**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI**
**PHÂN HIỆU TẠI TP. HỒ CHÍ MINH**
**BỘ MÔN CÔNG NGHỆ THÔNG TIN**


# BÁO CÁO BÀI TẬP LỚN

# ĐỀ TÀI: LẬP TRÌNH GAME SPACE INVADERS

Giảng viên hướng dẫn: ThS. TRẦN THỊ DUNG

Sinh viên thực hiện: NGUYỄN BÌNH MINH

Lớp: CQ.62.CNTT

Khoá: K62

Tp. Hồ Chí Minh, năm 2022

**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI**
**PHÂN HIỆU TẠI TP. HỒ CHÍ MINH**
**BỘ MÔN CÔNG NGHỆ THÔNG TIN**

# BÁO CÁO BÀI TẬP LỚN

## ĐỀ TÀI: LẬP TRÌNH GAME SPACE INVADERS

Giảng viên hướng dẫn: ThS. TRẦN THỊ DUNG

Sinh viên thực hiện: NGUYỄN BÌNH MINH

LÊ HỒNG PHÚC

NGUYỄN TẤN PHÁT

BÙI TUẤN LINH

Lớp: CQ.62.CNTT

Khoá: K62

Tp. Hồ Chí Minh, năm 2022

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI
**PHÂN HIỆU TẠI THÀNH PHỐ HỒ CHÍ MINH**

**CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM**
Độc lập – Tự do – Hạnh phúc

**NHIỆM VỤ THIẾT KẾ BÀI TẬP LỚN**
BỘ MÔN: **CÔNG NGHỆ THÔNG TIN**
-------***-------

**Mã sinh viên:** **Họ tên SV:**

**Khóa:** 62 **Lớp:** CQ.62.CNTT

## 1. Tên đề tài:

LẬP TRÌNH GAME SPACE INVADERS BẰNG NGÔN NGỮ C++

## 2. Mục đích, yêu cầu

- Mục đích:
  - Tạo ra được một game hoàn chỉnh
  - Cải thiện kĩ năng, tư duy lập trình
  - Tiếp thu thêm nhiều kiến thức từ một dự án thực tiễn
- Yêu cầu:
  - Sử dụng ngôn ngữ C++

## 3. Nội dung và phạm vi đề tài:

- Nội dung đề tài:
  - Lập trình hướng đối tượng game Space Invaders
- Phạm vi đề tài:
  - Dùng lập trình hướng đối tượng (ngôn ngữ C++)
  - Sử dụng thư viện đồ hoạ, cụ thể SFML

## 4. Công nghệ, công cụ và ngôn ngữ lập trình:

- Công cụ lập trình: Dev-C++
- Ngôn ngữ lập trình: C++

## 5. Các kết quả chính dự kiến sẽ đạt được và ứng dụng:

- Xây dựng thành công tựa game đơn giản bằng ngôn ngữ C++

## 6. Giáo viên và cán bộ hướng dẫn

Họ tên: Trần Thị Dung

Đơn vị công tác: Bộ môn Công Nghệ Thông Tin – Trường Đại học Giao Thông Vận tải phân hiệu tại Thành Phố Hồ Chí Minh

Điện thoại: 0388389579 Email: ttdung@st.utc2.edu.vn

| | |
|---|---|
| **Ngày tháng năm 2022** | **Đã giao nhiệm vụ TKTN** |
| **Trưởng BM Công nghệ Thông tin** | **Giáo viên hướng dẫn** |
| **ThS. Trần Phong Nhã** | **ThS. Trần Thị Dung** |

Đã nhận nhiệm vụ TKTN

Sinh viên: Ký tên:

Điện thoại: Email:

# LỜI CẢM ƠN

Đầu tiên, nhóm em xin bày tỏ lời cảm ơn đến nhà trường đã tạo điều kiện tốt nhất cho chúng em, để chúng em có một môi trường học tập thật tốt.

Kế tiếp, nhóm em xin cảm ơn đến Giảng viên Bộ môn Công Nghệ Thông Tin – cô Trần Thị Dung đã tạo điều kiện cho nhóm chúng em có cơ hội được làm việc chung với nhau và hoàn thành dự án này.

Cảm ơn đến các bạn thuộc lớp Công nghệ Thông Tin K62 đã luôn giúp đỡ nhóm mình trong thời gian học tập cùng nhau.

Với tư cách là nhóm trưởng, tôi – Nguyễn Bình Minh xin cảm ơn đến Lê Hồng Phúc, Nguyễn Tấn Phát, Bùi Tuấn Linh là các bạn trong nhóm đã cùng tôi hoàn thành dự án này.

Chúc sức khỏe đến tất cả các thành viên trong gia đình, toàn thể giảng viên của trường Đại học Giao thông Vận tải phân hiệu tại Thành phố Hồ Chí Minh đặc biệt là các giảng viên thuộc Bộ Môn Công nghệ Thông tin, các bạn thuộc lớp Công nghệ Thông tin K62, các bạn bè khác của tôi.

Xin cảm ơn!

# NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

*Tp. Hồ Chí Minh, ngày …… tháng …… năm ……*

**Giáo viên hướng dẫn**

**Trần Thị Dung**

# MỤC LỤC

# DANH MỤC HÌNH ẢNH

# PHẦN 1: DỊCH SÁCH

## Class

### 1. Khái niệm cơ bản về lớp

Tóm tắt về lớp đối tượng:

- Lớp là kiểu người dùng tự định nghĩa.
- Lớp bao gồm các thành phần mà phổ biến nhất là thành phần dữ liệu và các phương thức.
- Các hàm thành phần có thể định nghĩa cho việc khởi tạo, sao chép, di chuyển và huỷ.
- Các thành phần được truy cập bằng . (dấu chấm) cho các đối tượng và -> (mũi tên) cho con trỏ.
- Các toán tử như +, !, và [], có thể định nghĩa cho lớp.
- Các thành phần **công khai** cung cấp lớp truu và thành phần **riêng tư** cung cấp …
- **Cấu trúc** là một **lớp** mà các thành phần mặc định là **công khai**.

Ví dụ:

```
class X {
private:                    // the representation (implementation) is private
      int m;
public:                     // the user interface is public
      X(int i =0) :m{i} { } // a constructor (initialize the data member m)

      int mf(int i)         // a member function
      {
            int old = m;
            m = i;          // set a new value
            return old;     // return the old value
      }
};

X var {7}; // a variable of type X, initialized to 7

int user(X var, X* ptr)
{
      int x = var.mf(7);          // access using . (dot)
      int y = ptr->mf(9);         // access using -> (arrow)
      int z = var.m;              // error: cannot access private member
}
```

### 1.1 Các hàm thành phần

```
struct Date {
        int d, m, y;

        void init(int dd, int mm, int yy);      // initialize
        void add_year(int n);                   // add n years
        void add_month(int n);                  // add n months
        void add_day(int n);                    // add n days
};
```

Các hàm được khai báo trong lớp được gọi là *hàm thành phần* và chỉ được gọi cho một biến cụ thể của kiểu thích hợp bằng cách sử dụng chuẩn cú pháp để truy cập. Ví dụ:

```
Date my_birthday;

void f()
{
        Date today;

        today.init(16,10,1996);
        my_birthday.init(30,12,1950);

        Date tomorrow = today;
        tomorrow.add_day(1);
        // ...
}
```

Vì các cấu trúc khác nhau có thể có các hàm thành phần với tên giống nhau, chúng ta cần phải chỉ rõ tên lớp khi định nghĩa một hàm thành phần:

```
void Date::init(int dd, int mm, int yy)
{
        d = dd;
        m = mm;
        y = yy;
}
```

Trong hàm, tên các thành phần có thể được sử dụng mà không cần tham chiếu rõ đến một đối tượng.

### 1.2 Sao chép mặc định

Các đối tượng có thể sao chép. Mỗi lớp đối tượng có thể được khởi tạo với bằng bản sao của một đối tượng thuộc lớp đó. Ví dụ:

```
Date d1 = my_birthday;   // initialization by copy
Date d2 {my_birthday};   // initialization by copy
```

Tương tự, các lớp đối tượng mặc định có thể sao chép bởi phép gán. Ví dụ:

```
void f(Date& d)
{
      d = my_birthday;
}
```

### 1.3 Kiểm soát truy cập

```
class Date {
      int d, m, y;
public:
      void init(int dd, int mm, int yy);      // initialize

      void add_year(int n);                   // add n years
      void add_month(int n);                  // add n months
      void add_day(int n);                    // add n days
};
```

Bên trong một lớp có hai phần. Phần thứ nhất là *riêng tư*, phần này chỉ được sử dụng bởi các hàm thành phần. Thứ hai là *công khai*, trừu tượng hoá công khai cho các đối tượng của lớp. Cấu trúc đơn giản là một lớp mà các thành viên mặc định là công khai; các hàm thành phần được định nghĩa và sử dụng chính xác như trước đó. Ví dụ:

```
void Date::add_year(int n)
{
      y += n;
}
```

Tuy nhiên, các hàm không phải thành phần bị cấm sử dụng các thành phần riêng tư. Ví dụ:

```
void timewarp(Date& d)
{
      d.y -= 200;          // error: Date::y is private
}
```

Một số lợi ích có được từ việc hạn chế quyền truy cập tới cấu trúc dữ liệu của các hàm được khai báo. Ví dụ, bất kỳ lỗi khiến cho lớp Date nhận một giá trị không phù hợp (ví dụ: ngày 36 tháng 12 năm 2016) phải do các hàm thành phần gây ra.

Việc bảo vệ dữ liệu riêng tư nhờ vào từ việc hạn chế sử dụng sử dụng tên các thành phần của lớp.

### 1.4 class và struct

**class X{…};** là định nghĩa một lớp được gọi là X. Điều đó còn được gọi là khởi tạo lớp.

Một **struct** là một lớp mà trong đó các thành viên mặc định là công khai; rằng

```
struct S { /* ... */ };
```

đơn giản là viết tắt của

```
class S { public: /* ... */ };
```

Hai cách định nghĩa lớp **S** có thể thay cho nhau. Thường nên gắn bó với một phong cách tùy hoàn cảnh và sở thích

Các thành viên của **class** mặc định là riêng tư:

```
class Date1 {
        int d, m, y;            // private by default
public:
        Date1(int dd, int mm, int yy);
        void add_year(int n);       // add n years
};
```

Tuy nhiên, ta có thể dùng cụm chỉ định **private** để nói rằng các thành sau đó là riêng tư, và tương tự cho cụm chỉ định **public**:

```
struct Date2 {
private:
        int d, m, y;
public:
        Date2(int dd, int mm, int yy);
        void add_year(int n);       // add n years
};
```

Các cụm chỉ định có thể được sử dụng nhiều lần trong một khai báo lớp duy nhất. Ví dụ:

```
class Date4 {
public:
        Date4(int dd, int mm, int yy);
private:
        int d, m, y;
public:
        void add_year(int n);       // add n years
};
```

### 1.5 Các hàm khởi tạo

Sử dụng các hàm như là **init()** để khởi tạo cho các đối tượng của lớp là không phù hợp và dễ bị lỗi. Vì không có một đối tượng nào được chỉ rõ là phải khởi tạo. Cách tốt hơn là khai báo một hàm với mục đích khởi tạo rõ ràng các đối tượng. Vì một hàm xây dựng những giá trị cho một loại nhất định được gọi là *hàm khởi tạo*. Một phương thức khởi tạo được nhận dạng bằng cái tên giống với lớp của nó. Ví dụ:

```
class Date {
        int d, m, y;
public:
        Date(int dd, int mm, int yy);          // constructor
        // ...
};
```

Khi một lớp có hàm khởi tạo, tất cả đối tượng của lớp đó sẽ được khởi tạo bởi một hàm khởi tạo gọi ra. Phải cung cấp những đối số nếu hàm khởi tạo yêu cầu:

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);          // abbreviated form
Date my_birthday;               // error: initializer missing
Date release1_0(10,12);         // error: third argument missing
```

Chúng ta có nhiều cách khác nhau để khởi tạo. Chẳng hạn:

```
class Date {
        int d, m, y;
public:
        // ...

        Date(int, int, int);    // day, month, year
        Date(int, int);         // day, month, today's year
        Date(int);              // day, today's month and year
        Date();                 // default Date: today
        Date(const char*);      // date in string representation
};
```

Có nhiều hàm khởi tạo trong ví dụ lớp **Date**. Khi thiết kế một lớp, lập trình viên luôn dễ muốn thêm các tính năng không cần thiết. Cần phải nghĩ kỹ hơn để quyết định cẩn thận những tính năng thật sự chỉ cần thiết và nên có. Có một cách là sử dụng đối số mặc định. Đối với lớp **Date**, mỗi đối số được truyền một giá trị mặc định được hiểu là "chọn giá trị mặc định: **hôm nay**."

```
class Date {
        int d, m, y;
public:
        Date(int dd =0, int mm =0, int yy =0);
        // ...
};

Date::Date(int dd, int mm, int yy)
{
        d = dd ? dd : today.d;
        m = mm ? mm : today.m;
        y = yy ? yy : today.y;

        // check that the Date is valid
}
```

Giá trị đối số được sử dụng để biểu thị "chọn giá trị mặc định", thì giá trị ấy phải nằm ngoài tập hợp các giá trị có thể cho đối số ấy. Rõ ràng như là ví dụ trên truyền giá trị 0 cho **ngày** và **tháng**, **năm**.

Bằng cách đảm bảo khởi tạo các đối tượng một cách thích hợp, các hàm khởi tạo sẽ đơn giản hoá rất nhiều việc thực hiện các hàm thành phần. Có hàm khởi tạo, các hàm thành phần không phải đối phó với khả năng dữ liệu chưa được khởi tạo.

### 1.6 Các hàm khởi tạo tường minh

Một hàm khởi tạo được gọi bởi một đối số hoạt động như một ép kiểu ngầm định từ kiểu đối số sang kiểu của nó. Ví dụ:

```
complex<double> d {1};        // d=={1,0} (§5.6.2)
```

Những ép kiểu ngầm như vậy có thể cực kỳ hữu ích. Tuy nhiên, trong nhiều trường hợp, những ép kiểu như vậy là nguyên nhân gây ra đáng kể các nhầm lẫn và lỗi.

May mắn, chúng ta có thể chỉ định rằng một hàm khởi tạo không được sử dụng như một ép kiểu ngầm ngầm định. Một hàm khởi tạo được khai báo với từ khoá rõ ràng chỉ có thể được sử dụng để khởi tạo và ép kiểu tường minh.Ví dụ:

```
class Date {
        int d, m, y;
public:
        explicit Date(int dd =0, int mm =0, int yy =0);
        // ...
};

Date d1 {15};           // OK: considered explicit
Date d2 = Date{15};     // OK: explicit
Date d3 = {15};         // error: = initialization does not do implicit conversions
Date d4 = 15;           // error: = initialization does not do implicit conversions

void f()
{
        my_fct(15);         // error: argument passing does not do implicit conversions
        my_fct({15});       // error: argument passing does not do implicit conversions
        my_fct(Date{15});   // OK: explicit
        // ...
}
```

Một khởi tạo với dấu = được coi là *khởi tạo sao chép*. Nguyên tắc là một bản sao của trình khởi tạo được đặt vào đối tượng đã khởi tạo. Bỏ ra ngoài dấu = được gọi là *khởi tạo trực tiếp*.

Nếu một hàm khởi tạo được khai báo **tường minh** và được định nghĩa bên ngoài lớp, thì không thể lặp lại.

Sự khác biệt giữa khởi tạo trực tiếp và khởi tạo sao chép là được duy trì cho quá trình khởi tạo danh sách.

### 1.7 Các bộ khởi tạo trong lớp

Khi chúng ta sử dụng vài hàm khởi tạo. việc khởi tạo các thành phần có thể lặp lại. Ví dụ:

```
class Date {
        int d, m, y;
public:
        Date(int, int, int);        // day, month, year
        Date(int, int);             // day, month, today's year
        Date(int);                  // day, today's month and year
        Date();                     // default Date: today
        Date(const char*);          // date in string representation
        // ...
};
```

Có thể giải quyết bằng cách mở đầu các đối số mặc định để giảm số lượng hàm tạo. Ngoài ra có thể thêm trình khởi tạo vào các dữ liệu thành viên:

```
class Date {
        int d {today.d};
        int m {today.m};
        int y {today.y};
public:
        Date(int, int, int);        // day, month, year
        Date(int, int);             // day, month, today's year
        Date(int);                  // day, today's month and year
        Date();                     // default Date: today
        Date(const char*);          // date in string representation
        // ...
```

### 1.8 Định nghĩa hàm trong lớp

Một hàm thành phần được định nghĩa trong định nghĩa lớp – thay vì chỉ khai báo – đó là định nghĩa hàm trong lớp để các chức năng nhỏ được sử dụng thường xuyên.

Một thành viên có thể tham chiếu đến thành viên khác trong lớp của mình cách độc lập. Xem xét:

```
class Date {
public:
        void add_month(int n) { m+=n; }     // increment the Date's m
        // ...
private:
        int d, m, y;
};
```

Nghĩa là, các khai báo hàm và dữ liệu thành phần là độc lập. Tương ứng với:

```
class Date {
public:
        void add_month(int n) { m+=n; }     // increment the Date's m
        // ...
private:
        int d, m, y;
};

inline void Date::add_month(int n)  // add n months
{
        m+=n;     // increment the Date's m
}
```

Kiểu này thường được dùng để giữ cho các định nghĩa lớp đơn giản và dễ đọc. Nó cho khả năng tách biệt rõ về giao diện và triển khai của một lớp.

### 1.9 Khả năng thay đổi

Chúng ta có thể định nghĩa một đối tượng được đặt tên như một hằng số hoặc một biến. Nói cách khác cái tên có thể đề cập đến đối tượng chứa một giá trị không đổi

hay có thể thay đổi. Việc sử dụng có hệ thống các đối tượng không thay đổi dẫn đến mã dễ hiểu, nhiều lỗi được phát hiện sớm hơn và cải thiện hiệu suất. Đặc biệt, tính bất biến rất hữu ích trong đa luồng.

Để trở nên hữu ích ngoài định nghĩa về hằng số đơn giản của các kiểu có sẵn, chúng ta phải có khả năng xác định các hàm hoạt động trên các đối tượng **hằng** của kiểu do người dùng tự định nghĩa.

### 1.9.1 Các hằng hàm thành viên

Lớp **Date** cung cấp các hàm thành phần để cho một giá trị của **Date**. Nhưng lại không cho cách kiểm tra giá trị của **Date**. Vấn đề này khắc phục bằng thêm các hàm để đọc ngày, tháng và năm:

```
class Date {
        int d, m, y;
public:
        int day() const { return d; }
        int month() const { return m; }
        int year() const;

        void add_year(int n);       // add n years
        // ...
};
```

**const** sau danh sách đối số (trống) trong khai báo hàm chỉ ra các hàm này không thể sửa đổi trạng thái của lớp **Date**.

Khi một **hằng** hàm thành phần được định nghĩa bên ngoài lớp, hậu tố **const** là bắt buộc:

Một **hằng** hàm thành phần có thể gọi cho cả đối tượng **hằng** và không **hằng**, trong khi một hàm thành viên không **hằng** chỉ có thể gọi cho các đối tượng không phải **hằng**. Ví dụ:

```
void f(Date& d, const Date& cd)
{
        int i = d.year();       // OK
        d.add_year(1);          // OK

        int j = cd.year();      // OK
        cd.add_year(1);         // error: cannot change value of a const Date
}
```

### 1.9.2 Hằng số vật lý và logic

Một hàm thành viên về mặt logic là hằng nhưng vẫn cần thay đổi giá trị của một thành viên. Nghĩa là hàm không thay đổi trạng thái đối tượng của nó, nhưng một số chi tiết mà người dùng không thể quan sát trực tiếp được cập nhật. Điều này được gọi là *hằng số logic*. Ví dụ, lớp **Date** có thể có một hàm trả về biểu diễn chuỗi. Việc xây dựng biểu diễn này có thể tương đối tốn kém. Do đó nên giữ một bản sao để các yêu cầu lặp đi lặp lại chỉ trả lại bản sao đó, trừ khi giá trị **Date** thay đổi. Lưu trữ các giá trị như vậy phổ biến hơn các cấu trúc dữ liệu phức tạp hơn, nhưng hãy xem cách có được giá trị đó cho một **Date**:

```
class Date {
public:
        // ...
        string string_rep() const;      // string representation
private:
        bool cache_valid;
        string cache;
        void compute_cache_value(); // fill cache
        // ...
};
```

Những vấn đề như vậy có thể giải quyết bằng cách ép kiểu, nhưng cũng có cách giải quyết hợp lý mà không làm ảnh hưởng tới các loại quy tắc.

### 1.9.3 Tính thay đổi

Chúng ta có thể định nghĩa một thành viên của một lớp mang **tính thay đổi**, nghĩa là nó có thể sửa đổi thậm chí trong một đối tượng **hằng**:

```
class Date {
public:
        // ...
        string string_rep() const;              // string representation
private:
        mutable bool cache_valid;
        mutable string cache;
        void compute_cache_value() const;       // fill (mutable) cache
        // ...
};
```

Giờ chúng ta có thể định nghĩa **string_rep()** cách rõ ràng:

```
string Date::string_rep() const
{
        if (!cache_valid) {
                compute_cache_value();
                cache_valid = true;
        }
        return cache;
}
```

Ta giờ có thể sử dụng **string_rep()** cho cả những đối tượng **hằng** và không **hằng**. Ví dụ:

```
void f(Date d, const Date cd)
{
        string s1 = d.string_rep();
        string s2 = cd.string_rep();            // OK!
        // ...
}
```

### 1.10 Quyền truy cập thành viên

Một thành viên của lớp **X** có thể được truy cập bằng . (dấu chấm) hay bởi -> (mũi tên) cho một con trỏ đến một đối tượng thuộc lớp **X**. Ví dụ:

```
struct X {
        void f();
        int m;
};

void user(X x, X* px)
{
        m = 1;          // error: there is no m in scope
        x.m = 1;        // OK
        x->m = 1;       // error: x is not a pointer
        px->m = 1;      // OK
        px.m = 1;       // error: px is a pointer
}
```

Bên trong lớp thì không cần toán tử. Ví dụ:

```
void X::f()
{
        m = 1;          // OK: "this->m = 1;" (§16.2.10)
}
```

Một tên thành viên không đủ tiêu chuẩn hoạt động như thể nó đã được đặt **this** -> trước đó. Một hàm thành phần có thể tham chiếu đến tên của một thành viên trước khi cả khai báo:

```
struct X {
        int f() { return m; }    // fine: return this X's m
        int m;
};
```

Nếu muốn tham chiếu đến một thành viên nói chung, dùng **::** theo sau tên lớp:

```
struct S {
        int m;
        int f();
        static int sm;
};

int X::f() { return m; }          // X's f
int X::sm {7};                    // X's static member sm (§16.2.12)
int (S::*) pmf() {&S::f};         // X's member f
```

### 1.11 Các thành viên [tĩnh]

Một biến là một phần của lớp, nhưng không phải là một phần của một đối tượng của lớp đó, được gọi là thành viên **tĩnh**. Tương tự với cả hàm được gọi là hàm thành phần **tĩnh**.

Một thiết kế lại bảo đảm ý nghĩa của các giá trị phương thức khởi tạo mặc định cho **Date** mà không gặp vấn để bắt nguồn từ việc phụ thuộc vào biến toàn cục:

```
class Date {
        int d, m, y;
        static Date default_date;
public:
        Date(int dd =0, int mm =0, int yy =0);
        // ...
        static void set_default(int dd, int mm, int yy); // set default_date to Date(dd,mm,yy)
};
```

Sử dụng **set_dafault ()**, ta có thể thay đổi ngày mặc định thích hợp. Một thành viên **tĩnh** có thể được gọi như những thành viên khác, ngoài ra có thể được tham chiếu mà không cần đề cập đến một đối tượng. Chỉ cần tên của nó đủ điều kiện bởi tên của lớp. Ví dụ:

```
void f()
{
        Date::set_default(4,5,1945);    // call Date's static member set_default()
}
```

Nếu được sử dụng, một thành viên tĩnh phải được định nghĩa ở đâu đó. Từ khoá **static** không được lặp trong định nghĩa của một thành viên tĩnh. Ví dụ:

```
Date Date::default_date {16,12,1770};          // definition of Date::default_date

void Date::set_default(int d, int m, int y)    // definition of Date::set_default
{
      default_date = {d,m,y};                  // assign new value to default_date
}
```

### 1.12 Các kiểu thành viên

Các kiểu và bí danh có thể là thành viên của một lớp. Ví dụ:

```
template<typename T>
class Tree {
      using value_type = T;                    // member alias
      enum Policy { rb, splay, treeps };       // member enum
      class Node {                             // member class
            Node* right;
            Node* left;
            value_type value;
      public:
            void f(Tree*);
      };
      Node* top;
public:
      void g(const T&);
      // ...
};
```

Một lớp thành viên (thường được gọi là lớp lồng nhau) có thể tham chiếu đến các thành viên **tĩnh** của lớp bao quanh nó. Nó chỉ có thể tham chiếu đến các thành viên không **tĩnh** khi nó được cung cấp một đối tượng của lớp bao quanh để tham chiếu đến.

Các lớp thành viên là một sự tiện lợi về mặt ký hiệu hơn là một tính năng có tầm quan trọng cơ bản. Mặt khác, bí danh thành viên rất quan trọng như là cơ sở của kỹ thuật lập trình chung dựa trên các kiểu liên kết. Các thành viên **enum** thường là một sự thay thế cho các **enum**.

## 2. Nạp chồng toán tử

```
bool operator!=(Date, Date);        // inequality
bool operator<(Date, Date);         // less than
bool operator>(Date, Date);         // greater than
// ...

Date& operator++(Date& d) { return d.add_day(1); }          // increase Date by one day
Date& operator--(Date& d) { return d.add_day(-1); }         // decrease Date by one day

Date& operator+=(Date& d, int n) { return d.add_day(n); }   // add n days
Date& operator-=(Date& d, int n) { return d.add_day(-n); }  // subtract n days

Date operator+(Date d, int n) { return d+=n; }              // add n days
Date operator-(Date d, int n) { return d+=n; }              // subtract n days

ostream& operator<<(ostream&, Date d);                      // output d
istream& operator>>(istream&, Date& d);                     // read into d
```

Các toán tử này được định nghĩa trong **Chrono** cùng với **Date** để tránh quá tải và hưởng lợi từ tra cứu phụ thuộc đối số.

Đối với **Date**, những toán tử này có thể được coi là những tiện ích đơn thuần. Tuy nhiên, đối với nhiều loại - chẳng hạn như số phức, vecto, và các đối tượng giống hàm – việc sử dụng các toán tử thông thường khiến mọi người cho rằng định nghĩa của chúng gần như là bắt buộc.

# Toán tử đặc biệt

## 1. Giới thiệu

Nạp chồng không chỉ dành cho các phép toán số học và logic. Trên thực tế, các toán tử đóng vai trò quan trọng trong việc thiết kế vùng chứa, "con trỏ thông minh", iterator và các lớp khác liên quan đến quản lý tài nguyên.

## 2. Các toán tử đặc biệt

Các toán tử

```
[]  ()  ->  ++  --  new  delete
```

chỉ đặc biệt ở chỗ ánh xạ từ việc sử dụng chúng trong mã đến định nghĩa của lập trình viên hơi khác so với ánh xạ được sử dụng cho các toán tử đơn phân và nhị phân thông thường, chẳng hạn như +, <, và ~. Toán tử **[]** (chỉ số dưới) và **()** (gọi) là một trong những toán tử hữu ích nhất do người dùng xác định.

### 2.1 Toán tử gián tiếp

Toán tử gián tiếp, → (còn được gọi là toán tử mũi tên), có thể được định nghĩa là toán tử hậu tố một ngôi. Ví dụ:

```
class Ptr {
      // ...
      X* operator->();
};
```

Các đối tượng của lớp **Ptr** có thể được sử dụng để truy cập các thành viên của lớp **X** theo cách rất giống với cách con trỏ sử dụng. Ví dụ:

```
void f(Ptr p)
{
      p->m = 7;       // (p.operator->())->m = 7
}
```

Việc biến đổi đối tượng **p** thành con trỏ **p.operator → ()** không phụ thuộc vào thành viên **m** được trỏ tới. Đó là nghĩa mà **toán tử → ()** là một toán tử hậu tố một ngôi. Tuy nhiên, không có cú pháp mới nào được giới thiệu, vì vậy tên thành viên vẫn được yêu cầu sau dấu → Ví dụ:

```
void g(Ptr p)
{
      X* q1 = p->;              // syntax error
      X* q2 = p.operator->();   // OK
}
```

### 2.2 Tăng và giảm

Khi con người phát minh ra "con trỏ thông minh", họ thường quyết định cung cấp toán tử tăng ++ và toán tử giảm -- để phản ánh việc sử dụng các toán tử này cho các kiểu cài sẵn. Điều này đặc biệt rõ ràng và cần thiết khi mục đích là thay thế một loại con trỏ thông thường bằng một loại " con trỏ thông minh " có cùng ngữ nghĩa, ngoại trừ việc nó thêm một chút kiểm tra lỗi thực thi. Ví dụ: hãy xem xét một chương trình truyền thống:

```
void f1(X a)          // traditional use
{
      X v[200];
      X* p = &v[0];
      p--;
      *p = a;     // oops: p out of range, uncaught
      ++p;
      *p = a;     // OK
}
```

Ở đây, chúng ta có thể muốn thay thế **X*** bằng một đối tượng của một lớp **Ptr <X>** chỉ có thể được tham chiếu nếu nó thực sự trỏ đến **X**

```
void f2(Ptr<X> a)          // checked
{
        X v[200];
        Ptr<X> p(&v[0],v);
        p--;
        *p = a;     // run-time error: p out of range
        ++p;
        *p = a;     // OK
}
```

### 2.3 Cấp phát và giải phóng

Toán tử **new** có được bộ nhớ của nó bằng cách gọi một **toán tử new()**. Tương tự, toán tử **xóa** giải phóng bộ nhớ của nó bằng cách gọi một **toán tử delete()**. Người dùng có thể xác định lại toán tử toàn cầu **new()** và **delete()** hay định nghĩa toán tử **new()** và **delete()** cho một lớp cụ thể.

```
void* operator new(size_t);                 // use for individual object
void* operator new[](size_t);               // use for array
void operator delete(void*, size_t);        // use for individual object
void operator delete[](void*, size_t);      // use for array
```

Một cách tiếp cận chọn lọc hơn và thường tốt hơn là cung cấp các hoạt động này cho một lớp cụ thể. Lớp này có thể là cơ sở cho nhiều lớp dẫn xuất. Ví dụ, chúng tôi có thể muốn chào đón một lớp **Employee** cung cấp một trình phân bổ và phân bổ giao dịch chuyên biệt cho chính nó và tất cả các lớp dẫn xuất của nó

```
class Employee {
public:
      // ...

      void* operator new(size_t);
      void operator delete(void*, size_t);

      void* operator new[](size_t);
      void operator delete[](void*, size_t);
};
```

Việc giải phóng được thự hiện bởi hàm huỷ (thứ biết được kích thước của lớp của nó)

## 3. Các hàm hỗ trợ

Một tập hợp các hàm hữu ích, luồng I/O, hỗ trợ các vòng lặp phạm vi cho, so sánh và nối. Tất cả những điều này đều phản ánh các lựa chọn thiết kế được sử dụng

cho **std::string**. Cụ thể, << chỉ in các ký tự mà không cần thêm định dạng và >> bỏ qua khoảng trắng đầu tiên trước khi đọc cho đến khi tìm thấy khoảng trắng kết thúc.

```
ostream& operator<<(ostream& os, const String& s)
{
        return os << s.c_str();
}

istream& operator>>(istream& is, String& s)
{
        s = "";      // clear the target string
        is>>ws;
        char ch = ' ';
        while(is.get(ch) && !isspace(ch))
                s += ch;
        return is;
}
```

Toán tử == và **!**= cho so sánh:

```
bool operator==(const String& a, const String& b)
{
        if (a.size()!=b.size())
                return false;
        for (int i = 0; i!=a.size(); ++i)
                if (a[i]!=b[i])
                        return false;
        return true;
}

bool operator!=(const String& a, const String& b)
{
        return !(a==b);
}
```

### 4. Hàm bạn

Một hàm được khai báo **friend** được cấp quyền truy cập vào việc triển khai một lớp giống như một hàm thành viên nhưng độc lập với lớp đó.

Ví dụ, chúng ta có thể xác định một toán tử nhân **Ma trận** với một **Vectơ**. Đương nhiên, **Vectơ** và **Ma trận** ẩn các biểu diễn tương ứng của chúng và cung cấp một tập hợp các thao tác hoàn chỉnh để thao tác các đối tượng cùng loại. Tuy nhiên, thói quen nhân của chúng ta không thể là thành viên của cả hai. Ngoài ra, chúng ta không thực sự muốn cung cấp các chức năng truy cập cấp thấp để cho phép mọi người dùng có thể đọc và ghi toàn bộ biểu diễn của cả **Ma trận** và **Vectơ**. Để tránh điều này, chúng ta khai báo **toán tử *** là **bạn** của cả hai.

```
class Matrix;

class Vector {
        float v[rc_max];
        // ...
        friend Vector operator*(const Matrix&, const Vector&);
};

class Matrix {
        Vector v[rc_max];
        // ...
        friend Vector operator*(const Matrix&, const Vector&);
};
```

Bây giờ **toán tử ∗()** có thể tiếp cận việc triển khai cả **Vectơ** và **Ma trận**. Điều đó sẽ cho phép các kỹ thuật triển khai phức tạp, nhưng thực hiện đơn giản:

```
Vector operator*(const Matrix& m, const Vector& v)
{
        Vector r;
        for (int i = 0; i!=rc_max; i++) {          // r[i] = m[i] * v;
                r.v[i] = 0;
                for (int j = 0; j!=rc_max; j++)
                        r.v[i] += m.v[i].v[j] * v.v[j];
        }
        return r;
}
```

Một hàm thành viên của một lớp có thể là bạn của lớp khác. Ví dụ:

```
class List_iterator {
        // ...
        int* next();
};

class List {
        friend int* List_iterator::next();
        // ...
};
```

Có một cách viết tắt để biến tất cả các chức năng của lớp này trở thành bạn của lớp khác. Ví dụ:

```
class List {
        friend class List_iterator;
        // ...
};
```

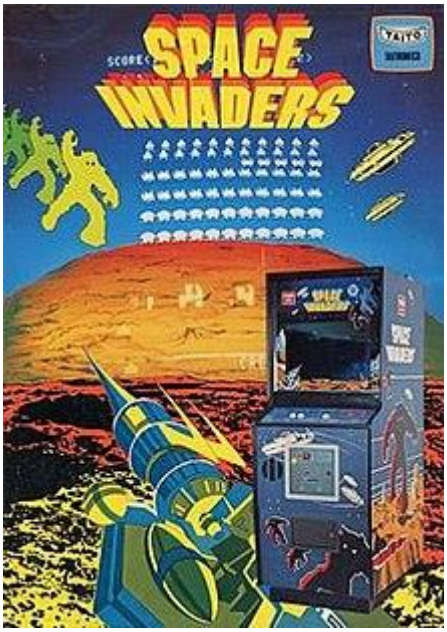
# PHẦN 2: CHƯƠNG TRÌNH

## 1. Lý do chọn đề tài

Trong những năm gần đây, nhu cầu giải trí của con người ngày càng cao, ai cũng muốn được giải toả tinh thần sau những ngày làm việc mệt nhọc. Tuy nhiên, không phải ai cũng có thời gian, cơ hội để mà tham gia các hoạt động vui trời ngoài trời hay chỉ đơn giàn là họ không thích đi đâu xa mà chỉ mong muốn ở nhà giải trí. Chính vì thế họ đã tìm tới các trò chơi ngay trên điện thoại di động, máy tính để tìm niềm vui. Khi biết được nhu cầu đó, nhóm em quyết định cùng nhau xây dựng một game nhằm đáp ứng điều đó.

Tuy nhiên, với trình độ, kinh nghiệm cùng với kiến thức hạn hẹp, nhóm em chưa thể tạo ra game có cho mình sức hút lớn, đồ hoạ đẹp cùng với lối chơi độc đáo như những sản phẩm có mặt trên thị trường game bây giờ. Nhưng vì mong muốn có thể thực hiện điều đó, nhóm em quyết định bắt đầu từ những game đơn giản để từ đó hiểu cách hoạt động xây dựng và làm tiền đề cho những game phức tạp hơn. Những game cuối thế kỷ XX thì lại rất phù hợp với điều đó và nhóm em đã tham gia cùng nhau làm ra game mang tên Space Invaders.

Space Invaders là trò chơi đầu tiên hầu hết các game thủ trải nghiệm khi nó được phát hành vào năm 1978. Tựa game đấy không khó cũng không quá dễ để xây dựng, phù hợp cho nhóm em áp dụng kiến thức đã học nói chung và lập trình hướng đối tượng nói riêng.




**Hình 1.1 Space Invaders**

## 2. Mô tả bài toán

Người chơi điều khiển một con tàu không gian di chuyển từ bên này sang bên kia dọc theo đáy màn hình để phá huỷ một đội quân xâm lược ngoài hành tinh đang đến. Ba loại khác nhau của người ngoài hành tinh được xếp thành năm hàng. Bốn mươi tám người ngoài hành tinh di chuyển trong hình khi chúng bò từ một phía của màn hình sang bên kia, rơi xuống khi chúng chạm tới cạnh của màn hình, sau đó quay ngược lại theo hướng ngược lại. Càng ít số người ngoài hành tinh còn sống, chúng càng di chuyển nhanh hơn cho đến khi chúng tới phạm vi của tàu không gian.

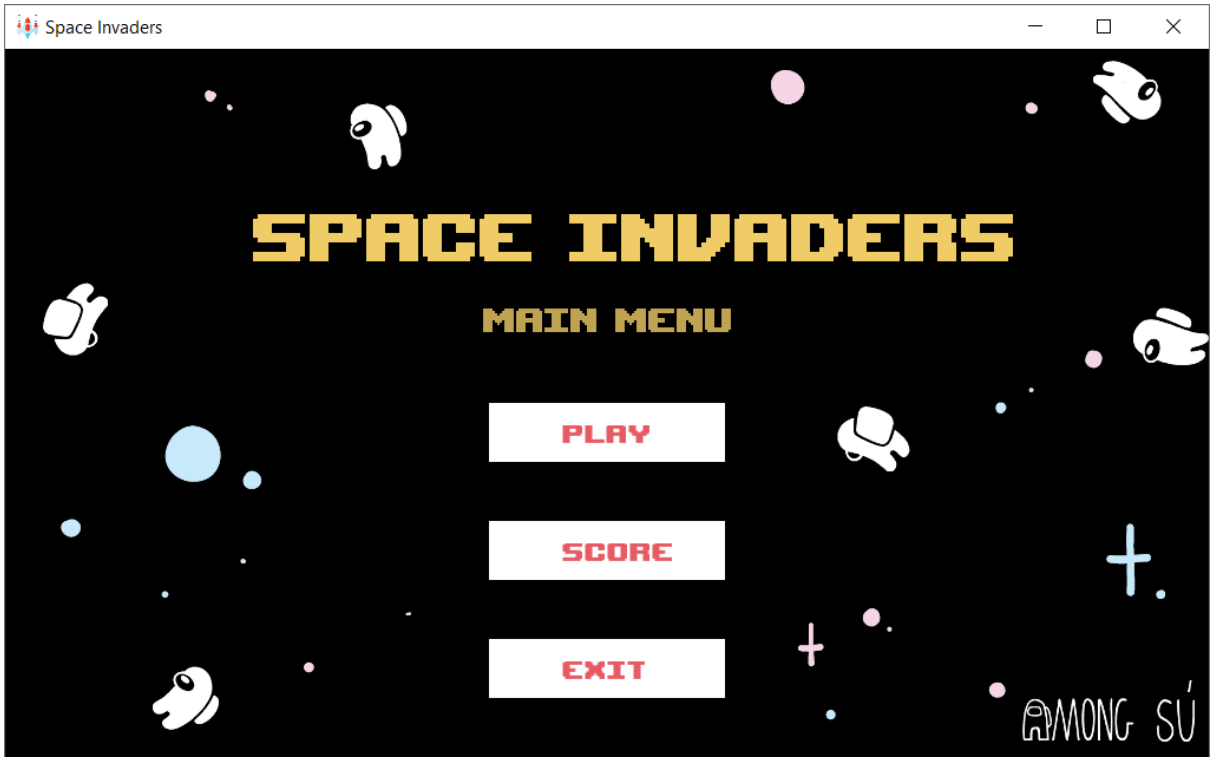
Tàu không gian chỉ có thể bắn ra một phát đạn. Đạn tiếp theo được bắn khi đạn trước không còn xuất hiện trên màn hình. Người chơi cũng phải điều khiển tàu không gian để né tránh những tên lửa được bắn trả của kẻ thù.

Người chơi ban đầu nhận được ba mạng sống và có thể đánh mất chúng bằng cách bị bắn bởi hoả lực của người ngoài hành tinh hoặc khi chúng xâm lấn đến phạm vi di chuyển của con tàu không gian người.

Người chơi chiến thắng ở mỗi đợt sóng tấn công nếu họ tiêu diệt tất cả kẻ thù đang tới và đối mặt với đợt sóng tiếp theo.

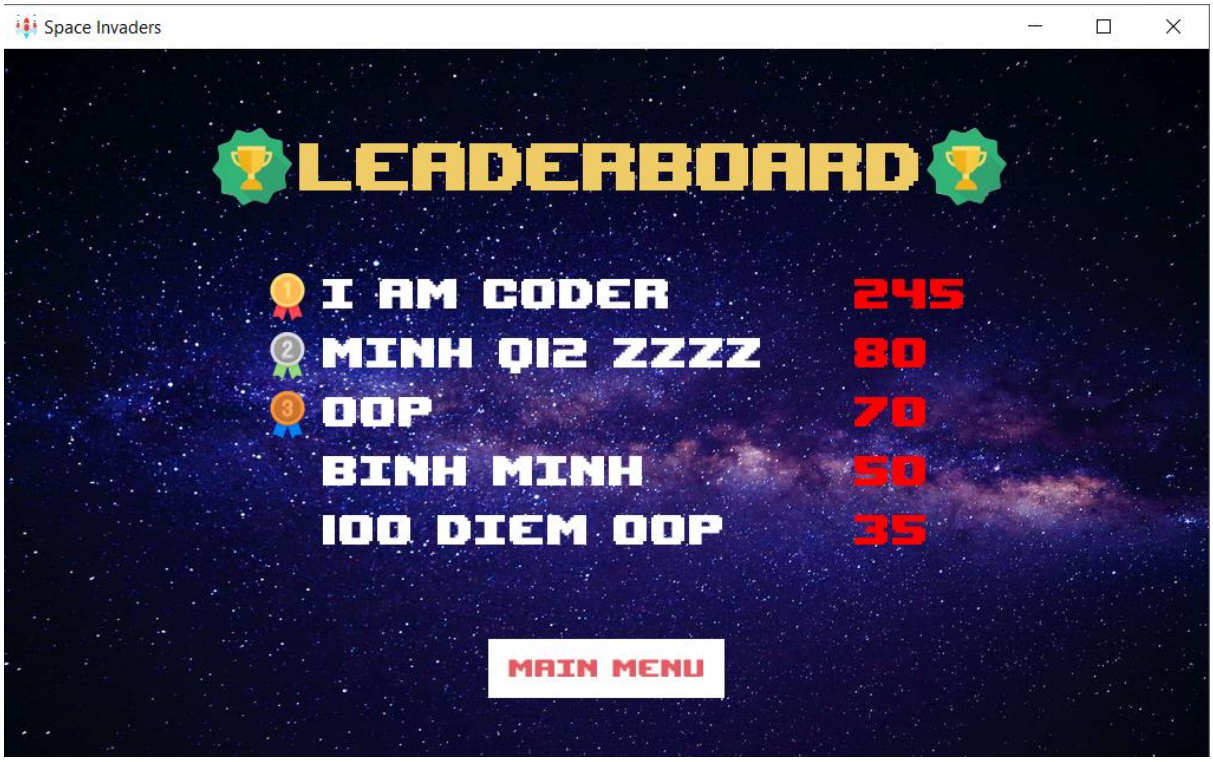
## 3. Giao diện chương trình

### 3.1 Giao diện khi bắt đầu game



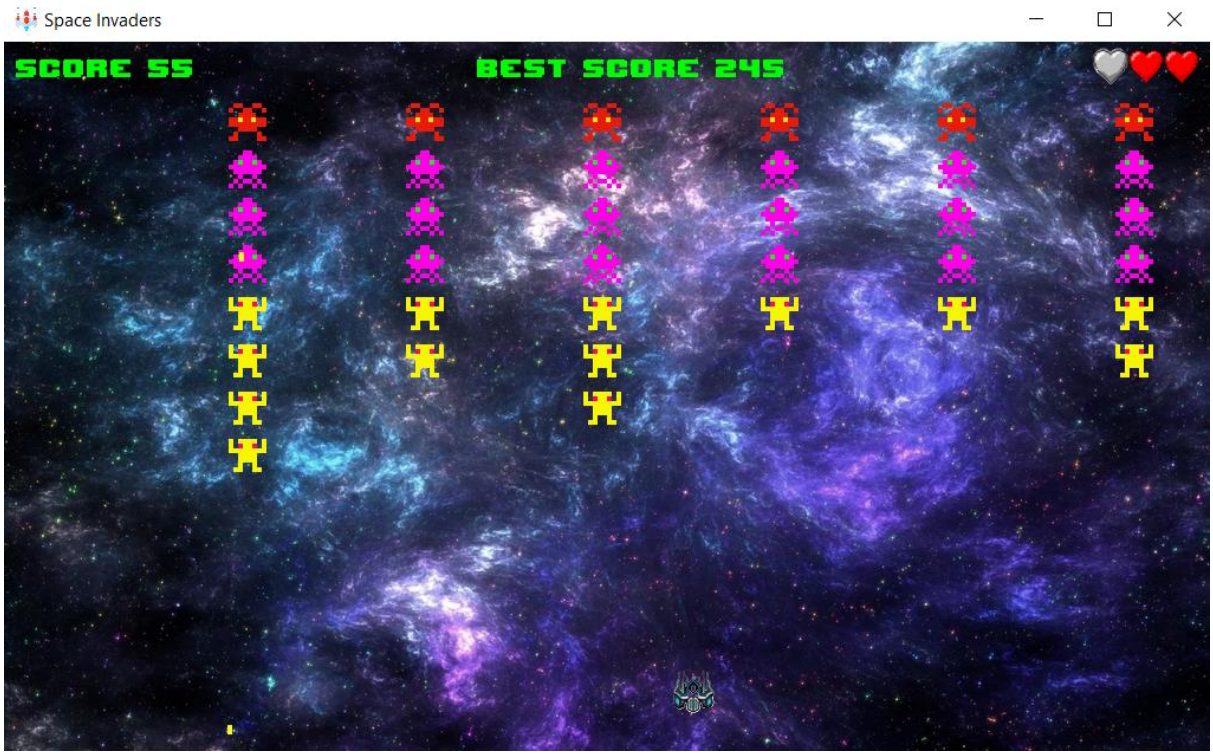

**Hình 3.1 Menu chính**

### 3.2 Giao diện hiển thị thông tin người chơi có điểm cao



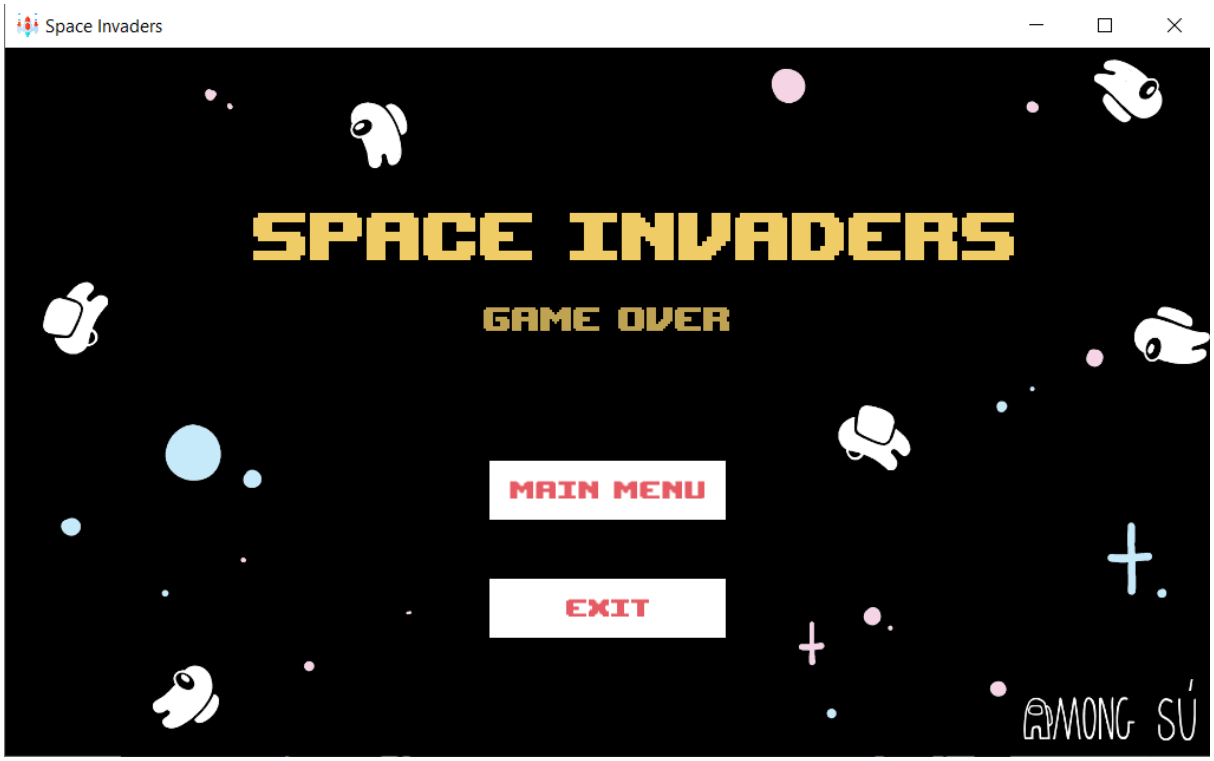

**Hình 3.2 Năm người chơi có điểm cao nhất**

### 3.3 Giao diện khi chơi game



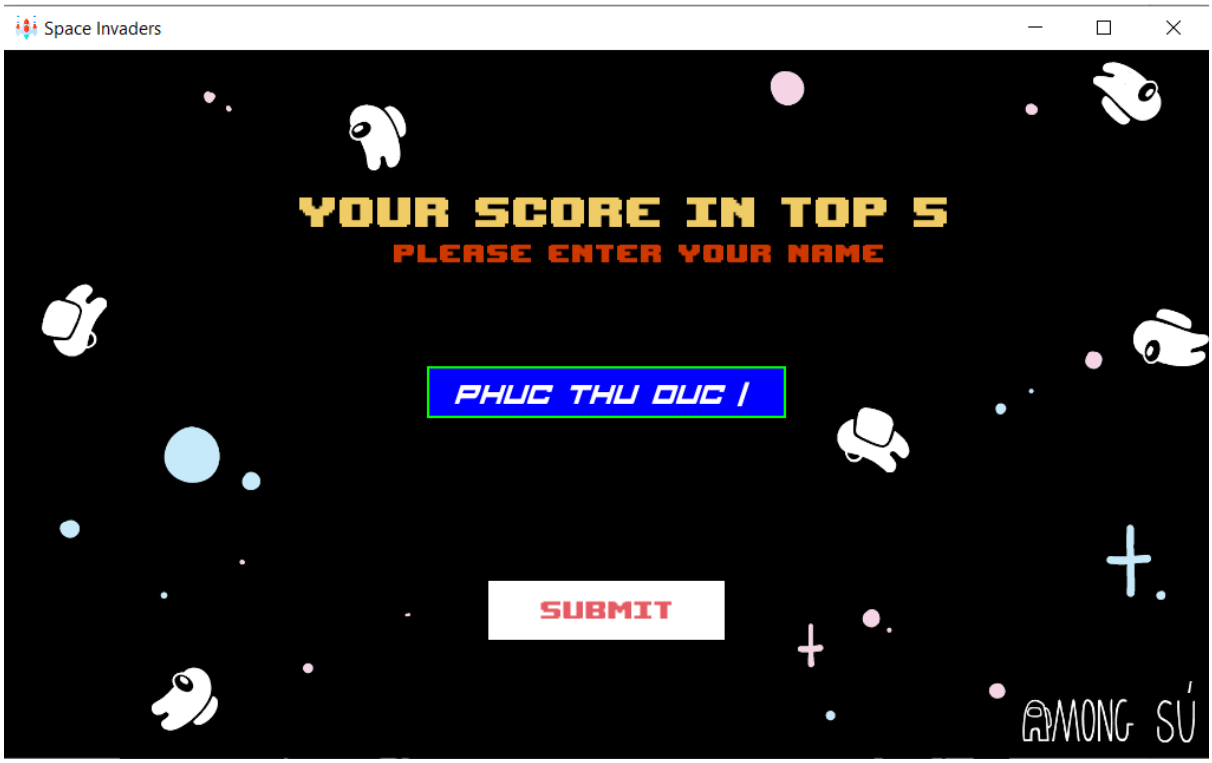

**Hình 3.3 Quá trình chơi game**

### 3.4 Giao diện thua cuộc




**Hình 3.4 Game over**

### 3.5 Giao diện nhập tên khi người chơi đạt điểm cao




**Hình 3.5 Nhập tên**

## 4. Nội dung lý thuyết

### 4.1 Tính chất hướng đối tượng

Chương trình được sử dụng đầy đủ 4 tính chất của hướng đối tượng: tính đóng gói, tính trừu tượng, tính kế thừa và tính đa hình

#### 4.1.1 Tính đóng gói

Tính đóng gói (Encapsulation) là một khái niệm của lập trình hướng đối tượng mà ràng buộc dữ liệu và các hàm mà thao tác dữ liệu đó, và giữ chúng an toàn bởi ngăn cản sự gây trở ngại và sự lạm dụng từ bên ngoài. Tính đóng gói dẫn đến khái niệm OOP quan trọng là Data Hiding.

Tính đóng gói - Data encapsulation là một kỹ thuật đóng gói dữ liệu, và các hàm mà sử dụng chúng và trừu tượng hóa dữ liệu là một kỹ thuật chỉ trưng bày tới các Interface và ẩn Implementation Detail (chi tiết trình triển khai) tới người sử dụng.

C++ hỗ trợ các thuộc tính của đóng gói và ẩn dữ liệu thông qua việc tạo các kiểu tự định nghĩa (user-defined), gọi là classes. Một lớp có thể chứa các thành viên private, protected và public. Theo mặc định, tất cả thuộc tính được định nghĩa trong một lớp là private.

```
class playerShip{
    private:
        bool alive;
        int xmax=980,xmin=0,speed=5;
        int width=36;
        int height=39;
    public:
        initShip();
        void draw (RenderWindow& app);
        void move();
};
```

**Hình 4.1 Tính chất đóng gói**

Cụ thể trong bài của nhóm, các biến alive, width, height,… là **private**. Nghĩa là chúng chỉ có thể truy cập bởi các thành viên khác của lớp playerShip, và không thể gọi bởi bất kỳ phần khác của chương trình. Đây là một cách thực hiện tính đóng gói trong C++.

#### 4.1.2 Tính trừu tượng

Trừu tượng hóa dữ liệu (Data abstraction) liên quan tới việc chỉ cung cấp thông tin cần thiết tới bên ngoài và ẩn chi tiết cơ sở của chúng.

### 4.1.3 Tính kế thừa

Một trong những khái niệm quan trọng nhất trong lập trình hướng đối tượng là **Tính kế thừa (Inheritance)**. Kế thừa trong C++ là sự liên quan giữa hai class với nhau, trong đó có class cơ sở (Base Class) và class con (Derived Class). Khi kế thừa class con được hưởng tất cả các phương thức và thuộc tính của class cơ sở. Tuy nhiên, nó chỉ được truy cập các thành viên public và protected của class cơ sở. Nó không được phép truy cập đến thành viên private của class cơ sở.

Tư tưởng của kế thừa trong C++ là có thể tạo ra một class mới được xây dựng trên các lớp đang tồn tại. Khi kế thừa từ một lớp đang tồn tại ta có sử dụng lại các phương thức và thuộc tính của lớp cơ sở, đồng thời có thể khai báo thêm các phương thức và thuộc tính khác.

Khi kế thừa từ một lớp cơ sở, lớp cơ sở đó có thể được kế thừa thông qua kiểu kế thừa là **public**, **protected** hoặc **private**. Kiểu kế thừa trong C++ được xác định bởi Access-specifier đã được giải thích ở trên.

Chúng ta hiếm khi sử dụng kiểu kế thừa **protected** hoặc **private**, nhưng kiểu kế thừa **public** thì được sử dụng phổ biến hơn. Trong khi sử dụng các kiểu kế thừa khác sau, bạn nên ghi nhớ các quy tắc sau:

**Kiểu kế thừa public**: Khi kế thừa từ một lớp cơ sở là public, thì các thành viên public của lớp cơ sở trở thành các thành viên public của lớp kế thừa; và các thành viên protected của lớp có sở trở thành các thành viên protected của lớp kế thừa. Một thành viên là private của lớp cơ sở là không bao giờ có thể được truy cập trực tiếp từ một lớp kế thừa, nhưng có thể truy cập thông qua các lời gọi tới các thành viên **public** và **protected** của lớp cơ sở đó.

**Kiểu kế thừa protected**: Khi kế thừa từ một lớp cơ sở là protected, thì các thành viên public và protected của lớp cơ sở trở thành các thành viên protected của lớp kế thừa.

**Kiểu kế thừa private**: Khi kế thừa từ một lớp cơ sở là private, thì các thành viên public và protected của lớp cơ sở trở thành các thành viên private của lớp kế thừa.

Cụ thể trong bài, nhóm em sử dụng kiểu kế thừa public.

```
class Position {
    private:
        int x, y;
        Position(){
            x=-20;
            y=-20;
        }
};

class Alien:public Position {
    private:
        bool alive;
        int height=32,width=32;
        bool explosion=false;
};

class missileAlien:public Position{
    private:
        int speed;
        int height=8,width=4;
        bool active=false;
    public:
        initMissile();
        void move();
        void draw(RenderWindow& app);
        void checkFire (listAliens a);
        bool checkCollisionShip(playerShip a);
        void checkBulletCollisionsShip(playerShip &ship);
};
```

**Hình 4.2 Tính chất kế thừa**

### 4.1.4 Tính đa hình

**Đa hình (polymorphism)** nghĩa là có nhiều hình thái khác nhau. Tiêu biểu là, đa hình xuất hiện khi có một cấu trúc cấp bậc của các lớp và chúng là liên quan với nhau bởi tính kế thừa.

Đa hình trong C++ nghĩa là một lời gọi tới một hàm thành viên sẽ làm cho một hàm khác để được thực thi phụ thuộc vào kiểu của đối tượng mà triệu hồi hàm đó.

Một hàm **virtual** là một hàm trong một lớp cơ sở mà được khai báo bởi sử dụng từ khóa virtual trong C++. Việc định nghĩa trong một lớp cơ sở một hàm virtual, với phiên bản khác trong một lớp kế thừa, báo cho compiler rằng: chúng ta không muốn Static Linkage cho hàm này. Những gì chúng ta làm là muốn việc lựa chọn hàm để được gọi tại bất kỳ điểm nào đã cung cấp trong chương trình là dựa trên kiểu đối tượng, mà với đó nó được gọi.

Cụ thể trong bài, nhóm em cho đa hình hàm **move()** (di chuyển)

```
class Action{
    public:
        virtual void move()=0;
};

class bulletShip:public Position,public Action;
class playerShip:public Position,public Action;


Action *pShip= new playerShip;
Action *pBullet_ship= new bulletShip;

pBullet_ship->move();
pShip->move();
```

**Hình 4.3 Tính chất đa hình**

### 4.2 Đồ hoạ

Nhóm em chọn thư viện SFML để làm đồ hoạ cho game. SFML (là viết tắt của Simple and Fast Multimedia Library) là một thư viện đa phương tiện viết từ C++. Bao gồm 5 moules: Audio, Graphics, Network, System, Window.

- System: gồm các class liên quan với hệ thống như làm thời gian, xử lí unicode.
- Window: liên quan tới việc tạo, đóng và xử lí sự kiện cửa sổ
- Graphics: bao gồm các class về việc biểu diễn đồ hoạ
- Audio: bao gồm các class về xử lý âm thanh, ta có thể dùng để phát một file nhạc hoặc ghi âm trong máy tính và lưu thành file

## 5. Kết luận và kiến nghị

### 5.1 Kết quả đạt được

Trong quá trình tìm hiểu và hoàn thành bài tập lớn với đề tài "Lập trình game Space Invaders" nhóm em đã đạt được những kết quả sau:

- Xây dựng và hoàn chỉnh được một game có đầy đủ các chức năng cơ bản nhất với yêu cầu được đưa ra ban đầu là lập trình hướng đối tượng bằng ngôn ngữ C++.
- Hiểu được cách tạo ra và hoạt động của một game nói chung và game Space Invaders nói riêng.
- Được ôn lại, rèn luyện và áp dụng các kiến thức đã học trong khi tạo ra sản phẩm.

- Được biết thêm nhiều điều mới, thoả sức sáng tạo đồng thời bản thân được nâng cao các kỹ năng tìm hiểu, xử lý vấn đề, kiên nhẫn, khả năng xây dựng và lập trình.

### 5.2 Kiến nghị

Do thời gian tìm hiểu, phân tích, thiết kế cùng với kiến thức còn hạn chế, game Space Invaders của nhóm em so với các sản phẩm cùng loại hiện nay vẫn còn một số thiếu sót như:

- Giao diện chưa thực sự đáp ứng được nhiều về mặt thẩm mỹ.
- Game không có được nhiều tính năng và lối chơi.
- Các lỗi vẫn còn tiềm ẩn trong game mà nhóm chưa tìm ra được.

### 5.3 Hướng phát triển

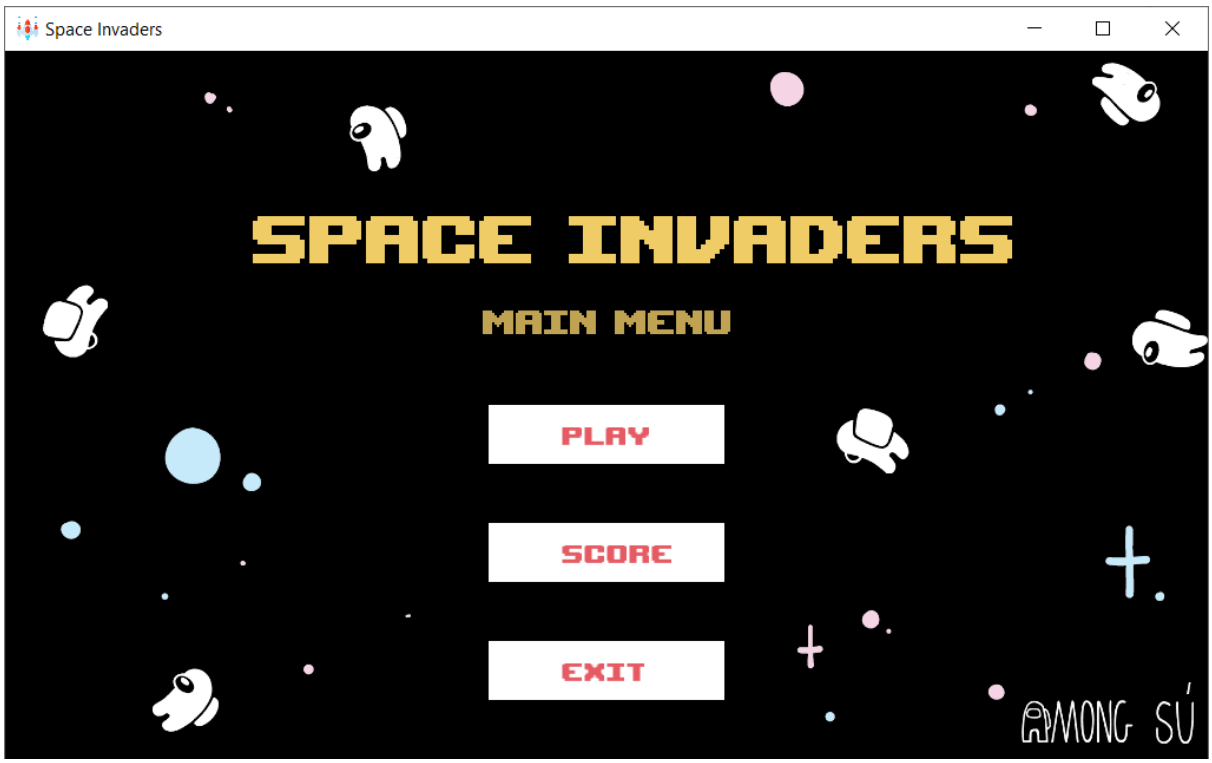
Trong tương lai nếu có điều kiện, sản phẩm sẽ được phát triển theo hướng sau:

- Tìm ra và loại bỏ các lỗi vẫn còn tiềm ẩn để tăng độ chính xác, đúng đắn vốn có của game.
- Tìm hiểu thêm các ngôn ngữ lập trình, các công cụ hỗ trợ khác để cải thiện giao diện, đồ hoạ trong game.
- Phát triển thêm nhiều tính năng, lối chơi để nâng cao trải nghiệm cho người dùng.

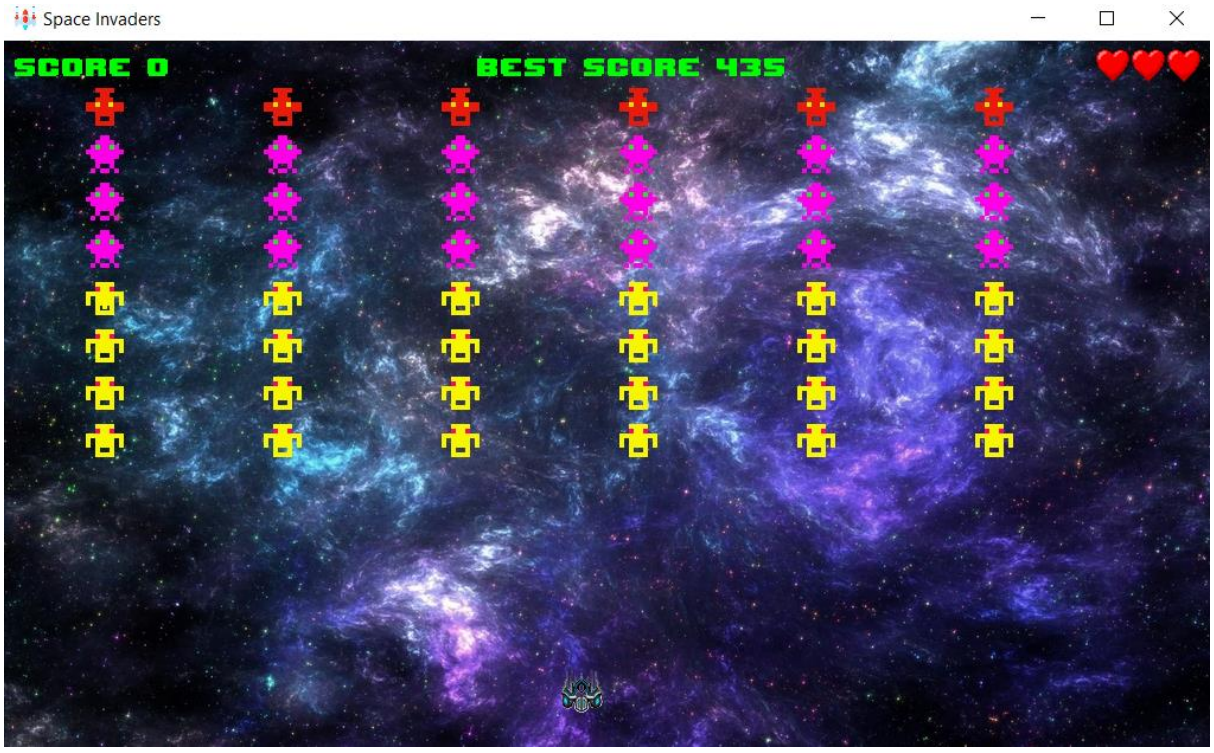
# PHỤ LỤC

## Hướng dẫn sử dụng

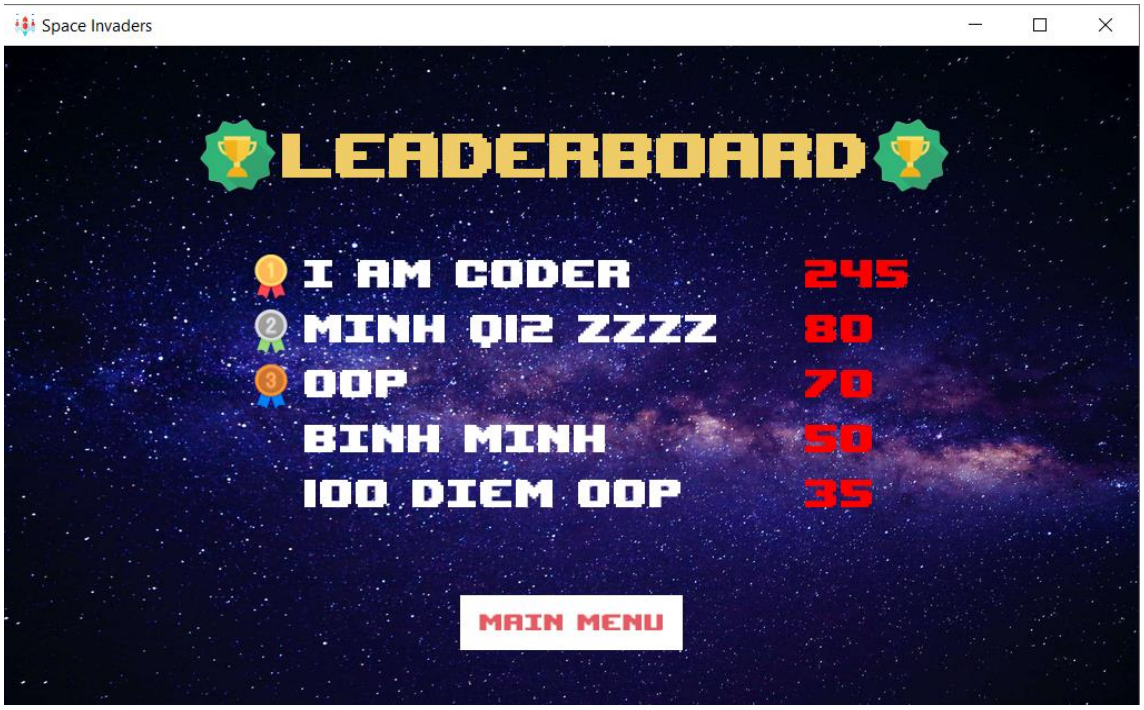
- Sau khi vào chạy chương trình, sẽ có một bảng Menu bao gồm 3 chức năng cho người chơi lựa chọn:




*Chức năng 1: Vào màn hình chơi chính của game:*

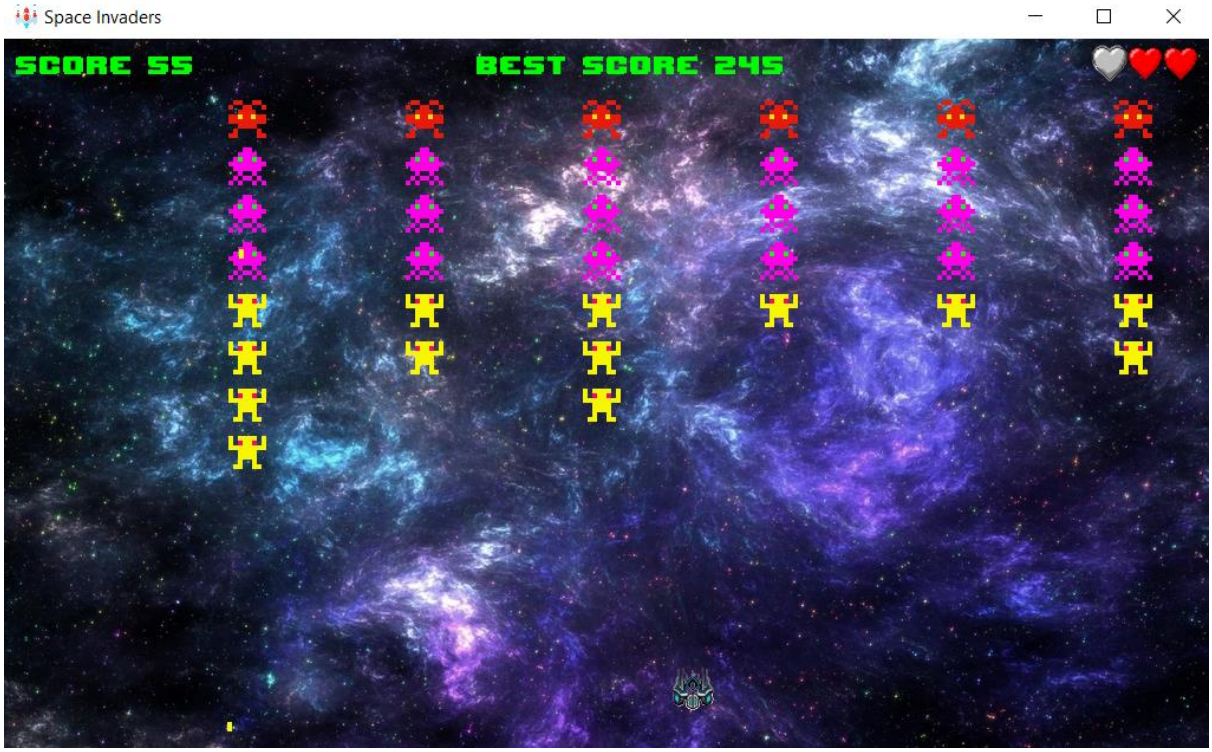


*Chức năng 2: Hiển thị danh sách tên và điểm của người chơi, có thể chọn "Main Menu" để quay lại màn hình chính:*

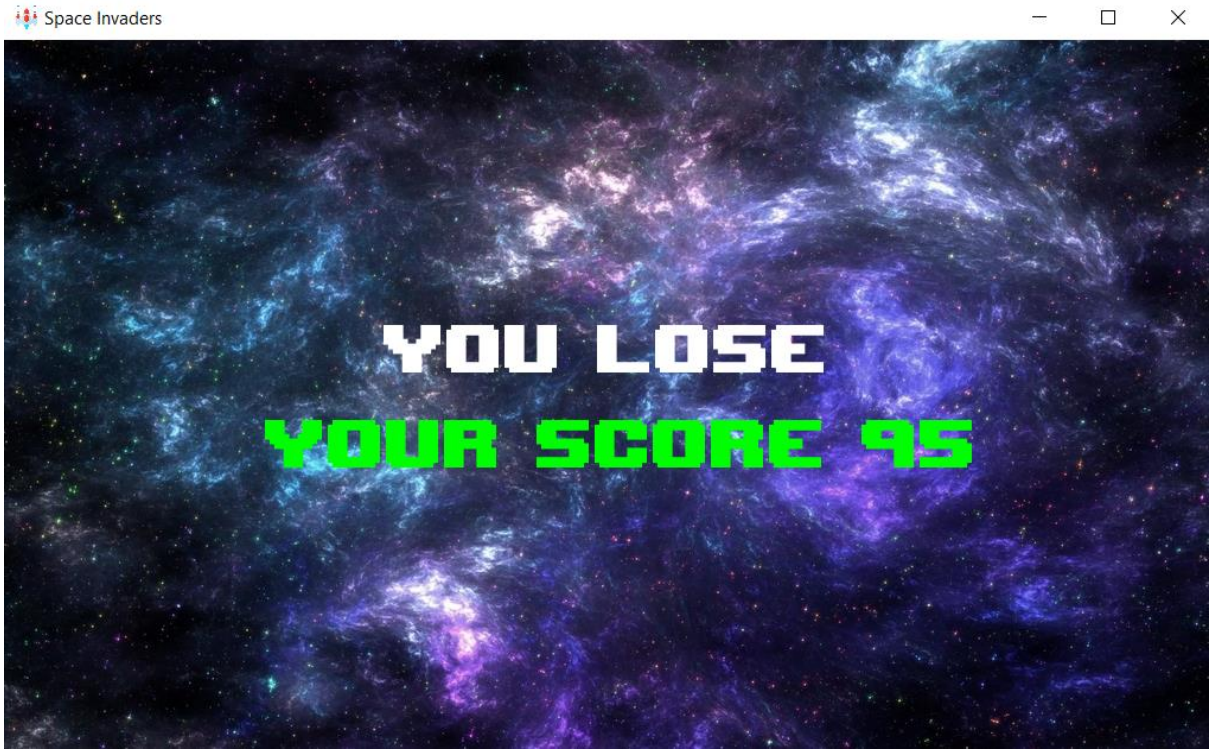


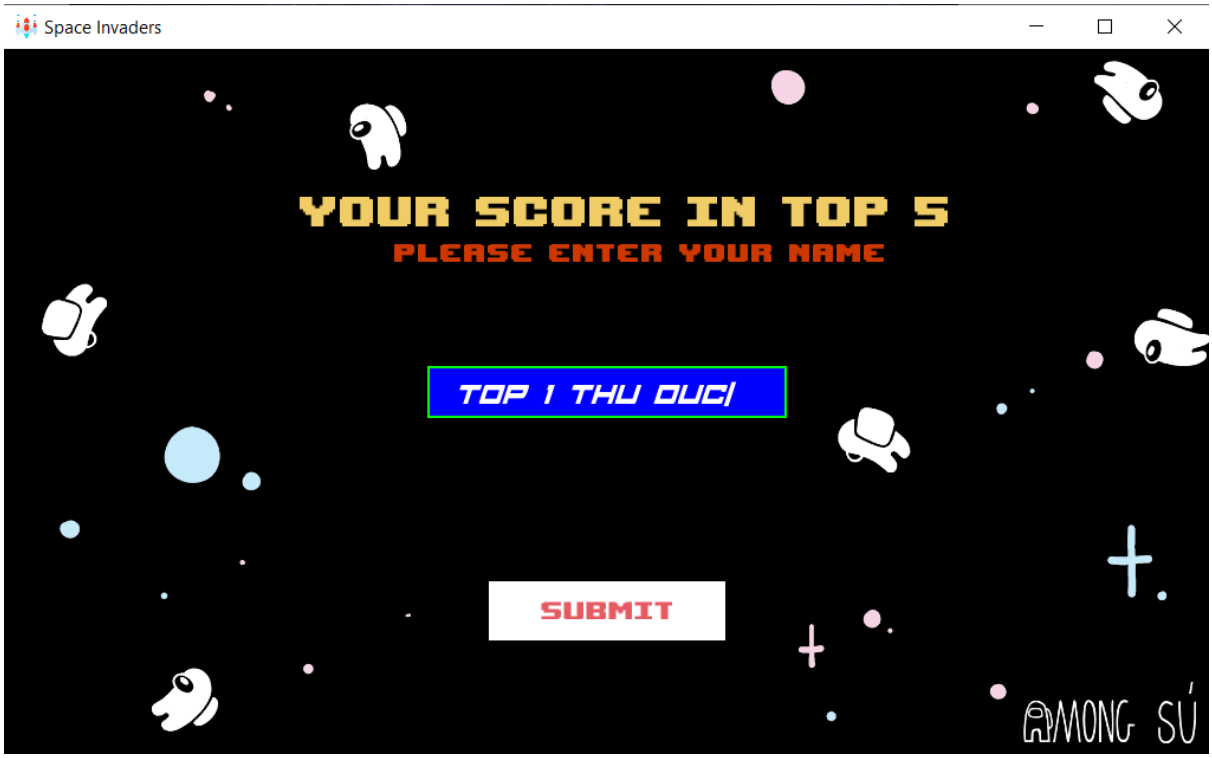

*Chức năng 3: Đóng chương trình*

- Khi chơi, người chơi dùng phím mũi tên (trái, phải để điều khiển) và dùng phím space để tấn công quái:

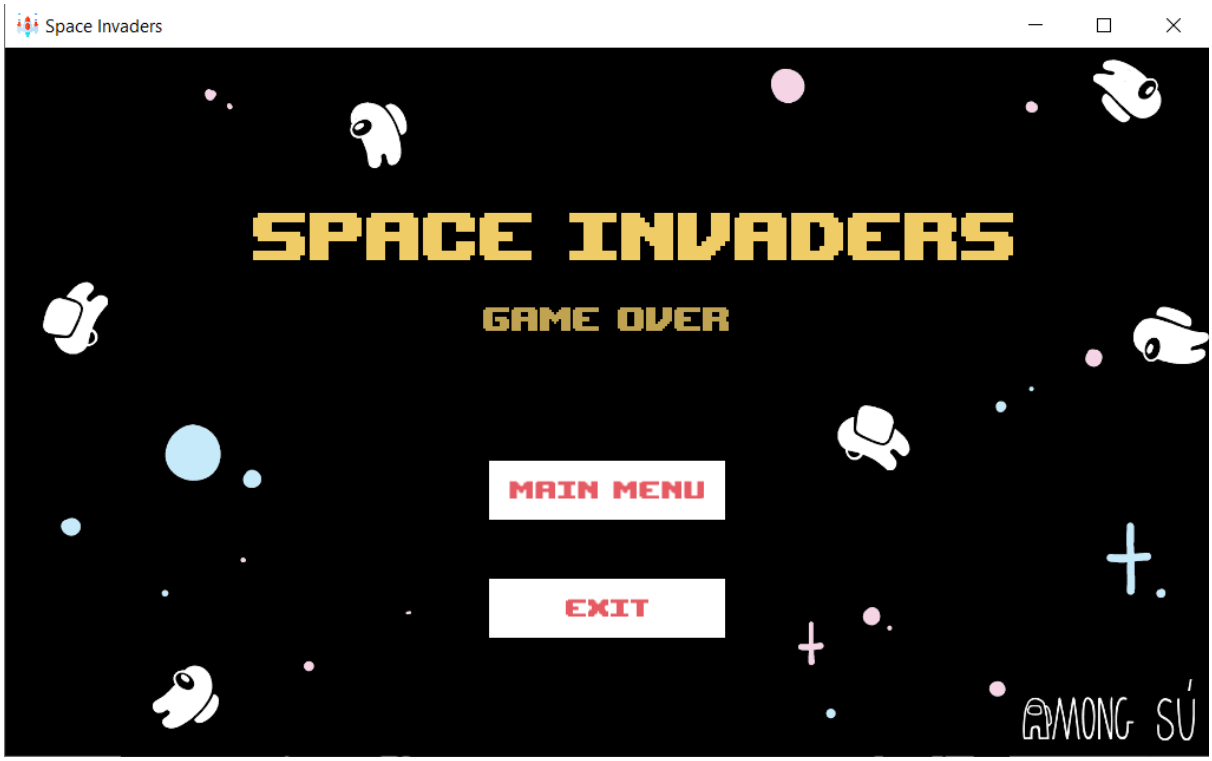



- Sau khi hoàn thành trò chơi, màn hình sẽ thông báo điểm của người chơi, nếu điểm nằm trong top 5 thì sẽ xuất hiện một bảng để người chơi nhập tên:




- Màn hình sẽ hiện ra một menu phụ sau khi nhập tên, người chơi có thể chọn "Main Menu" để quay lại menu chính, hoặc "Exit" để thoát.




**Hướng dẫn cài đặt**

Link github: https://github.com/Dawn-Zzz/ProjectOOP