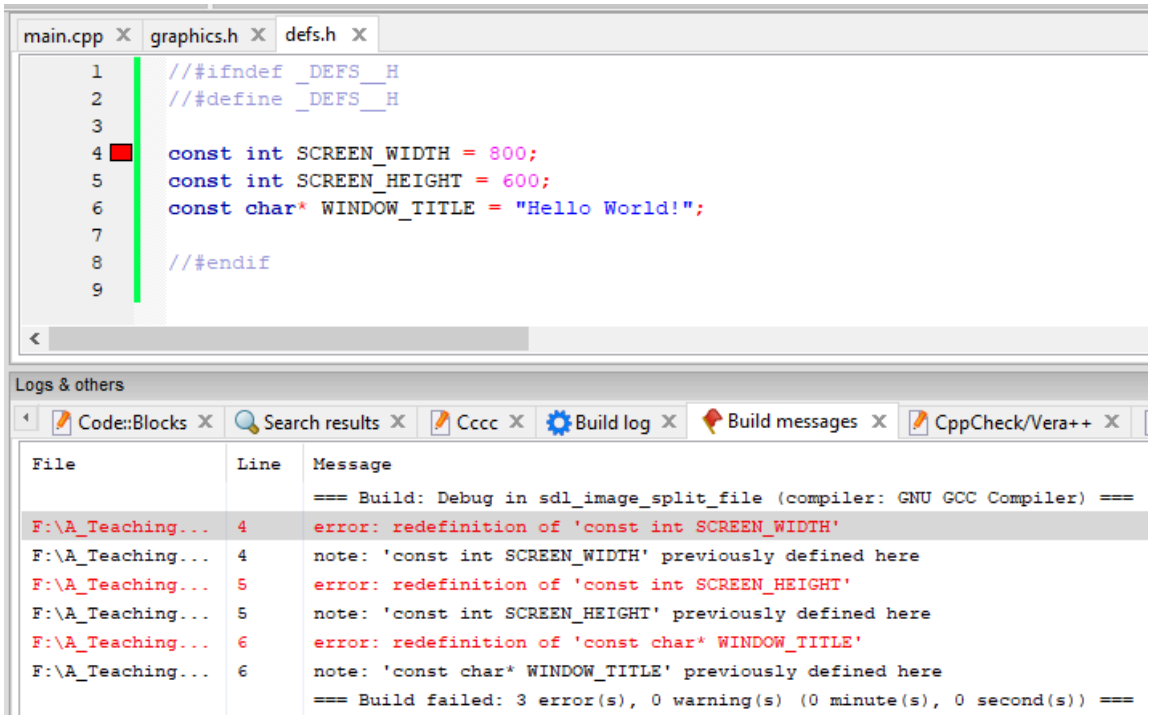
—----------

Build và chạy lại project để đảm bảo là cải tiến (refactor) vừa rồi không gây lỗi.
Ta đã tách thành công project thành ba file.

1. **main.cpp** chứa logic chính của chương trình. Hiện giờ hầu như không có gì mấy nhưng sau này sẽ thêm, và việc hiện giờ file này rất đơn giản dễ đọc sẽ tạo điều kiện thuận lợi cho việc thêm logic về sau.
   Hiện ở cuối hàm main có mấy dòng dọn dẹp mấy file ảnh có vẻ hơi kỹ thuật quá, không hợp cảnh. Đó là vì logic hiện giờ chỉ là demo ảnh. Sau này, khi ảnh được đóng gói vào các nhân vật / artifact trong game thì bạn sẽ thấy cả phần tạo ảnh lẫn hủy ảnh đều sẽ được để vào đúng chỗ của nó, tại code điều khiển các nhân vật và artifact tương ứng.
   Ta sẽ còn tiếp tục tách thêm file ra từ đây, khi mà logic của game tăng lên.
2. **defs.h** chứa các giá trị hằng để dùng chung cho toàn project. Hiện giờ chỉ có vài giá trị, nhưng sau này khi logic game được bổ sung thì sẽ có nhiều giá trị khác được bổ sung.
3. **graphics.h** chứa các chi tiết kỹ thuật vẽ SDL, hiện mới có các hàm cơ bản để khởi tạo, kết thúc, chuẩn bị và hiển thị màn hình game, nạp và vẽ ảnh. Sau này sẽ nảy sinh thêm các nhu cầu/tiện ích vẽ khác, ta sẽ bổ sung dần vào “thư viện” này.

Trên đây mới chỉ là cách tách file đơn giản dễ làm. Mỗi mô-đun (thư viện) của ta ở trên nằm hoàn toàn trong duy nhất một file header. Ta chưa quan tâm đến việc phân biệt giữa phần header và phần implement của mỗi mô-đun. Kỹ thuật này để sau.

Gợi ý thử nghiệm:
Bạn thử tạm bỏ (comment out) bộ khóa ở file defs.h và biên dịch lại project để xem lỗi redefinition



Thử xong nhớ bỏ comment out để chương trình hết lỗi biên dịch nhé.

Hiện giờ graphics.h chỉ được include ở main.cpp nên nếu bỏ khóa không bị lỗi. Tuy nhiên, không biết tương lai có dùng nhiều chỗ hay không, để không phải quan tâm đến lỗi này, ta tạo thói quen đặt khóa cho tất cả các file header.

Code: https://github.com/chauttm/gameProject/tree/main/03_split_file

Bài 3a. Tách tiếp graphics thành graphics.h và graphics.cpp (optional)

Nội dung này bạn có thể học cho biết chứ không cần thiết cho project bài tập lớn. Nếu bạn muốn học kỹ về C++ thì chủ đề này là có ích. Nó có ích cho việc tổ chức các mô-đun thư viện tái sử dụng cho các project khác. Nó không có ích và phức tạp không cần thiết khi dùng cho chỉ một project. Bạn có thể bỏ qua bài này.

Ở tình trạng của project bài trước, graphics.h chứa toàn bộ tất cả các khai báo, định nghĩa và cài đặt của struct Graphics và các hàm của nó. Tình trạng này rất tiện cho người viết chương trình, nhưng không chuẩn lắm theo quy tắc tách file thành header (API) và cpp (implementation). Để tách tiếp cho chuẩn thì graphics.h chỉ nên chứa định nghĩa struct với khai

báo các hàm thành viên, đó chính là API của mô-đun Graphics. Còn thân các hàm thành viên của Graphics là chi tiết cài đặt, không phải API, nên cần tách ra khỏi file header chứa API.

Lấy ví dụ hàm logErrorAndExit của Graphics cần được tách thành 2 phần:
Phần khai báo nằm tại graphics.h, ở bên trong định nghĩa struct Graphics. Để ý là khai báo hàm không có thân hàm mà chỉ có dấu chấm phảy kết thúc dòng khai báo.
—---------- graphics.h—---------
…
```
struct Graphics {
    SDL_Renderer *renderer;
    SDL_Window *window;

    void logErrorAndExit(const char* msg, const char* error);
…
```
—---------

Phần thân hàm chuyển sang file graphics.cpp.
—---------- graphics.cpp—---------
```
#include <SDL.h>
#include <SDL_image.h>
#include "graphics.h"

void Graphics::logErrorAndExit(const char* msg, const char* error)
{
    SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION, SDL_LOG_PRIORITY_ERROR,
                   "%s: %s", msg, error);
    SDL_Quit();
}
…
```
—---------
Để ý là graphics.cpp cần include các thư viện cho các hàm được dùng đến trong nội dung file, include cả graphics.h vì ở graphics.cpp có dùng đến kiểu dữ liệu Graphics.
Ngoài ra, vì hàm `logErrorAndExit` là thành viên của struct Graphics nhưng lại đang được viết ở bên ngoài phạm vi của struct Graphics (nó nằm ở file graphics.h), nên quan hệ “của Graphics” cần được ghi một cách tường minh, đó là lý do ta cần tiết đầu tố `Graphics::` đặt trước tên hàm `logErrorAndExit`. Tất cả các thành viên của struct khi được định nghĩa ở bên ngoài định nghĩa struct đều cần dùng cú pháp này để phân biệt với các biến/hàm loại thường (không thuộc về struct/class nào).
Ta làm tương tự với các hàm khác của struct Graphics.

Nếu trong mô-đun graphics có các hàm không thuộc struct Graphics, ta cũng tách thành hai phần tương tự như trên. Chẳng hạn nếu ta có hàm foo() ở mô-đun Graphics.
File graphics.h sẽ chứa khai báo hàm foo

```
void foo();
```

File graphics.cpp sẽ chứa định nghĩa hàm foo

```
void foo() {
    std::cout << "Hello";
}
```

Nhớ là foo() là hàm độc lập, nên ở đây ta không cần tiền tố Graphics:: hay gì đó tương tự cho foo().
Nảy sinh một vấn đề là lỗi định nghĩa nhiều lần đối với dòng sau trong defs.h
`const char* WINDOW_TITLE = "Hello World!";`
Để sửa một cách đơn giản, tại defs.h, ta thay dòng trên bằng
`#define WINDOW_TITLE = "Hello World!"`

Code:
https://github.com/chauttm/gameProject/blob/main/03_split_file_further/graphics.h
https://github.com/chauttm/gameProject/blob/main/03_split_file_further/graphics.cpp

Bài 4. Event chuột

Tóm tắt

Lấy tọa độ hiện tại của chuột: `SDL_GetMouseState(&x, &y);`

In ra màn hình để theo dõi: `cerr << x << ", " << y << endl;`

Cần poll event để lấy được tọa độ tiếp theo của chuột, nếu không sẽ bị tắc ở event chuột đầu tiên (thử mà xem).

```
SDL_Event event;
while (true) {
    SDL_PollEvent(&event);
    SDL_GetMouseState(&x, &y);
    cerr << x << ", " << y << endl;
    switch (event.type) {
        case SDL_QUIT:
            exit(0);
            break;
    }
    SDL_Delay(100);
}
```

Nạp texture cho ảnh con trỏ, làm một lần trước vòng lặp để dùng đi dùng lại.

```
SDL_Texture *targetterTexture = graphics.loadTexture("img/tinyBlackBox.png");
```

Vẽ hình con trỏ tại tọa độ chuột `graphics.renderTexture(targetterTexture, x, y);`

Kết quả là hình vẽ hiện song song với hình con trỏ thông thường. Để không hiện con trỏ, đặt vào cuối hàm init: `SDL_ShowCursor(0);`

Cần kết hợp các lệnh xóa màn hình để xóa dấu vết di chuyển của con trỏ.

Chi tiết

Ta cũng bắt đầu từ một bản sao của project bài trước (đã tách file defs và graphics). Nhớ sửa tên project mới thành mouse_event chẳng hạn để không bị lẫn.

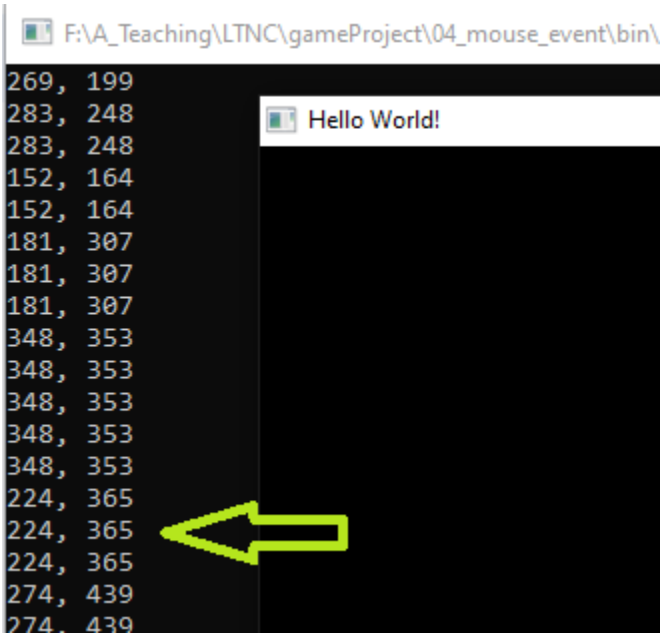
Cách lấy tọa độ chuột trong khi di chuyển.

Hãy sửa nội dung hàm main thành nội dung đơn giản sau để thấy những lệnh tối thiểu cần thiết để lấy được tọa độ chuột.

```
Graphics graphics;
```

```
graphics.init();

SDL_Event event;
int x, y;
while (true) {
    SDL_GetMouseState(&x, &y);
    cerr << x << ", " << y << endl;

    SDL_PollEvent(&event);
    switch (event.type) {
        case SDL_QUIT:
            exit(0);
            break;
    }
    SDL_Delay(100);
}

graphics.quit();
return 0;
```

Chạy thử, nhớ khua chuột trong cửa sổ Hello của chương trình, bạn sẽ thấy cửa sổ console đen bên dưới có in ra tọa độ x, y của chuột, nó thay đổi theo chuyển động của chuột.



Đó là kết quả in ra màn hình console kết quả của lệnh lấy tọa độ chuột
`SDL_GetMouseState(&x, &y);`

Lệnh trên sẽ không hoạt động như mong muốn nếu thiếu các điều kiện sau:

1. Đặt trong vòng lặp để được gọi đi gọi lại trong quá trình di chuyển chuột. cái này sau sẽ chính là game loop của chương trình game của các bạn.
2. Vòng lặp cần có độ trễ vừa phải, ở đây là trễ 100ms với lệnh SDL_Delay(100).
3. Trong vòng lặp cần lấy event (cả bàn phím lẫn chuột) từ lệnh `SDL_PollEvent(&event)` (quan trọng là có gọi, còn xử lý event nhận được như thế nào thì tùy logic của game). Ở đoạn code trên, event chỉ dùng để kiểm tra xem user đã click đóng cửa sổ game hay chưa.

Lưu ý rằng nội dung hàm main ở trên không vẽ gì, chỉ làm duy nhất việc demo lấy tọa độ chuột. Các hoạt động vẽ và xử lý logic nếu có sẽ ghép vào bên trong chính game loop sơ khai trong đó.

cerr: Như đã thấy ở kết quả chạy chương trình, cerr là lệnh in ra màn hình, cerr gần giống với cout. Điểm khác biệt là ở chỗ cerr là luồng output dành riêng cho báo lỗi và debug, (err là từ error) nó đổ lập tức nội dung ra màn hình và chỉ đổ ra màn hình. Trong khi đó, cout là luồng output có bộ nhớ đệm để giảm thời gian chương trình phải đợi việc ghi ra màn hình (ghi vào bộ nhớ đệm thì nhanh hơn ghi ra màn hình). Khi bộ nhớ đệm đầy thì nội dung cout mới được đổ ra màn hình.

Giờ ta có thể thêm một chút code để thử nghiệm thêm về việc sử dụng tọa độ chuột. Ta sẽ vẽ một hình chữ nhật, và liên tục kiểm tra xem chuột đang ở bên trong hay bên ngoài hình chữ nhật.

Vẽ hình chữ nhật. Ta không có nhu cầu vẽ đi vẽ lại màn hình, nên chỉ cần 1 đoạn ngắn nằm ngoài game loop (đoạn bôi đậm). Vẽ và hiển thị màn hình luôn một lần.

—----------main.cpp

```
Graphics graphics;
graphics.init();

SDL_Rect rect;
rect.x=100;
rect.y=100;
rect.h=100;
rect.w=100;
SDL_SetRenderDrawColor(graphics.renderer, 255, 255, 255, 0 );
SDL_RenderFillRect(graphics.renderer, &rect);
SDL_RenderPresent(graphics.renderer);

SDL_Event event;
int x, y;
```

—————————————

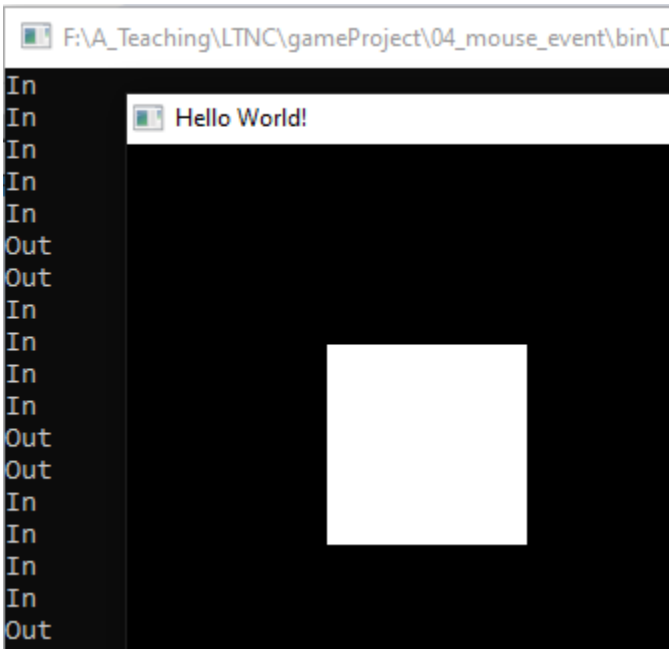
Kiểm tra vị trí chuột. Thay vì chỉ in x, y ra màn hình console như lúc trước, ta thay bằng một lệnh in theo điều kiện (dòng bôi đậm).

————————main.cpp————————

```
    SDL_GetMouseState(&x, &y);
    cerr << ((x > 100 && y > 100 && x < 200 && y < 200) ? "In\n" : "Out\n");


    SDL_PollEvent(&event);
```

—————————————

Kết quả chạy sẽ như sau, nhớ di chuột ra vào hình chữ nhật để thấy output thay đổi theo.



Cách lấy event click/thả chuột

Khi bạn làm gì đó với chuột (click, thả, phím trái, phải….scroll …), event đó đã được ghi nhận vào biến event bởi chính lệnh `SDL_PollEvent` đã có sẵn. Ta đã có sẵn lệnh switch với case dành cho event đóng cửa sổ game. Giờ ta chỉ cần thêm trường hợp xử lý cho trường hợp event chuột. Thêm hai case click và thả chuột vào trong lệnh switch như dưới đây, bạn có thể muốn comment out dòng cerr cũ về vị trí in/out chuột cho đỡ rối mắt.

————————main.cpp ————————

```
  while (true) {
        SDL_GetMouseState(&x, &y);
        //cerr << ((x > 100 && y > 100 && x < 200 && y < 200)...


        SDL_PollEvent(&event);
```

```
        switch (event.type) {
            case SDL_QUIT:
                 exit(0);
                 break;
            case SDL_MOUSEBUTTONDOWN:
                 cerr << "Down at (" << x << ", " << y << ")\n";
                 break;
            case SDL_MOUSEBUTTONUP:
                 cerr << "Up at (" << x << ", " << y << ")\n";
                 break;
        }
        SDL_Delay(100);
    }
```

——----------

Chạy thử và bạn sẽ thấy cửa sổ console ghi ra các sự kiện nhấn và thả chuột, ví dụ:

```
Down at (159, 135)
Up at (159, 135)
Down at (159, 135)
Up at (136, 177)
```

Hãy thử click (nhấn rồi thả nhanh) và kéo (nhấn, giữ, di chuột, rồi mới thả), thử tổ hợp trái và phải, xem hiệu ứng khác nhau như thế nào.
Đoạn code trên chưa quan tâm phân biệt chuột trái và phải. Nếu muốn phân biệt trái và phải, bạn cần kiểm tra event.button có giá trị là SDL_BUTTON_LEFT hay SDL_BUTTON_RIGHT. Về chi tiết, nếu bạn có thể tự tìm hiểu (https://wiki.libsdl.org/SDL2/SDL_MouseButtonEvent).

Lưu ý: các phần code ở trên là code tối thiểu với mục đích duy nhất là chỉ dẫn cách thực hiện một số công việc. Bạn cần tổ chức code tốt hơn (hằng, hàm, tên biến …..) khi viết vào logic chương trình.

Đến đây bạn đã có đủ công cụ kỹ thuật để làm game đơn giản như TicTacToe hay Memory rồi.

Bài 5. Game Tictactoe

Ta sẽ dùng các kỹ thuật đã học ở trên để làm game Tictactoe đơn giản.

Chuẩn bị code: Từ project của bài sự kiện chuột. Tạo project mới, chỉnh code hàm main để bỏ bớt code demo lần trước.

—-------main.cpp ------------------------

```
int main(int argc, char *argv[])
{
    Graphics graphics;
    graphics.init();

    int x, y;
    SDL_Event event;
    bool quit = false;
    while (! quit) {
        SDL_PollEvent(&event);
        switch (event.type) {
            case SDL_QUIT:
                quit = true;
                break;
            case SDL_MOUSEBUTTONDOWN:
                SDL_GetMouseState(&x, &y);
                cerr << "Mouse-down at (" << x << ", " << y << ")\n";
                break;
        }
        SDL_Delay(100);
    }

    graphics.quit();
    return 0;
}
```

—-------------------------------

Để ý là ta chỉ cần dùng đến sự kiện mouse down cho việc user click vào ô bàn cờ. Tại vị trí hiện giờ chỉ output thông tin event mouse down, ta sẽ xử lý logic việc đánh cờ và hiển thị bàn cờ đã có thêm nước mới.

Ta cũng chỉnh case SDL_QUIT để chỉ nhảy ra khỏi game loop chứ không đột ngột dừng chương trình như ở bài trước (nhớ là ta cần dọn dẹp SDL trước khi dừng hẳn). Chạy thử đi để biết là chương trình vẫn ghi nhận event chuột.

Bài này bắt đầu có chút logic game, nên ta bắt đầu đưa game vào code hàm main. Đầu tiên là khởi tạo game, vẽ màn hình đầu tiên của game. Tiếp theo là chỉnh phần xử lý event chuột để có thêm xử lý logic và vẽ tình trạng mới của game.

—-------main.cpp -------------------------

…

```
Graphics graphics;
graphics.init();

Tictactoe game;
game.init();
graphics.render(game);
```

…

```
            case SDL_MOUSEBUTTONDOWN:
                 SDL_GetMouseState(&x, &y);
                 processClick(x, y, game);
                 graphics.render(game);
                 break;
```

…

—------ ------------------

Bạn đã có khung xử lý của chương trình với đầy đủ các bước. Công việc hiện giờ là thêm dần chi tiết và logic cho từng bước cho đến khi hoàn thiện game. File main.cpp giờ đã xong.

Thêm hàm **processClick**vào file main.cpp với nội dung có tính chất chuẩn bị cho logic chi tiết (hiện chưa biết tính hàng cột từ tọa độ chuột x, y như thế nào), nhưng hàm này có nhiệm vụ chuyển từ tọa độ chuột sang tọa độ hàng cột và gọi hàm update logic game

—-------main.cpp -------------------------

```
void processClick(int x, int y, Tictactoe& game) {
    // chuyển tọa độ màn hình x, y thành tọa độ hàng cột của game
    int clickedCol = 0; // todo
```

```
    int clickedRow = 0; // todo
    game.move(clickedRow, clickedCol);
}
```

—-------main.cpp ------------------------

Hãy tạo một file mới, chẳng hạn logic.h, ở trong project, viết struct Tictactoe, tạm thời chỉ có mấy hàm rỗng để tránh lỗi biên dịch. Nhớ thêm dòng #include “logic.h” vào đầu file main.cpp

—--------logic.h-----------------------

```
#ifndef _LOGIC__H
#define _LOGIC__H

struct Tictactoe {
   void init() {    }
   void move(int row, int col) { }
};

#endif
```

—---------------------------------

Để ý thấy ta mới thêm lệnh graphics.render(game) vào hàm main. Do đó ta cũng cần bổ sung hàm render (Tictactoe& game) vào trong struct Graphics. Nhớ thêm dòng #include “logic.h” vào đầu file graphics.h

—-—----------graphics.h----------------------

```
struct Graphics {
   SDL_Renderer *renderer;
   SDL_Window *window;
   …
   void render(const Tictactoe& game) {
      // prepareScene: có thể xóa màn hình hoặc vẽ hình nền, hoặc không gì cả, tùy

      // vẽ game board

      presentScene();
   }
  …
```

—---------------

Đầu tiên là chuẩn bị màn hình, có thể xóa rỗng, có thể load hình nền, hoặc không làm gì cả nếu ta sẽ chỉ toàn vẽ đè như ở game này. Tiếp theo là vẽ bàn cờ. Cuối cũng là render màn hình (ta có sẵn hàm presentScene() rồi bạn nhớ không?).

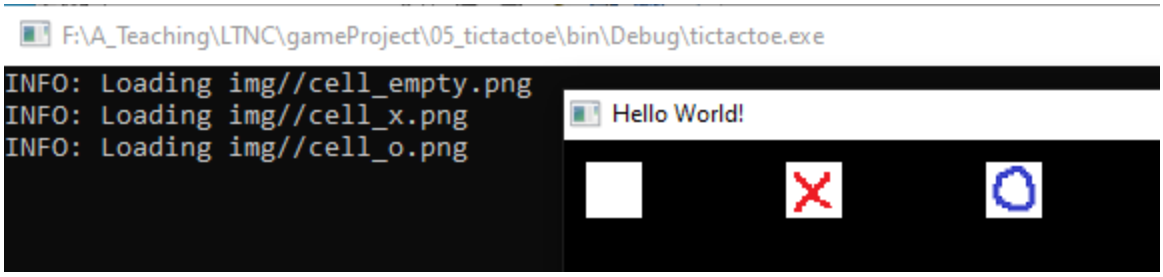
Đến đây vẫn chưa vẽ gì, nhưng bạn vẫn dịch và chạy thử đi để biết chắc là các đoạn code vừa thêm khớp nhau về cú pháp và không phá mất cái gì.

Chuẩn bị ảnh và vẽ thử: Vẽ 3 ảnh vuông kích thước 30x30 pixel cho 3 trạng thái của ô: trống, x, và o. Dùng Paint hoặc phần mềm nào tùy bạn hoặc download từ github (https://github.com/chauttm/gameProject/tree/main/05_tictactoe/img). Bàn cờ 3x3 sẽ được vẽ từ các ô này. Để cho gọn gàng cấu trúc file trong thư mục project, bạn tạo thư mục img trong thư mục project và đặt 3 file trên vào trong đó.

Sốt ruột muốn vẽ lắm rồi phải không? Bài 2 đã có kỹ thuật load texture và vẽ ảnh. Hãy thêm nhanh một đoạn code vào trong hàm render nói trên để nhanh chóng vẽ 3 cái ảnh kia lên. (Nhớ là công việc init sdl_image, các hàm loadTexture và renderTexture đã có sẵn trong hàm graphics.init rồi, nên không phải để ý đến nó nữa, quá tiện lợi!)

—----------graphics.h----------
```
struct Graphics {
  ….
   void render(const Tictactoe& game) {
       //prepareScene();
       SDL_Texture* cellEmpty = loadTexture("img//cell_empty.png");
       renderTexture(cellEmpty, 10, 10);

       SDL_Texture* cellX = loadTexture("img//cell_x.png");
       renderTexture(cellX, 110, 10);

       SDL_Texture* cellO = loadTexture("img//cell_o.png");
       renderTexture(cellO, 210, 10);

       presentScene();
    }
```
—----------

Yay!!! Chạy chương trình và đã thấy 3 ô vuông.



Giờ thì dọn dẹp trước khi vẽ tử tế.

Đầu tiên là 3 cái texture kia sẽ dùng rất nhiều trong khi vẽ bàn cờ, nhưng chỉ cần nạp một lần. Do đó ta sẽ chuyển 3 biến texture ra ngoài làm biến thành viên của struct Graphics và nạp nó khi khởi tạo graphics.

—---------- graphics.h ----------

```
struct Graphics {
    SDL_Renderer *renderer;
    SDL_Window *window;
    SDL_Texture *cellEmpty, *cellX, *cellO;

    void init() {
        initSDL();
        cellEmpty = loadTexture("img//cell_empty.png");
        cellX = loadTexture("img//cell_x.png");
        cellO = loadTexture("img//cell_o.png");
    }

    void render(const Tictactoe& game) {
        //prepareScene();
        renderTexture(cellEmpty, 10, 10);
        renderTexture(cellX, 110, 10);
        renderTexture(cellO, 210, 10);

        presentScene();
    }
    ...
    void quit()
    {
        SDL_DestroyTexture(cellEmpty);
```

```
        cellEmpty = nullptr;
        SDL_DestroyTexture(cellX);
        cellX = nullptr;
        SDL_DestroyTexture(cellO);
        cellO = nullptr;

        IMG_Quit();

        SDL_DestroyRenderer(renderer);
        SDL_DestroyWindow(window);
        SDL_Quit();
    }
…
```

—----------

Để ý là hàm init() giờ chỉ còn mấy dòng. Hàm init() cũ vốn chứa rất nhiều logic khởi tạo được đổi tên thành initSDL() và gọi từ bên trong hàm init() mới cho gọn. Ta cũng thêm các lệnh hủy texture vào đầu hàm quit() để cho đủ quy trình tạo và dọn dẹp. Để ý là việc này không ảnh hưởng đến API của code hàm main gọi graphics.init() hay graphics.quit().

Nhớ chạy lại để chắc chắn là bạn chưa gây lỗi gì.

Vẽ game board.

Để tính toán tọa độ vẽ mỗi ô, ta cần tọa độ góc trên trái của bàn cờ và kích thước của mỗi ô. Các giá trị này được dùng ở graphics.cpp cũng như chỗ nào đó sẽ xử lý event chuột, do đó ta để nó ở dạng hằng ở defs.h để dùng chung như là cấu hình game. Để ý là mấy cái ảnh của ta kích thước 30x30 nên kích thước ô bàn cờ CELL_SIZE cũng là 30 cho tiện. BOARD_X BOARD Y thì tùy ý

—----------defs.h----------

```
#ifndef _DEFS__H
#define _DEFS__H

const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;
const char* WINDOW_TITLE = "Hello World!";
#define BOARD_X 10
#define BOARD_Y 10
#define CELL_SIZE 30
…
```

—————————————

Ta sửa hàm render của Graphics để vẽ bàn cờ với trạng thái ô (trống, X, O) tạm quyết định bằng một logic đơn giản linh tinh không liên quan đến logic game (đã làm đâu mà dùng).

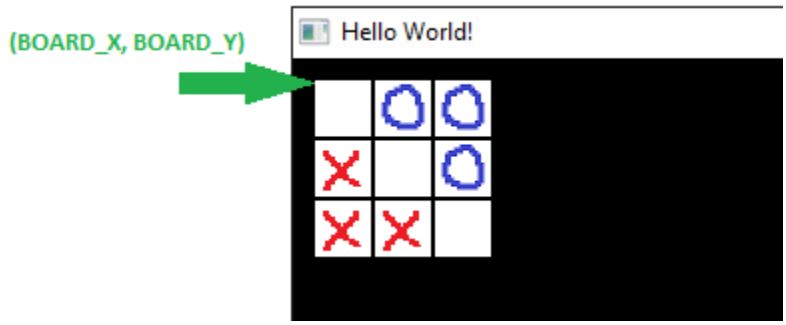
—————graphics.cpp ------

```
  void render(const Tictactoe& game) {
        //prepareScene();

        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++) {
                int x = BOARD_X + j * CELL_SIZE;
                int y = BOARD_Y + i * CELL_SIZE;
                if (i == j) renderTexture(cellEmpty, x, y);
                if (i > j) renderTexture(cellX, x, y);
                if (i < j) renderTexture(cellO, x, y);
            };

        presentScene();
    }
```

…

—————————

Chạy thử đi, không đến nỗi tệ phải không?



Cuối cùng là logic game và xử lý input.

Game đơn giản này chỉ cần có một mảng 2 chiều ghi trạng thái các ô và hàm move

—————————logic.h ----------

```
#ifndef _LOGIC__H
#define _LOGIC__H
```

```
#define BOARD_SIZE 3
#define EMPTY_CELL ' '
#define O_CELL 'o'
#define X_CELL 'x'

struct Tictactoe {
    char board[BOARD_SIZE][BOARD_SIZE];
    char nextMove = O_CELL;

    void init() {
        for (int i = 0; i < BOARD_SIZE; i++)
            for (int j = 0; j < BOARD_SIZE; j++) board[i][j] = EMPTY_CELL;
    }
    void move(int row, int column) {
        if (row >= 0 && row < BOARD_SIZE &&
            column >= 0 && column < BOARD_SIZE)
        {
            board[row][column] = nextMove;
            nextMove = (nextMove == O_CELL) ? X_CELL : O_CELL;
        }
    }
};

#endif
```

——————————————————

Chỉnh hàm render cho khớp logic game

——------graphics.cpp ------

```
void render(const Tictactoe& game) {
        //prepareScene();

        for (int i = 0; i < BOARD_SIZE; i++)
            for (int j = 0; j < BOARD_SIZE; j++) {
                int x = BOARD_X + j * CELL_SIZE;
```

```
                int y = BOARD_Y + i * CELL_SIZE;
                switch (game.board[i][j]) {
                    case EMPTY_CELL: renderTexture(cellEmpty, x, y); break;
                    case X_CELL: renderTexture(cellX, x, y); break;
                    case O_CELL: renderTexture(cellO, x, y); break;
                };
            };

        presentScene();
    }
```

—----------

Và đến giờ, khi logic chuyển đổi giữa tọa độ và hàng cột đã rõ ràng, ta hoàn thiện hàm processClickAt ở main.cpp

—-------main.cpp ------------------------

```
void processClickAt(int x, int y, Tictactoe& game) {
    // chuyển tọa độ màn hình x, y thành tọa độ hàng cột của game
    int clickedCol = (x - BOARD_X) / CELL_SIZE;
    int clickedRow = (y - BOARD_Y) / CELL_SIZE;
    game.move(clickedRow, clickedCol);
}
```

—-------main.cpp ------------------------

Build và chạy thử đi. Game chơi được rồi nhé!

Code: https://github.com/chauttm/gameProject/tree/main/05_tictactoe

Hạn chế:

- Hơi lag vì vòng lặp lúc nào cũng delay 100ms. Thực ra game loại này không cần delay, chỉ cần đợi event tiếp theo thôi. Bạn có thể tìm cách chỉnh code sau khi học bài về sự kiện bàn phím
- Chưa kiểm tra để cấm đánh trùng ô
- Chưa kiểm tra và báo thắng thua
- Chưa cho chơi nhiều ván.
- Kích thước bàn cờ và ô bị cố định theo kích thước ảnh (30x30)
- Chưa loại các event ở quanh ranh giới giữa hai ô, người dùng khó mà nhìn thấy được việc dịch 1 pixel thì game sẽ tính là ô bên cạnh. Tốt nhất là bỏ qua các eventclick vào mép ô.
- Kích thước màn hình và tiêu đề cửa sổ vẫn giữ nguyên như bài đầu tiên, nên chỉnh để đẹp hơn.
- Nền game hiện trống không có gì cả, có thể dùng ảnh vẽ nền cho đẹp hơn

- …

Bạn có thể tự mày mò sửa chương trình để cải thiện. ***Nhớ là bạn chỉ cần hiểu code ở trên thôi đã đủ để lấy 4 điểm bài tập lớn; bạn viết thêm code để game hoàn thiện hơn thì sẽ được điểm cao hơn.***

Bài 6. Event bàn phím

Chuẩn bị code: tạo project mới dựa trên sản phẩm Bài 4. Event chuột.

Cách đơn giản:

Để bước đầu bắt sự kiện bàn phím, hãy sửa hàm main() để có nội dung như dưới đây. Để ý các dòng code mầu lam. Dòng có vòng while chạy SDL_PollEvent sẽ đợi bắt một sự kiện chuột hoặc bàn phím. Hai khối case SDL_KEYDOWN và SDL_KEYUP xử lý hai trường hợp phím nhấn và phím thả, in ra scancode của phím đó. Khối default in ra dấu chấm để ta biết có một event không phải phím bấm, bạn click chuột chẳng hạn.

—----------main.cpp —----------

```
int main(int argc, char *argv[])
{
    Graphics graphics;
    graphics.init();

    SDL_Event event;
    while (true) {
        while (SDL_PollEvent(&event)) {
            switch (event.type) {
                case SDL_QUIT:
                    exit(0);
                    break;
                case SDL_KEYDOWN:
                    cerr << "\nDown: " << event.key.keysym.scancode;
                    break;
                case SDL_KEYUP:
                    cerr << "\nUp: " << event.key.keysym.scancode;
                    break;
                default: cerr << "\n.";
            }
        }
        SDL_Delay(100);
    }

    graphics.quit();
```

```
    return 0;
}
```

Hãy chạy thử, nhấn và thả các phím trên bàn phím xem chương trình in ra cái gì.

Bạn có thể nhìn thấy output tương tự như sau ở cửa sổ console

```
Down: 9
Down: 10
Down: 11
.
Up: 9
Up: 10
Up: 11
Down: 7
.
Down: 9
```

Scancode được in ra là mã của phím vật lý trên bàn phím, bạn thử gõ A hay a sẽ thấy như nhau. Bạn có thể tra xem scancode của các phím cụ thể tại bảng này: https://wiki.libsdl.org/SDL2/SDLScancodeLookup

Nếu muốn kiểm tra xem phím vừa nhận có phải phím C không chẳng hạn, bạn có thể dùng điều kiện (`event.key.keysym.scancode ==` SDL_SCANCODE_C). Cột *SDL_Scancode Constant* là các hằng được định nghĩa sẵn cho scancode của từng phím.

Chẳng hạn, bạn có thể ghép logic xử lý các phím mũi tên một cách đơn giản như sau

---

```
    case SDL_KEYDOWN:
        if (event.key.keysym.scancode == SDL_SCANCODE_UP) goUp();
        if (event.key.keysym.scancode == SDL_SCANCODE_DOWN) goDown();
        if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) goLeft();
        if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) goRight();
        break;
    …
```

Nếu bạn có nhịp game chậm, hoặc không có nhịp đều đặn (kiểu như tictactoe và memory), bạn muốn xử lý chính xác từng event phím up và down, ví dụ down thì làm một hoạt động nào đó, up thì làm một hoạt động khác, thì bạn nên làm theo cách trên.

Nếu game của bạn chạy nhịp nhanh, và bạn quan tâm phím nào đang được giữ, thì có một cách đơn giản hơn và nó cho phép dễ dàng xử lý tổ hợp phím (giữ một lúc nhiều phím). Ví dụ giữ đồng thời hai phím Up và Right thì nhân vật sẽ nhảy lên cao về bên phải, còn nếu chỉ nhấn Up thì nhân vật chỉ nhảy lên tại chỗ.

Sửa hàm main() thành như dưới đây. Lệnh SDL_PollEvent không thể bỏ qua, nó không được gọi thì trạng thái phím không được ghi nhận (còn nhớ event chuột cũng tương tự???) Ta cần đặt SDL_PollEvent trong vòng while để mỗi lần đều đọc hết các event đã xảy ra nhưng chưa được đọc.

—————main.cpp---------

```
int main(int argc, char *argv[])
{
    Graphics graphics;
    graphics.init();

    bool quit = false;
    SDL_Event event;
    while (!quit) {
        //Handle events on queue
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) quit = true;
        }

        const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

        if (currentKeyStates[SDL_SCANCODE_UP] ) cerr << " Up";
        if (currentKeyStates[SDL_SCANCODE_DOWN] ) cerr << " Down";
        if (currentKeyStates[SDL_SCANCODE_LEFT] ) cerr << " Left";
        if (currentKeyStates[SDL_SCANCODE_RIGHT] ) cerr << " Right";

        cerr << ".\n";

        SDL_Delay(100);
    }

    graphics.quit();
    return 0;
}
```

—————————————

Hãy chạy thử và thử giữ một phím và giữ tổ hợp các phím mũi tên xem kết quả in ra như thế nào. Bạn sẽ thấy các phím được giữ đồng thời được output trên cùng một dòng. Các nhịp không có event là phím mũi tên là các dòng chứa duy nhất một dấu chấm. Ví dụ:

.

.

```
 Left.
 Up Left.
 Up Left.
 Up Left.
 Up Left.
.
 Right.
 Right.
 Left Right.
 Left.
.
.
 Down Left Right.
```

Bạn thử thêm vào code các dòng if cho các phím khác để nghịch tiếp. Nhớ tra bảng https://wiki.libsdl.org/SDL2/SDLScancodeLookup để biết tên hằng ứng với các phím khác.

Lưu ý: Nếu nhịp game của bạn chậm quá, trong khi người chơi bấm phím nhanh quá, nhấn và thả ngay trong một nhịp game, thì cách xử lý trên sẽ bỏ qua sự kiện đó. Bạn hãy thử cho delay 1000ms và nhấn phím thật nhanh trong khi chạy chương trình để xem hiện tượng này xảy ra như thế nào.

Code: https://github.com/chauttm/gameProject/tree/main/06_keyboard_event

Bài 7a. Hoạt hình - Vật di chuyển

Chuẩn bị code: Bắt đầu từ sản phẩm của Bài 6 - Event bàn phím.

Ở bài này ta làm một vật chuyển động đều trong màn hình game, có thể dùng các phím mũi tên để đổi hướng.

Thông tin về một đối tượng chuyển động gồm có tọa độ và vận tốc. Tại mỗi nhịp game, ta thay đổi tọa độ của đối tượng theo vận tốc hiện hành. Ví dụ ta làm struct Mouse.

```
struct Mouse {
    int x, y;
    int dx = 0, dy = 0;
    int speed = INITIAL_SPEED;
    void move() {
        x += dx; y += dy;
    }
    void turnNorth() {
        dy = -speed; dx = 0;
    }
    void turnSouth() {
        dy = speed; dx = 0;
    }
    void turnWest() {
        dy = 0; dx = -speed;
    }
    void turnEast() {
        dy = 0; dx = speed;
    }
};
```

Ta làm sẵn các hàm đổi hướng để có thể gọi từ bên ngoài, game loop hoặc trong logic của game, mà không phải nhớ chi tiết về các thông tin bên trong.

Để vẽ vật di chuyển, tại mỗi nhịp game, ta cần làm đủ 3 bước: xóa màn hình, cập nhật tọa độ của đối tượng, render game với đối tượng ở vị trí mới.

Để thêm chút thú vị, ta dùng đoạn code dùng event bàn phím từ bài trước và ghép các lời gọi đổi hướng chuyển động

——————————

```
    Mouse mouse;
    mouse.x = SCREEN_WIDTH / 2;
    mouse.y = SCREEN_HEIGHT / 2;
….
    while (!quit && !gameOver(mouse)) {
        graphics.prepareScene();

        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) quit = true;
        }
        const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);
        if (currentKeyStates[SDL_SCANCODE_UP]) mouse.turnNorth();
        if (currentKeyStates[SDL_SCANCODE_DOWN]) mouse.turnSouth();
        if (currentKeyStates[SDL_SCANCODE_LEFT]) mouse.turnWest();
        if (currentKeyStates[SDL_SCANCODE_RIGHT]) mouse.turnEast();

        mouse.move();
        render(mouse, graphics);

        graphics.presentScene();
        SDL_Delay(10);
    }
```

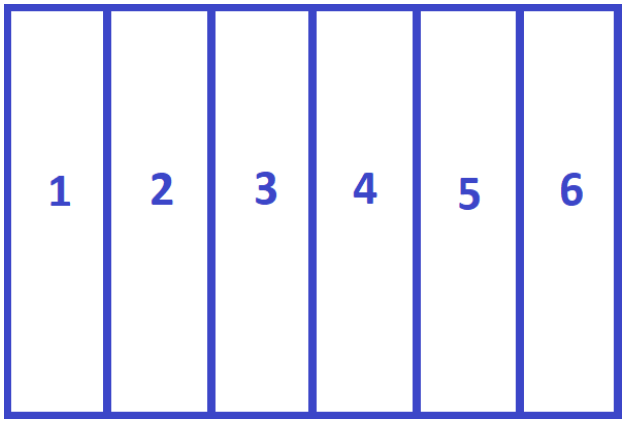
——————————

Code: https://github.com/chauttm/gameProject/tree/main/07a_moving_object

Bài 7. Hoạt hình - Background trôi

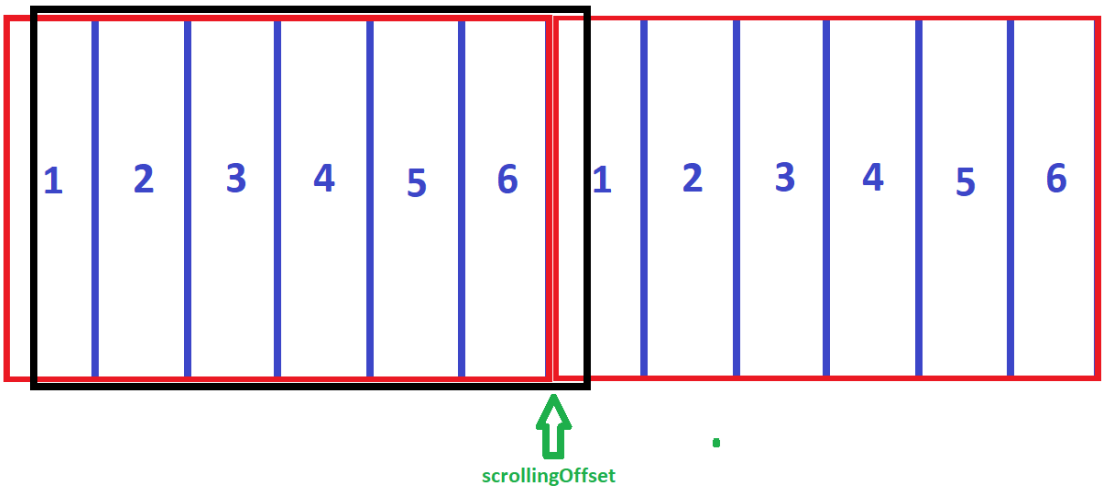
Chuẩn bị code: Bắt đầu từ sản phẩm của Bài 3 - Project có nhiều file.

Chuẩn bị ảnh: cần 1 ảnh có chiều rộng bằng chiều rộng của cửa sổ game, hoặc sửa chiều rộng cửa sổ game cho vừa bằng ảnh. Để dễ minh họa chiến lược trôi, ta tạm dùng ảnh đơn giản sau, bạn sẽ dễ thấy biên ảnh cũng như các sọc ảnh.



Chiến lược: Vẽ hình nền hai lần nối tiếp nhau, hình bên phải tại tọa độ (scrollingOffset, 0), hình bên trái tại tọa độ (scrollingOffset - width, 0). Ở hình minh họa bên dưới, khung chữ nhật mầu đen là cửa sổ game. Với tọa độ vẽ như trên thì mỗi thời điểm ta chỉ nhìn thấy khúc sau của hình bên trái và khúc đầu của hình bên phải. Khi giá trị scrollingOffset bằng chiều rộng màn hình thì ta chỉ nhìn thấy hình bên trái, khi scrollingOffset giảm dần thì hai hình dịch dần sang trái, ta nhìn thấy nền chạy dần sang trái. Khi scrollingOffset giảm về 0 thì ta reset nó về độ rộng của màn hình và mọi chuyện lặp lại.

Ta thay đổi giá trị của scrollingOffset khi nào? Đều đặn tại mỗi nhịp của gameloop.



Để điều khiển từ chương trình chính, ta thêm vài dòng code vào hàm main để (1) khởi tạo background, (2) cho ảnh nền trôi tại mỗi nhịp game, vẽ nền; và (3) giải phóng ảnh trước khi kết thúc chương trình.

—-- main.cpp----------

```
    Graphics graphics;
```

```
    graphics.init();

    ScrollingBackground background;
    background.setTexture(graphics.loadTexture(BACKGROUND_IMG));

    bool quit = false;
    SDL_Event e;
    while( !quit ) {
        while( SDL_PollEvent( &e ) != 0 ) {
            if( e.type == SDL_QUIT ) quit = true;
        }

        background.scroll(1);
        graphics.render(background);

        graphics.presentScene();
        SDL_Delay(100);
    }

    SDL_DestroyTexture( background.texture );
    graphics.quit();
```

—----------

ScrollingBackground sẽ là cấu trúc dữ liệu dùng để vẽ và quản lý một cái background có khả năng trôi. Dữ liệu về một background gồm có ảnh nền, giá trị scrollingOffset, kích thước chiều rộng của ảnh. Cấu trúc ScrollingBackground còn cần hai hàm xử lý dữ liệu: setTexture để lấy ảnh nền và lấy độ rộng ảnh. Scroll để thay đổi giá trị scrollingOffset.

Ta cũng cần thêm hàm render để vẽ nền theo chiến lược nói trên. Ta cài hàm này ở trong struct Graphics để tiện dùng các hàm tiện ích và bút vẽ sẵn có tại đó.

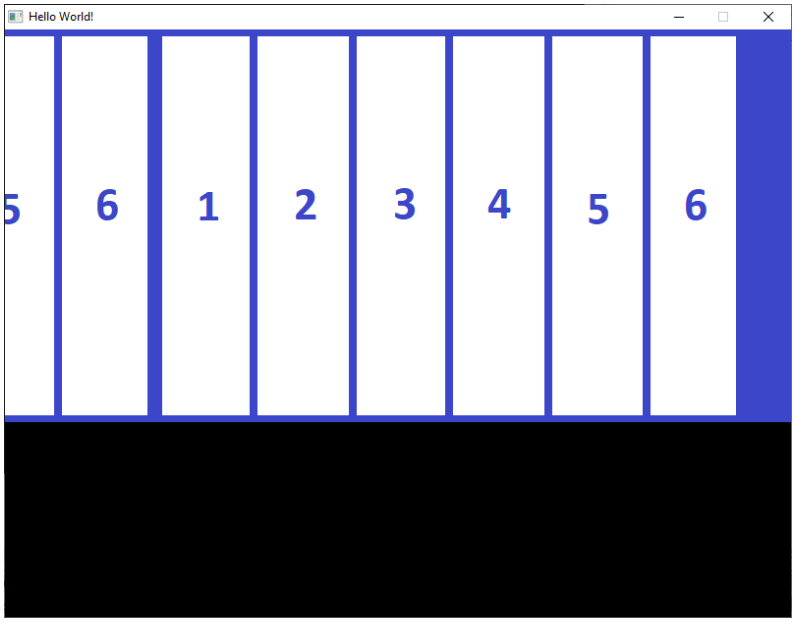
Như vậy là ta có kiểu dữ liệu tổng quát dùng cho việc tạo hình nền trôi.

—----------Graphics.h----------

```
struct ScrollingBackground {
    SDL_Texture* texture;
    int scrollingOffset = 0;
```

```
    int width, height;

    void setTexture(SDL_Texture* _texture) {
        texture = _texture;
        SDL_QueryTexture(texture, NULL, NULL, &width, &height);
    }

    void scroll(int distance) {
        scrollingOffset -= distance;
        if( scrollingOffset < 0 ) { scrollingOffset = width; }
    }
};

struct Graphics {
    ...
    void render(const ScrollingBackground& bgr) {
        renderTexture(bgr.texture, bgr.scrollingOffset, 0);
        renderTexture(bgr.texture, bgr.scrollingOffset - bgr.width, 0);
    }
};
```

Bạn chạy thử để thấy nó chạy đúng, hình sọc tuy xấu nhưng dễ phát hiện lỗi tọa độ. Chẳng hạn bạn thử tăng chiều rộng cửa sổ màn hình, đợi chương trình chạy một lúc xem có bị lỗi vẽ hụt nền không? Ví dụ:



Đấy là khi chiều rộng màn hình dài hơn chiều rộng ảnh nền, khi ảnh bên phải nằm trọn trong màn hình rồi dịch dần sang trái thì phần màn hình bên phải nó bị trống. Khúc mầu blue bên phải trong hình trên là vết của các lần vẽ trước (vì không xóa màn hình). Đấy là lý do bạn nên tạo ảnh nền rộng bằng cửa sổ, hoặc bạn cần chỉnh thuật toán hàm render background để vẽ nhiều hơn 2 hình nền nối tiếp (vẽ sao cho phủ đủ chiều rộng màn hình thì mới thôi).

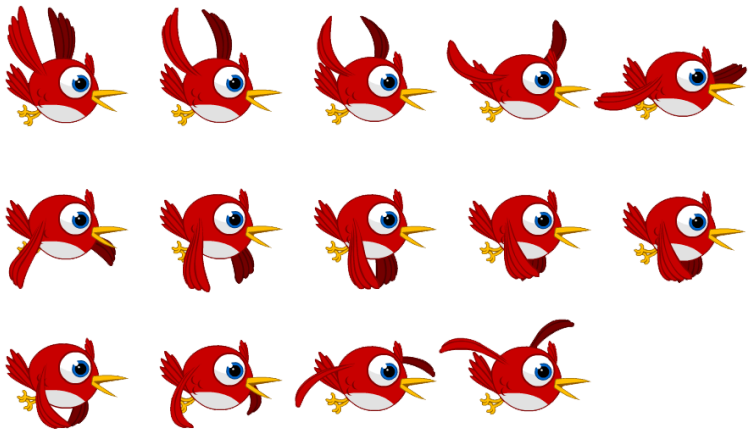
Sau khi chạy thấy ổn thì bạn thay bằng một hình nền đẹp phù hợp với game của bạn.

Code: https://github.com/chauttm/gameProject/tree/main/07_animation

Bài 8. Hoạt hình - Nhân vật hoạt động - Sprite

Vẽ một hình chuyển động về bản chất là vẽ một chuỗi các khung hình. Có thể là một bộ file ảnh, mỗi khung dùng một file ảnh. Hoặc có thể dùng 1 ảnh trên đó có nhiều khung hình, mỗi lần (frame) chỉ vẽ một phần.

Ví dụ Ảnh hoạt hình lấy từ LazyFoo (người đi) và từ https://freepngimg.com/download/sprite/83127-sprite-area-line-animated-bird-film.png (chim vỗ cánh)



Chuẩn bị code: Bắt đầu từ sản phẩm của Bài 3 - Project có nhiều file.

Để ghi thông tin về một bộ các khung hình, ta tạo cấu trúc Sprite, trong đó chứa thông tin về SDL_Texture từ ảnh chứa các khung hình, danh sách các khung hình là các SDL_Rect chứa tọa độ khung hình trong ảnh. Và để phục vụ cho việc lần lượt mỗi nhịp game lại vẽ một khung hình trong chuỗi, ta cần một biến ghi chỉ số của khung hình hiện tại, chỉ số này sẽ quay vòng trong chuỗi các khung hình.

—---------graphics.h --------

```
struct Sprite {
    SDL_Texture* texture;
    std::vector<SDL_Rect> clips;
    int currentFrame = 0;

    void init(SDL_Texture* _texture, int frames, const int _clips [][4]) {
        texture = _texture;
        SDL_Rect clip;
        for (int i = 0; i < frames; i++) {
            clip.x = _clips[i][0];
            clip.y = _clips[i][1];
            clip.w = _clips[i][2];
            clip.h = _clips[i][3];
            clips.push_back(clip);
```

```
        }
    }
    void tick() {
        currentFrame = (currentFrame + 1) % clips.size();
    }

    const SDL_Rect* getCurrentClip() const {
        return &(clips[currentFrame]);
    }
};
```

—————————————

Hàm init() có nhiệm vụ khởi tạo bộ ảnh hoạt hình từ texture, số khung hình (frames) và mảng chứa tọa độ các khung hình trong ảnh.

Hàm tick() đẩy chỉ số khung hình hiện tại tới khung tiếp theo trong chuỗi, hàm này sẽ được gọi ở mỗi nhịp gameloop.

Hàm getCurrentClip() trả về khung hình hiện tại để tiện cho việc vẽ.
Cả ba hàm này đều nhằm mục đích tạo điều kiện thuận lợi cho việc lập trình sử dụng và điều khiển hoạt hình từ các mô-đun khác mà không phải nhớ quy cách tổ chức dữ liệu ở bên trong Sprite.

Tiếp theo, ta cần một hàm vẽ texture mà chỉ vẽ một vùng hình chữ nhật trong ảnh thay vì vẽ toàn bộ ảnh.

Ở Bài 2, ta đã dùng lệnh sau để vẽ toàn bộ một ảnh ra màn hình vào vị trí của khung chữ nhật dest.

```
SDL_RenderCopy(renderer, texture, NULL, &dest);
```

Để ý là tham số thứ 3 khi đó là NULL, đó chính là tham số SDL_Rect* quy định khung chữ nhật chứa vùng ảnh cần vẽ.
Với các hàm đã chuẩn bị sẵn ở Sprite, ta có thể viết thêm một hàm sau trong struct Graphics với nhiệm vụ vẽ khung hình hiện tại của sprite đã cho tại tọa độ x,y.

—————————————————————

```
   void render(int x, int y, const Sprite& sprite) {
        const SDL_Rect* clip = sprite.getCurrentClip();
        SDL_Rect renderQuad = {x, y, clip->w, clip->h};
        SDL_RenderCopy(renderer, sprite.texture, clip, &renderQuad);
    }
```

—————————————————————

Đến đây ta có thể demo đơn giản tại gameloop

—---------- main.cpp ----------

```
    Sprite man;
    SDL_Texture* manTexture = graphics.loadTexture(MAN_SPRITE_FILE);
    man.init(manTexture, MAN_FRAMES, MAN_CLIPS);

    Sprite bird;
    SDL_Texture* birdTexture = graphics.loadTexture(BIRD_SPRITE_FILE);
    bird.init(birdTexture, BIRD_FRAMES, BIRD_CLIPS);

    bool quit = false;
    SDL_Event e;
    while( !quit ) {
        while( SDL_PollEvent( &e ) != 0 ) {
            if( e.type == SDL_QUIT ) quit = true;
        }
        man.tick(); bird.tick();

        graphics.prepareScene();
        graphics.render(100, 100, man);
        graphics.render(150, 100, bird);
        graphics.presentScene();
        SDL_Delay(100);
    }

    SDL_DestroyTexture( manTexture ); manTexture = NULL;
    SDL_DestroyTexture( birdTexture ); birdTexture = NULL;
```

—----------

Trong đó, các dữ liệu về hoạt hình man và bird được định nghĩa ở defs.h, chẳng hạn cho man:

—--------defs.h-----

```
const char* MAN_SPRITE_FILE = "img\\foo.png";
const int MAN_CLIPS[][4] = {
    {  0, 0, 64, 205},
    { 64, 0, 64, 205},
```

```
    {128, 0, 64, 205},
    {192, 0, 64, 205}};
const int MAN_FRAMES = sizeof(MAN_CLIPS)/sizeof(int)/4;
…
```

—----------

Để ý là cần xóa màn hình ở mỗi nhịp game để không vẽ đè khung hình sau lên khung hình trước; gọi hàm tick() ở mỗi nhịp game để chuyển tới khung hình tiếp theo. Nếu bạn muốn tăng hoặc giảm tốc độ hoạt hình thì cần điều chỉnh chút ít ở việc gọi tick() - mấy nhịp mới gọi một lần, hoặc ở trong nội dung tick() - tick mấy lần thì mới chuyển frame.

Code: https://github.com/chauttm/gameProject/tree/main/08_sprite

Bạn có thể thêm hiệu ứng theo nhiều kiểu:

- Ghép hoạt hình dạng này vào background trôi ở bài trước và giữ nguyên tọa độ vẽ nhân vật, ta sẽ có hiệu ứng nhân vật chuyển động đối với nền (trong khi thực tế là đứng nguyên tại chỗ đối với cửa sổ game).
- Thay đổi tọa độ vẽ nhân vật theo mỗi lần lặp của gameloop, ta sẽ có nhân vật di chuyển thực trong cửa sổ game. Hiện giờ code đang cố định tọa độ của hai nhân vật.
- Ghép với event chuột hoặc bàn phím, cộng với background trôi, bạn có thể điều khiển nhân vật nhảy lên tại chỗ (thay đổi tọa độ y). Hiệu ứng sẽ trông như là vừa đi vừa nhảy (nhảy khi nhận event). Ví dụ flappy bird thực ra chỉ thay đổi tọa độ y mỗi lần bấm phím, x không đổi, background trôi tạo hiệu ứng chim bay lên bay xuống.

Bài 9. Âm thanh và nhạc nền

(đang viết)

Chuẩn bị code: Bắt đầu từ sản phẩm của Bài 6 - Event bàn phím.

Cài thư viện và cấu hình project

Download SDL2_mixer-devel-2.8.0-mingw.zip từ
https://github.com/libsdl-org/SDL_mixer/releases

Làm tương tự SDL, SDL_image

- Gỡ nén vào một thư mục
- Xóa bỏ thư mục i686
- Copy **SDL2_mixer.dll** từ x64…\bin vào thư mục chứa code
- Thêm **-lSDL2_mixer** vào linker setting
- Thêm đường dẫn tới …**include\SDL2** vào Search directories | compiler
- Thêm đường dẫn tới …**lib** vào search directories|linker

Khởi tạo và dọn dẹp

Tương tự SDL_image, mixer cũng cần được khởi tạo bằng hàm `Mix_OpenAudio` và dọn dẹp bằng hàm Mix_Quit(). Hãy thêm đoạn sau vào cuối hàm init của struct Graphics (nhớ là hàm này luôn được gọi ở đầu chương trình)

——----------

```
//Initialize SDL_mixer
if( Mix_OpenAudio( 44100, MIX_DEFAULT_FORMAT, 2, 2048 ) < 0 ) {
   logErrorAndExit( "SDL_mixer could not initialize! SDL_mixer Error: %s\n",
                    Mix_GetError() );
}
```

——----------

Và lời gọi hàm sau vào đầu hàm quit() của struct Graphics

Mix_Quit();

Tất nhiên, ta còn cần include thư viện mixer

#include <SDL_mixer.h>

Chơi nhạc

——----------

```
    Graphics graphics;
    graphics.init();

    Mix_Music *gMusic = graphics.loadMusic("assets\\RunningAway.mp3");
    graphics.play(gMusic);
```

```
    bool quit = false;
    SDL_Event event;
    while (!quit ) {
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) quit = true;
        }
        const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);
        SDL_Delay(10);
    }
    if (gMusic != nullptr) Mix_FreeMusic( gMusic );

    graphics.quit();
```

—---

Các hàm chơi nhạc cần làm ở Graphics (để còn tiện dùng lại)

—---------struct graphics----------

```
    Mix_Music *loadMusic(const char* path)
    {
        Mix_Music *gMusic = Mix_LoadMUS(path);
        if (gMusic == nullptr) {
            SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION,
                           SDL_LOG_PRIORITY_ERROR,
                "Could not load music! SDL_mixer Error: %s", Mix_GetError());
        }
        return gMusic;
    }
    void play(Mix_Music *gMusic)
    {
        if (gMusic == nullptr) return;

        if (Mix_PlayingMusic() == 0) {
            Mix_PlayMusic( gMusic, -1 );
        }
        else if( Mix_PausedMusic() == 1 ) {
```

```
            Mix_ResumeMusic();
        }
    }
```

——————————

Tiếng động

——————————————

```
    Graphics graphics;
    graphics.init();

    Mix_Chunk *gJump = graphics.loadSound("assets\\jump.wav");

    bool quit = false;
    SDL_Event event;
    while (!quit ) {
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) quit = true;
        }
        const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

        if (currentKeyStates[SDL_SCANCODE_UP]) graphics.play(gJump);

        SDL_Delay(10);
    }
    if (gJump != nullptr) Mix_FreeChunk( gJump);

    graphics.quit();
    return 0;
```

——————————————

——————————————

```
    Mix_Chunk* loadSound(const char* path) {
        Mix_Chunk* gChunk = Mix_LoadWAV(path);
```

```
    if (gChunk == nullptr) {
        SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION,
                       SDL_LOG_PRIORITY_ERROR,
           "Could not load sound! SDL_mixer Error: %s", Mix_GetError());
    }
}
void play(Mix_Chunk* gChunk) {
    if (gChunk != nullptr) {
        Mix_PlayChannel( -1, gChunk, 0 );
    }
}
```

——---

Code: https://github.com/chauttm/gameProject/tree/main/09_sound_and_music

Bài 10. Hiển thị text

Cài thư viện và cấu hình project

Download SDL2_ttf-devel-2.22.0-mingw.zip từ https://github.com/libsdl-org/SDL_ttf/releases

Làm tương tự SDL, SDL_image

- Gỡ nén vào một thư mục
- Xóa bỏ thư mục i686
- Copy **SDL2_ttf.dll** từ x64…\bin vào thư mục chứa code
- Thêm **-lSDL2_ttf** vào linker setting
- Thêm đường dẫn tới …**include\SDL2** vào Search directories | compiler
- Thêm đường dẫn tới …**lib** vào search directories|linker

Chuẩn bị font: lấy font loại true-type (file .ttf) để vào một thư mục trong project của bạn. Ví dụ ….\assets\Purisa-BoldOblique.ttf

Include: `#include <SDL_ttf.h>`

Khởi tạo: Thêm vào cuối hàm init() của Graphics

—----------

```
        if (TTF_Init() == -1) {
            logErrorAndExit("SDL_ttf could not initialize! SDL_ttf Error: ",
                            TTF_GetError());
        }
```

—----------

Dọn dẹp: Thêm vào đầu hàm quit() của Graphics

—----------

```
       TTF_Quit();
```

—----------

Nạp font: nạp font bạn đã có file trong thư mục asset vào chương trình để sau này sẽ sinh các đoạn text vẽ theo font đó. Tham số hàm là đường dẫn tới file chứa font và font size (cỡ font)

—----------

```
    TTF_Font* loadFont(const char* path, int size)
    {
        TTF_Font* gFont = TTF_OpenFont( path, size );
        if (gFont == nullptr) {
            SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION,
                           SDL_LOG_PRIORITY_ERROR,
                           "Load font %s", TTF_GetError());
        }
    }
```

—————————————

Tạo texture chứa dòng text:

—————————————

```
    SDL_Texture* renderText(const char* text,
                            TTF_Font* font, SDL_Color textColor)
    {
        SDL_Surface* textSurface =
                TTF_RenderText_Solid( font, text, textColor );
        if( textSurface == nullptr ) {
            SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION,
                           SDL_LOG_PRIORITY_ERROR,
                           "Render text surface %s", TTF_GetError());
            return nullptr;
        }

        SDL_Texture* texture =
                SDL_CreateTextureFromSurface( renderer, textSurface );
        if( texture == nullptr ) {
            SDL_LogMessage(SDL_LOG_CATEGORY_APPLICATION,
                           SDL_LOG_PRIORITY_ERROR,
                           "Create texture from text %s", SDL_GetError());
        }
        SDL_FreeSurface( textSurface );
        return texture;
    }
```

—————————————

Demo sử dụng tại logic chương trình: bốn bước: nạp font; tạo ảnh chứa text; vẽ ảnh chứa text; sau khi dùng xong thì giải phóng ảnh(texture) và font.

—————————

```
    Graphics graphics;
    graphics.init();

    TTF_Font* font = graphics.loadFont("assets/Purisa-BoldOblique.ttf", 100);
    SDL_Color color = {255, 255, 0, 0};
    SDL_Texture* helloText = graphics.renderText("Hello", font, color);
    graphics.renderTexture(helloText, 200, 200);
```

```
    graphics.presentScene();
    waitUntilKeyPressed();

    SDL_DestroyTexture( helloText );
    TTF_CloseFont( font );
    helloText = NULL;

    graphics.quit();
```

——----------

Code: https://github.com/chauttm/gameProject/tree/main/10_text

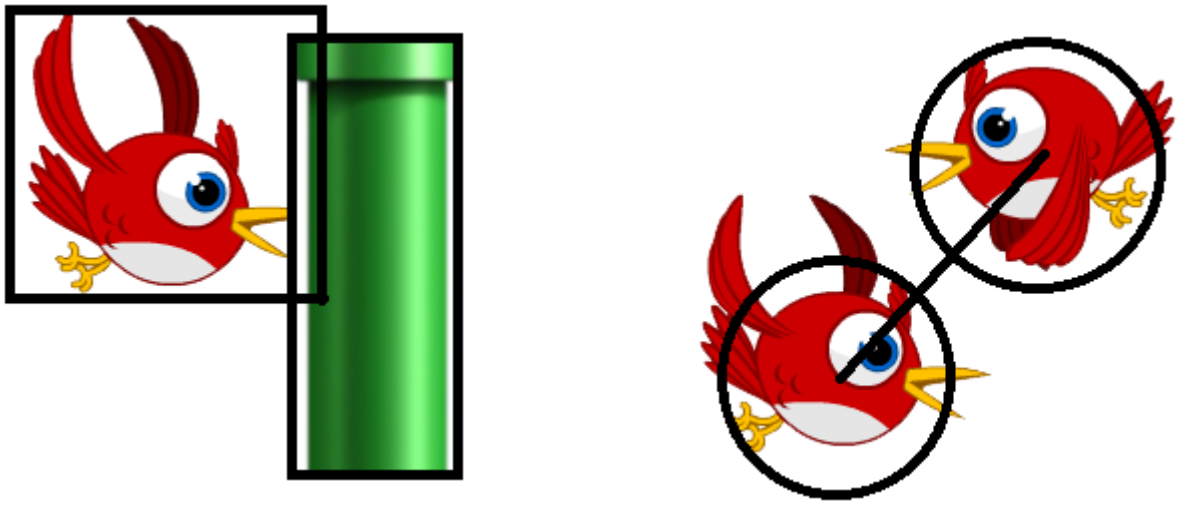
Bài 11. Xử lý va chạm

(đang viết)

Ý tưởng thô để xác định va chạm giữa hai vật trên màn hình rất đơn giản.

Cách dùng khung chữ nhật: ta có thể dùng hai khung chữ nhật chứa mỗi vật và tính tọa độ xem hai hình chữ nhật có vùng chồng lấn hay không.

Nếu vật có dạng tròn, ta có thể xác định hình tròn bao mỗi vật và tính khoảng cách giữa hai tâm hình tròn để xem chúng có đang chồng lên nhau hay không.



Ở hình minh họa bên trái, ta dùng các khung chữ nhật cho con chim và cột. Ở hình bên phải, ta dùng hai hình tròn làm khung cho hai con chim, đoạn thẳng mầu đen là khoảng cách giữa tâm hai hình tròn, nếu nó lớn hơn tổng bán kính của hai hình tròn thì ta có thể kết luận là có va chạm.

Tất nhiên, độ chính xác phụ thuộc vào khung chữ nhật hoặc khung tròn mà bạn dùng để xác định là khung bao của vật. Chẳng hạn nếu bạn muốn xem kiếm thủ có đâm trúng quái hay không thì bạn cần xác định va chạm giữa mũi kiếm và khung của quái thay vì giữa khung chứa toàn bộ kiếm thủ và khung chứa quái.

Ví dụ ứng dụng: Bắt đầu từ code bài 07a hoạt hình vật di chuyển.

Tại bài đó, ta đã có 1 con chuột (Mouse) di chuyển theo điều khiển từ bàn phím và game over khi chuột lao ra khỏi màn hình. Ta sẽ sửa để thêm 1 cục pho-mát (Cheese) và sẽ xử lý va chạm giữa chuột và pho mát để làm hiệu ứng chuột "ăn" pho mát. Ta sẽ dùng chính hình chữ nhật vẽ chuột và hình chữ nhật vẽ pho-mát làm khung va chạm.

Logic main thay đổi chút xíu (các dòng bôi đậm): (1) khai báo thêm một mẩu pho mát và khởi tạo tọa độ của mouse và cheese tại chỗ (hàm constructor sẽ được thêm vào struct Mouse và struct Cheese). (2) Thêm logic kiểm tra nếu chuột ăn được pho mát thì nó sẽ lớn thêm một chút. (3). Thêm việc vẽ mẩu pho-mát.

—------main.cpp ------

Mouse mouse(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2);

Cheese cheese(100, 100);

```
    bool quit = false;
    SDL_Event event;
    while (!quit && !gameOver(mouse)) {
        graphics.prepareScene();

        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) quit = true;
        }

        const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

        if (currentKeyStates[SDL_SCANCODE_UP]) mouse.turnNorth();
        if (currentKeyStates[SDL_SCANCODE_DOWN]) mouse.turnSouth();
        if (currentKeyStates[SDL_SCANCODE_LEFT]) mouse.turnWest();
        if (currentKeyStates[SDL_SCANCODE_RIGHT]) mouse.turnEast();

        mouse.move();
        if (mouse.canEat(cheese)) mouse.grow();

        render(mouse, graphics);
        render(cheese, graphics);

        graphics.presentScene();
        SDL_Delay(10);
    }
```

—----------

Code: https://github.com/chauttm/gameProject/tree/main/11_collision

Như vậy là ta đã có thêm 1 game đơn giản và chơi được. Bạn hoàn toàn có thể dùng game này cho bài tập lớn.

Phụ lục 0. Simplified Flappy Bird

Game đơn giản này có màn giới thiệu, màn chơi, và màn game over.

Viết kiểu đơn giản không dùng quan hệ thừa kế. Bạn nào muốn làm menu thì có thể tham khảo cách tổ chức hàm main.

https://github.com/chauttm/flappyCopy

Todos:

std::vector/list/array….

Khung game loại đơn giản chỉ có 1 player di chuyển lên xuống - Flappy bird

Khung game loại có object tự động di chuyển qua màn hình (đạn/máy bay…) (cần std::list, new/delete) -> lỗi sử dụng bộ nhớ động, cách giảm new/delete….

Code: https://github.com/chauttm/gameProject/tree/main/shooting

Đồng bộ thời gian mỗi vòng lặp của game

Phụ lục 1. GitHub và GitHub Desktop

Mở tài khoản tại https://github.com/ nếu bạn chưa có. Chọn một cái tên nghiêm túc nếu bạn không muốn sau này phải tạo cái khác để bỏ vào CV và dùng lâu dài.

Nếu bạn đã quen dùng github hay bitbucket thì bỏ qua phần còn lại của bài này.

1.1. Git

Nếu bạn chưa bao giờ dùng git thì cần download và cài (https://git-scm.com/downloads). Nói chung không phải làm gì ngoài việc cài theo chế độ mặc định.

1.2 GitHub Desktop

Có rất nhiều chương trình client có thể làm việc được với code đặt tại github và những chỗ tương tự khác. Ví dụ: cửa sổ dòng lệnh cmd sau khi đã cài git (https://git-scm.com/downloads), GitBash, GitGUI, TortoiseGit…một số IDE có sẵn các chức năng client như IntelliJ, Visual Code… Hướng dẫn này dành cho các bạn dùng CodeBlocks và chưa thành thạo git. Nếu bạn đã quen dùng một chương trình client cho git rồi thì có thể bỏ qua phần còn lại.

Cài GitHub Desktop từ https://desktop.github.com/. Nói chung không phải làm gì ngoài việc cài theo chế độ mặc định.
Tài liệu hướng dẫn khá là nhiều nội dung (https://docs.github.com/en/desktop/overview/getting-started-with-github-desktop). Tuy nhiên với ngữ cảnh đơn giản của môn học. Bạn sẽ chỉ dùng đến vài chức năng như giới thiệu dưới đây:

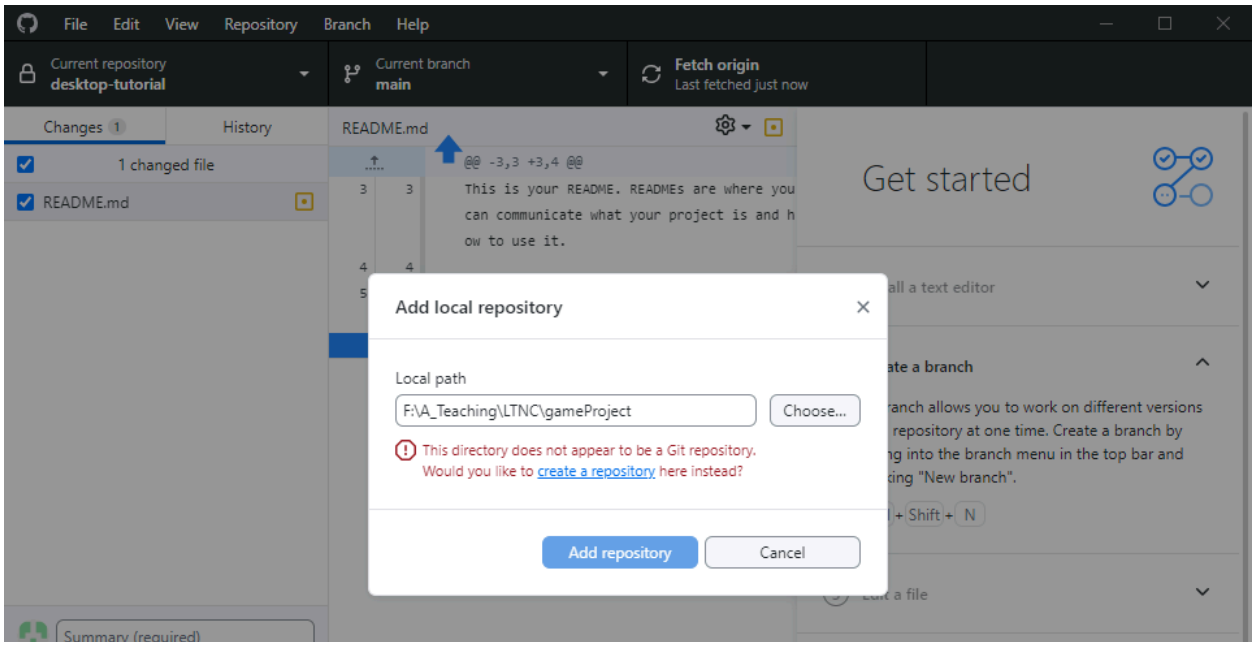
Thuật ngữ:

- git: một giao thức quản lý phiên bản của file, đặc biệt thông dụng cho việc quản lý phiên bản source code. Có một số giao thức tương tự như mercurial, cvs. Git rất tốt, nhưng không nhất thiết là cái tốt nhất, nhưng nó thông dụng nhất và những cái khác vì thế mà chết dần.
- GitHub: một dịch vụ hỗ trợ quản lý phiên bản sử dụng giao thức git (git server). Có vài dịch vụ dùng git, Bitbucket là một ví dụ. Bạn có thể tự cài một server chạy giao thức git để dùng riêng.
- git client: chương trình cài ở máy cá nhân và có thể tương tác với các git server theo giao thức git để đẩy code lên các server đó và lấy code về.
- Repository: một thư mục chứa file quản lý kiểu git. Có thể hiểu như một kho chứa. Bạn có thể tạo các repository của bạn và share cho người khác dùng chung (team work).

1.3 Bắt đầu repository cho bài tập lớn

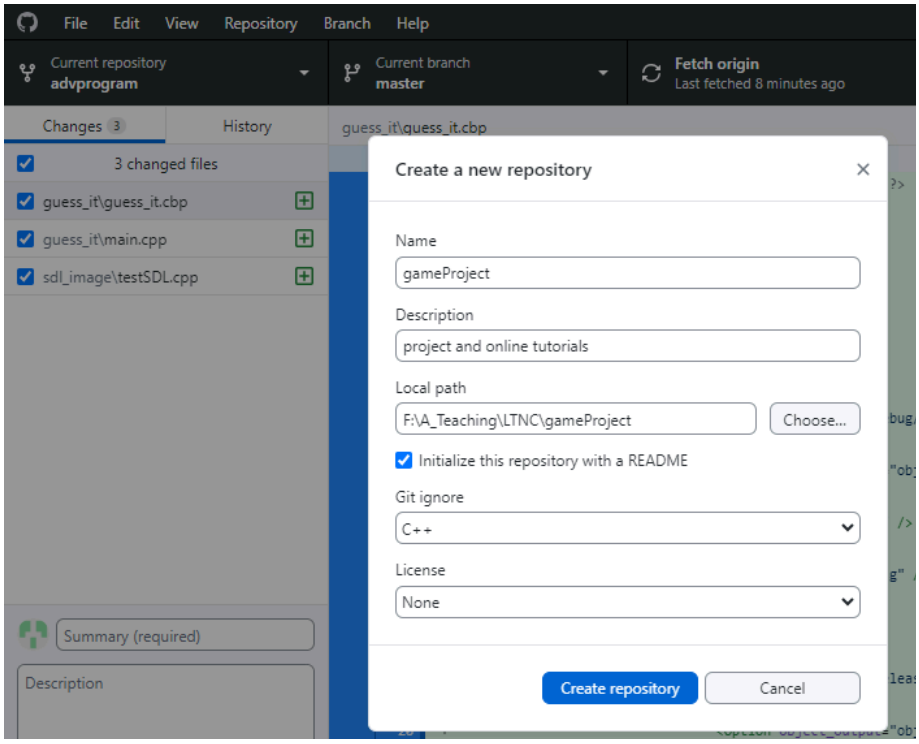
Giả sử ta đã có sẵn thư mục F:\A_Teaching\LTNC\gameProject, trong đó có sẵn một ít code nhưng hiện chỉ là thư mục bình thường. Ta muốn cho thư mục này thành repository dành cho

bài tập lớn. Chọn menu File | Add Local Repository …, chọn lấy thư mục gameProject đó, ta sẽ thấy cửa sổ dưới đây.



Vì là thư mục đã có sẵn và không phải thư mục do git quản lý, nên ta nhìn thấy dòng thông báo mầu đỏ. Click vào link Create a repository, git sẽ khởi tạo việc quản lý repository ở thư mục gameProject đó và biến nó thành một local repository. (Đừng bấm nút Add repository, nó sẽ bỏ qua thư mục ta đã chọn.)

Ta sẽ thấy form điền thông tin về repo muốn tạo.

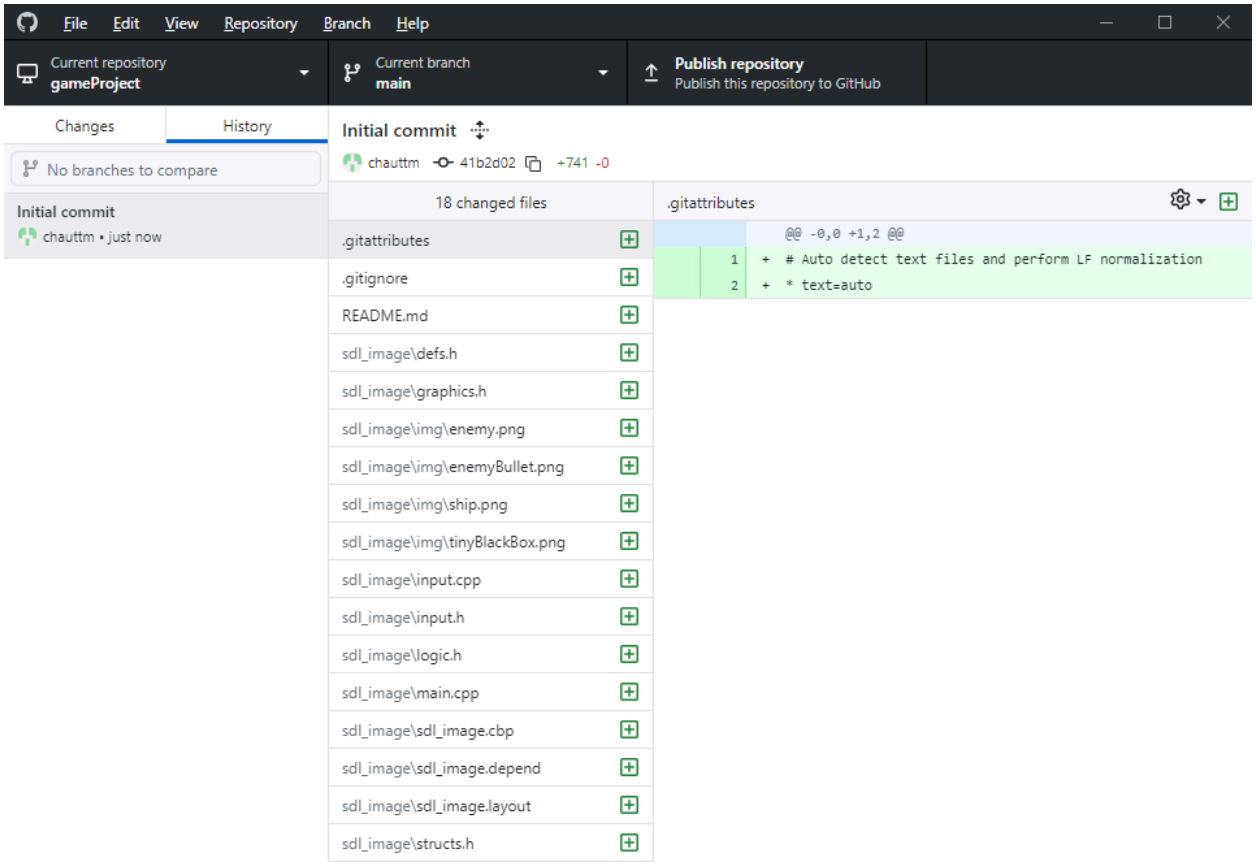


Ta nhập *Name* là tên của repository, *Description* là mô tả về repository (không bắt buộc). *Local path* là đường dẫn tới thư mục đã chọn và không được sửa. Nên tích vào *Initialize …. a README* để có file Readme sau này tiện cho việc mô tả project (mã sinh viên, tên, lớp….chủ đề bài tập lớn.). Việc cuối cùng nhưng rất quan trọng là chọn Git ignore loại C++, để sau này git sẽ tự động bỏ qua tất cả các file mã nhị phân do trình biên dịch tạo ra, không làm repository trở nên quá to do quản lý các file không cần thiết.

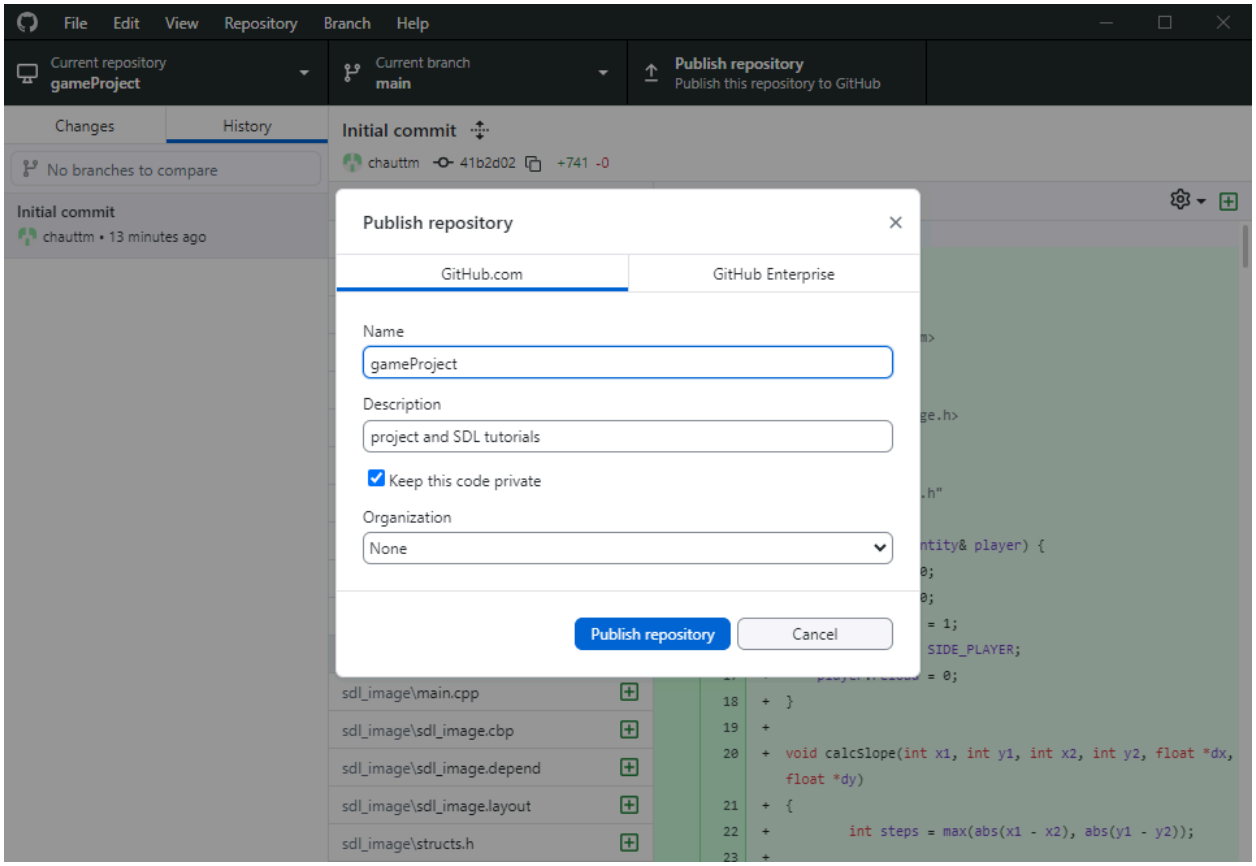
Bấm nút Create repository. Bạn sẽ thấy từ trái qua phải: và từ trên xuống dưới

- Current repository **gameProject** nghĩa là hiện tại đang xem thông tin của repository gameProject, click vào tam giác nhỏ bên phải sẽ có thể chuyển sang repo khác nếu muốn.
- Current branch **main** nghĩa là hiện đang ở nhánh main, nhánh mặc định của repo, click vào tam giác nhỏ bên phải sẽ có thể chuyển sang nhánh khác, nhưng hiện giờ chưa có branch nào ngoài branch main do git tự động tạo (nói chung bạn chắc là không cần tạo branch ở project đơn giản như môn này, branching là cơ chế cho phép phát triển song song nhiều phiên bản của code và sau có thể hợp lại (merge) nếu muốn).
- Publish repository là chức năng giúp đẩy local repository mà bạn vừa tạo lên remote server, ở đây là github.com. Lát nữa sẽ làm việc này.
- Changes: cửa sổ liệt kê các thay đổi ở các file trong repo mà git chưa ghi nhận (commit). Hiện không có gì.
- History: lịch sử các lần ghi nhận (commit) và các lần đẩy code lên remote server (push). Hiện mới có 1 commit **Initial commit** mà github desktop tự tạo để ghi nhận 3 file mà nó vừa tạo ra khi ta tạo repo mới.
    - README.md chứa description mà ta nhập khi tạo repo.
    - .gitignore là file liệt kê đặc điểm (ở đây là phần mở rộng của tên file, .exe chẳng hạn) của các file mà git sẽ bỏ qua. Đây là danh sách dành cho project C++, nếu khi tạo repo ta chọn loại khác thì danh sách này sẽ khác.
    - .gitattributes không quan trọng, ko cần để ý.

Và rất nhiều file mã nguồn, file ảnh, và file quản lý project của CodeBlocks trong project sdl mà tôi đã có sẵn trong thư mục con sdl_image. Git đã quét toàn bộ các file trong thư mục gameProject và ghi nhận tất cả những file không thuộc diện cần bỏ qua như liệt kê trong .gitignore. Ta có thể bấm vào từng file trong danh sách để xem nội dung.

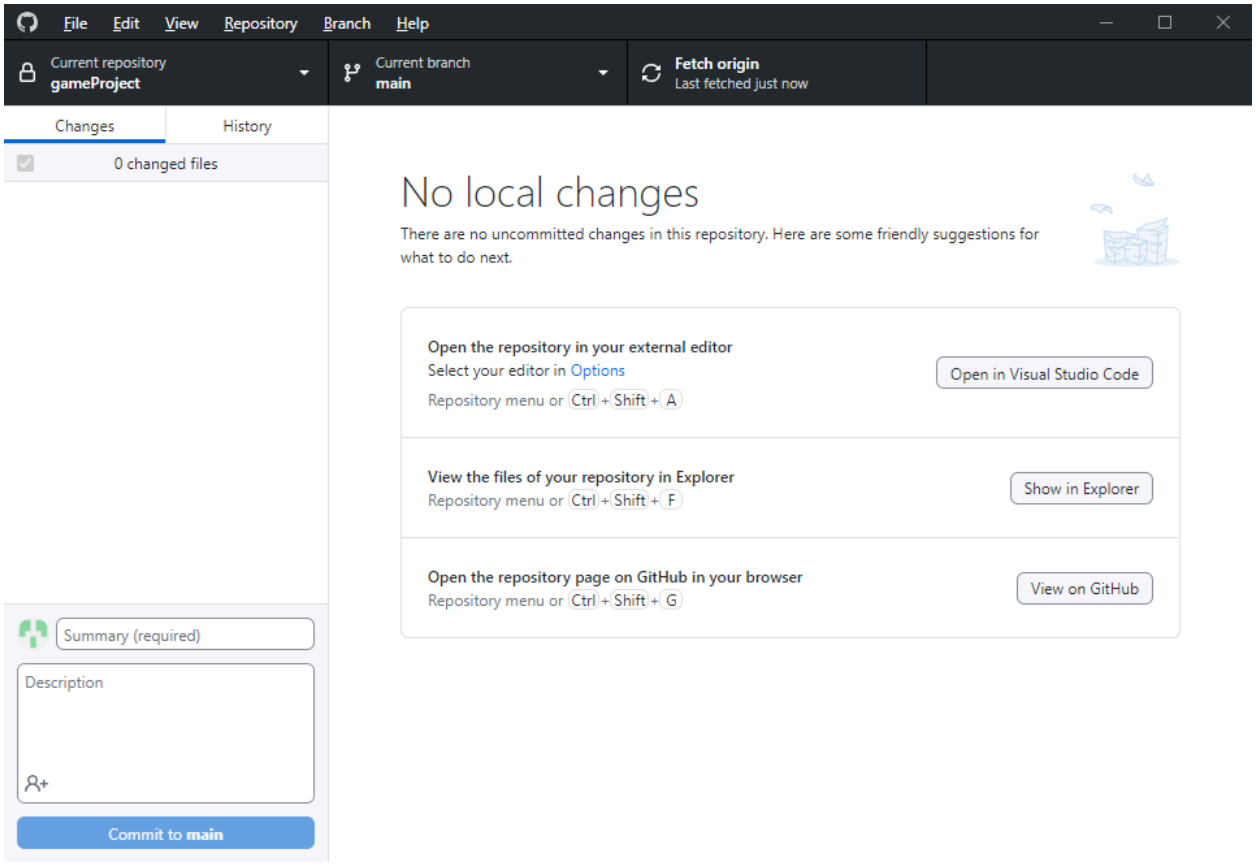


Nếu hài lòng với nội dung của commit đầu tiên, ta đẩy code lên github bằng cách click vào Publist repository

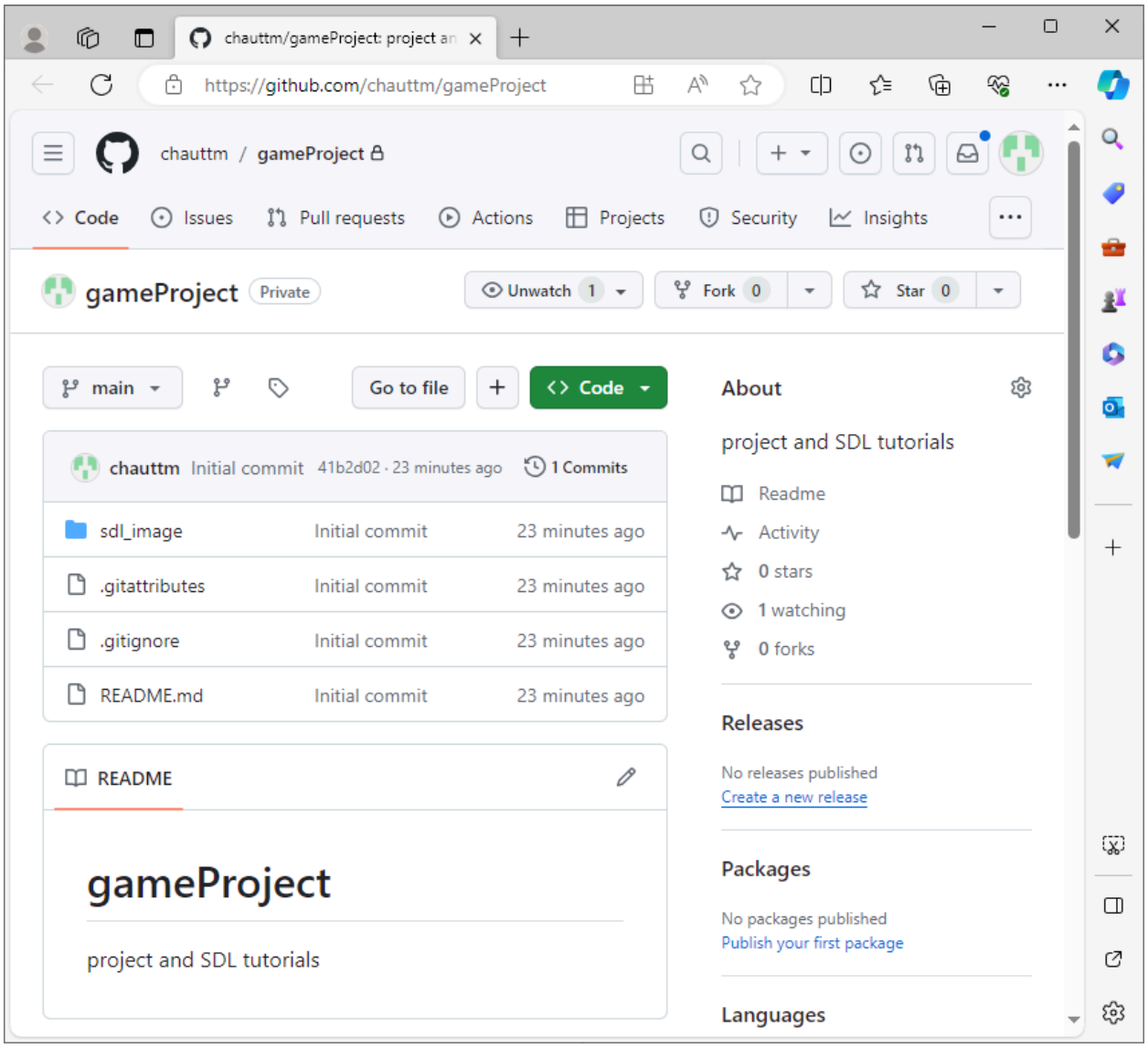


Click vào nút Publish, Github Desktop sẽ đẩy code lên remote repository. Và vì đây là lần đầu push, và bạn chưa hề tạo remote repository ở github.com, phần mềm này làm cả việc đó giúp bạn.

Bạn sẽ thấy kết quả như dưới đây. Ở cột thứ ba, thay vì Push repository như lúc trước, ta có Fetch Origin, chức năng này sẽ lấy phiên bản code mới nhất từ github nếu có. Không còn Push repository vì hiện tại tại local repo, ta không có commit nào mà chưa được đẩy lên remote repo.



Nếu bạn tò mò về remote repo vừa được tạo ra tại github, click vào nút View on GitHub ở góc phải dưới.

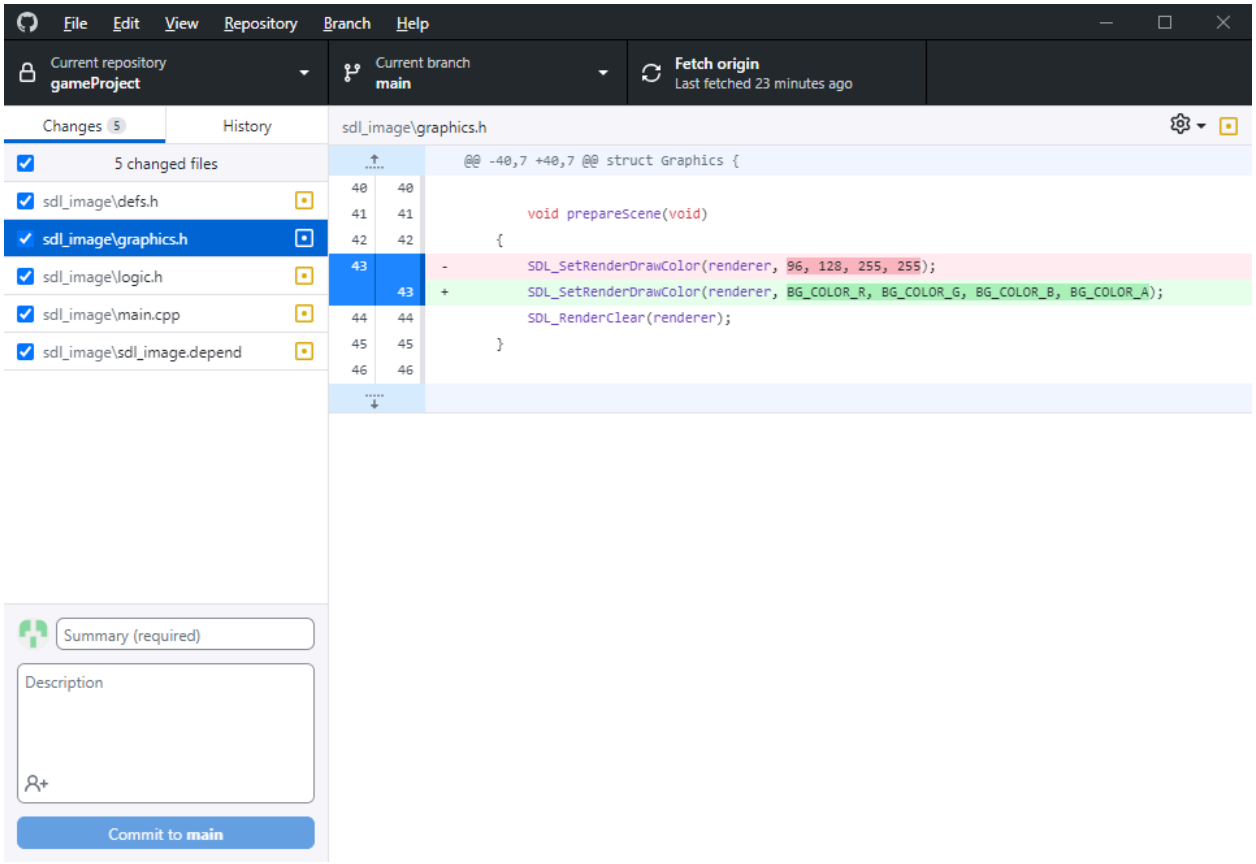


Bạn có thể thấy commit *Initial commit* với mã hiệu 41b2d02
Ta đã khởi tạo xong một repository.

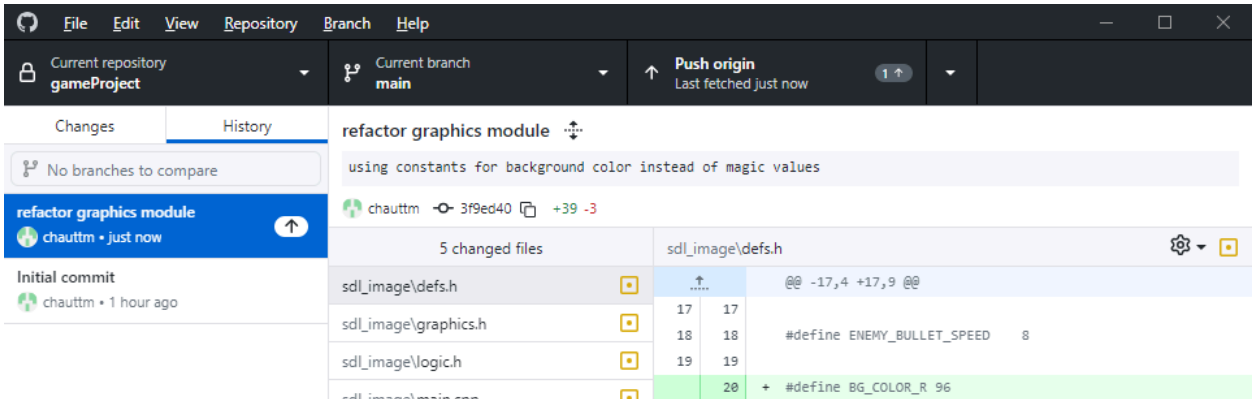
1.4. Code → commit → push

Tôi mở CodeBlocks sửa một ít code trong project, chạy thử thấy ổn và quay lại cửa sổ GitHub Desktop. Cột Changes (lúc đầu không có gì) liệt kê 05 file có thay đổi. Click vào mỗi file có thể thấy thay đổi ở đâu, phần mầu hồng là code cũ, phần mầu xanh là code mới, hồng đậm là code bị xóa, xanh đậm là code viết mới.

Cả 5 file đã được tick chọn để chuẩn bị cho commit mới (chưa tạo). Nếu bạn không muốn đưa file nào vào lần commit này (nghĩa là file vẫn ở tình trạng đã bị sửa, nhưng chưa muốn ghi nhận ngay lần này, thì bạn có thể bỏ tick file đó. Tất nhiên, nếu bạn chưa muốn commit gì bây giờ thì chỉ việc không làm gì ở Github Desktop.



Tôi muốn commit bây giờ. Tôi nhập “refactor graphics module” vào Summary, một ít text dài hơn (nhưng không bắt buộc) vào Description, rồi click nút Commit to main (ghi nhận vào branch *main*). Kết quả là chức năng Push origin hiện thay cho Fetch origin (giờ ta có commit mới để push), cột history có thêm commit Refactor…với mã hiệu 3f9e40.

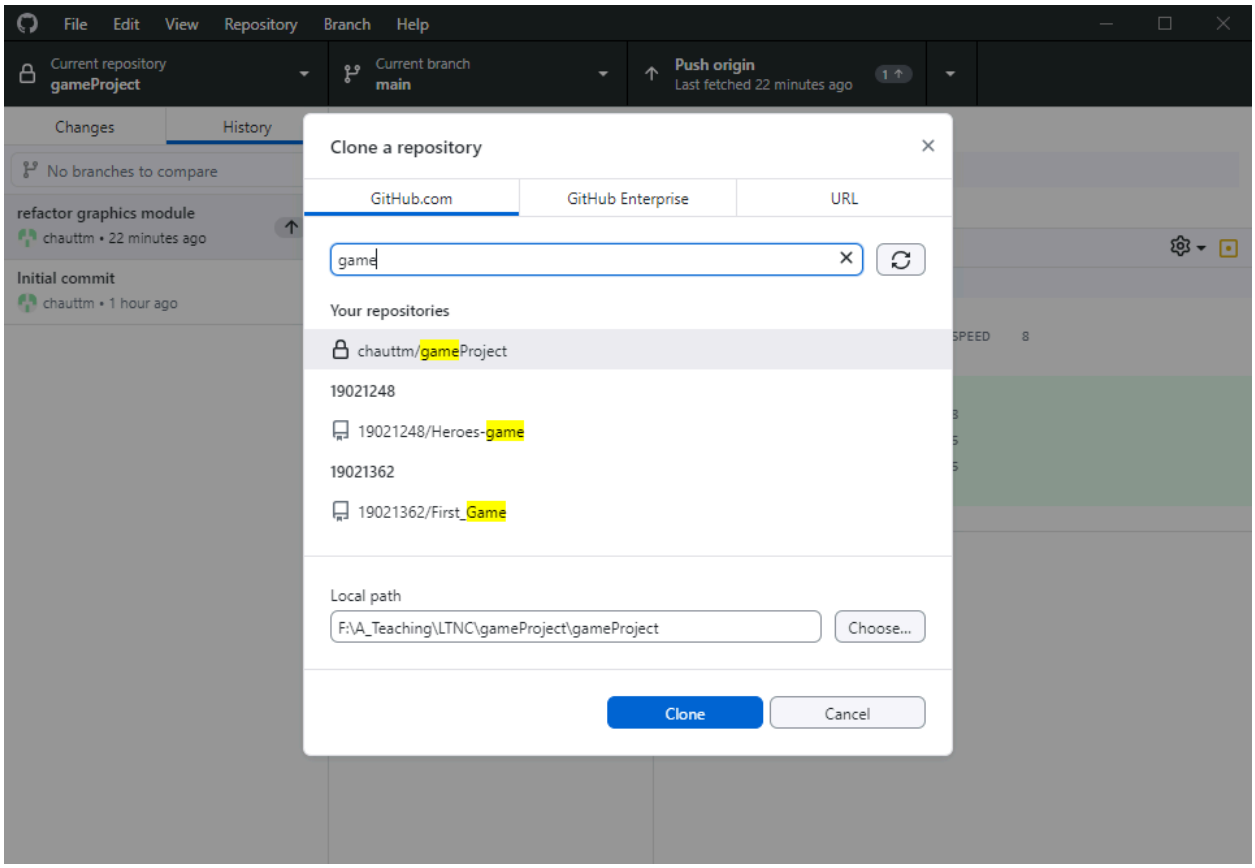


Click chọn Push origin. Tôi đã hoàn thành một chu kỳ code → commit → push.
Nếu không có internet nên push không thành công hoặc vì lý do nào đó chưa muốn push, thì lịch sử các commit vẫn nằm tại local repo. Và tất cả các commit đã tạo nhưng chưa push sẽ được đẩy lên remote server ở lần push tiếp theo với thông tin y hệt tại local.

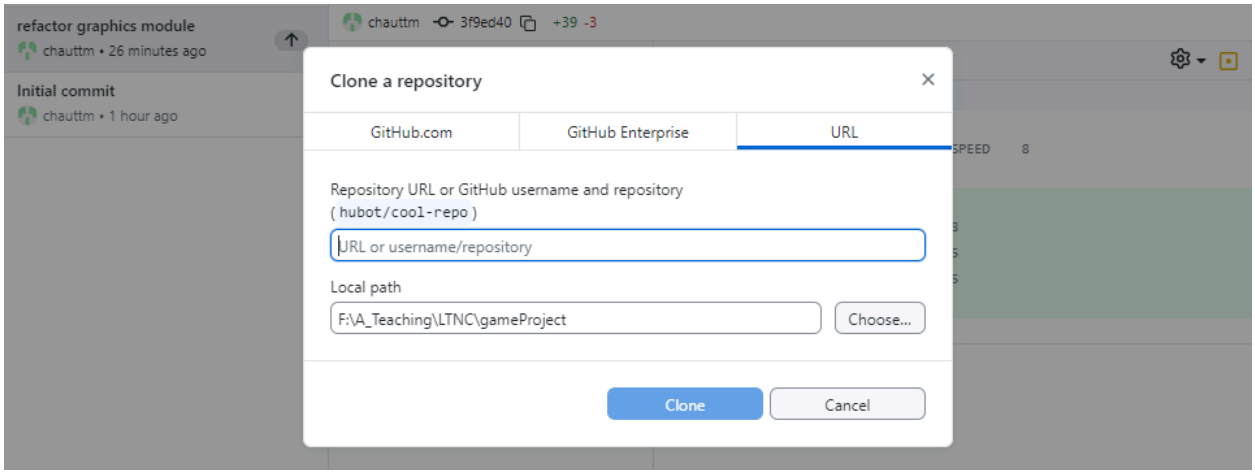
Từ nay, nếu không có gì thay đổi thì tôi có thể tiếp tục các chu kỳ code → commit → push cho đến khi nào tôi không muốn làm gì với project này nữa.

Các tình huống thay đổi:

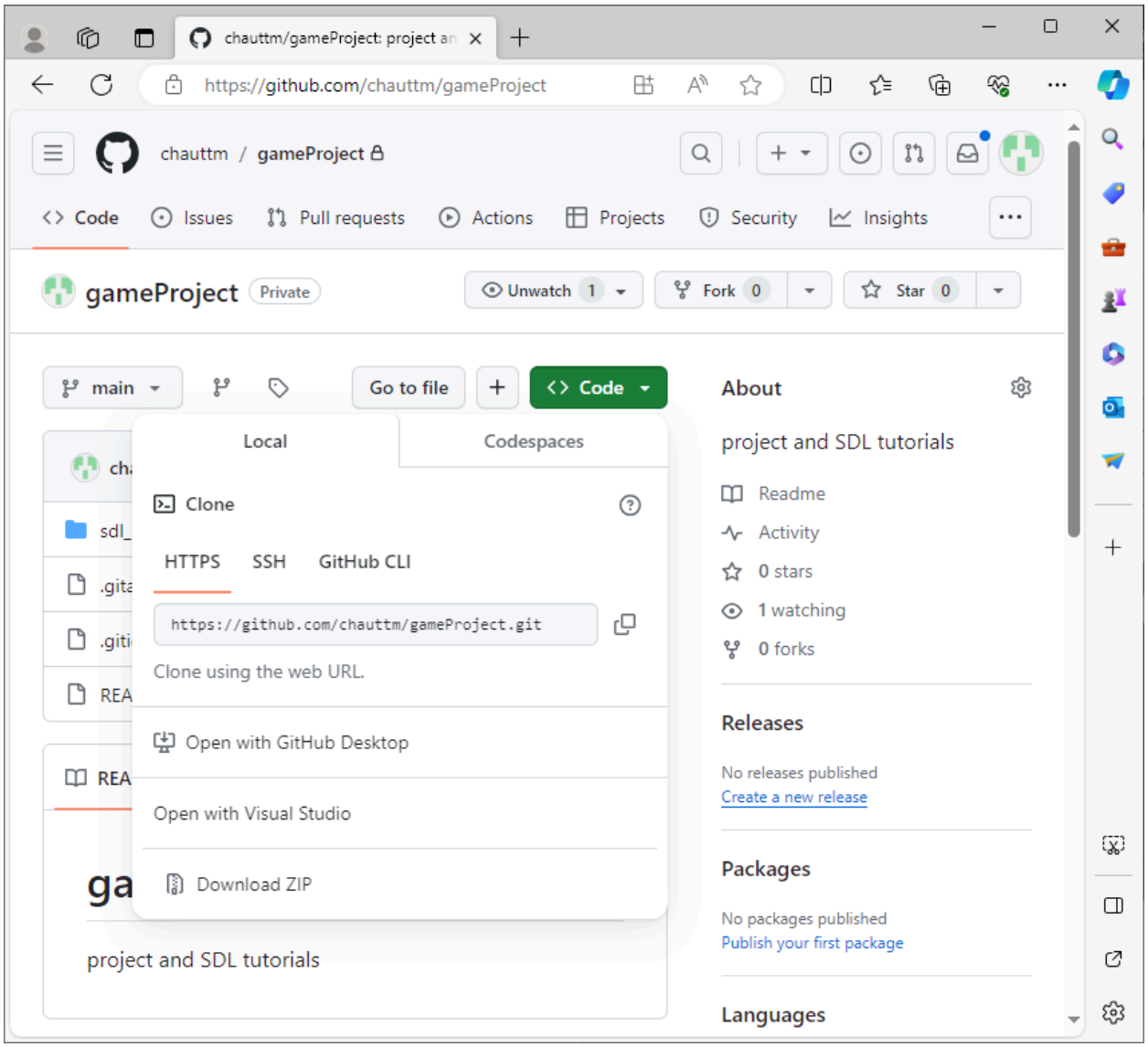
1. *Bạn muốn chuyển thư mục repo đi chỗ khác trên máy*:
   Bê nguyên xi thư mục gameProject đó đến chỗ bạn muốn. Dữ liệu của git nằm trong thư mục đó, cụ thể là trong thư mục con .git, nên nó vẫn là local repository bình thường. Có điều Github Desktop không tự động biết vị trí mới của gameProject. Khi mở GitHub Desktop lần tiếp theo, nó sẽ không tìm thấy, bạn sẽ cần File | Add Local Repository lần nữa. Nhưng lần này nó sẽ nhận ra thông tin có sẵn trong .git và không đòi "Create repository" lần nữa. Tình huống tương tự xảy ra khi bạn có sẵn một thư mục repo được tạo bằng client khác chứ không phải GitHub Desktop, bạn sẽ Add Local… để mở repo đó từ Github Desktop.
2. *Bạn có hai máy tính, bạn đã tạo local repo từ máy thứ nhất, và giờ muốn code chính project đó bằng máy thứ hai.*
   File | Clone a repository
   Bạn có thể filter theo tên để chọn lấy project mình cần từ danh sách (tab GitHub.com)



   Hoặc chọn tab URL và copy paste URL của remote repository



Có thể lấy URL của một project theo cách dưới đây (click vào nút [<> Code]):



(Nếu bạn đang mở sẵn github.com thì **Open with GitHub Desktop** là cách clone nhanh nhất)

Lưu ý: khi bạn có nhiều hơn 1 bản local của cùng một project thì sau khi bạn push commit mới ở bản local này thì khi mở bản local kia tại GitHub, bạn sẽ được nhắc nhở Fetch origin để lấy code mới từ remove về, tránh tình huống làm việc trên code cũ dẫn đến conflict code.

Kết
Đây là công cụ dễ dùng hơn rất nhiều so với các Git client khác, nó làm hộ bạn nhiều công đoạn mà lúc đầu có thể là phức tạp dễ lỗi. Giúp bạn đỡ vất vả hơn ở môn này. Tuy nhiên nó cũng che đi nhiều thứ ở mà có thể sau này bạn vẫn phải học (nhưng đó là chuyện ở môn học khác và khi bạn code project trong nhóm)