

# MỤC LỤC

<b>MỤC LỤC.....</b>	<b>1</b>
<b>LỜI NÓI ĐẦU .....</b>	<b>5</b>
<b>CHƯƠNG 1. CÁC CÚ PHÁP C/C++ CƠ BẢN.....</b>	<b>6</b>
<b>1.1 Các khái niệm cơ bản.....</b>	<b>6</b>
1.1.1 Biến, hằng .....	6
1.1.2 Kiểu dữ liệu trong C/C++ .....	8
1.1.3 Các loại toán tử .....	12
<b>1.2 Các cấu trúc điều khiển .....</b>	<b>16</b>
1.2.1 if...else.....	16
1.2.2 Vòng lặp while, do...while.....	17
1.2.3 Vòng lặp for .....	17
1.2.4 switch...case .....	18
1.2.5 break, continue, goto .....	18
<b>1.3 Hàm .....</b>	<b>21</b>
1.3.1 Truyền tham trị.....	21
1.3.2 Truyền tham chiếu .....	22
<b>1.4 Con trỏ .....</b>	<b>23</b>
1.4.1 Tổng quan .....	23
1.4.2 Các phép toán trên con trỏ .....	24
1.4.3 Con trỏ hằng (pointer to const) và hằng con trỏ (const pointer) .....	26
1.4.4 Môi quan hệ giữa mảng và con trỏ.....	27
1.4.5 Bản chất của xâu kí tự kiểu C .....	28
1.4.6 Cấp phát bộ nhớ động.....	29
1.4.7 Con trỏ hàm.....	30
1.4.8 Con trỏ với hàm.....	34
1.4.9 Con trỏ đa cấp .....	37
<b>1.5 Kiểu dữ liệu người dùng tự định nghĩa: .....</b>	<b>38</b>
1.5.1 struct .....	38
1.5.2 enum .....	42
1.5.3 typedef .....	43
1.5.4 using.....	44

<b>1.6 Chỉ thị tiền xử lý.....</b>	<b>45</b>
1.6.1 Giới thiệu .....	45
1.6.2 Chỉ thị chèn tệp (#include).....	45
1.6.3 Chỉ thị định nghĩa cho tên.....	46
1.6.4 Chỉ thị biên dịch có điều kiện .....	48
<b>1.7 Namespace .....</b>	<b>52</b>
1.7.1 Namespace là gì?.....	52
1.7.2 Tại sao phải sử dụng namespace?.....	53
1.7.3 Sử dụng namespace .....	53
1.7.4 Các trường hợp đặc biệt.....	55
<b>1.8 Bài tập.....</b>	<b>57</b>
<b>CHƯƠNG 2. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG .....</b>	<b>58</b>
<b>2.1 Tính chất của lập trình hướng đối tượng.....</b>	<b>58</b>
2.1.1 Tính đóng gói (encapsulation) và che dấu thông tin (information hiding) .....	58
2.1.2 Tính kế thừa (inheritance).....	58
2.1.3 Tính đa hình (polymorphism) .....	58
2.1.4 Tính trừu tượng (abstraction) .....	59
2.1.5 Ví dụ .....	59
<b>2.2 Lớp.....</b>	<b>61</b>
2.2.1 Thành phần của một lớp.....	61
2.2.2 Quyền truy cập các thành phần .....	62
2.2.3 Cú pháp khai báo .....	62
2.2.4 So sánh struct và class trong C++ .....	63
<b>2.3 Constructor, Destructor.....</b>	<b>64</b>
2.3.1 Hàm tạo (Constructor) .....	64
2.3.2 Hàm huỷ (Destructor) .....	66
2.3.3 Con trỏ this.....	67
<b>2.4 Getter, Setter.....</b>	<b>71</b>
2.4.1 Định nghĩa.....	71
2.4.2 Cú pháp .....	71
<b>2.5 Hàm bạn, lớp bạn, static .....</b>	<b>72</b>
2.5.1 Hàm bạn, lớp bạn.....	72
2.5.2 static .....	75
2.5.3 Bài tập.....	80

<b>2.6 Nạp chồng hàm, nạp chồng toán tử.....</b>	<b>82</b>
2.6.1 Nạp chồng hàm.....	82
2.6.2 Nạp chồng toán tử .....	84
2.6.3 Bài tập.....	88
<b>2.7 Tính Kế Thừa.....</b>	<b>90</b>
2.7.1 Tính kế thừa .....	90
2.7.2 Tại sao và khi nào cần dùng tính kế thừa.....	90
2.7.3 Cú pháp sử dụng.....	91
2.7.4 Các phạm vi kế thừa .....	91
2.7.5 Các loại kế thừa trong C++ .....	92
2.7.6 Bài tập.....	104
<b>2.8 Tính đa hình .....</b>	<b>108</b>
2.8.1 Định nghĩa.....	108
2.8.2 Phân loại .....	108
2.8.3 Cơ chế Virtual Table .....	120
2.8.4 Bài tập.....	123
<b>2.9 Template.....</b>	<b>124</b>
2.9.1 Function template .....	124
2.9.2 Class template .....	126
2.9.3 Non-type parameters for template.....	128
2.9.4 Tham số khuôn mẫu .....	129
2.9.5 Thuật toán tổng quát.....	129
2.9.6 Bài tập.....	133
<b>CHƯƠNG 3. TỔNG QUAN VỀ C++ STL .....</b>	<b>134</b>
<b>3.1 Giới thiệu .....</b>	<b>134</b>
<b>3.2 Iterator (Biến lặp) .....</b>	<b>135</b>
<b>3.3 Containers (Thư viện lưu trữ).....</b>	<b>136</b>
3.3.1 Iterator.....	136
3.3.2 Vector (Mảng động).....	137
3.3.3 Deque (Hàng đợi hai đầu).....	141
3.3.4 List (Danh sách liên kết).....	142
3.3.5 Stack (Ngăn xếp) .....	143
3.3.6 Queue (Hàng đợi) .....	143
3.3.7 Set (Tập hợp).....	144
3.3.8 Multiset (Tập hợp).....	147

3.3.9 Map (Ánh xạ) .....	148
3.3.10 Multimap (Ánh xạ) .....	150
<b>3.4 STL Algorithms (Thư viện thuật toán).....</b>	<b>151</b>
3.4.1 Min, max .....	151
3.4.2 Sắp xếp.....	152
3.4.3 Tìm kiếm nhị phân.....	153
<b>3.5 Lambda Function.....</b>	<b>154</b>
3.5.1 Định nghĩa.....	154
3.5.2 Khái quát .....	155
3.5.3 Cú pháp .....	155
3.5.4 Ví dụ .....	155
<b>CHƯƠNG 4. ĐỀ THI THAM KHẢO.....</b>	<b>161</b>
<b>CHƯƠNG 5. MỘT SỐ PROJECT NHỎ.....</b>	<b>170</b>
<b>TÀI LIỆU THAM KHẢO.....</b>	<b>175</b>



## LỜI NÓI ĐẦU

Cuốn Kỹ thuật lập trình C/C++ được biên soạn nhằm hỗ trợ các bạn sinh viên của Đại học Bách khoa Hà Nội có cuốn tài liệu chất lượng, phục vụ cho việc học Ngôn ngữ lập trình C++, ôn tập và luyện thi, cũng là nền tảng để cho các bạn học các ngôn ngữ lập trình khác như Java, C#, Python...

Cuốn tài liệu này do mình – Nguyễn Văn Long – CTTN Điện tử Truyền thông K63 – phụ trách môn Kỹ thuật lập trình và các thành viên trong Team BCORN cùng nhau tìm hiểu, sưu tầm những vấn đề đặc trưng, những vấn đề khó hiểu của ngôn ngữ lập trình C++ ở nhiều nguồn khác nhau, tập hợp lại ngắn gọn, súc tích, để cho mọi người đọc cuốn này có thể hiểu được. Trong quá trình tìm hiểu, sưu tầm không tránh khỏi những sai sót nên mình mong các bạn đóng góp ý kiến để tài liệu ngày càng hoàn thiện, mọi đóng góp xin vui lòng gửi về:

- Facebook: [fb.com/vanlong5512](https://fb.com/vanlong5512)
- Email: [nguyenvanlong12092000@gmail.com](mailto:nguyenvanlong12092000@gmail.com)

Ngoài ra, các bạn có thể tham gia Group BCORN - HỖ TRỢ HỌC TẬP SINH VIÊN BK của tụi mình để nhận các tài liệu hữu ích nhất.

Lời cuối, chúc mọi người sử dụng tài liệu này hiệu quả và có những kỳ học ở ngôi trường Bách khoa thật thành công.

# CHƯƠNG 1. CÁC CÚ PHÁP C/C++ CƠ BẢN

## 1.1 Các khái niệm cơ bản

### 1.1.1 Biến, hằng

#### 1.1.1.1 Biến

- Giá trị của biến có thể thay đổi bất cứ lúc nào trong suốt quá trình mà chương trình chạy. Biến dùng để lưu trữ dữ liệu mà cần được thay đổi trong quá trình chương trình thực hiện

- Các khai báo biến: <Kiểu dữ liệu> <tên biến>

```
// Ví dụ
int a;
char c;
bool check;
```

- Có thể khai báo nhiều biến có cùng kiểu dữ liệu trên một dòng, được cách nhau bởi một dấu phẩy

```
// Ví dụ:
int a, b, c;
char ch1, ch2;
```

- Khi khai báo biến, có thể khởi tạo giá trị ban đầu cho biến, có 2 cú pháp khởi tạo dữ liệu, một là dùng toán tử =, hai là dùng toán tử ()

```
// Ví dụ:
int a = 5, b = 10;
char ch1('a'), ch2('b');
```

- Quy tắc đặt tên biến, tên hằng, tên hàm:

- + Tên không được có kí tự số ở đầu. Ví dụ: 1st, 3ab...
- + Tên không được có dấu cách
- + Tên không được chứa các toán tử
- + Tên không được chứa các ký tự đặc biệt như @, -, ... (trừ dấu gạch dưới \_)
- + Tên không được trùng với từ khoá
- + Hai biến bất kì trong một hàm tên không được giống nhau
- + Tên biến không trùng với tên hằng

- + Tên biến, hằng không trùng với tên hàm
- Cách đặt tên biến: Có 2 cách đặt tên biến chính
  - + Từ đầu tiên của tên biến viết thường, các từ sau viết hoa chữ cái đầu, có ý nghĩa để gọi nhắc ta về chức năng của tên biến đó. Ví dụ: fullName, totalMark...
  - + Viết thường hết, mỗi từ cách nhau bằng dấu gạch dưới \_. Ví dụ: full\_name, total\_mark...

### 1.1.1.2 Hằng

- Là một giá trị hằng số không cho phép thay đổi trong quá trình chạy chương trình. Như vậy, chúng ta dùng hằng khi không muốn giá trị bị thay đổi trong suốt thời gian chương trình chạy.

- Cú pháp khai báo hằng:

+ Dùng #define:

```
// Ví dụ
#define PI 3.14
#define MAX_PLAYER 4
```

+ Dùng từ khoá const:

```
// Ví dụ
const float PI = 3.14;
const int MAX_PLAYER 4;
```

- Cách đặt tên hằng: Tất cả các kí tự trong tên hằng đều viết hoa, mỗi từ cách nhau bằng dấu gạch dưới \_. Ví dụ: MAX, PI, MAX\_PLAYER,...

### 1.1.1.3 Cú pháp nhập xuất

- Cách nhập xuất trong C++: Nhập xuất cơ bản trong C++ đơn giản hơn C, ta chỉ cần dùng cin, cout với biến ta cần nhập và xuất

+ Xuất:

Cú pháp:

```
cout << [chuỗi] << [tên biến 1] << [tên biến 2]...
```

Ví dụ:

```
cout << "Ho va ten: " << hoTen << "\t Tuoi: " << tuoi << "\n";
```

+ Nhập:

Cú pháp:

```
cin >> [biến 1] >> [biến 2] >> ...;
```

Nếu nhập nhiều biến, sau mỗi biến là dấu backspace hoặc enter

Ví dụ:

```
cin >> x >> y;
```

## 1.1.2 Kiểu dữ liệu trong C/C++

### 1.1.2.1 Kiểu số nguyên

Type	Kích thước	Phạm vi giá trị
char	1 byte	-128 đến 127
unsigned char	1 byte	0 đến 255
int	2 or 4 bytes	-32,768 đến 32,767 hoặc -2,147,483,648 đến 2,147,483,647
unsigned int	2 or 4 bytes	0 đến 65,535 hoặc 0 đến 4,294,967,295
short	2 bytes	-32,768 đến 32,767
unsigned short	2 bytes	0 đến 65,535
long	8 bytes	-9223372036854775808 đến 9223372036854775807
unsigned long	8 bytes	0 đến 18446744073709551615

### 1.1.2.2 Kiểu số thực

Type	Kích thước	Phạm vi giá trị	Độ chính xác
float	4 bytes	1.2E-38 to 3.4E+38	6 chữ số thập phân
double	8 bytes	2.3E-308 to 1.7E+308	15 chữ số thập phân
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 chữ số thập phân

Ví dụ:

```
#include <float.h> // for float, double macros
#include <iostream>
#include <limits.h> // for int, char macros

using namespace std;

int main()
{
    // Displaying ranges with the help of macros
    cout << "char ranges from: " << CHAR_MIN << " to " << CHAR_MAX;
```



```

    cout << "\n\nunsigned char ranges from: " << 0 << " to " << UCHAR_MAX;
    cout << "\n\nshort int ranges from: " << SHRT_MIN << " to " <<
SHRT_MAX;
    cout << "\n\nunsigned short int ranges from: " << 0 << " to " <<
USHRT_MAX;
    cout << "\n\nint ranges from: " << INT_MIN << " to " << INT_MAX;
    cout << "\n\nunsigned int ranges from: " << 0 << " to " << UINT_MAX;
    cout << "\n\nlong int ranges from: " << LONG_MIN << " to " <<
LONG_MAX;
    cout << "\n\nunsigned long int ranges from: " << 0 << " to " <<
ULONG_MAX;
    cout << "\n\nlong long int ranges from: " << LLONG_MIN << " to " <<
LLONG_MAX;
    cout << "\n\nunsigned long long int ranges from : " << 0 << " to "
<< ULLONG_MAX;
    cout << "\n\nfloat ranges from: " << FLT_MIN << " to " << FLT_MAX;
    cout << "\n\nnegative float ranges from: " << -FLT_MIN << " to " <<
-FLT_MAX;
    cout << "\n\ndouble ranges from: " << DBL_MIN << " to " << DBL_MAX;
    cout << "\n\nnegative double ranges from: " << -DBL_MIN << " to " <<
+DBL_MAX;

    return 0;
}

```

Kết quả:

```

char ranges from: -128 to 127
unsigned char ranges from: 0 to 255
short int ranges from: -32768 to 32767
unsigned short int ranges from: 0 to 65535
int ranges from: -2147483648 to 2147483647
unsigned int ranges from: 0 to 4294967295
long int ranges from: -2147483648 to 2147483647
unsigned long int ranges from: 0 to 4294967295
long long int ranges from: -9223372036854775808 to 9223372036854775807
unsigned long long int ranges from: 0 to 18446744073709551615
float ranges from: 1.17549e-38 to 3.40282e+38
negative float ranges from: -1.17549e-38 to -3.40282e+38
double ranges from: 2.22507e-308 to 1.79769e+308
negative double ranges from: -2.22507e-308 to 1.79769e+308

```

### 1.1.2.3 Xâu ký tự C (cstring hay string.h)

- Chuỗi ký tự là một dãy gồm các ký tự hoặc một mảng các ký tự được kết thúc bằng ký tự '\0' (còn được gọi là ký tự NULL trong bảng mã ASCII).

- Khai báo:

+ Khai báo theo mảng: `char tên_biến[độ dài]`

```
// Ví dụ
char fullName[30];
char gender[10];
```

+ Khai báo theo con trỏ: `char *tên_biến`. Muốn lưu dữ liệu khi khai báo kiểu này, ta phải cấp phát động cho nó.

- Các thao tác trên chuỗi ký tự:

+ Nhập xuất chuỗi không khoảng cách, dùng cin bình thường như các biến khác.

+ Nhập xuất chuỗi có khoảng cách:

```
std::cin.getline(char * __s, std::streamsize __n, char __delim = '\\n');
```

Trong đó \_\_s là chuỗi cần nhập, \_\_n là số ký tự tối đa nhập vào, \_\_delim là ký tự kết thúc quá trình nhập chuỗi (mặc định là Enter)

```
// Ví dụ:
int main() {
    char fullName[30];
    cout << "Nhập họ và tên: ";
    cin.getline(fullName, 30);
    return 0;
}
```

- Một số hàm xử lý chuỗi:

+ Cộng chuỗi – Hàm strcat():

+ Xác định độ dài chuỗi – strlen()

+ Copy chuỗi – strcpy()

+ So sánh chuỗi – strcmp()

+ Đổi từ chuỗi ra số, hàm atoi(), atof(), atol()

#### 1.1.2.4 Chuỗi ký tự C++ (string)

- String là một kiểu đặc biệt của container, thiết kế đặc biệt để hoạt động với các chuỗi ký tự.

- Khai báo: #include <string>

- Cú pháp: string tên\_xâu

```
// Ví dụ:
string fullName;
string birthday;
```

- Các thao tác trên chuỗi ký tự:

+ Nhập xuất chuỗi không khoảng cách, dùng cin bình thường như các biến khác.

+ Nhập xuất chuỗi có khoảng cách:

```
std::getline(std::istream & __is, std::string & __str, char __delim = '\\n');
```

Trong đó, \_\_is là stream input, thường là cin, \_\_str là xâu muốn nhập, \_\_delim là kí tự kết thúc xâu

- Iterator: Tương tự như trong container, string cũng hỗ trợ các iterator như begin, end, rbegin, rend với ý nghĩa như trước.

- Trong string bạn có thể sử dụng các toán tử “=” để gán giá trị cho string, hay toán tử “+” để nối hai string với nhau, hay toán tử “==”, “!=” để so sánh. Chức năng này khá giống với xâu ở trong ngôn ngữ pascal.

- Capacity:

- size: trả về độ dài của string
- length: trả về độ dài của string
- clear : xóa string
- empty: return true nếu string rỗng, false nếu ngược lại

- Truy cập đến phần tử:

- operator [chỉ\_số]: lấy kí tự vị trí chỉ\_số của string
- at(chỉ\_số): như trên

- Chỉnh sửa:

- push\_back: chèn kí tự vào sau string
- insert (n,x): chèn x vào string ở trước vị trí thứ n. x có thể là string, char,...
- erase (pos,n): xóa khỏi string “n” kí tự bắt đầu từ vị trí thứ “pos”.
- erase (iterator): xóa khỏi string phần tử ở vị trí iterator.
- replace (pos, size, s1) : thay thế string từ vị trí “pos”, số phần tử thay thế là “size” và thay bằng xâu s1.
- swap (string\_cần\_đổi): đổi giá trị 2 xâu cho nhau.

- String operations:

- c\_str : chuyển xâu từ dạng string trong C++ sang string trong C.
- substr (pos,length): return string được trích ra từ vị trí thứ “pos”, và trích ra “length” kí tự.

### 1.1.3 Các loại toán tử

#### 1.1.3.1 Toán tử số học

Toán tử số học là một loại toán tử trong C++, được sử dụng để thực hiện các phép toán: cộng, trừ, nhân, chia,... trên các giá trị số (biến và hằng). Đây là các toán tử cần sự tham gia của 2 giá trị số nên được phân loại là các toán tử 2 ngôi.

Toán tử	Ý nghĩa
+	Phép toán cộng
-	Phép toán trừ
*	Phép toán nhân
/	Phép toán chia (chia lấy phần nguyên nếu kiểu dữ liệu là nguyên)
%	Chia lấy phần dư với kiểu dữ liệu nguyên

#### 1.1.3.2 Toán tử gán

Toán tử gán là toán tử trong ngôn ngữ C++ được dùng để gán giá trị cho 1 biến trong ngôn ngữ lập trình C++. Bao gồm các toán tử sau:

Toán tử	Viết gọn	Viết đầy đủ
=	a = b	a = b
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

#### 1.1.3.3 Toán tử tăng giảm

Toán tử tăng, giảm là 1 loại toán tử trong C và là các toán tử 1 ngôi, bao gồm 2 toán tử sau:

- Toán tử ++: Tăng giá trị lên 1 đơn vị
- Toán tử --: Giảm giá trị đi 1 đơn vị

So sánh i++ và ++i:

- ++i: tăng giá trị của i lên 1 và trả về giá trị mới đó.

- `i++` cũng tương tự nhưng giá trị trả về là giá trị ban đầu của `i` trước khi được tăng lên 1.

Một điểm đáng lưu ý ở đây, không nên nhầm lẫn là `i++` sẽ trả về giá trị `i` cho phép gán trước khi nó được tăng lên. Phép gán luôn thực hiện sau cùng, nên điều đó là không thể nhé.

Về bản chất chương trình sẽ tạo ra một biến tạm (temp) để lưu giá trị ban đầu của `i` và trả về giá trị đó cho phép gán sau khi phép toán `i++` thực hiện xong.

#### 1.1.3.4 Toán tử quan hệ

Toán tử quan hệ là một loại toán tử trong C++ được dùng để thực hiện các phép kiểm tra mối quan hệ giữa 2 toán hạng. Nếu quan hệ kiểm tra là đúng thì nó trả về giá trị `true` và trả về `false` trong trường hợp ngược lại.

Toán tử	Ý nghĩa	Ví dụ
<code>==</code>	so sánh bằng	<code>7 == 3</code> , cho kết quả bằng 0 (false)
<code>&gt;</code>	so sánh lớn hơn	<code>7 &gt; 3</code> , cho kết quả là 1 (true)
<code>&lt;</code>	so sánh nhỏ hơn	<code>3 &lt; 1</code> , cho kết quả là 0 (false)
<code>!=</code>	so sánh khác	<code>7 != 3</code> cho kết quả là 1 (true)
<code>&gt;=</code>	lớn hơn hoặc bằng	<code>7 &gt;= 3</code> , cho kết quả 1 (true)
<code>&lt;=</code>	nhỏ hơn hoặc bằng	<code>7 &lt;= 3</code> , cho kết quả 0 (false)

#### 1.1.3.5 Toán tử logic

Toán tử logic là một toán tử trong C. Toán tử logic bao gồm các toán tử sau:

- Toán tử `&&`: là toán tử AND, trả về `true` khi và chỉ khi tất cả các toán hạng đều đúng.
- Toán tử `||`: là toán tử OR, trả về `true` khi có ít nhất 1 toán hạng đúng.
- Toán tử `!`: là toán tử NOT, phủ định giá trị của toán hạng.

#### 1.1.3.6 Toán tử thao tác trên bit

- ❖ AND: Toán tử thao tác bit AND lấy 2 toán hạng nhị phân có chiều dài bằng nhau và thực hiện phép toán lý luận AND trên mỗi cặp bit tương ứng bằng cách nhân chúng lại với nhau. Nhờ đó, nếu cả hai bit ở vị trí được so sánh đều là 1, thì bit

hiển thị ở dạng nhị phân sẽ là 1 ( $1 \times 1 = 1$ ); ngược lại thì kết quả sẽ là 0 ( $1 \times 0 = 0$ ).

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Ví dụ:

```
0101 (số thập phân 5)
AND 0011 (số thập phân 3)
= 0001 (số thập phân 1)
```

Trong C, C++, Java, C#, toán tử thao tác bit AND được biểu diễn bằng ký hiệu "&" (dấu và)

- ❖ NOT: Toán tử thao tác bit NOT, hay còn gọi là còn được gọi là toán tử lấy phần bù (complement), là toán tử một ngôi thực hiện phủ định luận lý trên từng bit, tạo thành bù 1 (one's complement) của giá trị nhị phân cho trước. Bit nào là 0 thì sẽ trở thành 1, và 1 sẽ trở thành 0

A	~A
1	0
0	1

Ví dụ:

```
NOT 0111 (số thập phân 7)
= 1000 (số thập phân 8)
```

Trong các ngôn ngữ lập trình C, C++, Java, C#, toán tử thao tác bit NOT được biểu diễn bằng ký hiệu "~" (dấu ngã).

- ❖ OR: Phép toán trên thao tác bit OR lấy hai dãy bit có độ dài bằng nhau và thực hiện phép toán lý luận bao hàm OR trên mỗi cặp bit tương ứng. Kết quả ở mỗi vị trí sẽ là 0 nếu cả hai bit là 0, ngược lại thì kết quả là 1

A	B	A   B
0	0	0
0	1	1
1	0	1

1	1	1
---	---	---

Ví dụ

```
0101 (số thập phân 5)
OR 0011 (số thập phân 3)
= 0111 (số thập phân 7)
```

Trong C, C++, Java, C#, toán tử thao tác bit OR được biểu diễn bằng ký hiệu "|" (vạch đứng).

- ❖ XOR: Phép toán thao tác bit XOR lấy hai dãy bit có cùng độ dài và thực hiện phép toán logic bao hàm XOR trên mỗi cặp bit tương ứng. Kết quả ở mỗi vị trí là 1 chỉ khi bit đầu tiên là 1 hoặc nếu chỉ khi bit thứ hai là 1, nhưng sẽ là 0 nếu cả hai là 0 hoặc cả hai là 1. Ở đây ta thực hiện phép so sánh hai bit, kết quả là 1 nếu hai bit khác nhau và là 0 nếu hai bit giống nhau.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Ví dụ:

```
0101 (số thập phân 5)
XOR 0011 (số thập phân 3)
= 0110 (số thập phân 6)
```

- ❖ Dịch trái

Ký hiệu: <<

Phép dịch trái n bit tương đương với phép nhân cho 2<sup>n</sup>.

Ví dụ:

A	0000 1100
B = A << 2	0011 0000

- ❖ Dịch phải

Ký hiệu: >>

Phép dịch phải n bit tương đương với phép chia cho 2<sup>n</sup>.

Ví dụ:

A	0000 1100
---	-----------

```
B = A >> 2      0000 0011
```

## 1.2 Các cấu trúc điều khiển

### 1.2.1 if...else...

```
if (condition) {
    /* statement_1 */
} else {
    /* statement_2 */
}
```

Nếu *condition* là true, *statement\_1* được thực hiện, nếu không *statement\_1* bị bỏ qua (không thực hiện) và chương trình tiếp tục thực hiện lệnh *statement\_2*.

```
if (condition_1) {
    /* statement_1 */
} else if (condition_2) {
    /* statement_2 */
} else {
    /* statement_3 */
}
```

Ví dụ 1: Viết chương trình giải hệ phương trình bậc nhất 2 ẩn.

```
#include <iostream>

int main()
{
    float a1, b1, c1, a2, b2, c2;
    std::cin >> a1 >> b1 >> c1;
    std::cin >> a2 >> b2 >> c2;
    float D, Dx, Dy;
    D = a1 * b2 - a2 * b1;
    Dx = c1 * b2 - c2 * b1;
    Dy = a1 * c2 - a2 * c1;
    if (D == 0)
    {
        if (Dx + Dy == 0)
            std::cout << "He phuong trinh co vo so nghiem";
        else
            std::cout << "He phuong trinh vo nghiem";
    }
    else
    {
        float x = Dx / D;
        float y = Dy / D;
        std::cout << "Nghiem: x = " << x << ", y = " << y;
    }
    return 0;
}
```

Ví dụ 2: Viết chương trình giải phương trình bậc 2 một ẩn.

```
#include <cmath>
#include <iostream>

int main()
{
    float a, b, c;
    std::cin >> a >> b >> c;
```



```

float delta = b * b - 4 * a * c;
if (delta == 0)
{
    std::cout << "Phuong trinh co nghiem kep: x1 = x2 = " << -b / (2
* a);
}
else if (delta > 0)
{
    std::cout << "Phuong trinh co 2 nghiem: x1 = " << (-b -
sqrt(delta)) / (2 * a)
    << "x2 = " << (b + sqrt(delta)) / (2 * a);
}
else
    std::cout << "Phuong trinh vo nghiem";
return 0;
}

```

### 1.2.2 Vòng lặp while, do...while...

```

while (expression) {
    /* statement */
}
do {
    /* statement */
} while (expression);

```

- Lặp lại statement khi điều kiện expression còn thoả mãn

### 1.2.3 Vòng lặp for

```

for (initialization; condition; increase) {
    /* statement */
}

```

- Lặp lại statement chừng nào condition còn mang giá trị đúng
- initialization: đặt một giá trị ban đầu cho biến điều khiển
- increase: thay đổi giá trị của biến điều khiển

Ví dụ 1: Tính tổng từ 1 đến n

```

#include <iostream>

int main() {
    int n;
    std::cin >> n;
    int sum = 0;
    for (int i = 0; i <= n; ++i)
        sum += i;
    std::cout << "Tong tu 1 den n la: " << sum << '\n';
    return 0;
}

```

Ví dụ 2: Viết chương trình nhập vào số nguyên dương n, kiểm tra số đó là số nguyên tố hay không?

```

#include <cmath>
#include <iostream>

```

```

int main()
{
    int n;
    std::cin >> n;
    if (n <= 1)
        std::cout << n << " không phải số nguyên tố";
    else
    {
        bool res = true;
        for (int i = 2; i < sqrt(n); ++i)
        {
            if (n % i == 0)
            {
                res = false;
                break;
            }
        }
        if (res == true)
            std::cout << n << " là số nguyên tố";
        else
            std::cout << n << " không phải số nguyên tố";
    }
    return 0;
}

```

#### 1.2.4 swich...case

```

switch (expression)
{
case /* constant-expression */:
    /* code */
    break;

default:
    break;
}

```

Kiểm tra *expression* giống case nào thì thực hiện case đó, không thì thực hiện block default

#### 1.2.5 break, continue, goto

##### 1.2.5.1 break

Câu lệnh break trong C++ có hai cách sử dụng như sau:

- Khi gặp câu lệnh break trong một vòng lặp, vòng lặp bị kết thúc ngay lập tức và câu lệnh kế tiếp sau vòng lặp được thực thi.
- Lệnh break có thể được sử dụng để kết thúc một case trong câu lệnh switch.

Nếu bạn sử dụng vòng lặp lồng nhau, câu lệnh break sẽ dừng việc thực hiện vòng lặp trong cùng và bắt đầu thực hiện câu lệnh kế tiếp sau vòng lặp trong cùng.

Ví dụ:

```
#include <iostream>
```

```
using namespace std;

int main () {
    int a = 10;
    while (a < 20) {
        cout << "Gia tri cua a: " << a << endl;
        a++;
        if (a > 15) {
            /* ket thuc vong lap khi a lon hon 15 */
            break;
        }
    }

    return 0;
}
```

Kết quả:

```
Gia tri cua a: 10
Gia tri cua a: 11
Gia tri cua a: 12
Gia tri cua a: 13
Gia tri cua a: 14
Gia tri cua a: 15
```

### 1.2.5.2 continue

Câu lệnh continue trong C++ hoạt động giống như câu lệnh break. Thay vì buộc kết thúc vòng lặp, nó buộc trở về kiểm tra điều kiện để thực hiện vòng lặp tiếp theo và bỏ qua các lệnh bên trong vòng lặp hiện tại sau lệnh continue.

Đối với vòng lặp for, câu lệnh continue làm cho điều khiển chương trình tăng hoặc giảm biến đếm của vòng lặp. Đối với vòng lặp while và do-while, câu lệnh continue làm cho điều khiển chương trình quay về đầu vòng lặp và kiểm tra điều kiện của vòng lặp.

Ví dụ:

```
#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    do
    {
        if (a == 15)
        {
```

```

        // quay ve do khi a = 15 (bo qua lenh print)
        a = a + 1;
        continue;
    }
    cout << "Gia tri cua a: " << a << endl;
    a++;
} while (a < 20);
return 0;
}

```

Kết quả:

```

Gia tri cua a: 10
Gia tri cua a: 11
Gia tri cua a: 12
Gia tri cua a: 13
Gia tri cua a: 14
Gia tri cua a: 16
Gia tri cua a: 17
Gia tri cua a: 18
Gia tri cua a: 19

```

### 1.2.5.3 goto

Câu lệnh goto trong C++ cung cấp một bước nhảy vô điều kiện từ 'goto' đến một câu lệnh có nhãn trong cùng một hàm.

Chú ý: Việc sử dụng câu lệnh goto không được khuyến khích sử dụng trong bất kỳ ngôn ngữ lập trình nào vì nó rất khó để theo dõi luồng điều khiển của chương trình, làm cho chương trình khó hiểu và khó bảo trì. Bất kỳ chương trình nào sử dụng goto đều có thể được viết lại theo cách bình thường.

Ví dụ:

```

#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    TEST:

```

```

do
{
    if (a == 15)
    {
        // quay ve do khi a = 15 (bo qua lenh print)
        a = a + 1;
        goto TEST;
    }
    cout << "Gia tri cua a: " << a << endl;
    a++;
} while (a < 20);
return 0;
}

```

Kết quả:

```

Gia tri cua a: 10
Gia tri cua a: 11
Gia tri cua a: 12
Gia tri cua a: 13
Gia tri cua a: 14
Gia tri cua a: 16
Gia tri cua a: 17
Gia tri cua a: 18
Gia tri cua a: 19

```

## 1.3 Hàm

```
type name(argument1, argument2,...) statement
```

Chú ý: Truyền tham chiếu, truyền tham trị

### 1.3.1 Truyền tham trị

Truyền tham trị là truyền giá trị của biến (không phải là địa chỉ ô nhớ), khi đó phương thức sẽ tự động tạo ra một địa chỉ ô nhớ mới để lưu trữ giá trị này, do đó nó chỉ được thay đổi trong phương thức hiện hành và giá trị của biến không bị thay đổi bên ngoài phương thức hiện hành.

Khi truyền đối số kiểu tham trị, chương trình biên dịch sẽ copy giá trị của đối số để gán cho tham số của hàm và không tác động trực tiếp đến biến số truyền vào.

### 1.3.2 Truyền tham chiếu

Truyền tham chiếu là truyền địa chỉ ô nhớ của biến, do đó khi thay đổi giá trị của biến bên trong phương thức thì giá trị của biến cũng bị thay đổi bên ngoài phương thức.

Đối với truyền tham chiếu, địa chỉ của biến sẽ được truyền vào hàm, chính vì thế việc thực hiện các phép tính toán trong hàm đối với biến này đều thực hiện trực tiếp trên địa chỉ chứa giá trị của biến đó.

Cú pháp:

- Đối với C: Thông qua con trỏ

```
//Ví dụ
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int a = 4, b = 5;
    swap(&a, &b)
}
```

- Đối với C++: Dùng & trước tên mỗi biến

```
//Ví dụ
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int a = 4, b = 5;
    swap(a, b)
}
```

## 1.4 Con trỏ

### 1.4.1 Tổng quan

Con trỏ trong C++ cũng chỉ là biến, cũng có thể khai báo, khởi tạo và lưu trữ giá trị và có địa chỉ của riêng nó. Nhưng biến con trỏ không lưu giá trị bình thường, nó là biến trỏ tới 1 địa chỉ khác, tức mang giá trị là 1 địa chỉ.

Chúng ta cùng thống nhất 1 số khái niệm khi làm việc với con trỏ:

- Giá trị của con trỏ: địa chỉ mà con trỏ trỏ đến.
- Địa chỉ của con trỏ: địa chỉ của bản thân biến con trỏ đó.
- Giá trị của biến nơi con trỏ đang trỏ tới.
- Địa chỉ của biến nơi con trỏ đang trỏ tới = giá trị của con trỏ.

Khai báo: `type * pointer_name;` trong đó `type` là kiểu dữ liệu được trỏ tới

```
// Ví dụ:
int *p_i; // khai báo con trỏ để trỏ tới biến kiểu nguyên
int *p, val; // khai báo con trỏ p kiểu int, biến val (không phải con trỏ) kiểu int
float *p_f; // khai báo con trỏ để trỏ tới biến kiểu thực
char *p_char; // khai báo con trỏ để trỏ tới biến kiểu ký tự
void *p_v; // con trỏ kiểu void (không kiểu)
```

#### 1.4.1.1 Toán tử lấy địa chỉ (&)

Nó được dùng như là một tiền tố của biến và có thể được dịch là "địa chỉ của", vì vậy `&variable1` có thể được đọc là "địa chỉ của `variable1`".

#### 1.4.1.2 Toán tử tham chiếu (\*)

Nó chỉ ra rằng cái cần được tính toán là nội dung được trỏ bởi biểu thức được coi như là một địa chỉ. Nó có thể được dịch là "giá trị được trỏ bởi"..

`*mypointer` được đọc là "giá trị được trỏ bởi `mypointer`"

Gán giá trị cho con trỏ: Sau khi khai báo con trỏ, muốn sử dụng được nó, ta phải gán cho nó giá trị là địa chỉ của biến mà ta muốn con trỏ của mình đại diện. Ví dụ:

```
int *p, value;
value = 5;
p = &value; // khởi tạo giá trị cho con trỏ p là địa chỉ của value
```

Nên khởi tạo con trỏ bằng địa chỉ NULL hoặc nullptr nếu chưa cần dùng theo cách sau: `int *p = NULL` hoặc `int *p = nullptr`. Khi đó con trỏ p nó không chỉ vào thẳng nào cả.

Chú ý: `int *p` chỉ là cho ta biết p là kiểu con trỏ trỏ tới giá trị nguyên (dấu \* cho ta biết đây là kiểu con trỏ, không có ý nghĩa toán học), còn \*p (toán tử tham chiếu) là để lấy giá trị mà con trỏ trỏ tới.

```
//Ví dụ:
int a = 5;
int *ptr // * ở đây chỉ cho ta biết ptr là kiểu con trỏ
ptr = &a;
(*ptr)++; // * ở đây là lấy giá trị mà con trỏ trỏ tới, ở đây là giá trị của a
```

## 1.4.2 Các phép toán trên con trỏ

### 1.4.2.1 Phép gán

Phép gán đối với con trỏ thì tham khảo phần khởi tạo nhưng có 1 vài yếu tố xâu đây:

- + Tất cả các loại con trỏ đều có phép gán
- + Phép gán với con trỏ yêu cầu vế trái là 1 con trỏ và vế phải là 1 địa chỉ
- + Phép gán yêu cầu sự tương xứng về kiểu dữ liệu, nếu ko tương xứng chúng ta phải ép kiểu. Ví dụ: `p = (int *)2000` (Do 2000 là hằng số nguyên nên để gán được cho con trỏ p kiểu dữ liệu `int *` thì phải ép kiểu rồi mới thực hiện phép gán)
- + Phép gán với 1 con trỏ kiểu `void` ko cần thiết phải tương xứng hoàn toàn về kiểu dữ liệu, `void*` có thể tương ứng với tất cả, thậm chí là vượt cấp (vượt hẳn 2 cấp).

```
void *ptr;
int a;
void **ptr2;
ptr = &a; //OK
ptr = &ptr2; //OK
```




#### 1.4.2.2 Phép so sánh

- Phép so sánh ngang bằng dùng để kiểm tra 2 con trỏ có trỏ vào cùng 1 vùng nhớ hay không, hoặc kiểm tra 1 con trỏ có phải là đang trỏ vào NULL hay không (trong trường hợp cấp phát động, mở file, mở resource,.....)

- Phép so sánh lớn hơn nhỏ hơn: > , < , >= , <= sử dụng để kiểm tra về độ thấp cao giữa 2 địa chỉ . Con trỏ nào nhỏ hơn thì trỏ vào địa chỉ thấp hơn.

- Được quyền so sánh mọi con trỏ với 0, vì 0 chính là NULL. Ngoài ra thì khi so sánh 2 con trỏ hoặc con trỏ với 1 địa chỉ xác định (số nguyên) cần có sự tương xứng về kiểu dữ liệu



```
int main()
{
    int a=197,*p=&a;
    double b=0,*x=&b;

    // so sánh 2 con trỏ
    (int)p==(int)x;
    p==(int *)x;
    (double*)p==x;
    (void*)p==(void*)x;
    p==(void*)x;
    (float*)p==(float*)x;

    //so sánh con trỏ với số nguyên
    p==(int*)9999;
    int(p)==9999;
}
```

- Con trỏ void có thể đem ra so sánh với tất cả các con trỏ khác

#### 1.4.2.3 Phép cộng trừ và phép tăng giảm: + += - -= ++ --

Bản chất của việc tăng/giảm con trỏ p đi 1 đơn vị là cho p trỏ đến ô nhớ bên cạnh phía dưới/trên.

Chú ý:

+ Khi tăng giảm con trỏ p đi 1 đơn vị không có nghĩa là trỏ sang byte bên cạnh

- + Việc tăng giảm con trỏ đi 1 đơn vị phụ thuộc vào kiểu dữ liệu và nó trỏ đến, quy tắc là  $p+1 \gg>$  giá trị chứa trong  $p + \text{sizeof}(\text{kiểu dữ liệu của biến mà } p \text{ trỏ đến})$
- + Không có phép tăng giảm trên con trỏ void
- + Không có phép tăng giảm trên con trỏ hàm
- + Không có phép cộng 2 con trỏ với nhau
- + Phép trừ 2 con trỏ trả về độ lệch pha giữa 2 con trỏ

### 1.4.3 Con trỏ hằng (pointer to const) và hằng con trỏ (const pointer)

#### 1.4.3.1 Con trỏ hằng

Con trỏ hằng là con trỏ trỏ đến vùng dữ liệu hằng. Ta ko thể thay đổi giá trị mà nó đang trỏ đến. Nhưng có thể thực hiện tăng giảm địa chỉ con trỏ hay cho nó trỏ đi nơi khác.

```
void main()
{
    int x = 5;
    int y = 10;
    int *q;
    q = &y;
    const int* p = &x;
    *p = 6; // lỗi biên dịch
    p = q; // OK
}
```

#### 1.4.3.2 Hằng con trỏ

Hằng con trỏ là 1 con trỏ nếu trỏ vào 1 ô nhớ nào đó. Thì nó sẽ nằm “dính chặt” vào ô nhớ đó. Ta không thể cho nó trỏ đi nơi khác và tăng giảm địa chỉ của con trỏ. Nhưng ta có thể thay đổi được giá trị mà nó trỏ đến.

```
void main()
{
    int x = 5;
    int y = 10;
    int *q;
    q = &y;
    int* const p = &x;
}
```

```

p = &q; // ERROR
*p = 6; // OK
}

```

#### 1.4.4 Mỗi quan hệ giữa mảng và con trỏ

Khi ta khai báo mảng thì tương đương với xin cấp phát 1 vùng nhớ có kích thước như bạn khai báo và khai báo ra 1 hằng con trỏ trỏ vào đầu vùng nhớ đó

Ví dụ: `int a[100];`

+ Có thể coi a là 1 hằng con trỏ trỏ vào phần tử thứ 0 của mảng, a mang đầy đủ tính chất của 1 hằng con trỏ nhưng có thêm 1 số khác biệt nhỏ (ví dụ khi dùng `sizeof`)

+ Các phép toán nhằm làm a trỏ tới vùng khác (thay đổi giá trị của a) là ko thể (`++` `--` `=`)

+ a tương đương với `&a[0]`

+ `a+i` tương đương với `&a[i]`

+ `*a` tương đương với `a[0]`

+ `*(a+i)` tương đương với `a[i]`

Chú ý: trình biên dịch luôn hiểu `a[i]` là `*(a+i)`

Ví dụ:

```

#include <stdio.h>
#include <conio.h>

void main()
{
    int a[100]={0,1,2,3,4,5,6};
    printf("%d",2[a]); //in ra 2, tại sao vậy ?

    getch();
}

```

Chắc chắn lúc nhìn thấy `2[a]` ko ít người sẽ thấy là lạ, nghĩ nó là lỗi,..... Nhưng không lỗi đâu, kết quả phần trên hoàn toàn đúng. Vì `2[a]` trình biên dịch sẽ hiểu là `*(2+a)`. `*(2+a)` hoàn toàn tương đương với `*(a+2)` mà `*(a+2)` chính là `a[2]`. Vậy `2[a]` cũng đơn giản là `a[2]`.

### 1.4.5 Bản chất của chuỗi ký tự kiểu C

#### 1.4.5.1 Một số sai lầm khi làm việc với chuỗi ký tự

- Chưa có bộ nhớ đã sử dụng như đúng rồi

```
char *xau;
gets(xau);
// vẫn biên dịch được
// nhưng khi chạy sẽ sinh ra lỗi run-time
// ở 1 số trình biên dịch cũ bắt ngày xưa thì có thể ko bị lỗi
// nhưng sai thì vẫn là sai, code này sai thuộc loại chưa cấp phát
```

- Thay đổi giá trị của một hằng

```
char *xau="BCORN - Ho tro sinh vien Bach khoa";
xau[6]='A'; // vẫn biên dịch được
//nhưng khi chạy sẽ sinh ra lỗi run-time
// lỗi này là lỗi cố tình thay đổi giá trị của 1 hằng
```

Nguyên nhân sâu xa của vấn đề như sau: Khi khai báo `char *xau="BCORN - Ho tro sinh vien Bach khoa";` thì bản chất là

- + Trong vùng nhớ data của chương trình sẽ có 1 hằng chuỗi "BCORN – Ho tro sinh vien Bach khoa", đã là hằng thì ko thể bị thay đổi

- + Cho con trỏ `xau` trỏ đến đầu của vùng nhớ đó.

Câu lệnh tiếp theo `xau[6]='A';` cố tình thay đổi giá trị của hằng, rõ ràng là sinh ra lỗi rồi

- Cố tình thay đổi giá trị của hằng con trỏ

```
char xau[100];
xau="Nguyen Van Long"; // không biên dịch được
// vì phép toán trên có nghĩa là khai báo 1 chuỗi "Nguyen Van Long" trong vùng nhớ code
// rồi sau đó cho hằng con trỏ xau trỏ vào đó
// rất tiếc xau là hằng con trỏ nên ko thể trỏ đi đâu khác được
// ngoài vị trí đã được khởi tạo trong câu lệnh khai báo
```

Chú ý `char xau[100] = "Nguyen Van Long";` hoặc `char xau[100] = {0};` thì hoàn toàn hợp lệ

- Dùng phép toán so sánh để so sánh nội dung 2 xâu

```
void main()
{
    char xau[100] = "quangxeng";
    if (xau == "quangxeng")
        // code này ko sai về ngữ pháp, ko sinh ra lỗi runtime
        // nhưng mang lại kết quả ko như người dùng mong muốn
        // vì như ở trên ta có
        // Phép so sánh ngang bằng dùng để kiểm tra 2 con trỏ có trỏ vào cùng
        // 1 vùng nhớ hay không,
        // hoặc kiểm tra 1 con trỏ có phải là đang trỏ vào NULL hay không
        // (trong trường hợp cấp phát động, mở file, mở resource,.....)
        // chứ ko phải là phép so sánh nội dung của xâu
        // để so sánh nội dung của xâu ta phải dùng những hàm strcmp (string
        // compare) hoặc strcmp
        // hoặc những hàm bạn tự định nghĩa
}
```

#### 1.4.6 Cấp phát bộ nhớ động

Bộ nhớ là 1 tài nguyên quan trọng trong hệ thống , được phân phối điều phối phức tạp. Ở đây, chúng ta sẽ xét tương quan như sau cho dễ hiểu: đất là 1 tài nguyên quý giá trong thành phố và được điều phối bởi sở tài nguyên môi trường.

Bộ nhớ quan trọng như thế ko thể tùy tiện sử dụng, vì trong máy có nhiều tiến trình cần bộ nhớ, tùy tiện sử dụng thì lấy đâu ra mà dùng ? Ai cũng cần có đất, cứ tùy tiện lấy gì cần gì sở quản lý, như thế sẽ có người không có đất mà dùng ?

Cấp phát bộ nhớ hoạt động như thế nào: Bản chất là trao quyền sử dụng. Hôm nay 1 người (1 con trỏ) xin cấp phát (malloc, new) 1 vùng đất, anh ta được trao quyền sử dụng 1 vùng đất đó.

Khi này :

+ Có thể trên mảnh đất này đang trống rỗng (toàn null) nhưng cũng có thể trên đó đầy cỏ rác, hoa màu hoặc nhà cửa do người trước đó sử dụng vẫn còn. (điều này đối với anh ta mà nói nó hoàn toàn là ngẫu nhiên, ko ai có thể khẳng định được cả) => Khi ta xin cấp phát xong thì vùng nhớ đó hoàn toàn có giá trị ngẫu nhiên

+ Nếu như khi anh ta xin cấp phát mà có đứt lốt để ra thêm điều kiện hoặc trước đó anh ta có thuê thêm 1 người quét dọn xây sửa thì anh sẽ có được 1 mảnh đất có sẵn cái anh ta muốn

Anh ta dùng xong đất rồi, ko cần nữa, đem trả quyền sử dụng mảnh đất lại cho khô chủ.

Giải bộ nhớ hoạt động như thế nào: bản chất là trả lại quyền sử dụng, khi này:

+ Sở sẽ thu lại quyền sử dụng đất của anh, thế là đủ rồi, sở ko cần phải động chạm sửa chữa lại gì vùng đất đó. Nên khi đó dù vùng đất đó đã được giải phóng nhưng mà hoa màu của a ta trên đó thì vẫn còn, ko có biến mất được. Điều này giải thích tại sao khi mà trong 1 số hệ điều hành đời cổ, nhiều bạn thử giải phóng bộ nhớ rồi, delete rồi, free rồi mà khi in thử ra vẫn thấy kết quả như thế. Đơn giản thôi, anh ta trả lại mảnh đất đó, nhưng chưa có ai thuê nó cả, chưa có ai sử dụng nên hôm sau ra xem thử thì nó vẫn như hôm trước khi a ta trả lại

Nếu không giải phóng bộ nhớ thì sao? Nó sẽ gây ra memory leak. Như vậy sau vài lần chạy sẽ ko còn memory để cấp phát nữa.

```
pointer = new type; //Cấp phát bộ nhớ cho 1 phần tử
```

hoặc

```
pointer = new type[elements]; //Cấp phát bộ nhớ cho một mảng nhiều phần tử
```

Giải phóng bộ nhớ:

```
delete pointer; //Giải phóng bộ nhớ cấp phát cho 1 phần tử
```

hoặc

```
delete[] pointer; //Giải phóng bộ nhớ cấp phát cho nhiều phần tử
```

### 1.4.7 Con trỏ hàm

#### 1.4.7.1 Con trỏ hàm

Con trỏ hàm là một biến lưu trữ địa chỉ của một hàm, thông qua biến đó, có thể gọi hàm mà nó trỏ tới. Điều này rất thuận tiện khi muốn định nghĩa các chức năng khác nhau cho một nhóm các đối tượng khá giống nhau.

Cú pháp: <kiểu tra` về> (\*<tên con trỏ>)(<danh sách đối số>;

```

void thefunc(int a)
{
    // DO SOMETHING
}

int main()
{
    // ...
    void (*func)(int);
    func = &thefunc;
    // ...
    func(1);
    // OR
    (*func)(1);
}

```

#### 1.4.7.2 Hằng con trỏ hàm

Khái niệm hằng con trỏ hàm cũng gần gần giống như khái niệm hằng con trỏ với mảng 1 chiều, Khi bạn khai báo 1 hàm, thì tên của hàm chính là 1 hằng con trỏ hàm, con trỏ này trỏ cố định vào vùng nhớ của hàm.

```

(int)p==int(main);
p==(int*)main;
(int(*)())p==main;
p==(void*)main;

```

Chúng ta thấy đó, chúng ta khai báo ra 1 hàm main. vậy rõ ràng ta có 1 hằng con trỏ main, là 1 hằng thì ta hoàn toàn có thể sử dụng để so sánh rồi

#### 1.4.7.3 Ứng dụng của con trỏ hàm

- Trường hợp đơn giản tất cả chúng ta đều sử dụng không ít lần rồi, nhưng vẫn không hiểu, không biết là mình dùng, đó là `cout<<endl;`

`endl` được định nghĩa như thế này:

```

ostream& endl (ostream& os)
{
    os.push('\n');
    return os;
}

```

Toán tử << có 1 hàm overload như này:

```
friend ostream& operator<<(ostream &os, ostream& (*p)(ostream&))
{
    return p(os);
}
```

- Sử dụng trong các hàm mẫu, lớp mẫu, có tính tùy chọn cao:

```
#include <iostream>
using namespace std;

void sort(double *arr, int size, bool (*compare)(double, double))
{
    for (int i = 0; i < size; ++i)
    {
        for (int j = i + 1; j < size; ++j)
        {
            if (compare(arr[i], arr[j]) == true)
            {
                swap(arr[i], arr[j]);
            }
        }
    }
}

bool compare_less(double a, double b)
{
    return a < b;
}

bool compare_big(double a, double b)
{
    return a > b;
}

int main()
{
    double a[100] = {1., 2., 3., 4., 6., 5.};
    int n = 6;
    sort(a, n, compare_less);
}
```



```

for (int i = 0; i < n; i++)
    cout << a[i] << " ";
cout << endl;

double b[100] = {1, 2, 6, 3, 5, 4};
int m = 6;
sort(b, m, compare_big);
for (int i = 0; i < n; i++)
    cout << b[i] << " ";

return 0;
}

```

#### 1.4.7.4 *std::function*

C++11 cung cấp một cách thay thế cho việc sử dụng con trỏ hàm bằng cách sử dụng kiểu dữ liệu `std::function` thuộc thư viện `<functional>` trong namespace `std`.

```

void sort(double *arr, int size, std::function<bool(double, double)>
compare)
{
    for (int i = 0; i < size; ++i)
    {
        for (int j = i + 1; j < size; ++j)
        {
            if (compare(arr[i], arr[j]) == true)
            {
                swap(arr[i], arr[j]);
            }
        }
    }
}

bool compare_less(double a, double b)
{
    return a < b;
}

bool compare_big(double a, double b)
{

```

```

    return a > b;
}

int main()
{
    double a[100] = {1., 2., 3., 4., 6., 5.};
    int n = 6;
    sort(a, n, compare_less);
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[100] = {1, 2, 6, 3, 5, 4};
    int m = 6;
    sort(b, m, compare_big);
    for (int i = 0; i < n; i++)
        cout << b[i] << " ";

    return 0;
}

```

### 1.4.8 Con trỏ với hàm

#### 1.4.8.1 Giải thích nguyên tắc truyền đối số trong C/C++

- Hàm trong C ko hề có tham biến, hàm trong C đều hoạt động theo nguyên tắc sau: Khi gọi hàm, 1 bản sao của tham số được tạo ra (cấp phát vùng nhớ mới, copy giá trị sang. quá trình này gọi là shadow copy, là 1 yếu tố cần quan tâm, 1 C/C++ Developer đừng bao giờ quên điều này), và hàm sẽ làm việc với bản sao này (trong C++ nó sẽ dùng hàm tạo sao chép để tiến hành quá trình shadow copy này)

- Vậy khi làm việc với con trỏ thì hàm làm thế nào, hàm vẫn cứ làm theo nguyên tắc 1 và 1 bản sao của con trỏ được tạo ra, và hàm làm việc với bản sao hàm, và trước khi gọi hàm con trỏ trỏ vào đâu thì nó vẫn được trỏ vào đấy. Chứng minh:

```

#include <stdio.h>
#include <conio.h>

int ham(int *a)
{

```

```

    *a=2;

    a++;

}

void main()
{
    int *a;
    printf("Truoc : %x",a); //trước và sau khi gọi hàm
    ham(a);                //con trỏ a trỏ vào đâu
    printf("Sau %x",a);     // thì nó vẫn trỏ vào đó
    getch();
}

```

- Vậy tại sao lại có sự thay đổi và tại sao lại sử dụng con trỏ trong hàm? Con trỏ ko thay đổi thì cái gì thay đổi được? Các bạn chú ý nhé, giá trị nằm trong vùng nhớ trỏ đến thay đổi. Vâng đúng thế đấy bạn à, do biến của ta nằm trong vùng nhớ được trỏ đến nên nó được thay đổi:

```

#include <stdio.h>
#include <conio.h>

int ham(int *a)
{
    *a=2; // làm việc với địa chỉ nhận được
}

void main()
{
    int n;
    ham(&n); // truyền địa chỉ của n vào đây
    // do đó sau hàm này n =2
    getch();
}

```

- Một lỗi các bạn hay mắc phải đó là trong hàm chúng ta cấp phát bộ nhớ rồi cho bản sao đang làm việc trỏ đến. ra khỏi hàm rồi thì x của ta vẫn chưa có trỏ vào bộ nhớ nào cả. Ví dụ:

```

#include <stdio.h>
#include <conio.h>

void nhap(int *a,int n)

```

```

{
    //a=new int[n]; //sai lầm
    a=(int*)malloc(n * sizeof(int)); //sai lầm
    for(int i=0;i<n;i++)
        cin>>a[i];
}

void main()
{
    int *x;
    int n=6;
    nhap(x,n);
    //xuat
    delete[] x; // sản sinh ra lỗi run-time , tung là 1 exception, do x
    chưa trở vào đâu mà đòi giải phóng
}

```

- Cách giải quyết vấn đề trên như thế nào, ta có 2 cách để thay đổi giá trị của một con trỏ qua hàm

+ Cách 1. Dùng tham chiếu trong C++

```

void ham(int *&a)
{
    a=new int[100];
}

void ham(int **&a)
{
    a=new int*[100];
}

```

+ Cách 2. Up level của \* dùng con trỏ cấp cao hơn của con trỏ hiện tại

```

#include <stdio.h>
#include <conio.h>

void ham(int **a)
{
    *a=(int*)malloc(100*sizeof(int));
    //a[0]=(int*)malloc(100*sizeof(int));
    // 2 cách này như này
}

```

```
void main()
{
    int *a;
    ham(&a);

    free(a);
}
```

### 1.4.9 Con trỏ đa cấp

#### 1.4.9.1 Giới thiệu

- Ta có thể tạm hiểu đó là những con trỏ có dạng 2 hoặc nhiều \*.

```
int **a; // cấp 2
char ***b; // cấp 3
int *****a; // cấp ??
```

Phép toán trên con trỏ cấp n ( $n > 1$  và con trỏ cấp 2 thuần túy như trong ví dụ vừa khai báo trên) tương tự như với con trỏ cấp 1 tương ứng

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int **a=NULL;
    printf("%x\n",a); //0x0
    a++;
    printf("%x",a); //0x4
    getch();
}
```

#### 1.4.9.2 Con trỏ đa cấp dùng để làm gì

- Con trỏ đa cấp thường được dùng trong trường hợp cần thay đổi giá trị của 1 con trỏ cấp thấp hơn khi ra khỏi hàm.

- Con trỏ cấp 2 còn được dùng như là "con trỏ" trỏ tới một "con trỏ", có thể dùng để xử lý 1 matrix 2 chiều

- Con trỏ cấp 3 còn được dùng như là con trỏ trỏ tới một "con trỏ", mà con trỏ này đang trỏ tiếp tới 1 con trỏ khác, có thể dùng như matrix 3 chiều

## 1.5 Kiểu dữ liệu người dùng tự định nghĩa:

### 1.5.1 struct

#### 1.5.1.1 Giới thiệu

```
struct model_name {
    type1 element1;
    type2 element2;
    type3 element3;
    ...
} object_name;
```

Để truy cập các biến của `model_name`, ta dùng toán tử “.”

\* Con trỏ trỏ đến cấu trúc:

Biểu thức	Mô tả	Tương đương với
<code>object_name.element1</code>	Phần tử <code>element1</code> của cấu trúc <code>object_name</code>	
<code>object_name-&gt;element1</code>	Phần tử <code>element1</code> của của cấu trúc được trỏ bởi <code>object_name</code>	<code>(*object_name).element1</code>
<code>*object_name.element1</code>	Giá trị được trỏ bởi phần tử <code>element1</code> của cấu trúc <code>object_name</code>	<code>*(object_name.element1)</code>

Ví dụ: Viết chương trình quản lý điểm trung bình của học sinh. Sắp xếp học sinh theo thứ tự điểm giảm dần. In ra danh sách trước và sau khi sắp xếp

```
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;

struct Student
```

```
{
    string fullName;
    int age;
    float mark;
};

Student list[100];
int n;

void input()
{
    cout << "So hoc sinh: ";
    cin >> n;

    for (int i = 0; i < n; ++i)
    {
        cout << "Hoc sinh thu " << i + 1 << ":\n";
        cin.ignore(1);
        cout << "\tHo va ten: ";
        getline(cin, list[i].fullName);
        cout << "\tTuoi: ";
        cin >> list[i].age;
        cout << "\tDiem trung binh: ";
        cin >> list[i].mark;
        cout << endl;
    }
}

void sort()
{
    sort(list, list + n, [](Student &a, Student &b) { return a.mark >
b.mark; });
}

void show()
{
    for (int i = 0; i < n; ++i)
    {
        cout << "Hoc sinh thu " << i + 1 << ":\n";
```

```

        cout << "\tHo va ten: " << list[i].fullName << "\n";
        cout << "\tTuoi: " << list[i].age << "\n";
        cout << "\tDiem trung binh: " << list[i].mark << "\n";
        cout << endl;
    }
}

int main()
{
    input();
    cout << "Truoc sap xep: \n";
    show();
    cout << "Sau sap xep: \n";
    sort();
    show();
    return 0;
}

```

#### 1.5.1.2 Tổ chức bộ nhớ trong struct

Trước hết, hãy xem ví dụ sau:

```

#include <algorithm>
#include <iostream>

using namespace std;

struct Point
{
    int x;
    int y;
    int z;
};

int main()
{
    Point p;
    cout << (int *)&p << endl;
    cout << (int *)&p.x << endl;
    cout << (int *)&p.y << endl;
}

```



```
cout << (int *) &p.z << endl;
return 0;
}
```

Kết quả:

```
0x61fde4
0x61fde4
0x61fde8
0x61fdec
```

Có thể thấy địa chỉ của biến p chính là địa chỉ của biến p.x và 3 biến p.x, p.y, p.z được nằm trên các ô nhớ liên tiếp nhau. Do đó có thể nhận xét rằng các biến bên trong struct sẽ được nằm cạnh nhau trên bộ nhớ và địa chỉ của struct chính là địa chỉ của biến đầu tiên.

Để tính kích thước của một struct, bạn cần ghi nhớ 3 quy tắc:

- Địa chỉ của struct giả sử bắt đầu từ 0
- Biến được khai báo sau sẽ luôn có địa chỉ lớn hơn biến được khai báo trước
- Địa chỉ của 1 biến bên trong struct luôn chia hết cho kích thước của biến đó

Ví dụ:

```
#include <iostream>

using namespace std;

struct MyStruct
{
    char a;
    short b;
    short c;
    char d;
    short e;
    int f;
};

int main()
{
    cout << sizeof(MyStruct);
}
```

```
return 0;
}
```

Kết quả:

```
16
```

Giải thích:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	empty	b		c		d	empty	e		empty	empty	f			

- Biến a là biến đầu tiên nên sẽ có địa chỉ là 0
- Biến b có địa chỉ lớn hơn biến a, và phải chia hết cho 2, nên địa chỉ của biến b là 2
- Biến c có địa chỉ lớn hơn biến b, và phải chia hết cho 2, nên địa chỉ của biến c là 4
- Biến d có địa chỉ lớn hơn biến c, và phải chia hết cho 1, nên địa chỉ của biến d là 6
- Biến e có địa chỉ lớn hơn biến d, và phải chia hết cho 2, nên địa chỉ của biến e là 8
- Biến f có địa chỉ lớn hơn biến e, và phải chia hết cho 4 nên địa chỉ của biến f là 12

## 1.5.2 enum

### 1.5.2.1 Định nghĩa

Enum là một tập hợp các giá trị có thể có của 1 thuộc tính, chẳng hạn Giới tính thì có thể Nam, Nữ (nay thì có thêm Gay, Less), hay tình trạng hôn nhân của 1 người thì có thể là Single, Engaged, Complicated, Married. Kiểu dữ liệu mặc định của Enum là int, phần tử đầu tiên có giá trị là 0 và các phần tử tiếp theo có giá trị mặc định tăng lên 1.

Tác dụng của Enum:

- Giới hạn các giá trị mà 1 thuộc tính có thể có (như trên thì MaritalStatus chỉ có 4 giá trị khác nhau)
- Enum giúp chúng ta dễ dàng nhận ra các giá trị có thể có của 1 thuộc tính, giúp dễ dàng đọc hiểu code

### 1.5.2.2 Cú pháp

```
enum ten_enum {trang_thai_1: gia_tri_1, trang_thai_2: gia_tri_2,...}
```

Ví dụ:

```
#include<stdio.h>
```

```
enum dayOfWeek{Mon=2, Tue=3, Wed=4, Thu=5, Fri=6, Sat=7, Sun=8};

int main()
{
    enum dayOfWeek day;
    day = Wed;

    printf("Wednesday: %d \n",day);

    printf("All day of week: %d - %d - %d - %d - %d - %d - %d", Mon, Tue,
Wed, Thu, Fri, Sat, Sun);
    return 0;
}
```

Trường hợp bạn không truyền giá trị cho các trạng thái trong enum thì nó sẽ tự nhận các giá trị tăng dần từ 0. Hoặc tăng dần theo giá trị của trạng thái trước đó.

Lưu ý

- Các trạng thái trong một enum có thể có giá trị bằng nhau.
- Các trạng thái trong enum chỉ nhận giá trị là kiểu integer.
- Trong cùng một phạm vi (scope), thì 2 enum không thể có trạng thái trùng tên.

### 1.5.3 typedef

Dùng để định nghĩa một tên khác cho một kiểu dữ liệu

Ví dụ:

```
int T; // T có kiểu dữ liệu là int
typedef int T; // T là một (alias) tên gọi khác của kiểu int
int A[100]; // A là một mảng 100 phần tử int
typedef int A[100]; // A là một alias (tên gọi khác) của kiểu "mảng 100 phần tử int"
```

Một số cú pháp nâng cao:

- Định nghĩa kiểu dữ liệu con trỏ hàm thông thường

```
void (*f)(int); // f là "con trỏ tới hàm có 1 tham số kiểu int, trả về void"
typedef void (*f)(int); // f là alias của kiểu "con trỏ tới hàm có
```

```
// 1 tham số kiểu int, trả về void
```

- Định nghĩa kiểu dữ liệu con trỏ hàm trỏ đến phương thức non-static của class

```
void (Foo::*pmf)(int); // pmf là một alias của "con trỏ tới hàm thành
viên của Foo
// có một tham số int trả về void"
typedef void (Foo::*pmf)(int); // pmf là một alias của kiểu "con trỏ tới
hàm
// thành viên của Foo có một tham số int, trả về void
```

- Định nghĩa cùng lúc nhiều kiểu dữ liệu bằng typedef

```
int *x, (*p)(); // x kiểu int*, p kiểu int(*)()
typedef int *x, (*p)(); // x là alias của int*, p là alias của int(*)()
```

### 1.5.4 using

Từ C++11 chúng ta có thể sử dụng cú pháp using thay thế typedef

```
using i = int; // kiểu int
using arr = int[100]; // kiểu dữ liệu array có 100 phần tử int
using FP = void(*) (int); // kiểu dữ liệu "con trỏ tới hàm có 1 tham số
kiểu int, trả về void"
using MP = void (Foo::*) (int); // kiểu dữ liệu "con trỏ tới hàm thành
viên của Foo có một tham số int,
// trả về void
```

Tạo một alias kiểu dữ liệu bằng using có tác dụng chính xác giống như với typedef. Nó chỉ đơn giản là một cú pháp thay thế để hoàn thành điều tương tự.

```
#include <iostream>
using namespace std;

template <typename T>
class Sample {
private:
    T t;

public:
    void setValue(T t) {
        this->t = t;
    }
}
```

```

void printf() {
    cout << t << endl;
}

};

using Sample_int = Sample<int>;
//hoặc typedef Sample<int> Sample_int

int main() {
    Sample_int sample;
    sample.setValue(10);
    sample.printf();
    return 0;
}

```

## 1.6 Chỉ thị tiền xử lý

### 1.6.1 Giới thiệu

Chỉ thị tiền xử lý là những chỉ thị cung cấp cho bộ tiền xử lý để xử lý thông tin trước khi bắt đầu quá trình biên dịch.

Tất cả các chỉ thị tiền xử lý đều bắt đầu với # và không kết thúc bằng dấu chấm phẩy ; khi kết thúc.

Để cho dễ phân biệt chúng ta chia thành 3 nhóm chính đó là:

- Chỉ thị chèn tệp (#include).
- Chỉ thị định nghĩa cho tên (#define macro).
- Chỉ thị biên dịch có điều kiện (#if, #else, #elif, #endif, ...).

### 1.6.2 Chỉ thị chèn tệp (#include)

- Ở nhóm này chỉ có một chỉ thị đó là #include. Đây là chỉ thị cho phép chèn nội dung một file khác vào file đang viết.

- Cú pháp 1: #include <file\_name>

Với cú pháp 1 (dùng dấu ngoặc nhọn), bộ tiền xử lý sẽ tìm file\_name có sẵn trong SDK và chèn vào file đang viết, nếu tìm không thấy file\_name sẽ gây ra lỗi. Các file có sẵn trong SDK như stdio.h, math.h, conio.h,....

- Cú pháp 2: `#include "file_name"`

Khi sử dụng cú pháp 2 (dùng dấu nháy đôi), bộ tiền xử lý sẽ tìm `file_name` trong các thư mục trên máy tính, khi tìm không thấy thì tiếp tục tìm trong các file có sẵn trong SDK. Nếu tìm được `file_name` thì chèn nội dung `file_name` vào file đang thao tác, nếu vẫn không tìm thấy `file_name` thì sinh ra báo lỗi.

### 1.6.3 Chỉ thị định nghĩa cho tên

Ở nhóm này gồm các chỉ thị `#define`, `#undef`.

#### 1.6.3.1 Chỉ thị `#define`

\* Chỉ thị `#define` không có đối số

- Cú pháp: `#define identifier replacement-list`

- Chỉ thị này có tác dụng thay thế tên (`identifier`) bằng một dãy kí tự sau nó, khi dãy kí tự thay thế quá dài và sang dòng mới thì có thể sử dụng dấu `\` vào cuối dòng trước.

Ví dụ: `#define BCORN "BCORN - Hỗ trợ sinh viên Bách khoa" // định nghĩa cho BCORN`

Trong hàm main ta thực hiện lệnh sau:

```
printf(BCORN); // tương đương với printf("BCORN - Hỗ trợ sinh viên Bách khoa");
```

- Phạm vi của tên được định nghĩa bởi `#define` là lúc từ khi nó được định nghĩa cho đến cuối tệp.

- Có thể dùng `#define` định nghĩa như tên hàm, một biểu thức, một đoạn chương trình bằng một tên, với cách sử dụng này thì chương trình của chúng ta sẽ ngắn gọn và dễ hiểu hơn. Ví dụ:

```
#define output printf("BCORN - Hỗ trợ sinh viên Bách khoa");
```

Trong hàm main ta thực hiện lệnh sau:

```
output; // printf("BCORN - Hỗ trợ sinh viên Bách khoa ");
```

- Những điểm cần chú ý của chỉ thị `#define` cho cách sử dụng trên:

- Khi định nghĩa một biểu thức ta nên đặt nó trong trong cặp dấu ngoặc tròn. Ví dụ: `#define SUM 5+8`. Khi ta gán `size = SUM` không xảy ra vấn đề gì nhưng

khi gán `size = 5 * SUM` thì tương đương với `size = 5 * 5+8` chứ không phải là `size = 5 * (5 + 8)`. Do đó, người ta thường dùng `#define SUM (5+8)`.

- Khi định nghĩa đoạn chương trình gồm nhiều câu lệnh thì ta nên đặt trong cặp ngoặc { và }.

\* Chỉ thị `#define` có đối số

Ngoài cách sử dụng `#define` như trên, chúng ta còn có thể dùng `#define` để định nghĩa các macro có đối giống như hàm. Để rõ hơn thì bạn theo dõi ví dụ định nghĩa một macro tính tổng của 2 giá trị.

```
#define SUM(x,y) (x)+(y)
```

Khi đó câu lệnh `int z = SUM(x*2, y*3);` được thay thế bằng `int z = (x*2) + (y*3);`

Các điểm cần lưu ý:

- Giữa macro và dấu ( không được tồn tại khoảng trắng.
- Để tránh rủi ro không mong muốn thì khi viết các biểu thức định nghĩa cho macro, các đối tượng hình thức (như x và y ở ví dụ trên) thì nên có cặp ngoặc ( và ) bao quanh. Để minh họa cho điều này thì ta đến với ví dụ sau:

```
#define MUL(x,y) x*y

void main()
{
    printf("%d",MUL(5+3, 10));
}
```

Khi đó trình biên dịch thay `MUL(5+3, 10)` bằng `5+3*10` và ta nhận đáp án 35 thay vì 80 như ta mong muốn.

#### 1.6.3.2 Chỉ thị `#undef`

- Cú pháp: `#undef identifier`

- Khi ta cần định nghĩa lại một tên mà ta đã định nghĩa trước đó thì ta sử dụng `#undef` để hủy bỏ định nghĩa đó và sử dụng `#define` định nghĩa lại cho tên đó.

Ví dụ:

```
#define BCORN "Hello BCORN"          // Định nghĩa cho tên BCORN là "Hello
BCORN "

#undef BCORN                          // Hủy bỏ định nghĩa cho tên BCORN

#define BCORN "Welcome to BCORN " // Định nghĩa lại cho tên BCORN là
"Welcome to BCORN"
```

### 1.6.4 Chỉ thị biên dịch có điều kiện

Ở nhóm này gồm các chỉ thị `#if`, `#elif`, `#else`, `#ifdef`, `#ifndef`.

#### 1.6.4.1 Các chỉ thị `#if`, `#elif`, `#else`

- Cú pháp:

```
#if constant-expression_1
// Đoạn chương trình 1

#elif constant-expression_2
// Đoạn chương trình 2

#else
//Đoạn chương trình 3

#endif
```

Nếu `constant-expression_1` true thì chỉ có đoạn chương trình 1 sẽ được biên dịch, trái lại nếu `constant-expression_1` false thì sẽ tiếp tục kiểm tra đến `constant-expression_2`. Nếu vẫn chưa đúng thì đoạn chương trình trong chỉ thị `#else` được biên dịch.

Các `constant-expression` là biểu thức mà các toán hạng trong đó đều là hằng, các tên đã được định nghĩa bởi các `#define` cũng được xem là các hằng.

#### 1.6.4.2 Các chỉ thị `#ifdef`, `#ifndef`

- Một cách biên dịch có điều kiện khác đó là sử dụng `#ifdef` và `#ifndef`, được hiểu như là Nếu đã định nghĩa và Nếu chưa được định nghĩa.

- Chỉ thị `#ifdef`

```
#ifdef identifier
//Đoạn chương trình 1
```



```
#else
    //Đoạn chương trình 2

#endif
```

Nếu identifier đã được định nghĩa thì đoạn chương trình 1 sẽ được thực hiện. Ngược lại nếu identifier chưa được định nghĩa thì đoạn chương trình 2 sẽ được thực hiện.

- Chỉ thị #ifndef

```
#ifndef identifier
    //Đoạn chương trình 1

#else
    //Đoạn chương trình 2

#endif
```

Với chỉ thị #ifndef thì cách thức hoạt động ngược lại với #ifdef.

- Ví dụ:

```
#ifdef    MAX                // Nếu MAX đã được định nghĩa
    #undef MAX              // Hủy bỏ MAX
    #define MAX 100         // Định nghĩa lại MAX

#else
    #define MAX 1           // Nếu MAX chưa được định nghĩa
                             // Định nghĩa MAX

#endif
```

- Các chỉ thị điều kiện ở trên, thường được sử dụng cho việc xử lý xung đột thư viện khi chúng ta #include nhiều thư viện như ở ví dụ dưới đây:

Tôi có một file A.h.

```
//file A.h
```

Source code B

Giả sử có các file B.h và C.h và 2 file này đều cần nội dung của file A.h, vì thế tôi #include "A.h" vào file B.h và C.h.

```
//file B.h
#include"A.h"
```

Source code B

```
//file C.h
#include"A.h"
```

Source code C

File main.cpp của tôi #include "B.h" và #include "C.h", khi đó nội dung file main.cpp trở thành.

```
//file main.cpp
#include"B.h"
#include"C.h"
```

Source code file main

Chúng ta có thể hình dung file main.cpp như sau:

```
//file main.cpp
//#include"B.h"
    Source code A
    Source code B

//#include"C.h"
    Source code A
    Source code C
```

Source code file main

Như ta thấy nội dung file A.h sẽ được chép 2 lần sang file main.cpp, bởi vì khi ta #include "B.h" thì nội dung file B.h(có cả nội dung file A.h) đã được chép sang file main.cpp. Ta tiếp tục #include "C.h" thì nội dung file C.h (có cả nội dung file A.h) đã được chép sang file main.cpp. Vì thế, nội dung của file A.h được chép 2 lần trong file main.cpp và khi ta biên dịch thì trình biên dịch sẽ báo lỗi. Để khắc phục lỗi này thì tôi sử dụng chỉ thị #ifndef, #define vào trong file A.h.

```

#ifndef    __A_H__
#define    __A_H__

    Source code A

#endif    // __AH__

```

Khi đó nội dung file main.cpp chúng ta có thể hình dung:

```

//file main.cpp

//#include"B.h"
#ifndef    __A_H__
#define    __A_H__

    Source code A

#endif    // __AH__

    Source code B

//#include"C.h"
#ifndef    __A_H__
#define    __A_H__

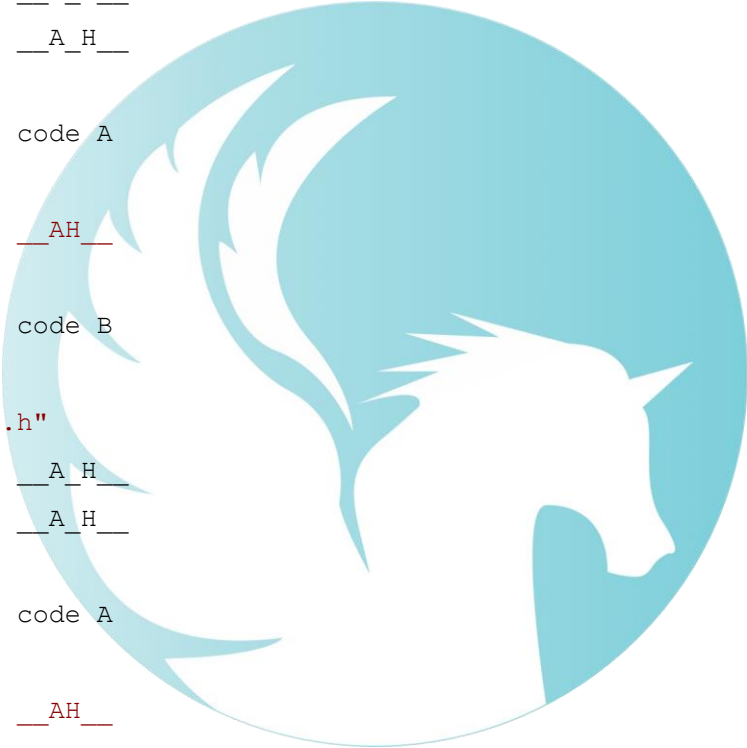
    Source code A

#endif    // __AH__

    Source code C

    Source code file main

```



- Cơ chế hoạt động:

- Từ dòng 4-9: `__A_H__` chưa được định nghĩa nên cho phép chép nội dung file A.h vào main.cpp.
- Dòng 11: Nội dung file B.h.

- Từ dòng 14-19: Ở trên \_\_A\_H\_\_ đã được định nghĩa nên không cho phép chép nội dung file A.h vào main.cpp.
- Dòng 21: Nội dung file C.h.

## 1.7 Namespace

### 1.7.1 Namespace là gì?

#### \* Tình huống

Khi đang lập trình trong một file A bạn include 2 file B và C, nhưng 2 file này có cùng định nghĩa một hàm function() giống nhau về tên và tham số truyền vào, nhưng xử lý của mỗi hàm ở mỗi file là khác nhau, vấn đề đặt ra là code làm sao để trình biên dịch hiểu được khi nào bạn muốn gọi function của file B, khi nào bạn muốn gọi function của file C. Khi gọi hàm function() ở file A, trình biên dịch sẽ không biết được hàm function() bạn muốn gọi là hàm được định nghĩa ở file B hay file C. Vì vậy trình biên dịch chương trình sẽ báo lỗi.

fileB.hpp:

```
#include <iostream>

using namespace std;

void function() { cout << "function in fileB running." << endl; }
```

fileC.hpp:

```
#include <iostream>

using namespace std;

void function() { cout << "function in fileC running." << endl; }
```

fileA.cpp:

```
#include <iostream>
#include "fileB.hpp"
#include "fileC.hpp"

using namespace std;
```

```
int main() {
    function();

    return 0;
}
```

Trình biên dịch báo lỗi:

```
In file included from fileA.cpp:3:0:
fileC.hpp: In function 'void function()':
fileC.hpp:5:6: error: redefinition of 'void function()'
    void function(){ cout << "function file1 running." << endl;}
        ^
In file included from fileA.cpp:2:0:
fileB.hpp:5:6: note: 'void function()' previously defined here
    void function(){ cout << "function file1 running." << endl;}
```

=> Namespace sẽ giải quyết chọn vẹn vấn đề này!

\* Định nghĩa: Namespace là từ khóa trong C++ được sử dụng để định nghĩa một phạm vi nhằm mục đích phân biệt các hàm, lớp, biến, ... cùng tên trong các thư viện khác nhau.

### 1.7.2 Tại sao phải sử dụng namespace?

Trong các project lớn, sẽ có các hàm, lớp, ... cùng tên được định nghĩa, nhưng nội dung khác nhau (giống như vd trên phần 1). Trong những trường hợp này cần đặt các hàm, lớp, ... này vào các namespace khác nhau để trình biên dịch có thể phân biệt được, cũng như việc tường minh code cho những người cần đọc và sử dụng sau đó.

### 1.7.3 Sử dụng namespace

- Cú pháp:

```
namespace ten_namespace{
    //code
}
```

Tiếp tục ví dụ ở phần 1 khi sử dụng namespace.

fileB.hpp:

```
#include <iostream>

using namespace std;

namespace fileB{
    void function() { cout << "function in fileB running." << endl; }
}
```

fileC.hpp:

```
#include <iostream>

using namespace std;

namespace fileC{
    void function() { cout << "function in fileC running." endl; }
}
```

fileA.cpp

```
#include <iostream>
#include "fileB.hpp"
#include "fileC.hpp"

using namespace std;

int main() {
    fileB::function();
    fileC::function();

    return 0;
}
```

Kết quả:

```
function fileB running.
function fileC running.
```

- Khai báo namespace:

Khi bạn cảm thấy việc gọi hàm thông qua gọi tên của namespace là dài dòng và không cần thiết, trong trường hợp này bạn có thể sử dụng từ khóa khai báo namespace để sử dụng.

```
using namespace ten_namespace;
```

Khi khai báo như này bạn có thể gọi trực tiếp các hàm, ... được định nghĩa trong namespace. Bạn sẽ thấy quen thuộc hơn trong việc khai báo sử dụng namespace std.

fileB.hpp:

```
#include <iostream>

using namespace std;

namespace fileB{
    void function() { cout << "function in fileB running." << endl; }
}
```

fileA.cpp

```
#include <iostream>
#include "fileB.hpp"

using namespace std;
using namespace fileB;

int main() {
    function();

    return 0;
}
```

#### 1.7.4 Các trường hợp đặc biệt

##### 1.7.4.1 namespace không kề nhau

Là 1 namespace nhưng được định nghĩa ở nhiều file khác nhau.

Ví dụ:

fileB.hpp:

```
#include <iostream>
```

```
using namespace std;

namespace file{
    void function1() { cout << "function in fileB running." << endl; }
}
```

**fileC.hpp:**

```
#include <iostream>

using namespace std;

namespace file{
    void function2() { cout << "function in fileC running." << endl; }
}
```

**fileA.cpp:**

```
#include <iostream>
#include "fileB.hpp"
#include "fileC.hpp"

using namespace std;
using namespace file;

int main() {
    function1();
    function2();

    return 0;
}
```

#### 1.7.4.2 namespace lồng nhau

Có nghĩa là bạn định nghĩa một namespace trong một namespace khác

```
namespace parent_namespace{
    //code
    namespace child_namespace{
        //code
    }
}
```



}

## 1.8 Bài tập

**Bài 1.** Tính tổng  $S = 1/1! + 1/2! + 1/3! + \dots + 1/n!$  với  $n$  nhập từ bàn phím

**Bài 2.** Biết  $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ . Tính  $e^2$  với độ chính xác  $10^{-6}$

**Bài 3.** Viết chương trình nhập vào ngày, tháng, năm. Kiểm tra ngày và tháng nhập có hợp lệ hay không. Tính thứ trong tuần của ngày đó.

Năm nhuận (leap year) tính theo lịch Gregorian (từ 1582): năm phải chia hết cho 4 và không chia hết cho 100, hoặc năm phải chia hết cho 400. Thứ trong tuần tính theo công thức Zeller:

```
dayofweek = (d + y + y/4 - y/100 + y/400 + (31 * m)/12) % 7
```

với:

+)  $a = (14 - \text{month})/12$

+)  $y = \text{year} - a$

+)  $m = \text{month} + 12*a - 2$

dayofweek: 0 (chủ nhật), 1 (thứ hai), 2 (thứ ba), ...

**Bài 4.** Viết chương trình nhập vào một năm ( $> 1582$ ), in lịch của năm đó. Tính thứ cho ngày đầu năm bằng công thức Zeller ở bài 3.

**Bài 5.** Xây dựng struct phân số, thực hiện chức năng sau: In phân số dạng  $a/b$ , rút gọn phân số, thực hiện phép toán cộng, trừ, nhân, chia trên phân số (trả về kết quả là phân số tối giản)

**Bài 6.** Xây dựng 1 struct Doubly Linked List với chức năng thêm phần tử ở đầu, ở cuối, ở vị trí bất kì, xóa phần tử đầu, cuối, vị trí bất kì

## CHƯƠNG 2. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

### 2.1 Tính chất của lập trình hướng đối tượng

#### 2.1.1 Tính đóng gói (*encapsulation*) và che dấu thông tin (*information hiding*)

- Trạng thái đối tượng được bảo vệ và không cho các truy cập từ code bên ngoài như thay đổi trong thái hay nhìn trực tiếp. Việc cho phép môi trường bên ngoài tác động vào các dữ liệu nội tại của một đối tượng theo cách nào là hoàn toàn tùy thuộc vào người viết mã. Đây là tính chất đảm bảo sự toàn vẹn, bảo mật.

- Tính đóng gói được thể hiện thông qua phạm vi truy cập.

#### 2.1.2 Tính kế thừa (*inheritance*)

- Tính kế thừa cho phép ta xây dựng một lớp mới dựa trên các định nghĩa của một lớp đã có. Lớp đã có gọi là lớp Cha, lớp mới phát sinh là lớp Con và đương nhiên kế thừa tất cả các thành phần của lớp Cha, có thể chia sẻ hay mở rộng các đặc tính sẵn có mà không phải tiến hành định nghĩa lại.

#### 2.1.3 Tính đa hình (*polymorphism*)

- Một tác vụ được thực hiện nhiều cách khác nhau được gọi là tính đa hình.

- Đối với tính chất đa hình, nó được thể hiện rõ qua việc gọi phương thức của đối tượng. Các phương thức hoàn toàn có thể giống nhau, nhưng việc xử lý luồng có thể khác nhau. Nói một cách khác: Tính đa hình cung cấp khả năng cho phép người lập trình gọi trước một phương thức của đối tượng, tuy chưa xác định đối tượng có phương thức muốn gọi hay không. Đến khi thực hiện, chương trình mới xác định được đối tượng và gọi phương thức tương ứng của đối tượng đó. Kết nối trở lại chương trình được uyển chuyển hơn, chỉ yêu cầu đối tượng cung cấp đúng phương thức cần thiết là đủ.

- Trong C++, chúng ta sử dụng nạp chồng phương thức (method overloading) và ghi đè phương thức (method overriding) để có tính đa hình:

+ Nạp chồng (Overloading): Đây là khả năng cho phép một lớp có nhiều thuộc tính, phương thức cùng tên nhưng với các tham số khác nhau về loại cũng như về số lượng. Khi được gọi, dựa vào tham số truyền vào, phương thức tương ứng sẽ được thực hiện.


+ Ghi đè (Overriding): là hai phương thức cùng tên, cùng tham số, cùng kiểu trả về nhưng con viết lại và dùng theo cách của nó, và xuất hiện ở lớp cha và tiếp tục xuất hiện ở lớp con. Khi dùng override, lúc thực thi, nếu lớp Con không có phương thức riêng, phương thức của lớp Cha sẽ được gọi, ngược lại nếu có, phương thức của lớp Con được gọi.

#### 2.1.4 Tính trừu tượng (abstraction)

- Tính trừu tượng là một tiến trình ẩn các chi tiết trình triển khai, chỉ hiển thị tính năng tới người dùng. Tính trừu tượng cho phép bạn loại bỏ tính chất phức tạp của đối tượng bằng cách chỉ đưa ra các thuộc tính và phương thức cần thiết của đối tượng.

- Tính trừu tượng giúp tập trung vào những cốt lõi cần thiết của đối tượng thay vì quan tâm đến cách mà nó thực hiện.

#### 2.1.5 Ví dụ



```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Animal
{
private:
    string name;

public:
    Animal() {}
    Animal(string name) : name(name) {}
    virtual ~Animal() {}

    virtual void sayHello() {}

    string getName() const { return name; }
};

class Cat : public Animal
{

```

```

public:
    Cat() : Animal() {}
    Cat(string name) : Animal(name) {}
    ~Cat() {}

    void sayHello()
    {
        cout << "Hi, I'm Cat. My name is " << Animal::getName();
    }
};

class Dog : public Animal
{
public:
    Dog() : Animal() {}
    Dog(string name) : Animal(name) {}
    ~Dog() {}

    void sayHello()
    {
        cout << "Hi, I'm Dog. My name is " << Animal::getName();
    }
};

class Zoo
{
private:
    vector<Animal *> animals;

public:
    void add(Animal *animal) { animals.push_back(animal); }
    void pop() { animals.pop_back(); }
    void show()
    {
        for (auto animal : animals)
        {
            animal->sayHello();
            cout << "\n";
        }
    }
}

```



```

    }
};

int main(int argc, char const *argv[])
{
    Animal *cat = new Cat("Tom");
    Animal *dog = new Dog("Milu");
    Zoo zoo;
    zoo.add(cat);
    zoo.add(dog);
    zoo.show();
    return 0;
}

```

Kết quả:

```

Hi, I'm Cat. My name is Tom
Hi, I'm Dog. My name is Milu

```

Các tính chất của lập trình hướng đối tượng được sử dụng thông qua ví dụ trên:

- Tạo abstract class Animal có phương thức sayHello. Abstract class này thể hiện tính trừu tượng, có nghĩa ta định ra rằng dù là con vật gì đi nữa thì nó cũng có phương thức sayHello.
- Tạo 2 lớp Cat và Dog kế thừa từ Animal. Khi khởi tạo chúng sẽ có tên. Chúng override lại phương thức sayHello để chào hỏi theo cách riêng của chúng. Điều này thể hiện tính đóng gói (đóng gói biến tên và phương thức sayHello với nhau) và tính thừa kế.
- Tạo lớp Zoo để quản lý nhiều Animal, có 1 phương thức add, pop để thêm, bớt các Animal (các đối tượng của các lớp thừa kế từ Animal), 2 phương thức show để gọi sayHello của tất cả đối tượng nó quản lý. Điều này thể hiện tính đa hình, Zoo gọi chỉ gọi một phương thức sayHello, nhưng tùy con vật mà lời chào hỏi sẽ khác nhau.

## 2.2 Lớp

### 2.2.1 Thành phần của một lớp

+ Trường: (hay còn gọi là thuộc tính, biến thành viên): Là các thuộc tính thể hiện đặc điểm tính chất cho đối tượng.

+ Phương thức: (hay còn được gọi là hành vi, hàm thành viên): Thể hiện các hành động của đối tượng.

### 2.2.2 Quyền truy cập các thành phần

+ private: Bảo mật cao nhất, chỉ được truy cập bởi các hàm thành viên của lớp, và hàm bạn lớp bạn của lớp đó.

+ protected: Quyền bảo mật có lựa chọn, cho phép thêm các lớp kế thừa có thể truy cập được.

+ public: Quyền tự do.

private và protected thường được dùng cho các thuộc tính, để đảm bảo không cho phép tự do can thiệp thuộc tính 1 đối tượng.

public: Thường được sử dụng cho các phương thức, vì các phương được gọi ở các hàm khác trong chương trình.

### 2.2.3 Cú pháp khai báo

```
class Tên_lớp {
private:
    /* Thuộc tính hoặc phương thức */
protected:
    /* Thuộc tính hoặc phương thức */
public:
    /* Thuộc tính hoặc phương thức */
};
```

Ví dụ:

```
class Student {
private:
    std::string fullName;
    int age;
    str::string birthday;
public:
    void show();
    void input();
};

void Student::input() {
    std::cout << "Ho va ten: ";
```

```

std::cin.ignore(0);
std::getline(std::cin, fullName);
std::cout << "Tuoi: ";
std::cin >> age;
std::cout << "Ngày sinh: ";
std::cin.ignore();
std::getline(std::cin, birthday);
}
int main() {
    Student s;
    s.input();
    s.show();
    Student *ptr = &s;
    ptr->input();
    ptr->show();
    return 0;
}

```

#### 2.2.4 So sánh struct và class trong C++

Trong C++, các class và struct là các bản thiết kế được sử dụng để tạo ra thể hiện của một lớp. Struct được sử dụng cho các đối tượng nhẹ như hình chữ nhật, màu sắc, điểm, v.v.

Không giống như lớp, các struct trong C++ là kiểu giá trị hơn là kiểu tham chiếu. Nó rất hữu ích khi chúng ta có dữ liệu mà chúng ta không có ý định sửa đổi sau khi tạo struct.

Struct trong C++ là một tập hợp các loại dữ liệu khác nhau. Nó tương tự như lớp chứa các loại dữ liệu khác nhau.

So sánh struct và class:

- Giống nhau:

- Bản thiết kế để tạo ra thể hiện.
- Chứa các dữ liệu khác nhau.

- Khác nhau:

Struct	Class
Nếu chỉ định truy cập không được khai báo rõ ràng, thì theo mặc định truy cập sẽ được public.	Nếu chỉ định truy cập không được khai báo rõ ràng, thì theo mặc định truy cập sẽ được private.
Thể hiện của struct được gọi là "Biến cấu trúc"	Thể hiện của lớp được gọi là "Đối tượng của lớp".

## 2.3 Constructor, Destructor

### 2.3.1 Hàm tạo (Constructor)

- Hàm khởi tạo là một hàm thành viên đặc biệt của một lớp. Nó sẽ được tự động gọi đến khi một đối tượng của lớp đó được khởi tạo.

- Sự khác biệt giữa hàm tạo và hàm thành viên thông thường:

- + Có tên trùng với tên lớp
- + Có tên trùng với tên lớp
- + Tự động được gọi khi một đối tượng thuộc lớp được tạo ra
- + Nếu chúng ta không khai báo một hàm tạo, trình biên dịch C++ sẽ tự động tạo một hàm tạo mặc định cho chúng ta (sẽ là hàm không có tham số nào và có phần thân trống)

- Có 4 loại hàm tạo:

a) *Hàm tạo không tham số (Default Constructor)*: Hàm tạo loại này sẽ không truyền vào bất kì một đối số nào

```
class sinhvien
{
private:
    string ten;
    int tuoi;
public:
    sinhvien()
```



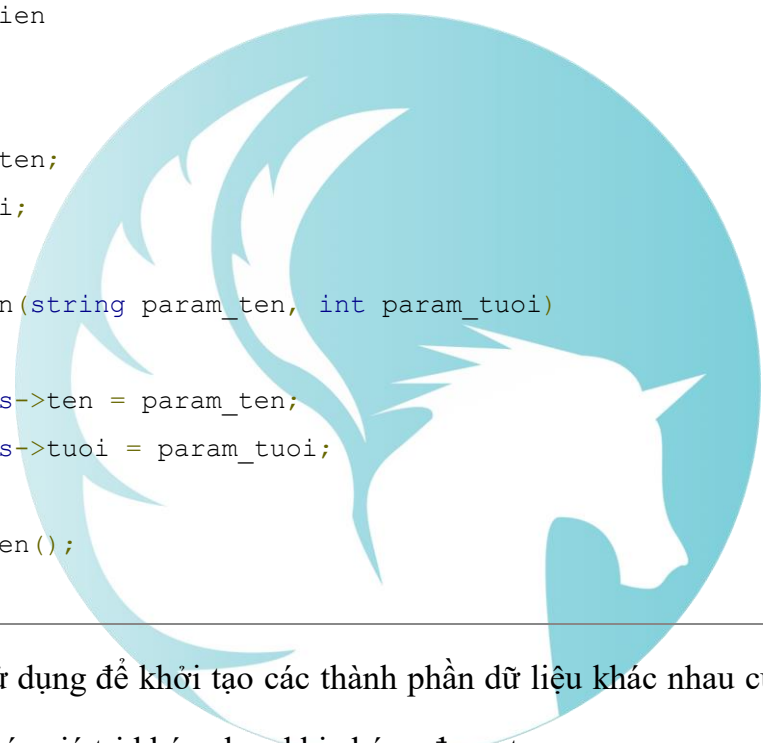
```

{
    this->ten = "";
    this->tuoi = 0;
}

~sinhvien();
};

```

b) *Hàm khởi tạo có tham số (Parameterized Constructor)*: Với loại hàm tạo này ta có thể truyền đối số cho chúng. Thông thường, các đối số này giúp khởi tạo một đối tượng khi nó được tạo.



```

class sinhvien
{
private:
    string ten;
    int tuoi;
public:
    sinhvien(string param_ten, int param_tuoi)
    {
        this->ten = param_ten;
        this->tuoi = param_tuoi;
    }
    ~sinhvien();
};

```

- Nó được sử dụng để khởi tạo các thành phần dữ liệu khác nhau của các đối tượng khác nhau với các giá trị khác nhau khi chúng được tạo.

- Nó được sử dụng để nạp chồng các hàm khởi tạo.

c) *Hàm tạo sao chép (Copy Constructor)*: là một hàm tạo mà tạo một đối tượng bằng việc khởi tạo nó với một đối tượng của cùng lớp đó, mà đã được tạo trước đó.

- Một hàm khởi tạo sao chép sẽ có nguyên mẫu chung như sau:

```

ClassName(const ClassName &old_obj)
{
    // Code
}

```

- Hàm khởi tạo sao chép sẽ được gọi khi:

- + Khi một đối tượng của lớp được trả về bằng một giá trị
- + Khi một đối tượng của lớp được truyền đối số dưới dạng tham số của một hàm
- + Khi một đối tượng được tạo ra dựa trên một đối tượng khác cùng lớp.
- + Khi trình biên dịch tạo một đối tượng tạm thời.

- Tuy nhiên trên thực tế thì không chắc chắn rằng hàm khởi tạo sao chép sẽ được gọi trong tất cả 4 trường hợp ở phía trên. Vì C++ tiêu chuẩn sẽ cho phép trình biên dịch tối ưu hoá bản sao trong một số trường hợp nhất định.

- Chú ý: Nếu một hàm tạo sao chép không được định nghĩa trong một lớp, trình biên dịch sẽ tự nó định nghĩa nó (bằng cách sao chép từng bit dữ liệu). Vì thế phải thật lưu ý nếu lớp có các biến con trỏ hoặc có sử dụng cấp phát bộ nhớ động thì nên viết lại hàm.

*d) Hàm tạo di chuyển (Move Constructor)*

### **2.3.2 Hàm huỷ (Destructor)**

- Hàm huỷ cũng là một hàm thành viên đặc biệt giống như hàm tạo, nó được dùng để phá huỷ hoặc xoá một đối tượng trong lớp.

- Hàm huỷ được gọi tự động khi một đối tượng thoát khỏi phạm vi của nó (Scope):

- + Một chức năng kết thúc.
- + Chương trình kết thúc.
- + Một khối chứa các biến cục bộ kết thúc.
- + Một toán tử delete được gọi

- Đặc điểm hàm huỷ:

+ Cũng giống với hàm tạo, hàm huỷ có tên trùng với tên của lớp, nhưng điểm khác biệt ở đây là sẽ có thêm ~ ở đầu.

- + Hàm huỷ là một hàm không có đối số truyền vào, và cũng không trả về giá trị.
- + Một lớp chỉ có duy nhất một hàm huỷ.
- Khi nào cần tự định nghĩa hàm huỷ: Thông thường thì hàm huỷ này hoạt động khá tốt, nhưng khi bài toán có sử dụng con trỏ, hoặc cấp phát bộ nhớ động thì bạn nên khai báo một hàm huỷ riêng để tránh rò rỉ bộ nhớ.

### 2.3.3 Con trỏ this

“this” là một con trỏ đặc biệt dùng để trỏ đến địa chỉ của đối tượng hiện tại. Như vậy để truy cập đến các thuộc tính, phương thức của đối tượng hiện tại thì ta sẽ sử dụng con trỏ this. Ví dụ:



```
#include <iostream>
using namespace std;
class NhanVien {
    int msnv;
    string ten;
    int tuoi;
public:
    void setData(int msnv, string ten, int tuoi) {
        this->msnv = msnv;
        this->ten = ten;
        this->tuoi = tuoi;
    }
    void showData() {
        cout << "Ten nhan vien: " << this->ten << endl;
        cout << "Ma so nhan vien: " << this->msnv << endl;
        cout << "Tuoi: " << this->tuoi << endl;
    }
};
```

Chú ý: Trong các phương thức bình thường (không phải hàm khởi tạo) nếu bạn sử dụng tên của biến thì sẽ có hai trường hợp xảy ra.

- Nếu biến đó không tồn tại trong phương thức mà nó lại trùng với tên thuộc tính thì mặc nhiên nó sẽ hiểu đó là thuộc tính.
- Nếu biến đó có khai báo trong phương thức thì ta sẽ hiểu đó là biến bình thường, không phải là thuộc tính.

Ví dụ:

```
#include <iostream>
using namespace std;
class NhanVien {
    int msnv;
    string ten;
    int tuoi;
public:
    NhanVien(int msnv, string ten, int tuoi) {
        cout << "Trong ham xay dung: " << endl;
        cout << "    msnv: " << msnv << endl;
        cout << "    ten: " << ten << endl;
        cout << "    Tuoi: " << tuoi << endl;
        msnv = msnv;
        ten = ten;
        tuoi = tuoi;
    }
    void HienThi() {
        cout << "Ham in thong tin cua doi tuong nhan vien: " << endl;
        cout << ten << endl;
        cout << "    Ma so nhan vien: " << msnv << endl;
        cout << "    Tuoi: " << tuoi << endl;
    }
};

int main() {
    NhanVien n1 = NhanVien(111231, "Nguyen Van A", 25);
    n1.HienThi();
    return 0;
}
```

Kết quả:

Trong ham xây dựng:

msnv: 111231

ten: Nguyen Van A

Tuoi: 25

Ham in thông tin của đối tượng nhân viên:

Ma so nhan vien: 6487488

Tuoi: 0

Đây là một kết quả mà chúng ta không hề mong đợi đúng không? Mình mong muốn in ra thông tin của sinh viên nhưng có vẻ như nó in ra một dãy số nào đó, có thể là một địa chỉ trong một ô nhớ nào đó. Lý do tại sao? Mình xin được giải thích kết quả của chương trình trên như sau:

- Khi chúng ta khai báo tên của tham số hàm trùng tên với dữ liệu thành viên của lớp, thì bên trong hàm xây dựng chương trình hiểu là biến tham số chứ không phải dữ liệu thành viên của lớp
- Như vậy ở ví dụ trên, bên trong thân hàm xây dựng ta gán `msnv = msnv`, `ten = ten`, `tuoi = tuoi`, thì chương trình hiểu `mssv`, `ten`, `tuoi` chính là biến truyền vào từ hàm xây dựng, chính vì vậy nó không cập nhật vào các thuộc tính của đối tượng.
- Khi các dữ liệu thành viên như `msnv`, `ten`, `tuoi` không được khởi tạo giá trị nó sẽ có giá trị tự động cho chương trình tạo ra mà chúng ta không hề biết trước

Con trỏ this trong C++ giúp chúng ta giải quyết được vấn đề trên. Chúng ta sẽ dùng con trỏ this trong ví dụ trên như sau:

```
#include <iostream>
using namespace std;
class NhanVien {
    int msnv;
    string ten;
    int tuoi;
public:
    NhanVien(int msnv, string ten, int tuoi) {
        cout << "Trong ham xay dung: " << endl;
```

```

        cout << "    msnv: " << msnv << endl;
        cout << "    ten: " << ten << endl;
        cout << "    Tuổi: " << tuoi << endl;
        this->msnv = msnv;
        this->ten = ten;
        this->tuoi = tuoi;
    }

    void HienThi() {
        cout << "Ham in thong tin cua doi tuong nhan vien: " << endl;
        cout << ten << endl;
        cout << "    Ma so nhan vien: " << msnv << endl;
        cout << "    Tuổi: " << tuoi << endl;
    }
};

int main() {
    NhanVien n1 = NhanVien(111231, "Nguyen Van A", 25);
    n1.HienThi();
    return 0;
}

```

### Kết quả:

Trong ham xay dung:

```

msnv: 111231
ten: Nguyen Van A
Tuoi: 25

```

Ham in thong tin cua doi tuong nhan vien:

```

Nguyen Van A
    Ma so nhan vien: 111231
    Tuổi: 25

```

Như vậy con trỏ this trong C++ dùng để tham chiếu đến thể hiện hiện tại của lớp. Con trỏ this có thể sử dụng trong 3 cách như sau:

- Nó có thể được sử dụng để truyền đối tượng hiện tại làm tham số cho phương thức khác.

- Nó có thể được sử dụng để tham chiếu đến thể hiện hiện tại của lớp (như ở ví dụ trên).
- Nó có thể được sử dụng để khai báo các chỉ mục.

## 2.4 Getter, Setter

### 2.4.1 Định nghĩa

Setter và Getter là 2 phương thức sử dụng để cập nhật hoặc lấy ra giá trị thuộc tính, đặc biệt dành cho các thuộc tính ở phạm vi private.

Việc sử dụng Setter và Getter cần thiết trong việc kiểm soát những thuộc tính quan trọng mà ta thường được sử dụng và yêu cầu giá trị chính xác. Ví dụ thuộc tính age lưu tuổi con người, thực tế thì phạm vi tuổi là từ 0 đến 100, thì ta không thể cho chương trình lưu giá trị age âm hoặc quá 100 được.

### 2.4.2 Cú pháp

#### 2.4.2.1 Setter

```
void set<tên thuộc tính> (<tham số giá trị mới>) {
    this-> <tên thuộc tính> = <tham số giá trị mới>;
}
```

#### 2.4.2.2 Getter

```
<kiểu dữ liệu thuộc tính> get<tên thuộc tính> () {
    return this-> <tên thuộc tính>;
}
```

Chú ý: Setter, Getter đều ở public

#### 2.4.2.3 Ví dụ

Ta sẽ tạo phương thức setter và getter cho thuộc tính age lớp Person

```
#include <iostream>
#include <string>

using namespace std;
```

```

class Person
{
private:
    string name;
    int age;

public:
    Person(string name = "", int age = 0)
    {
        this->name = name;
        this->age = age;
    }

    void setAge(int age)
    {
        this->age = age;
    }

    int getAge() const { return this->age; }
};

```

## 2.5 Hàm bạn, lớp bạn, static

### 2.5.1 Hàm bạn, lớp bạn

#### 2.5.1.1 Hàm bạn

- Hàm bạn trong C++ là hàm tự do, không thuộc lớp. Tuy nhiên hàm bạn trong c++ có quyền truy cập các thành viên private, protected của lớp.

- Một lớp trong C++ có thể có nhiều hàm bạn, và chúng phải nằm bên ngoài class

- Ưu điểm:

+ Kiểm soát các truy nhập ở cấp độ lớp. Nghĩa là không thể áp đặt hàm bạn cho một lớp, nếu như chưa khai báo hàm bạn trong lớp

+ Giải quyết được vấn đề cần truy cập dữ liệu của lớp như trên.

- Cú pháp: Đặt từ khoá `friend` phía trước, sau đó khai báo như một hàm thông thường

- Ví dụ:



```
#include <bits/stdc++.h>
using namespace std;

class giangvien;
class sinhvien
{
private:
    string masinhvien;
public:
    sinhvien()
    {
        masinhvien = "";
    }
    ~sinhvien()
    {
        masinhvien = "";
    }

    void set()
    {
        cout << "Nhap Ma Sinh Vien";
        cin.ignore(0);
        getline(cin, this->masinhvien);
    }

    friend void get(sinhvien a, giangvien b);
};

class giangvien
{
private:
    string magiangvien;
public:
    giangvien()
    {
        magiangvien = "";
    }
    ~giangvien()
    {
```

```

        magiangvien = "";
    }

    void set()
    {
        cout << "Nhap Ma Giang Vien: ";
        cin.ignore(0);
        getline(cin, this->magiangvien);
    }

    friend void get(sinhvien a, giangvien b);
};

void get(sinhvien a, giangvien b)
{
    cout << "Ma Sinh Vien: " << a.masinhvien << endl;
    cout << "Ma Giang Vien: " << b.magiangvien << endl;
}

int main()
{
    sinhvien a;
    giangvien b;
    a.set(); b.set();
    get(a,b);
    return 0;
}

```



#### 2.5.1.2 Lớp bạn

- Tương tự như hàm bạn, lớp bạn trong C++ cũng cho phép lớp bạn của lớp kia truy cập các thành viên private, protected.

- Tính chất:

+ Khai báo lớp A là bạn của lớp B không có nghĩa lớp B là bạn của lớp A (chỉ có tính 1 chiều). Điều đó có nghĩa là chỉ có lớp A truy cập được thành viên của lớp B, nhưng ngược lại lớp B không thể truy cập ngược lại của lớp A.

+ Không đối xứng.

+ Không bắt cầu.

- Cú pháp: Dùng từ khoá `friend` để khai báo giống như khai báo hàm bạn

```
class A
{
private:
    int i;

public:
    friend class B; //Có lớp bạn là B
};

class B
{
public:
    void Change(A obj)
    {
        obj.i++;
    }
};
```

## 2.5.2 static

### 2.5.2.1 Biến static

- Biến static trong Hàm: Khi một biến được khai báo với từ khóa `static`, vùng nhớ cho nó tồn tại theo vòng đời của chương trình. Ngay cả khi hàm được gọi nhiều lần, vùng nhớ cho biến static chỉ được cấp nhất một lần và giá trị của biến trong những lần gọi trước đó được lưu lại và được sử dụng để thực hiện thông qua các lượt gọi hàm tiếp theo. Điều này rất hữu ích để triển khai các ứng dụng nào khác mà trạng thái chức năng trước đó cần được lưu trữ.

```
#include <iostream>
using namespace std;

void demo()
{
    static int count = 0;
    cout << count << " ";
}
```

```

    count++;
}

int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}

```

Kết quả:

```
0 1 2 3 4
```

- Các biến static trong class: Vì các biến được khai báo là tĩnh chỉ được khởi tạo một lần khi chúng được cấp phát một địa chỉ trong bộ lưu trữ tĩnh riêng biệt, do đó, các biến tĩnh trong một lớp được chia sẻ bởi các đối tượng. Chúng ta không tạo ra các bản sao cho cùng một biến tĩnh của các đối tượng khác nhau. Cũng vì lý do này mà các biến tĩnh không thể được khởi tạo bằng cách sử dụng các hàm khởi tạo.

```

#include<iostream>
using namespace std;

class A
{
public:
    static int i;

    A() { i++; };
};

int A::i = 0;

int main()
{
    A obj1;
    A obj2;
    cout << obj1.i<<" "<<obj2.i;
    return 0;
}


```

Kết quả:

```
1 2
```

### 2.5.2.2 Thành viên static của class

- Các đối tượng của class là static: Cũng giống như các biến, các đối tượng cũng khi được khai báo là static có thời gian tồn tại bằng với thời gian tồn tại của chương trình.



```
#include <iostream>
using namespace std;

class A
{
    int i = 0;

public:
    A()
    {
        i = 0;
        cout << "Inside Constructor\n";
    }

    ~A()
    {
        cout << "Inside Destructor\n";
    }
};

int main()
{
    int x = 0;
    if (x == 0)
    {
        A obj;
    }

    cout << "End of main\n";
    return 0;
}
```

Kết quả:

```
Inside Constructor
Inside Destructor
End of main
```

```
#include <iostream>
using namespace std;

class A
{
    int i = 0;

public:
    A()
    {
        i = 0;
        cout << "Inside Constructor\n";
    }

    ~A()
    {
        cout << "Inside Destructor\n";
    }
};

int main()
{
    int x = 0;
    if (x == 0)
    {
        static A obj;
    }

    cout << "End of main\n";
    return 0;
}
```



**Kết quả:**

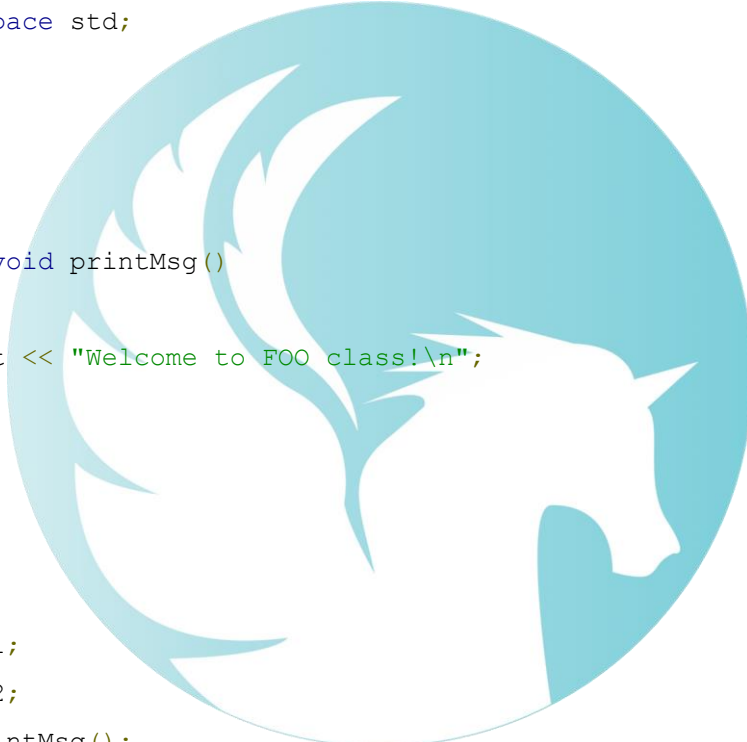
```
Inside Constructor
End of main
Inside Destructor
```

- Các hàm static trong class: Giống như các thành viên kiểu static hoặc các biến static bên trong class, các hàm static cũng không phụ thuộc vào đối tượng của class. Cho phép gọi một hàm thành viên static bằng cách sử dụng đối tượng và toán tử "." . Nhưng nên gọi các thành viên static bằng cách sử dụng tên lớp và toán tử phân giải phạm vi. Các hàm thành viên tĩnh chỉ được phép truy cập các thành viên dữ liệu kiểu static hoặc các hàm thành viên static khác, chúng không thể truy cập các thành viên không phải kiểu static của class.

```
#include <iostream>
using namespace std;

class FOO
{
public:
    static void printMsg()
    {
        cout << "Welcome to FOO class!\n";
    }
};

int main()
{
    FOO foo1;
    FOO foo2;
    foo1.printMsg();
    foo2.printMsg();
    FOO::printMsg();
    return 0;
}
```



**Kết quả:**

```
Welcome to FOO class!
Welcome to FOO class!
Welcome to FOO class!
```

### 2.5.3 Bài tập

**Bài 1.** Hoàn thành đoạn code sau:

```
#include <cmath>
#include <iostream>

class Point
{
private:
    float x = 0.0f;
    float y = 0.0f;

public:
    explicit Point() = default;
    ~Point(){};

    float getX() const { return x; }
    float getY() const { return y; }

    void setX(float x) { Point::x = x; }
    void setY(float y) { Point::y = y; }

    float dist(Point p) { return sqrt(pow(x - p.x, 2) + pow(y - p.y, 2)); }

    friend std::istream &operator>>(std::istream &is, Point &p)
    {
        is >> p.x >> p.y;
        return is;
    }
};

class Triangle
{
private:
    Point A, B, C;

public:
    explicit Triangle(){};
    Triangle(Point A, Point B, Point C) : A(A), B(B), C(C){};
};
```



```

~Triangle(){};

bool isTriangle();

float area();

float perimeter();

bool pointInTriangle(Point p);
};

```

## **Bài 2.** Lập chương trình quản lý danh sách sinh viên

\* Đối tượng Student.

*Các thuộc tính:*

- studentID: Khi đối tượng sinh viên được tạo ra, mã số sinh viên được khởi tạo bắt đầu từ 20190001
- fullName: Họ và tên sinh viên
- birthday: Ngày, tháng, năm sinh
- math, physics, english: Điểm của sinh viên

*Hàm tạo, hàm hủy:*

- Hàm tạo không đối số
- Hàm tạo 6 đối số với các đối số như thuộc tính

*Các phương thức:*

- Hàm set, get của từng thuộc tính
- input: Nhập vào thông tin sinh viên
- print: Hiển thị thông tin sinh viên

\* Đối tượng StudentList (quản lý danh sách sinh viên)

*Thuộc tính:*

- list: để quản lý danh sách sinh viên (dùng vector)
- size: số lượng sinh viên

*Phương thức:*

- input: Để nhập danh sách sinh viên (sau khi nhập mỗi sinh viên, ấn 1 để tiếp tục, phím còn lại để thoát nhập)

- show: Để xuất danh sách sinh viên dưới dạng như sau:

```
Sinh vien 1:
MSSV: 20190001
Ho va ten: Nguyen Van A
Ngay sinh: xx/xx/20xx
Diem:
    Toan: 8
    Ly: 7.5
    Anh: 8.5
```

```
Sinh vien 2:
MSSV: 20190002
Ho va ten: Pham Quynh B
Ngay sinh: xx/xx/20xx
Diem:
    Toan: 10
    Ly: 6.25
    Anh: 7.8
```

## 2.6 Nạp chồng hàm, nạp chồng toán tử

### 2.6.1 Nạp chồng hàm

- Nạp chồng hàm (function overloading) là tính năng của ngôn ngữ C++ (không có trong C). Kỹ thuật này cho phép sử dụng cùng một tên gọi cho nhiều hàm (có cùng mục đích). Nhưng khác nhau về kiểu dữ liệu tham số hoặc số lượng tham số.

```
#include <iostream>
using namespace std;

int sum(int val1, int val2) { return val1 + val2; }

float sum(float val1, float val2) { return val1 + val2; }

float sum(float val1, float val2, float val3)
```

```

{
    return val1 + val2 + val3;
}

float sum(float *arr, int n)
{
    float res = 0;
    for (int i = 0; i < n; ++i)
        res += arr[i];
    return res;
}

int main()
{
    cout << "8 + 4 = " << sum(8, 4) << endl; //sum(int, int)
    cout << "7.2 + 1.3 = " << sum(7.2f, 1.3f) << endl; //sum(float,
float)
    cout << "3.2 + 1.3 + 2 = " << sum(3.2, 1.3, 2) << endl; //sum(float,
float, float)

    float arr[] = {2, 3, 4, 1.2, 2.3};
    cout << "Sum of arr: " << sum(arr, 5) << endl; //sum(float*, int)

    return 0;
}

```

**Kết quả:**

```

8 + 4 = 12
7.2 + 1.3 = 8.5
3.2 + 1.3 + 2 = 6.5
Sum of arr: 12.5

```

- Một số hàm không thể nạp chồng trong C++:

+ Hàm chỉ khác nhau kiểu trả về

```

int foo() {
    return 10;
}

```

```
double foo() { // compiler error
    return 0.5;
}
```

+ Tham số hàm kiểu typedef

```
typedef int myint;
void print(myint value)
{
    cout << value << '\n';
}
void print(int value) // compiler error
{
    cout << value << '\n';
}
```

+ Tham số hàm kiểu con trỏ \* và mảng []

```
int foo(int *x);
int foo(int x[]); // compiler error
```

+ Nạp chồng hàm và từ khóa const

### 2.6.2 Nạp chồng toán tử

- Nạp chồng toán tử (Operator Overloading) được dùng để định nghĩa toán tử cho có sẵn trong c++ phục vụ cho dữ liệu riêng do bạn tạo ra.

- Các loại toán tử:

+ C++ chỉ cho phép người dùng overloading lại các toán tử có sẵn trong c++

+ Một toán tử có thể được định nghĩa cho nhiều kiểu dữ liệu khác nhau

+ Toán tử đơn: Toán tử đơn là toán tử một ngôi (unary operator), có thể được dùng làm toán tử trước (prefix operator) và toán tử sau (postfix operator). Ví dụ phép tăng (++) hay phép giảm (--)

+ Toán tử đôi là toán tử có 2 ngôi (binary operator).

+ Các toán tử có thể nạp chồng và không thể nạp chồng:

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Non-Overloadable operators			
.	.*	::	?:

- 2 cách khai báo nạp chồng toán tử: khai báo trong class, dùng friend

- Nạp chồng toán tử nhập, xuất: Việc nạp chồng toán tử nhập xuất cho phép người dùng dùng cin, cout nhập xuất nhanh một đối tượng mà không cần gọi lại cin, cout cho từng thuộc tính của dữ liệu dựa trên việc được định nghĩa trước.

+ Đối với nạp chồng toán tử nhập >> và toán tử xuất << ta sẽ dùng cách nạp chồng toán tử 2 ngôi.

+ Nạp chồng toán tử nhập >>:

```
friend std::istream &operator>>(std::istream &is, Object &obj);
```

+ Nạp chồng toán tử xuất <<:

```
friend std::ostream &operator<<(std::ostream &is, const Object &obj);
```

- Nạp chồng toán tử gán (phải có khi có cấp phát bộ nhớ động, thường đi đôi với Copy Constructor). Nếu ta không định nghĩa lại toán tử =, khi dùng nó, trình biên dịch tự sinh ra hàm gán bằng cách sao chép từng bit dữ liệu của 2 đối tượng cho nhau:

```
Object &operator=(const Object &obj);
```

- Nạp chồng toán tử ép kiểu:

```
operator Type() const;
```

Ví dụ:

```
class Fraction
{
private:
    int numerator = 0;
    int denominator = 1;
```

```

public:
    explicit Fraction() = default;
    Fraction(int numerator, int denominator) : numerator(numerator),
denominator(denominator) {}

    // Rut gon phan so
    Fraction reduce() const
    {
        result Fraction(numerator / __gcd(numerator, denominator),
                        denominator / __gcd(numerator, denominator));
    }

    //Nap chong toan tu cong voi phan so
    Fraction operator+(const Fraction &other) const
    {
        return Fraction(numerator * other.denominator + other.numerator *
denominator,
                        denominator * other.denominator)
.reduce();
    }

    //Nap chong toan tu cong voi so nguyen
    Fraction operator+(const int &other) const
    {
        return Fraction(numerator + other * denominator,
denominator).reduce();
    }

    // Nap chong toan tu lay doi xung
    Fraction operator-() const
    {
        return Fraction(-numerator, denominator).reduce();
    }

    //Nap chong toan tu tru voi phan so
    Fraction operator-(const Fraction &other) const
    {

```

```

        return Fraction(numerator * other.denominator - other.numerator *
denominator,
                        denominator * other.denominator)
        .reduce();
    }

    //Nap chong toan tu tru voi phan so
    Fraction operator-(const int &other) const
    {
        return Fraction(numerator - other * denominator).reduce();
    }

    //Nap chong toan tu nhan voi phan so
    Fraction operator*(const Fraction &other) const
    {
        return Fraction(numerator * other.numerator, denominator *
other.denominator).reduce();
    }

    //Nap chong toan tu nhan voi so nguyen
    Fraction operator*(const int &other) const
    {
        return Fraction(numerator * other, denominator).reduce();
    }

    //Nap chong toan tu chia voi phan so
    Fraction operator/(const Fraction &other) const
    {
        return Fraction(numerator * other.denominator, denominator *
other.numerator).reduce();
    }

    //Nap chong toan tu chia voi so nguyen
    Fraction operator/(const int &other) const
    {
        return Fraction(numerator, denominator * other).reduce();
    }

    //Nap chong toan tu ep kieu sang double

```

```
operator double() const { return (double)numerator / denominator; }
};
```

### 2.6.3 Bài tập

#### **Bài 1.** Hoàn thiện code sau

```
#include <cmath>
#include <iostream>

class Polynomial
{
private:
    float *data = nullptr;
    int sz = 0;

public:
    explicit Polynomial() = default;
    Polynomial(float *arr, int sz);
    Polynomial(int n, float val = 0.0f);
    Polynomial(const Polynomial &v);
    ~Polynomial();

    Polynomial &operator=(const Polynomial &v);

    int size() const; // Bac cua da thuc

    float &operator[](int index); // Lay gia he so ung voi bac la index

    Polynomial operator+(const Polynomial &other); // Cong 2 da thuc
    Polynomial operator-(); // Lay doi cua cac phan tu trong da thuc
    Polynomial operator-(const Polynomial &other); // Tru 2 da thuc
    Polynomial operator*(const Polynomial &other); // Nhan 2 da thuc
    Polynomial operator*(const float &val); // Nhan da thuc voi 1 so
    Polynomial operator/(const float &val); // Chia da thuc voi 1 so
    float operator()(float val); // Tinh gia tri cua da thuc tai val
    Polynomial pow(int n); // Tinh da thuc sau khi mu n lan

    friend std::ostream &operator<<(std::ostream &os, const Polynomial
&pol);
```



```
friend std::istream &operator>>(std::istream &is, Polynomial &pol);
};
```

**Bài 2.** Hoàn thiện code sau:

```
#include <cmath>
#include <iostream>

class Complex
{
private:
    float r = 0.0f;
    float i = 0.0f;

public:
    explicit Complex() = default;
    Complex(int r, int i) : r(r), i(i){};
    ~Complex(){};

    float imag() const { return i; }
    float real() const { return r; }

    Complex conjg(); // So phuc lien hop
    float abs();     // modulus cua so phuc
    float arg();

    Complex operator+(const Complex &comp);
    Complex operator-();
    Complex operator-(const Complex &comp);
    Complex operator*(const Complex &comp);
    Complex operator*(float val);
    Complex operator/(const Complex &comp);
    Complex operator/(float val);

    friend std::istream &operator>>(std::istream &is, Complex &comp);
    friend std::ostream &operator<<(std::ostream &os, const Complex
&comp);
};
```

## 2.7 Tính Kế Thừa

### 2.7.1 Tính kế thừa

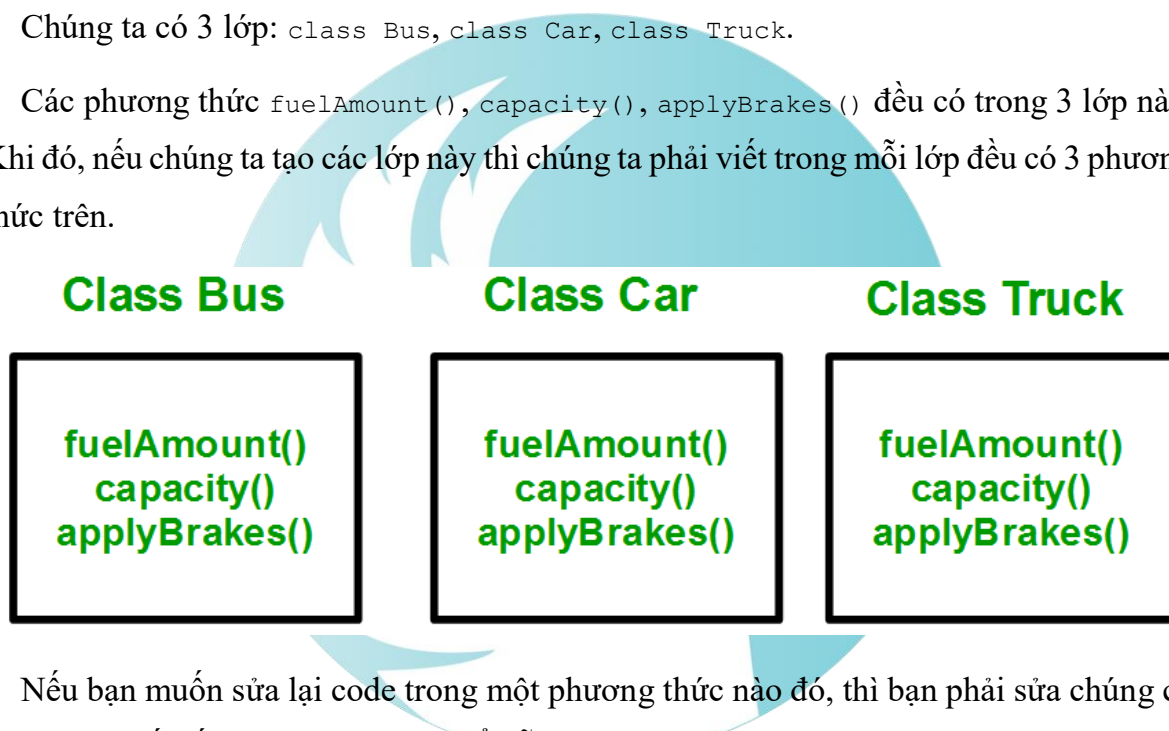
- Tính kế thừa là một trong những đặc tính quan trọng nhất của lập trình hướng đối tượng. Nó là khả năng lấy một thuộc tính, đặc tính của một lớp cha để áp dụng lên lớp con.

- Lớp kế thừa các thuộc tính từ một lớp khác được gọi là Lớp con hoặc Lớp dẫn xuất. Lớp có các thuộc tính được kế thừa bởi lớp con được gọi là Lớp cha hoặc Lớp cơ sở.

### 2.7.2 Tại sao và khi nào cần dùng tính kế thừa

Chúng ta có 3 lớp: `class Bus`, `class Car`, `class Truck`.

Các phương thức `fuelAmount()`, `capacity()`, `applyBrakes()` đều có trong 3 lớp này. Khi đó, nếu chúng ta tạo các lớp này thì chúng ta phải viết trong mỗi lớp đều có 3 phương thức trên.

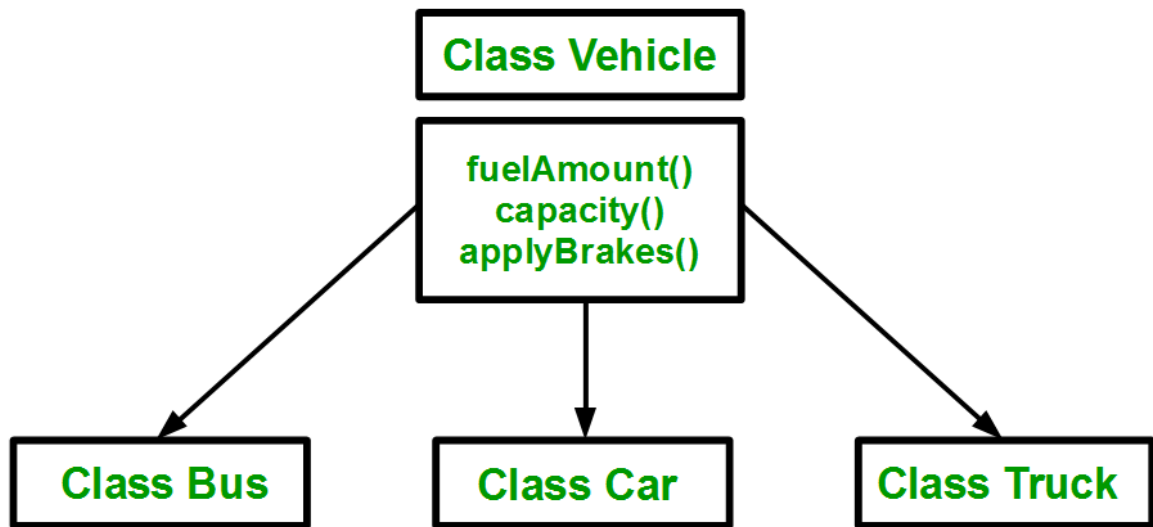


Nếu bạn muốn sửa lại code trong một phương thức nào đó, thì bạn phải sửa chúng cả ở 3 lớp, sẽ rất tốn thời gian và có thể dễ sai sót.

Vì thế để tránh điều này và cũng đảm bảo dữ liệu sẽ không bị dư thừa, tính kế thừa sẽ được sử dụng ở đây.

Khi áp dụng tính kế thừa, đầu tiên ta sẽ tạo một lớp: `class Vehical`. Trong lớp này sẽ có cả 3 phương thức `fuelAmount()`, `capacity()` và `applyBrakes()`.

Sau đó mới tạo 3 lớp: `class Bus`, `class Car`, `class Truck`. Rồi cho 3 lớp này kế thừa từ lớp `class Vehical`.



Điều này sẽ tránh việc sai sót khi sửa và tăng khả năng sử dụng lại.

Khả năng sử dụng lại ở đây là: Nếu bạn muốn thêm một lớp class Taxi chẳng hạn, bạn chỉ cần khai báo nó kế thừa từ `class Vehical` là cũng có thể dùng được 3 phương thức trên.

### 2.7.3 Cú pháp sử dụng

```

class subclass_name : access_mode base_class_name
{
    //body of subclass
};
  
```

Trong đó:

- `subclass_name` là tên lớp con sẽ áp dụng kế thừa
- `base_class_name` là tên lớp cha
- `access_mode` có thể là `public`, `private` hoặc `protected`

### 2.7.4 Các phạm vi kế thừa

- `public`: Nếu kế thừa ở dạng này, sau khi kế thừa, tất cả các thành viên dạng `public` lớp cha sẽ `public` ở lớp con, dạng `protected` ở lớp cha vẫn sẽ là `protected` ở lớp con.
- `protected`: Nếu dùng `protected` thì sau khi kế thừa, tất cả các thành viên dạng `public` lớp cha sẽ trở thành `protected` tại lớp con.
- `private`: Trường hợp ta sử dụng `private`, thì sau khi kế thừa, tất cả các thành viên dạng `public` và `protected` ở lớp cha sẽ thành `private` tại lớp con.

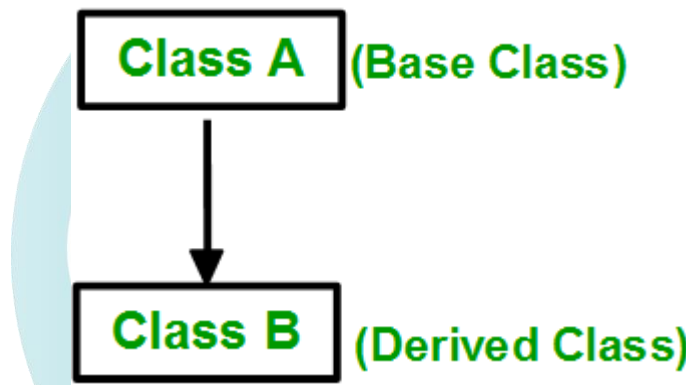
Bảng tóm tắt các phạm vi kế thừa cho cả 3 loại public, protected, và private

Base class member access specifier	Type of Inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accessible

### 2.7.5 Các loại kế thừa trong C++

#### 2.7.5.1 Đơn kế thừa (Single Inheritance)

Đơn kế thừa: nghĩa là một lớp chỉ được kế thừa từ đúng một lớp khác. Hay nói cách khác, lớp con chỉ có duy nhất một lớp cha.



Cú pháp khai báo đơn kế thừa:

```

class subclass_name : access_mode base_class
{
    //body of subclass
};
  
```

Ví dụ:

```

#include <iostream>
using namespace std;

class Vehicle
{
public:
    Vehicle()
    {
  
```

```

        cout << "This is a Vehicle" << endl;
    }
};

class Car : public Vehicle
{
};

int main()
{
    Car obj;
    return 0;
}

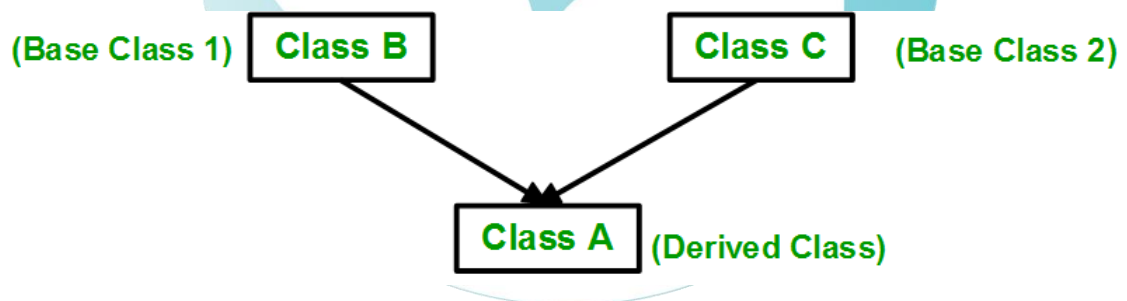
```

Kết quả:

```
This is a vehicle
```

#### 2.7.5.2 Đa kế thừa (Multiple Inheritance)

Đa kế thừa là một tính năng của ngôn ngữ C++. Trong đó một lớp có thể kế thừa từ nhiều hơn một lớp khác. Nghĩa là một lớp con được kế thừa từ nhiều hơn một lớp cơ sở.



Cú pháp khai báo:

```

class subclass_name : access_mode base_class1, access_mode base_class2,
....
{
    //body of subclass
};

```

Ở đây, các lớp cơ sở sẽ được phân tách bằng dấu phẩy , và phạm vi truy cập cho mọi lớp cơ sở phải được chỉ định.

Ví dụ 1:


```
#include <iostream>
using namespace std;

class A
{
public:
    A() { cout << "A's constructor called" << endl; }
};

class B
{
public:
    B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A // Note the order
{
public:
    C() { cout << "C's constructor called" << endl; }
};

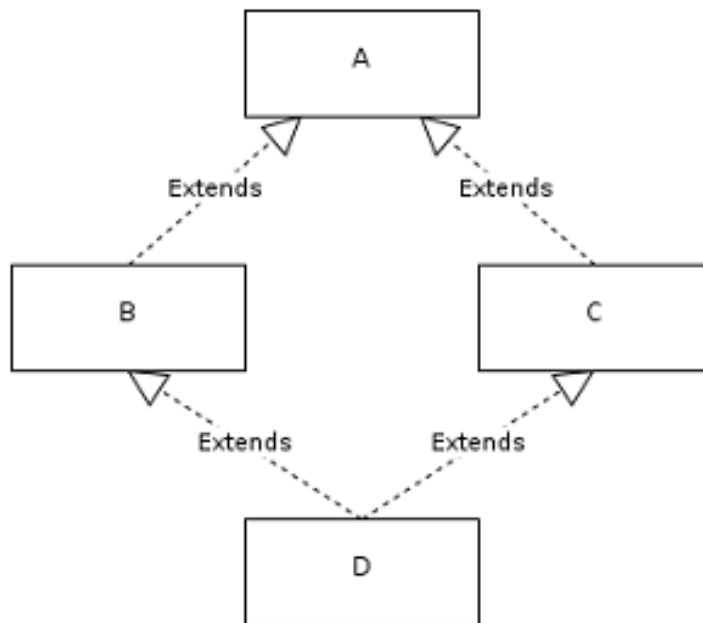
int main()
{
    C c;
    return 0;
}
```



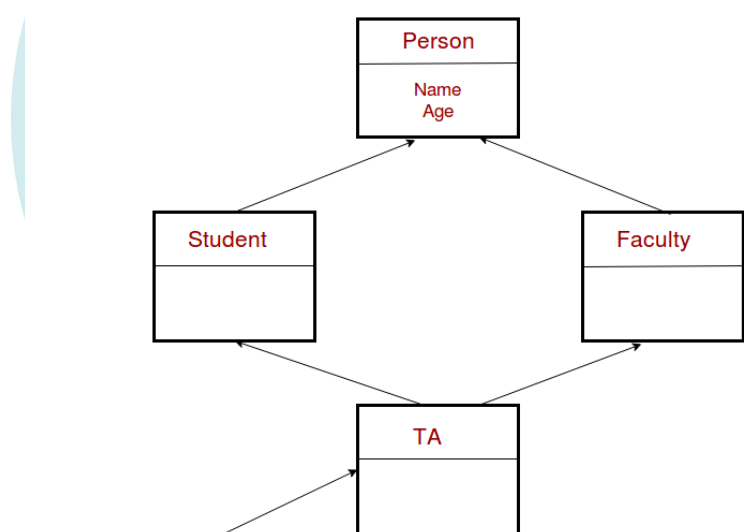
Kết quả:

```
B's constructor called
A's constructor called
C's constructor called
```

"diamond problem" (đôi khi được gọi là "Deadly Diamond of Death" – Kim cương chết chóc) là một sự mơ hồ sinh ra khi 2 lớp B và C kế thừa từ lớp A, và lớp D kế thừa từ cả B và C



Ta xét ví dụ sau: Lớp Student và lớp Faculty kế thừa từ lớp Person. Lớp TA kế thừa từ lớp Student và Faculty



Name and Age needed only once

```

#include <iostream>
using namespace std;
class Person {
// Data members of person
public:
    Person(int x) { cout << "Person::Person(int) called" << endl; }
};

class Faculty : public Person {

```

```
// data members of Faculty
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int) called"<< endl;
    }
};

class Student : public Person {
// data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

### Kết quả:

```
Person::Person(int) called
Faculty::Faculty(int) called
Person::Person(int) called
Student::Student(int) called
TA::TA(int) called
```

Trong chương trình trên, hàm tạo của ‘Person’ được gọi hai lần. Bộ hủy của ‘Person’ cũng sẽ được gọi hai lần khi đối tượng ‘ta1’ bị hủy. Vì vậy, đối tượng ‘ta1’ có hai bản sao của tất cả các thành viên của ‘Person’, điều này gây ra sự mơ hồ. Giải pháp cho vấn đề này là từ khóa virtual. Chúng tôi đặt các lớp ‘Faculty’ và ‘Student’ làm lớp cơ sở ảo để tránh hai bản sao của ‘Person’ trong lớp ‘TA’. Ví dụ, hãy xem xét chương trình sau:



```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int) called" << endl; }
    Person()      { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

### Kết quả:

```
Person::Person() called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

Trong chương trình trên, hàm tạo của ‘Person’ được gọi một lần. Một điều quan trọng cần lưu ý trong kết quả ở trên là, hàm tạo mặc định của ‘Person’ được gọi. Khi chúng ta sử dụng từ khóa ‘virtual’, hàm tạo mặc định của lớp ông được gọi theo mặc định ngay cả khi các lớp cha gọi hàm tạo tham số một cách rõ ràng.

Làm thế nào để gọi hàm tạo tham số của lớp ‘Person’? Hàm tạo phải được gọi trong lớp ‘TA’. Ví dụ, hãy xem chương trình sau đây:

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int) called" << endl; }
    Person() { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x) : Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

}

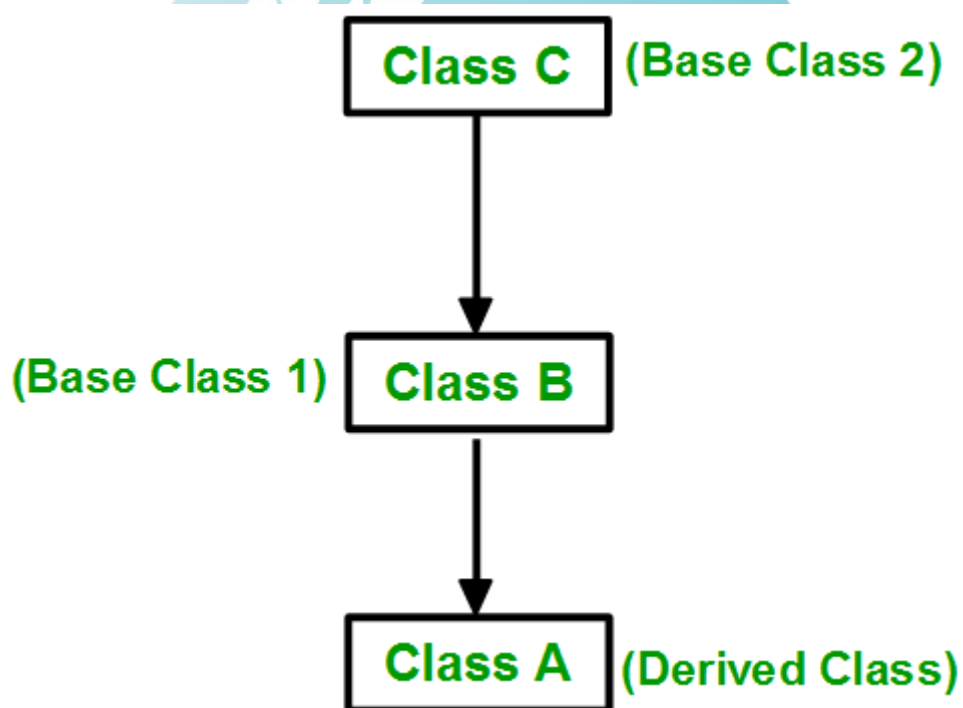
Kết quả:

```
Person::Person(int) called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

Nói chung, không được phép gọi hàm tạo của ông trực tiếp, nó phải được gọi thông qua lớp cha. Nó chỉ được phép khi từ khóa "virtual" được sử dụng.

### 2.7.5.3 Kế thừa đa cấp (Multilevel Inheritance)

Kế thừa đa cấp: Trong kiểu thừa kế này, một lớp dẫn xuất được tạo từ một lớp dẫn xuất khác.



Ví dụ:

```
#include <iostream>
using namespace std;

class Vehicle
{
public:
```

```
Vehicle()
{
    cout << "This is a Vehicle" << endl;
}

};

class fourWheeler : public Vehicle
{
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles" << endl;
    }
};

class Car : public fourWheeler
{
public:
    car()
    {
        cout << "Car has 4 Wheels" << endl;
    }
};

int main()
{
    Car obj;
    return 0;
}
```

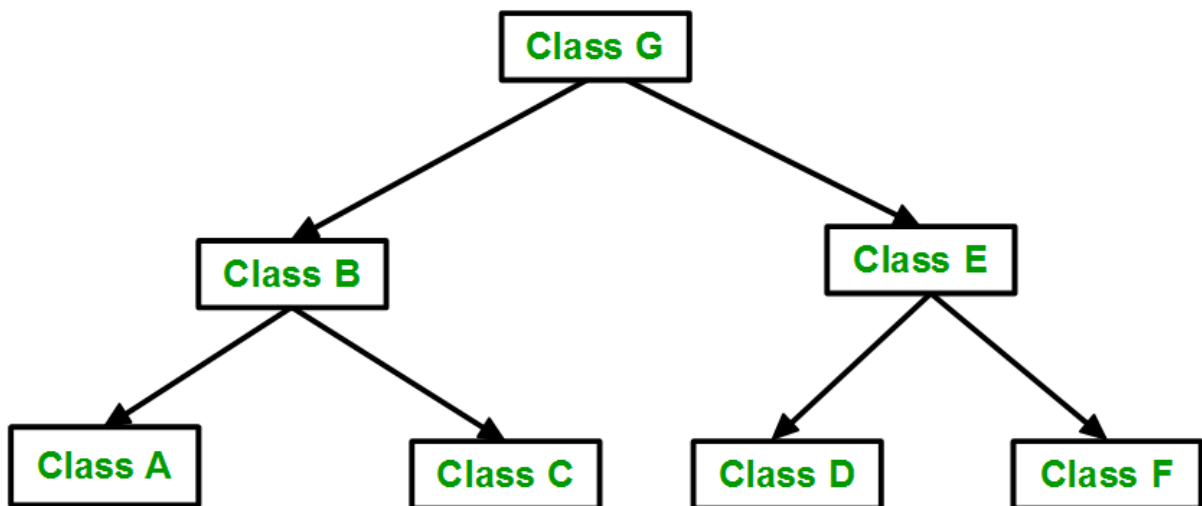


**Kết quả:**

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

#### 2.7.5.4 Kế thừa phân cấp (Hierarchical Inheritance)

**Kế thừa phân cấp:** Trong kiểu thừa kế này, sẽ có nhiều hơn một lớp con được kế thừa từ một lớp cha duy nhất.



Ví dụ:

```
#include <iostream>
using namespace std;

class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler : public Vehicle
{
public:
    FourWheeler()
    {
        cout << "Object with 4 wheels are vehicle" << endl;
    }
};

class MultiWheeler : public Vehicle
{
public:
    MultiWheeler()
    {
```

```
        cout << "Object with more 4 wheels are vehicle" << endl;
    }
};

class Car : public FourWheeler
{
};

class MiniCar : public FourWheeler
{
};

class Bus : public MultiWheeler
{
};

class Container : public MultiWheeler
{
};

int main()
{
    Car car;
    cout << endl;

    MiniCar miniCar;
    cout << endl;

    Bus bus;
    cout << endl;

    Container container;
    cout << endl;

    return 0;
}
```



Kết quả:

This is a Vehicle

Object with 4 wheels are vehicle

This is a Vehicle

Object with 4 wheels are vehicle

This is a Vehicle

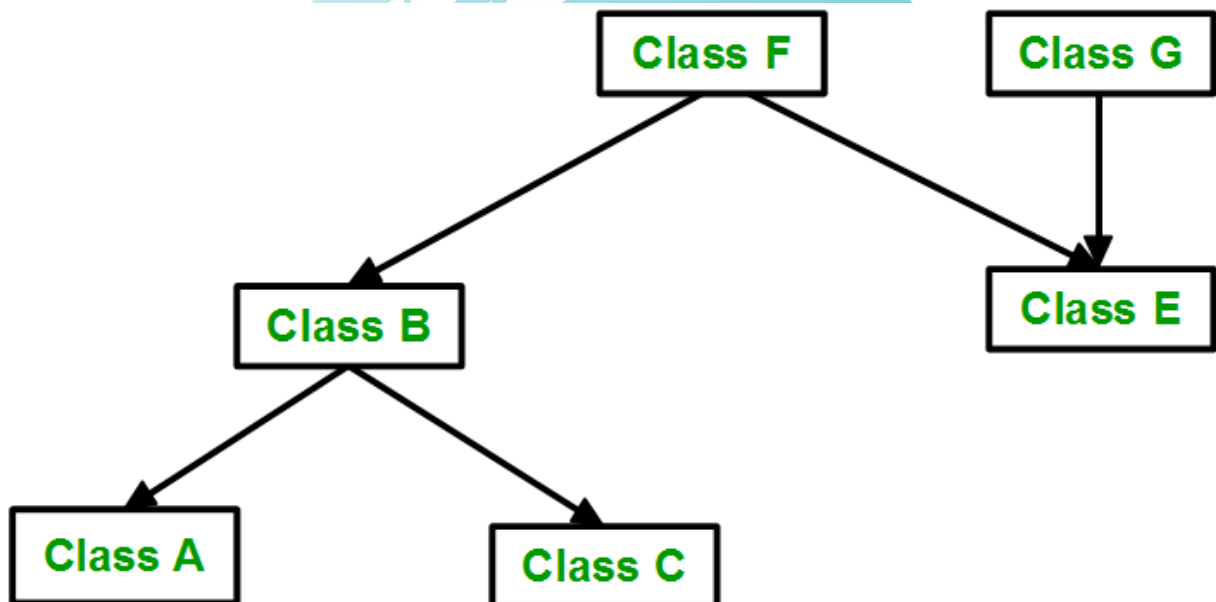
Object with more 4 wheels are vehicle

This is a Vehicle

Object with more 4 wheels are vehicle

### 2.7.5.5 Kế thừa lai (Kế thừa ảo) – Hybrid (Virtual) Inheritance

Kế thừa lai (Kế thừa ảo): được thực hiện bằng cách kết hợp nhiều hơn một loại thừa kế. Ví dụ: Kết hợp kế thừa phân cấp và đa kế thừa.



Ví dụ:

```

#include <iostream>
using namespace std;

class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
}
  
```

```
    }  
};  
  
class Fare  
{  
public:  
    Fare()  
    {  
        cout << "Fare of Vehicle\n";  
    }  
};  
  
class Car : public Vehicle  
{  
};  
  
class Bus : public Vehicle, public Fare  
{  
};  
  
int main()  
{  
    Bus obj2;  
    return 0;  
}
```

Kết quả:

```
This is a Vehicle  
Fare of Vehicle
```

### 2.7.6 Bài tập

#### **Bài 1.**

Tạo class People gồm:

- Thuộc tính: name, age, address để lưu lần lượt các giá trị tên, tuổi và địa chỉ.
- Phương thức trong class People gồm: set(), get() là hàm nhập và xuất; hàm khởi tạo không tham số và hàm hủy.



Tạo class Students kế thừa từ class People.

Class Students sẽ có thêm:

- Thuộc tính id để lưu mã sinh viên, math lưu điểm môn toán, physical để lưu điểm môn vật lý, chemistry để lưu điểm môn hoá học.
- Phương thức: set(), get(), GPA() để tính điểm trung bình 3 môn học

## **Bài 2.**

Xây dựng lớp Color gồm:

- Thuộc tính: TenMau, MaMau
- Phương thức:
  - Hàm tạo không tham số
  - Hàm tạo có tham số
  - Hàm hủy
  - Nạp chồng toán tử nhập
  - Nạp chồng toán tử xuất
  - getTenMau() : hàm trả về TenMau

Xây dựng lớp Point gồm:

- Thuộc tính: int x, y
- Phương thức:
  - Hàm tạo không tham số
  - Hàm tạo có tham số
  - Hàm hủy
  - Nạp chồng toán tử nhập
  - Nạp chồng toán tử xuất
  - CheoChinh: hàm kiểm tra Point có thuộc đường chéo chính hay không (1 điểm thuộc đường chéo chính khi và chỉ khi tung độ bằng hoành độ).

Xây dựng lớp Pixel kế thừa từ lớp Color và Point bao gồm thêm:

- Phương thức:
  - Hàm tạo không tham số
  - Hàm tạo có tham số
  - Nạp chồng toán tử nhập

- Nạp chồng toán tử xuất
- KiemTra: hàm kiểm tra Pixel thuộc đường chéo chính và có màu “Xanh” hay không?

Chương trình chính: Nhập vào từ bàn phím n Pixel (n nhập từ bàn phím). Hiển thị thông tin các Pixel thuộc đường chéo chính và có màu xanh.

**Bài 3.** Dự đoán kết quả của các chương trình sau

a)

```
#include<iostream>
using namespace std;

class A
{
    int x;
public:
    void setX(int i) {x = i;}
    void print() { cout << x; }
};

class B : public A
{
public:
    B() { setX(10); }
};

class C : public A
{
public:
    C() { setX(20); }
};

class D : public B, public C {
};

int main()
{
    D d;
```



```
d.print();  
return 0;  
}
```

b)

```
#include <iostream>  
using namespace std;  
  
class A  
{  
    int x;  
public:  
    A(int i) { x = i; }  
    void print() { cout << x; }  
};  
  
class B : virtual public A  
{  
public:  
    B() : A(10) { }  
};  
  
class C : virtual public A  
{  
public:  
    C() : A(10) { }  
};  
  
class D : public B, public C {  
};  
  
int main()  
{  
    D d;  
    d.print();  
    return 0;  
}
```

## 2.8 Tính đa hình

### 2.8.1 Định nghĩa

Từ đa hình có nghĩa là có nhiều dạng. Nói một cách đơn giản, chúng ta có thể định nghĩa đa hình là khả năng của một thông điệp được hiển thị dưới nhiều dạng.

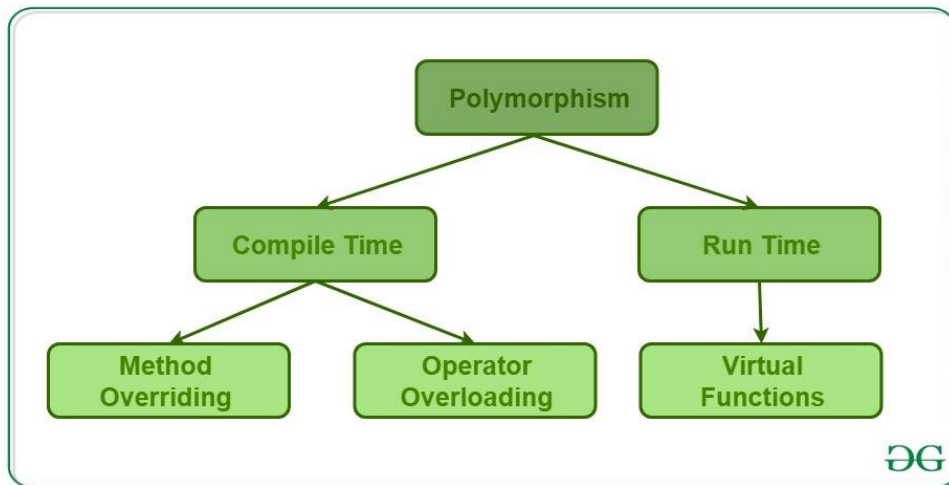
Ví dụ: Một người cùng một lúc có thể có đặc điểm khác nhau. Giống như một người đàn ông đồng thời là một người cha, một người chồng, một nhân viên. Vì vậy, cùng một người sở hữu những hành vi khác nhau trong các tình huống khác nhau. Điều này được gọi là đa hình.

Đa hình được coi là một trong những tính năng quan trọng của Lập trình hướng đối tượng.

### 2.8.2 Phân loại

Trong ngôn ngữ C ++, tính đa hình chủ yếu được chia thành hai loại:

- Compile time Polymorphism
- Runtime Polymorphism



#### 2.8.2.1 Compile time Polymorphism

Tính đa hình này được sử dụng bằng cách nạp chồng hàm hoặc nạp chồng toán tử như đã nói ở trên

#### 2.8.2.2 Runtime Polymorphism

##### a) Upcasting và Downcasting

```

#include <iostream>
using namespace std;
class Person
{
    //content of Person
};
class Employee : public Person
{
public:
    Employee(string fName, string lName, double sal)
    {
        FirstName = fName;
        LastName = lName;
        salary = sal;
    }
    string FirstName;
    string LastName;
    double salary;
    void show()
    {
        cout << "First Name: " << FirstName << " Last Name: " << LastName
        << " Salary: " << salary << endl;
    }
    void addBonus(double bonus)
    {
        salary += bonus;
    }
};
class Manager : public Employee
{
public:
    Manager(string fName, string lName, double sal, double comm) :
    Employee(fName, lName, sal)
    {
        Commision = comm;
    }
    double Commision;
    double getComm()
    {

```

```

        return Commision;
    }
};

class Clerk : public Employee
{
public:
    Clerk(string fName, string lName, double sal, Manager *man) :
Employee(fName, lName, sal)
    {
        manager = man;
    }
    Manager *manager;
    Manager *getManager()
    {
        return manager;
    }
};

void congratulate(Employee *emp)
{
    cout << "Happy Birthday!!!" << endl;
    emp->addBonus(200);
    emp->show();
};

int main()
{
    //pointer to base class object
    Employee *emp;

    //object of derived class
    Manager m1("Steve", "Kent", 3000, 0.2);
    Clerk c1("Kevin", "Jones", 1000, &m1);

    //implicit upcasting
    emp = &m1;

    //It's ok
    cout << emp->FirstName << endl;
    cout << emp->salary << endl;
}

```



```
//Fails because upcasting is used
//cout<<emp->getComm();

congratulate(&c1);
congratulate(&m1);

cout << "Manager of " << c1.FirstName << " is " << c1.getManager()-
>FirstName;
}
```

- Upcasting: là một quá trình tạo một con trỏ hoặc một tham chiếu của đối tượng lớp dẫn xuất như một con trỏ lớp cơ sở. Bạn không cần phải upcast theo cách thủ công. Bạn chỉ cần gán con trỏ lớp dẫn xuất (hoặc tham chiếu) cho con trỏ lớp cơ sở.

```
//pointer to base class object
Employee* emp;
//object of derived class
Manager m1("Steve", "Kent", 3000, 0.2);
//implicit upcasting
emp = &m1;
```

Khi bạn sử dụng upcasting, đối tượng không thay đổi. Tuy nhiên, khi bạn upcast một đối tượng, bạn sẽ chỉ có thể truy cập các hàm thành viên và các thành viên dữ liệu được xác định trong lớp cơ sở:

```
//It's ok
emp->FirstName;
emp->salary;
//Fails because upcasting is used
emp->getComm();
```

### Ví dụ về sử dụng upcasting

```
void congratulate(Employee* emp)
{
    cout << "Happy Birthday!!!" << endl;
    emp->show();
    emp->addBonus(200);
};
```

Hàm này sẽ hoạt động với tất cả các lớp có nguồn gốc từ lớp Nhân viên. Khi bạn gọi nó với các đối tượng kiểu Manager và Person, chúng sẽ tự động được chuyển đến lớp Employee

```
//automatic upcasting
congratulate(&c1);
congratulate(&m1);
```

- Downcasting: là một quá trình ngược lại với upcasting. Nó chuyển đổi con trỏ lớp cơ sở thành con trỏ lớp dẫn xuất. Việc downcasting phải được thực hiện thủ công. Có nghĩa là bạn phải chỉ định typecast rõ ràng. Downcasting không an toàn như upcast. Bạn biết rằng một đối tượng lớp dẫn xuất luôn có thể được coi là một đối tượng lớp cơ sở. Tuy nhiên, điều ngược lại là không đúng. Ví dụ, một Nhà quản lý luôn luôn là một Người; nhưng một Người không phải lúc nào cũng là Người quản lý vì người ấy cũng có thể là một Thư ký.

```
//pointer to base class object
Employee* emp;
//object of derived class
Manager m1("Steve", "Kent", 3000, 0.2);
//implicit upcasting
emp = &m1;
//explicit downcasting from Employee to Manager
Manager* m2 = (Manager*) (emp);
```

Code này biên dịch và chạy mà không gặp bất kỳ sự cố nào vì emp trỏ đến một đối tượng của lớp Manager.

Điều gì sẽ xảy ra, nếu chúng ta cố gắng downcasting một con trỏ lớp cơ sở đang trỏ đến một đối tượng của lớp cơ sở chứ không phải đến một đối tượng của lớp dẫn xuất? Biên dịch và chạy code này:

```
Employee e1("Peter", "Green", 1400);
//try to cast an employee to Manager
Manager* m3 = (Manager*) (&e1);
cout << m3->getComm() << endl;
```

b) Tính đa hình được thể hiện ở cách nạp chồng hàm trong kế thừa.

```
#include <bits/stdc++.h>
```



```
using namespace std;

class base
{
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show()
    {
        cout << "show base class" << endl;
    }
};

class derived : public base
{
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    bptr->print();
    bptr->show();

    return 0;
}
```



```
}
```

Sau biên dịch, ta được kết quả:

```
print derived class
show base class
```

Trong ví dụ trên mình đã thêm từ khóa `virtual` vào hàm `print()` trong lớp cơ sở `base`. Từ khóa `virtual` này dùng để khai báo một hàm là hàm ảo.

Khi khai báo hàm ảo với từ khóa `virtual` nghĩa là hàm này sẽ được gọi theo loại đối tượng được trả (hoặc tham chiếu), chứ không phải theo loại của con trả (hoặc tham chiếu). Và điều này dẫn đến kết quả khác nhau:

- Nếu không khai báo hàm ảo `virtual` trình biên dịch sẽ gọi hàm tại lớp cơ sở `base`
- Nếu dùng hàm ảo `virtual` trình biên dịch sẽ gọi hàm tại lớp dẫn xuất `derived`

Mục đích của hàm ảo: Các hàm ảo sẽ cho phép chúng ta tạo một danh sách các con trả lớp cơ sở và các phương thức của bất kỳ lớp dẫn xuất nào mà không cần biết loại đối tượng của lớp dẫn xuất.

Trong C++, các hàm ảo có thể là `private` và có thể bị ghi đè bởi lớp dẫn xuất. Ví dụ, chương trình sau đây:

```
#include<iostream>
using namespace std;

class Derived;

class Base {
private:
    virtual void fun() { cout << "Base Fun"; }
friend int main();
};

class Derived: public Base {
public:
    void fun() { cout << "Derived Fun"; }
};

int main()
```

```
{
Base *ptr = new Derived;
ptr->fun();
return 0;
}
```

### c) Các hàm ảo thuần túy và các lớp trừu tượng trong C++

Đôi khi việc thực hiện tất cả các hàm trong lớp cơ sở là không thể vì chúng ta không biết việc thàn đó thực hiện như thế nào. Ví dụ, lấy Shape là lớp cơ sở, ta không biết hàm draw() thực hiện như thế nào, nhưng ta biết nhưng lớp dẫn xuất của nó như Hexagon, Square,... đều có thể thực hiện hàm draw(). Khi đó Shape được gọi là lớp trừu tượng (Abstract Class). Lớp trừu tượng không thể tạo đối tượng.

Một hàm thuần ảo (hoặc hàm trừu tượng) trong C++ là một hàm ảo mà không có thân hàm, ta chỉ khai báo nó. Một hàm thuần khai báo bằng cách gán 0 trong khai báo.

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Chú ý:

- Một lớp là trừu tượng nếu nó có ít nhất một hàm thuần ảo.
- Chúng ta có thể có các con trỏ và tham chiếu kiểu lớp trừu tượng
- Nếu chúng ta không nạp chồng hàm thuần ảo trong lớp dẫn xuất, thì lớp dẫn xuất cũng trở thành lớp trừu tượng
- Một lớp trừu tượng có thể có các hàm tạo.

Một giao diện (Interface) trong C++ khi tất cả các phương thức là hàm thuần ảo:

```
#include <iostream>
using namespace std;
```


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

class Animal
{
public:
    virtual void speak() = 0;
};

class Vehicle
{
public:
    virtual void numberOfWheels() = 0;
    virtual void nameOfVehicle() = 0;
};

class Cat : public Action, public Animal
{
public:
    void run()
    {
        cout << "The cat can run!" << endl;
    }
    void speak()
    {
        cout << "The cat speak \"Meo Meo\\\"!" << endl;
    }
};

class Car : public Action, public Vehicle
{
public:
    void run()
    {
        cout << "The car can run!" << endl;
    }
}
```



```

void numberOfWheels ()
{
    cout << "The car has 4 wheels" << endl;
}

void nameOfVehicle ()
{
    cout << "Honda" << endl;
}
};

int main ()
{
    Car car;
    car.run ();
    Cat cat;
    cat.run ();
}

```

#### d) So sánh giữa hàm ảo và hàm thuần ảo

##### - Giống nhau:

- Đều là Runtime Polymorphism
- Khai báo của cả hai hàm này được giữ nguyên trong suốt chương trình.
- Hàm này không khai báo global hay static được

##### - Khác nhau:

Hàm ảo	Hàm thuần ảo
Một hàm ảo là một hàm thành viên của lớp cơ sở có thể được định nghĩa lại bởi lớp dẫn xuất.	Một hàm thuần ảo là một hàm thành viên của lớp cơ sở mà phần khai báo duy nhất được cung cấp trong lớp cơ sở và phải được định nghĩa trong lớp dẫn xuất nếu không lớp dẫn xuất cũng trở thành trừu tượng.
Các lớp có hàm ảo không trừu tượng.	Lớp cơ sở chứa hàm thuần ảo trở thành trừu tượng – lớp trừu tượng

Cú pháp: <code>virtual&lt;func_type&gt;&lt;func_name&gt;() { //code }</code>	Cú pháp: <code>virtual&lt;func_type&gt;&lt;func_name&gt;() = 0;</code>
Định nghĩa được đưa ra trong lớp cơ sở.	Không có định nghĩa nào được đưa ra trong lớp cơ sở.
Lớp cơ sở có hàm ảo có thể được khởi tạo, tức là có thể tạo đối tượng của nó.	Lớp cơ sở có hàm thuần ảo trở thành lớp trừu tượng, tức là không thể khởi tạo nó.
Nếu lớp dẫn xuất không định nghĩa lại hàm ảo của lớp cơ sở, thì nó không ảnh hưởng đến quá trình biên dịch.	Nếu lớp dẫn xuất không xác định lại hàm thuần ảo của lớp cơ sở thì không có lỗi biên dịch mà lớp dẫn xuất cũng trở thành lớp trừu tượng giống như lớp cơ sở.
Tất cả các lớp dẫn xuất có thể hoặc không định nghĩa lại hàm ảo của lớp cơ sở.	Tất cả các lớp dẫn xuất phải định nghĩa lại hàm thuần ảo của lớp cơ sở, nếu không lớp dẫn xuất cũng trở thành lớp trừu tượng giống như lớp cơ sở.

## e) Hàm hủy ảo

- Ví dụ:

```
#include <iostream>
using namespace std;

class Base
{
public:
    ~Base() { cout << "Base's destructor\n"; }
};

class Derived : public Base
{
public:
    ~Derived() { cout << "Derived's destructor\n"; }
};

int main()
{
    Base *base = new Derived();
    delete base;
}
```

```
    return 0;
}
```

**Kết quả:**

```
Base's destructor
```

Khi upcasting, không có từ khoá `virtual`, phương thức gọi tới luôn là phương thức của lớp cơ sở (do khi biên dịch thì trình biên dịch không thể xác định được con trỏ của lớp cơ sở trỏ tới đối tượng nào) nên khi gọi toán tử `delete` thì destructor được gọi là destructor của lớp cha và destructor của lớp con không được gọi dẫn tới sự không an toàn cho chương trình (do tài nguyên của đối tượng thuộc lớp con không được giải phóng, gây rò rỉ bộ nhớ). Để giải quyết vấn đề này, ta khai báo từ khoá `virtual` cho destructor của lớp cơ sở

```
#include <iostream>
using namespace std;

class Base
{
public:
    virtual ~Base() { cout << "Base's destructor\n"; }
};

class Derived : public Base
{
public:
    ~Derived() { cout << "Derived's destructor\n"; }
};

int main()
{
    Base *base = new Derived();
    delete base;
    return 0;
}
```

**Kết quả:**

```
Derived's destructor
```

Base's destructor

Lúc này, bộ nhớ đã được giải phóng hoàn toàn. Nếu muốn tạo ra một lớp và cho phép các lớp khác kế thừa, và sử dụng upcasting thì luôn luôn khai báo virtual destructor

### 2.8.3 Cơ chế Virtual Table

#### 2.8.3.1 Định nghĩa

Để triển khai các hàm ảo, C++ sử dụng một dạng liên kết trễ (late binding) đặc biệt được gọi là bảng ảo (virtual table). Bảng ảo là một bảng tra cứu các hàm được sử dụng để giải quyết các lệnh gọi hàm theo cách liên kết động/trễ (dynamic/late binding manner). Bảng ảo đôi khi có các tên khác, chẳng hạn như “vtable”, “bảng chức năng ảo” (virtual function table), “bảng phương thức ảo”(virtual method table) hoặc “bảng điều phối”(dispatch table).

```
class Base
{
public:
    virtual void function1() { cout << "Base: function1.\n"; };
    virtual void function2() { cout << "Base: function2.\n"; };
};

class D1: public Base
{
public:
    virtual void function1() { cout << "D1: function1.\n"; };
};

class D2: public Base
{
public:
    virtual void function2() { cout << "D2: function2.\n"; };
};

int main() {
    Base b = new Base();
    Base d1 = new D1();
    Base d2 = new D2();
}
```



```

    b.function1();
    b.function2();

    d1.function1();
    d2.function2();

    return 0;
}

```

**Kết quả:**

```

Base: function1.
Base: function2.
D1: function1.
D2: function2.

```

Cơ chế virtual table cho phép thực hiện việc gọi đến hàm function1(), function2 (được định nghĩa trong class D1, D2) của đối tượng d1, d2.

### 2.8.3.2 Cách thức hoạt động

#### a) Con trỏ \_vptr

- Con trỏ \_vptr là con trỏ được tạo ra mỗi khi một đối tượng (class định nghĩa đối tượng này có khai báo virtual function) được tạo ra khai báo.

- Khi một đối tượng được tạo ra, đồng thời con trỏ \_vptr cũng sẽ được tạo ra để trỏ đến virtual table của đối tượng đó. vd: Đối tượng b được tạo ra, thì con trỏ \_vptr cũng được khai báo để trỏ tới virtual table của đối tượng b.

#### b) Virtual table được tạo ra thế nào

- Trong vd trên, vì ở lớp Base có 2 hàm được định nghĩa virtual nên mỗi bảng sẽ có 2 mục (1 cho function1() và 1 cho function2()).

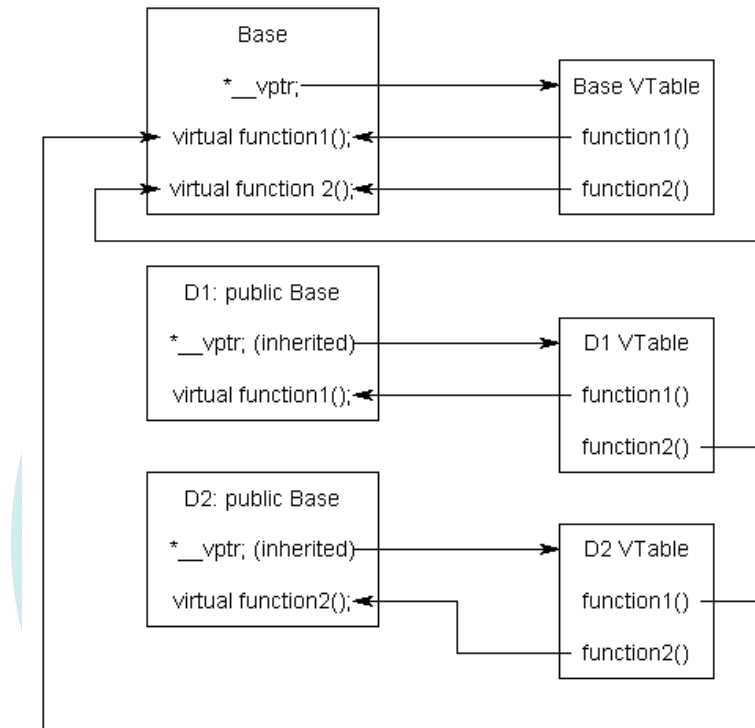
- Bảng ảo của các đối tượng Base rất đơn giản, một đối tượng kiểu Base chỉ có thể truy cập các thành viên của class Base, nên mục nhập function1() trỏ đến Base::function1() và mục nhập cho function2() trỏ đến Base::function2().

- Bảng ảo cho các đối tượng D1 sẽ phức tạp hơn, một đối tượng kiểu D1 có thể truy cập tới các thành viên của class Base và class D1. Hàm function1() đã được ghi đè trong

class D1, nên mục nhập cho function1() sẽ trỏ đến D1::function1(), còn hàm function2() không được định nghĩa nên mục nhập cho function2() trỏ đến Base::function2().

- Tương tự, các đối tượng kiểu D2, mục nhập function1() trỏ đến Base::function1(), và mục nhập cho function2() trỏ đến D2::function2().

Khi gọi function1() và function2() của các đối tượng kiểu D1 và D2 con trỏ \_vptr thực hiện nhiệm vụ để gọi chính xác hàm cần gọi.



### 2.8.3.3 Ưu điểm, nhược điểm

a) Ưu điểm: giải quyết trọn vẹn việc xử lý các hàm ảo, ngay cả khi chỉ sử dụng một con trỏ hoặc tham chiếu đến một lớp cơ sở

b) Nhược điểm:

- Việc gọi hàm ảo sẽ chậm việc gọi các hàm thông thường vì một số lý do sau:
  - Việc sử dụng con trỏ \_vptr để đến bản ảo và gọi hàm thích hợp
  - Phải lập chỉ mục bảng ảo để tìm đúng hàm cần gọi, sau đó mới có thể gọi đúng hàm
- Gây tốn tài nguyên do mỗi đối tượng sẽ có thêm một con trỏ.

#### 2.8.4 Bài tập

**Bài 1.** Một đơn vị sản xuất gồm có các cán bộ là công nhân, kỹ sư, nhân viên. Mỗi cán bộ cần quản lý các dữ liệu: Họ tên, tuổi, giới tính(name, nữ, khác), địa chỉ.

- Cấp công nhân sẽ có thêm các thuộc tính riêng: Bậc (1 đến 10).
- Cấp kỹ sư có thuộc tính riêng: Ngành đào tạo.
- Các nhân viên có thuộc tính riêng: công việc.

Yêu cầu 1: Xây dựng các lớp CongNhan, KySu, NhanVien kế thừa từ lớp CanBo.

Yêu cầu 2: Xây dựng lớp QLCB(quản lý cán bộ) cài đặt các phương thức thực hiện các chức năng sau:

- Thêm mới cán bộ.
- Tìm kiếm theo họ tên.
- Hiện thị thông tin về danh sách các cán bộ.
- Thoát khỏi chương trình.

**Bài 2.** Các thí sinh dự thi đại học bao gồm các thí sinh thi khối A, B, và khối C. Các thí sinh cần quản lý các thông tin sau: Số báo danh, họ tên, địa chỉ, mức ưu tiên.

Thí sinh thi khối A thi các môn: Toán, Lý, Hoá.

Thí sinh thi khối B thi các môn: Toán, Hoá, Sinh.

Thí sinh thi khối C thi các môn: Văn, Sử, Địa.

Yêu cầu 1: Xây dựng các lớp để quản lý các thí sinh dự thi đại học.

Yêu cầu 2: Xây dựng lớp TuyenSinh có các chức năng:

- Thêm mới thí sinh.
- Hiện thị thông tin của thí sinh và khối thi của thí sinh.
- Tìm kiếm theo số báo danh.
- Thoát khỏi chương trình.

**Bài 3.** Một thư viện cần quản lý các tài liệu bao gồm Sách, Tạp chí, Báo. Mỗi tài liệu gồm có các thuộc tính sau: Mã tài liệu(Mã tài liệu là duy nhất), Tên nhà xuất bản, số bản phát hành.

Các loại sách cần quản lý thêm các thuộc tính: tên tác giả, số trang.

Các tạp chí cần quản lý thêm: Số phát hành, tháng phát hành.

Các báo cần quản lý thêm: Ngày phát hành.

Yêu cầu 1: Xây dựng các lớp để quản lý tài liệu cho thư viện một cách hiệu quả.

Yêu cầu 2: Xây dựng lớp QuanLySach có các chức năng sau

- Thêm mới tài liệu: Sách, tạp chí, báo.
- Xoá tài liệu theo mã tài liệu.
- Hiện thị thông tin về tài liệu.
- Tìm kiếm tài liệu theo loại: Sách, tạp chí, báo.
- Thoát khỏi chương trình.

## 2.9 Template

- Template: là một khuôn mẫu, đại diện cho nhiều kiểu dữ liệu khác nhau
- Có 2 loại template: Function template, Class template

### 2.9.1 Function template

Function template là 1 function đặc biệt, có thể sử dụng được cho nhiều hơn 1 kiểu tham số truyền vào (khác với override function cần phải định nghĩa nhiều hàm với các kịch bản param được cố định. Function template chỉ cần định nghĩa 1 lần).

Trong C++, chúng ta có thể hoàn toàn sử dụng template parameters, để viết các hàm cần sử dụng cho nhiều kiểu tham số truyền vào.

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

Ví dụ:

```
template <class T>
T getMax (T x, T y) {
    return x > y ? x : y;
}
```

Ở đây một hàm đã được tạo ra với T làm parameters template của nó. Parameters template này đại diện cho một loại chưa được chỉ định, nhưng có thể được sử dụng trong function template như thể đó là một kiểu tham số thông thường (kiểu tham số thông thường như int, double, ...). Như chúng ta có thể thấy, hàm GetMax trả về giá trị lớn hơn của hai tham số của loại vẫn chưa được xác định này.

Sử dụng function template, theo dõi việc gọi và sử dụng nó:

```
int x = 5, y = 8;
getMax <int> (5,8);
```

Theo dõi ví dụ:

```
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a > b) ? a : b;
    return (result);
}

int main () {
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMax<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

Trong ví dụ này, trước khi chúng ta sử dụng function template getMax(), ở lần gọi đầu tiên sử dụng với kiểu int và lần gọi thứ 2 sử dụng với tham số kiểu long. Mỗi lần gọi đều xác định kiểu tham số trong dấu <> (<int>, <long>).

Tuy nhiên, trên thực tế thì khi thực hiện compiler, trình compiler có thể tự động xác định kiểu dữ liệu truyền vào mà không cần người gọi phải xác định trước (bằng cách để trong dấu <>).

Trường hợp chúng ta muốn sử dụng nhiều parameters template khác nhau.

```
template <class T, class U>
T GetMin (T a, U b) {
    return (a < b ? a : b);
}
```

### Ưu điểm

- Tiết kiệm được mã nguồn
- Tính mở: nâng cao tính sử dụng lại, thuật toán viết một lần sử dụng vô số lần
- Cho phép xây dựng các thư viện chuẩn rất mạnh như các thư viện thuật toán thông dụng: sao chép, tìm kiếm, sắp xếp, lựa chọn,...

### Nhược điểm

- Không che giấu được mã nguồn thực thi, vì compiler phải biết mã nguồn khi biên dịch
- Theo dõi, tìm kiếm lỗi phức tạp, đôi khi lỗi nằm ở phần sử dụng nhưng compiler lại báo trong phần định nghĩa khuôn mẫu hàm

### Tóm lược:

- Khi sử dụng compiler cần biết mã nguồn thực hiện khuôn mẫu hàm, do vậy khai báo và định nghĩa khuôn mẫu hàm nên để ở header file => sử dụng template sẽ công khai hết phần thực hiện
- Mã hàm khuôn mẫu chỉ được thực sự sinh ra khi và chỉ khi khuôn mẫu hàm được sử dụng với kiểu cụ thể
- Một khuôn mẫu hàm được sử dụng nhiều lần với các kiểu khác nhau thì nhiều hàm khuôn mẫu được tạo ra
- Một khuôn mẫu hàm được sử dụng nhiều lần với cùng một kiểu, thì chỉ có một hàm khuôn mẫu được tạo

### 2.9.2 Class template

Giả sử cần xây dựng một class mà trong đó các hàm, member có thể thực hiện với bất kì kiểu dữ liệu nào? Khi đó cần xây một kiểu class template.

Class template là một mẫu class, mà trong đó các template được định nghĩa để đại diện cho các member, các member template đại diện cho nhiều kiểu dữ liệu khác nhau.

#### Cú pháp:

```
template <class T>
class class_name {
    T value;
    ...
}
```

};

Ví dụ:

```

#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer
{
    T element;

public:
    mycontainer(T arg) { element = arg; }
    T increase() { return ++element; }
};

// class template specialization:
template <>
class mycontainer<char>
{
    char element;

public:
    mycontainer(char arg) { element = arg; }
    char uppercase()
    {
        if ((element >= 'a') && (element <= 'z'))
            element += 'A' - 'a';
        return element;
    }
};

int main()
{
    mycontainer<int> myint(7);
    mycontainer<char> mychar('j');
    cout << myint.increase() << endl;
    cout << mychar.uppercase() << endl;
}

```

```
return 0;
}
```

Chú ý: `mycontainer<int> myint (7);` đối với class template cần xác định rõ kiểu dữ liệu của class trong 2 dấu ngoặc nhọn `<>`

### 2.9.3 Non-type parameters for template

Bên cạnh những template được sử dụng để đại diện cho những kiểu khác nhau, thì chúng ta cũng có thể định nghĩa template là một kiểu thông thường.

Ví dụ:

```
#include <iostream>
using namespace std;

template <class T, int N>
class mysequence
{
    T memblock[N];

public:
    void setmember(int x, T value);
    T getmember(int x);
};

template <class T, int N>
void mysequence<T, N>::setmember(int x, T value)
{
    memblock[x] = value;
}

template <class T, int N>
T mysequence<T, N>::getmember(int x)
{
    return memblock[x];
}

int main()
{
    mysequence<int, 5> myints;
```



```

mysequence<double, 5> myfloats;
myints.setmember(0, 100);
myfloats.setmember(3, 3.1416);
cout << myints.getmember(0) << '\n';
cout << myfloats.getmember(3) << '\n';

return 0;
}

```

#### 2.9.4 Tham số khuôn mẫu

```

template <class T = int, class N = 10>
class Array{
    T data[N];
public:
    ...
};

void main(){
    Array<int,10> a;
    Array<int> b;
    Array<> c;
}

```

#### 2.9.5 Thuật toán tổng quát

Ví dụ: Hãy viết hàm sắp xếp các phần tử của một mảng theo thứ tự từ nhỏ đến lớn

```

void sort(int *p, int n){
    for(int i = 0; i < n-1; i++)
        for(int j = n-1; j > i; --j)
            if(p[j] < p[j-1])
                swap(p[j], p[j-1]);
}

void swap(int &a, int &b){
    int t = a;
    a = b;
    b = t;
}

```

- Tổng quát hoá kiểu dữ liệu của các phần tử

```

template <class T>

```

```

void sort(T *p, int n){
    for(int i = 0; i < n-1; i++)
        for(int j = n-1; j > i; --j)
            if(p[j] < p[j-1])
                swap(p[j], p[j-1]);
}

void swap(T &a, T &b){
    T t = a;
    a = b;
    b = t;
}

```

- Làm thế nào để ta có thể sắp xếp lại từ lớn đến nhỏ mà không cần phải viết lại hàm.

**Giải pháp:** cho thêm tham biến vào khai báo hàm

```

enum comparetype
{
    less,
    greater,
    abs_less,
    abs_greater
};

template <class T>
void sort(T *p, int n, comparetype c)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = n - 1; j > i; --j)
        {
            switch (c)
            {
                case less:
                {
                    if (p[j] < p[j - 1])
                        swap(p[j], p[j - 1]);
                    break;
                }
                case greater:
                {

```

```

        if (p[j] > p[j - 1])
            swap(p[j], p[j - 1]);
        break;
    }
    case abs_less:
    {if(abs(p[j]) < abs(p[j-1]))
        swap(p[j], p[j-1]);
        break;
    }
    case abs_greater:
    {
        if (abs(p[j]) > abs(p[j - 1]))
            swap(p[j], p[j - 1]);
        break;
    }
    }
}
}
}

```

- Nhược điểm:

- Hiệu quả không cao
- Tốc độ chậm
- Không có tính năng mở: ví dụ nếu muốn số sánh số phức theo phần thực thì không dùng được cách trên

#### - Tổng quát hóa phép toán

```

template <class T, class Compare>
void sort(T *p, int n, Compare comp)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = n - 1; j > i; --j)
        {
            if (comp(p[j], p[j - 1]))
                swap(p[j], p[j - 1]);
        }
    }
}

```

- Kiểu Compare có thể là
  - Một hàm
  - Một đối tượng thuộc lớp có định nghĩa lại toán tử gọi hàm
- Kiểu Compare là một hàm

```
template <class T>
inline bool less(const T &a, const T &b) {
    return a < b; //return operator<(a,b)
}

template <class T>
inline bool greater(const T &a, const T &b) {
    return a > b; //return operator>(a,b)
}
```

Sử dụng:

```
int v[100];
double d[100];
sort(v,100,less<int>);
sort(d,100,greater<double>)
```

- Kiểu compare là một đối tượng có định nghĩa lại toán tử gọi hàm

```
template <class T>
struct Less {
    bool operator()(const T &a, const T &b) {
        return a<b;
    }
};

template <class T>
struct Greater {
    bool operator()(const T &a, const T &b) {
        return a>b;
    }
};
```

Sử dụng:

```
int v[100];
double d[100];
sort(v,100,Less<int>());
```

```
sort(d,100, Greater<double>())
```

### 2.9.6 Bài tập

**Bài 1.** Xây dựng một khuôn mẫu hàm cho phép tìm giá trị lớn nhất/hoặc nhỏ nhất trong một mảng.

**Bài 2.** Áp dụng tổng quát hóa kiểu dữ liệu và tổng quát hóa phép toán để xây dựng các hàm cần thiết thực hiện các phép toán cộng, trừ, nhân, chia từng phần tử của hai mảng. Sau đó viết chương trình minh họa cách sử dụng.



## CHƯƠNG 3. TỔNG QUAN VỀ C++ STL

### 3.1 Giới thiệu

“C++ được đánh giá là ngôn ngữ mạnh vì tính mềm dẻo, gần gũi với ngôn ngữ máy. Ngoài ra, với khả năng lập trình theo mẫu (template), C++ đã khiến ngôn ngữ lập trình trở thành khái quát, không cụ thể và chi tiết như nhiều ngôn ngữ khác. Sức mạnh của C++ đến từ STL, viết tắt của Standard Template Library - một thư viện template cho C++ với những cấu trúc dữ liệu cũng như giải thuật được xây dựng tổng quát mà vẫn tận dụng được hiệu năng và tốc độ của C. Với khái niệm template, những người lập trình đã đề ra khái niệm lập trình khái lược (generic programming), C++ được cung cấp kèm với bộ thư viện chuẩn STL.

Bộ thư viện này thực hiện toàn bộ các công việc vào ra dữ liệu (iostream), quản lý mảng (vector), thực hiện hầu hết các tính năng của các cấu trúc dữ liệu cơ bản (stack, queue, map, set...). Ngoài ra, STL còn bao gồm các thuật toán cơ bản: tìm min, max, tính tổng, sắp xếp (với nhiều thuật toán khác nhau), thay thế các phần tử, tìm kiếm (tìm kiếm thường và tìm kiếm nhị phân), trộn. Toàn bộ các tính năng nêu trên đều được cung cấp dưới dạng template nên việc lập trình luôn thể hiện tính khái quát hóa cao. Nhờ vậy, STL làm cho ngôn ngữ C++ trở nên trong sáng hơn nhiều.”

- STL khá là rộng nên tài liệu này mình chỉ viết để định hướng về cách sử dụng STL cơ bản để các bạn ứng dụng trong việc giải các bài toán tin học đòi hỏi đến cấu trúc dữ liệu và giải thuật.
- Mình chủ yếu sử dụng các ví dụ, cũng như nguồn tài liệu từ trang web [www.cplusplus.com](http://www.cplusplus.com), các bạn có thể tham khảo chi tiết ở đó nữa.
- Để có thể hiểu được những gì mình trình bày trong này, các bạn cần có những kiến thức về các cấu trúc dữ liệu, cũng như một số thuật toán như sắp xếp, tìm kiếm...

Thư viện mẫu chuẩn STL trong C++ chia làm 4 thành phần là:

- Containers Library : chứa các cấu trúc dữ liệu mẫu (template)
  - Sequence containers
    - vector

- deque
- list
- array
- Forward\_list
- Containers adaptors
  - stack
  - queue
  - priority\_queue
- Associative containers
  - set
  - map
  - multiset
  - multimap
- Unordered associative containers
  - unordered\_set
  - unordered\_map
  - unordered\_multiset
  - unordered\_multimap
- Algorithms Library: một số thuật toán để thao tác trên dữ liệu
- Iterator Library: giống như con trỏ, dùng để truy cập đến các phần tử dữ liệu của container.
- Numeric library

Để sử dụng STL, bạn cần khai báo từ khóa “using namespace std;” sau các khai báo thư viện (các “#include”, hay “#define”,...), hoặc sử dụng std:: trước mỗi keyword

Việc sử dụng các hàm trong STL tương tự như việc sử dụng các hàm như trong class

### 3.2 Iterator (Biến lặp)

Trong C++, một biến lặp là một đối tượng bất kì, trỏ tới một số phần tử trong 1 phạm vi của các phần tử (như mảng hoặc container), có khả năng để lặp các phần tử trong phạm vi bằng cách sử dụng một tập các toán tử (operators) (như so sánh, tăng (++),...)

Dạng rõ ràng nhất của iterator là một con trỏ: Một con trỏ có thể trỏ tới các phần tử trong mảng, và có thể lặp thông qua sử dụng toán tử tăng (++). Tuy nhiên, cũng có các

dạng khác của iterator. Ví dụ: mỗi loại container (chẳng hạn như vector) có một loại iterator được thiết kế để lặp các phần tử của nó một cách hiệu quả.

Iterator có các toán tử như:

- So sánh: == , != giữa 2 iterator.
- Gán: = giữa 2 iterator.
- Cộng trừ: +, - với hằng số và ++, --
- Lấy giá trị: \*

### 3.3 Containers (Thư viện lưu trữ)

Một container là một đối tượng cụ thể lưu trữ một tập các đối tượng khác (các phần tử của nó). Nó được thực hiện như các lớp mẫu (class templates).

Container quản lý không gian lưu trữ cho các phần tử của nó và cung cấp các hàm thành viên (member function) để truy cập tới chúng, hoặc trực tiếp hoặc thông qua các biến lặp (iterator – giống như con trỏ).

Container xây dựng các cấu trúc thường sử dụng trong lập trình như: mảng động - dynamic arrays (vector), hàng đợi - queues (queue), hàng đợi ưu tiên - heaps (priority queue), danh sách liên kết - linked list (list), cây - trees (set), mảng ánh xạ - associative arrays (map),...

Nhiều container chứa một số hàm thành viên giống nhau. Quyết định sử dụng loại container nào cho nhu cầu cụ thể nói chung không chỉ phụ thuộc vào các hàm được cung cấp mà còn phải dựa vào hiệu quả của các hàm thành viên của nó (độ phức tạp (từ giờ mình sẽ viết tắt là ĐPT) của các hàm). Điều này đặc biệt đúng với container dãy (sequence containers), mà trong đó có sự khác nhau về độ phức tạp đối với các thao tác chèn/xóa phần tử hay truy cập vào phần tử.

#### 3.3.1 Iterator

Tất cả các container ở 2 loại: Sequence container và Associative container đều hỗ trợ các iterator như sau (ví dụ với vector, những loại khác có chức năng cũng vậy)

```
/*khai báo iterator "it"*/
vector<int>::iterator it;

/* trỏ đến vị trí phần tử đầu tiên của vector */
```



```

it = vector.begin();

/*trở đến vị trí kết thúc (không phải phần tử cuối cùng nhé) của
vector) */
it = vector.end();

/* khai báo iterator ngược "rit" */
vector<int>::reverse_iterator rit;
rit = vector.rbegin();

/* trở đến vị trí kết thúc của vector theo chiều ngược (không phải
phần tử đầu tiên nhé*/
rit = vector.rend();

```

Tất cả các hàm iterator này đều có độ phức tạp  $O(1)$

Để cho ngắn gọn câu lệnh, ta có thể dùng từ khoá auto để trình biên dịch tự động xác định kiểu dữ liệu dữ trên biến truyền vào, ví dụ:

```

vector<int> v;
auto it = v.begin();

```

Do `v.begin()` trả về kiểu dữ liệu là `vector<int>::iterator` nên từ khoá auto tự động xác nhận kiểu dữ liệu cho `it` là `vector<int>::iterator`

Các hàm với iterator:

- `begin()`: trả về 1 iterator ở vị trí đầu tiên của một container
- `end()`: trả về 1 iterator ở vị trí kết thúc của một container (ví dụ container có 7 phần tử, vị trí kết thúc là ở vị trí có index là 7)
- `cbegin()`: trả về 1 constant iterator ở vị trí đầu tiên của một container
- `cend()`: trả về 1 constant iterator ở vị trí đầu tiên của một container
- `rbegin()`: trả về 1 iterator ngược ở vị trí phần tử cuối cùng của một container
- `rend()`: trả về 1 iterator ngược ở vị trí kết thúc của một container (tính từ phần tử cuối cùng lên phần tử đầu tiên)
- Tương tự `crbegin()`, `crend()`

### 3.3.2 Vector (Mảng động)

- Khai báo vector:

- Khai báo `#include <vector>` ở đầu chương trình
- **Cú pháp:** `vector<data_type>` (nếu đã khai báo `using namespace std;`) hoặc `std::vector<data_type>` (nếu chưa khai báo `using namespace std;`)

```
#include <vector>

...

/* Vector 1 chiều */

/* tạo vector rỗng kiểu dữ liệu int */
vector <int> first;

//tạo vector với 4 phần tử là 100
vector <int> second (4,100);

// lấy từ đầu đến cuối vector second
vector <int> third (second.begin(),second.end())

//copy từ vector third
vector <int> four (third)

/* Vector 2 chiều*/

/* Tạo vector 2 chiều rỗng */
vector < vector <int> > v;

/* khai báo vector 5x10 */
vector < vector <int> > v (5, 10);

/* khai báo 5 vector 1 chiều rỗng */
```

```
vector < vector <int> > v (5);

//khai báo vector 5*10 với các phần tử khởi tạo giá trị là 1
vector < vector <int> > v (5, vector <int> (10,1));
```

#### - Capacity:

- `size()`: trả về số lượng phần tử của vector. ĐPT  $O(1)$ .
- `empty()`: trả về true nếu vector rỗng, ngược lại là false. ĐPT  $O(1)$ .

#### - Truy cập tới phần tử:

- `operator [index]`: trả về giá trị phần tử ở vị trí index. ĐPT  $O(1)$ .
- `at(index)`: tương tự như trên. ĐPT  $O(1)$ .
- `front()`: trả về giá trị phần tử đầu tiên. ĐPT  $O(1)$ .
- `back()`: trả về giá trị phần tử cuối cùng. ĐPT  $O(1)$ .

#### - Chỉnh sửa:

- `push_back(value)`: thêm phần tử value vào cuối vector. ĐPT  $O(1)$ .
- `pop_back()`: loại bỏ phần tử ở cuối vector. ĐPT  $O(1)$ .
- `insert (iterator, x)`: chèn “x” vào trước vị trí “iterator” ( x có thể là phần tử hay iterator của 1 đoạn phần tử...). ĐPT  $O(n)$ .
- `erase(iterator)`: xóa phần tử ở vị trí iterator. ĐPT  $O(n)$ .
- `swap(vecor_2)`: đổi 2 vector cho nhau (ví dụ: `first.swap(second);`). ĐPT  $O(1)$ .
- `clear()`: xóa vector. ĐPT  $O(n)$ .

#### - Sử dụng vector sẽ tốt khi:

- Truy cập đến phần tử riêng lẻ thông qua vị trí của nó  $O(1)$
- Chèn hay xóa ở vị trí cuối cùng  $O(1)$ .
- Vector làm việc giống như một “mảng động”.

Ví dụ. Ví dụ này chủ yếu để làm quen sử dụng các hàm chứ không có đề bài cụ thể.

```
#include <iostream>
#include <vector>
using namespace std;
vector <int> v; // Khai báo vector
vector <int>::iterator it; // Khai báo iterator
```

```

vector<int>::reverse_iterator rit;    // Khai báo iterator ngược
int i;

int main() {
    for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
    cout << v.front() << endl;        // In ra 1
    cout << v.back() << endl;         // In ra 5
    cout << v.size() << endl;          // In ra 5
    v.push_back(9);                    // v={1,2,3,4,5,9}
    cout << v.size() << endl;          // In ra 6

    v.clear();                          // v={}
    cout << v.empty() << endl;         // In ra 1 (vector rỗng)

    for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
    v.pop_back();                       // v={1,2,3,4}
    cout << v.size() << endl;          // In ra 4

    v.erase(v.begin()+1);               // Xóa ptừ thứ 1 v={1,3,4}
    v.erase(v.begin(),v.begin()+2);    // v={4}
    v.insert(v.begin(),100);            // v={100,4}
    v.insert(v.end(),5);                // v={100,4,5}

    /*Duyệt theo chỉ số phần tử*/
    for (i=0;i<v.size();i++)
        cout << v[i] << " ";          // 100 4 5
    cout << endl;

    /*
    Chú ý: Không nên viết
    for (i=0;i<=v.size()-1;i++) ...
    Vì nếu vector v rỗng, v.size() là kiểu unsigned int, nên v.size()-1
    sẽ bằng 2^32-1
    */

    /*Duyệt theo iterator*/
    for (it=v.begin();it!=v.end();it++)

```

```

        cout << *it << " ";           // In ra giá trị mà iterator đang
trở tới "100 4 5"
        cout << endl;

        /*Duyệt iterator ngược*/
        for (rit=v.rbegin();rit!=v.rend();rit++)
            cout << *rit << " ";       // 5 4 100
        cout << endl;

        system("pause");
    }

```

### 3.3.3 Deque (Hàng đợi hai đầu)

- Deque là từ viết tắt từ double-ended queue (hàng đợi hai đầu)
- Deque có các ưu điểm sau:
  - Các phần tử có thể truy cập thông qua chỉ số vị trí của nó. ĐPT  $O(1)$
  - Chèn hoặc xóa phần tử ở cuối hoặc đầu dãy. ĐPT  $O(1)$
- Khai báo: `#include <deque>`
- Cú pháp: `deque<data_type>` (nếu đã khai báo `using namespace std;`) hoặc `std::deque<data_type>` (nếu chưa khai báo `using namespace std;`)
- Capacity:
  - `size()`: trả về số lượng phần tử của deque
  - `empty()`: trả về true nếu deque rỗng, ngược lại là false.
- Truy cập phần tử:
  - `operator[index]`: trả về giá trị phần tử ở vị trí index.
  - `at(index)`: tương tự như trên.
  - `front()`: trả về giá trị phần tử đầu tiên.
  - `back()`: trả về giá trị phần tử cuối cùng.
- Chỉnh sửa:
  - `push_back(value)`: thêm phần tử value vào cuối deque.
  - `push_front(value)`: thêm phần tử value vào đầu deque
  - `pop_back()`: loại bỏ phần tử ở cuối deque.

- `pop_front()`: loại bỏ phần tử ở đầu deque
- `insert (iterator, x)`: chèn “x” vào trước vị trí “iterator” ( x có thể là phần tử hay iterator của 1 đoạn phần tử...).
- `erase(iterator)`: xóa phần tử ở vị trí iterator
- `swap(vecor_2)`: đổi 2 deque cho nhau (ví dụ: `first.swap(second)`);).
- `clear()`: xóa deque.

### 3.3.4 List (Danh sách liên kết)

- List được thực hiện như danh sách nối kép (doubly-linked list). Mỗi phần tử trong danh sách nối kép có liên kết đến một phần tử trước đó và một phần tử sau nó

- Do đó, list có các ưu điểm như sau:

- Chèn và loại bỏ phần tử ở bất cứ vị trí nào trong container.  $O(1)$ .

- Điểm yếu của list là khả năng truy cập tới phần tử thông qua vị trí.  $O(n)$

- Khai báo: `#include <list>`

- Cú pháp: `list<data_type>` (nếu đã khai báo `using namespace std;`) hoặc `std::list<data_type>` (nếu chưa khai báo `using namespace std;`)

- Capacity:

- `size()`: trả về số lượng phần tử của list
- `empty()`: trả về true nếu list rỗng, ngược lại là false.

- Truy cập phần tử:

- `front()`: trả về giá trị phần tử đầu tiên.
- `back()`: trả về giá trị phần tử cuối cùng.

- Chỉnh sửa:

- `push_back(value)`: thêm phần tử value vào cuối list.
- `push_front(value)`: thêm phần tử value vào đầu list
- `pop_back()`: loại bỏ phần tử ở cuối list.
- `pop_front()`: loại bỏ phần tử ở đầu list
- `insert (iterator, x)`: chèn “x” vào trước vị trí “iterator” ( x có thể là phần tử hay iterator của 1 đoạn phần tử...).
- `erase(iterator)`: xóa phần tử ở vị trí iterator

- `swap(vecor_2)`: đổi 2 list cho nhau (ví dụ: `first.swap(second)`);).
- `clear()`: xóa list.

- Operations

- `splice()`: di chuyển phần tử từ list này sang list khác
- `remove (const)`: loại bỏ tất cả phần tử trong list bằng `const`
- `remove_if (function)`: loại bỏ tất cả các phần tử trong list nếu hàm `function` return `true`
- `unique()`: loại bỏ các phần tử bị trùng lặp hoặc thỏa mãn hàm nào đó. Lưu ý: Các phần tử trong list phải được sắp xếp.
- `sort()`: sắp xếp các phần tử của list theo mặc định hoặc thỏa mãn một hàm nào đó
- `reverse()`: đảo ngược lại các phần tử của list

### 3.3.5 Stack (Ngăn xếp)

- Stack là một loại container adaptor, được thiết kế để hoạt động theo kiểu LIFO (Last - in first - out) (vào sau ra trước), tức là một kiểu danh sách mà việc bổ sung và loại bỏ một phần tử được thực hiện ở cuối danh sách. Vị trí cuối cùng của stack gọi là đỉnh (top) của ngăn xếp.

- Khai báo: `#include <stack>`

- Cú pháp: `stack<data_type>` (nếu đã khai báo `using namespace std;`) hoặc `std::stack<data_type>` (nếu chưa khai báo `using namespace std;`)

- Các hàm thành viên:

- `size()`: trả về kích thước hiện tại của stack.
- `empty()`: true stack nếu rỗng, và ngược lại.
- `push(value)`: đẩy phần tử vào stack.
- `pop()`: loại bỏ phần tử ở đỉnh của stack.
- `top()`: truy cập tới phần tử ở đỉnh stack.

### 3.3.6 Queue (Hàng đợi)

- Queue là một loại container adaptor, được thiết kế để hoạt động theo kiểu FIFO (First - in first - out) (vào trước ra trước), tức là một kiểu danh sách mà việc bổ sung được thực hiện ở cuối danh sách và loại bỏ ở đầu danh sách.

- Trong queue, có hai vị trí quan trọng là vị trí đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử cuối cùng được thêm vào.

- Khai báo: `#include <queue>`

- Cú pháp: `queue<data_type>` (nếu đã khai báo `using namespace std;`) hoặc `std::queue<data_type>` (nếu chưa khai báo `using namespace std;`)

- Các hàm thành viên:

- `size()`: trả về kích thước hiện tại của queue.
- `empty()`: true nếu queue rỗng, và ngược lại.
- `push(value)`: đẩy vào cuối queue.
- `pop()`: loại bỏ phần tử (ở đầu).
- `front()`: trả về phần tử ở đầu.
- `back()`: trả về phần tử ở cuối.

### 3.3.7 Set (Tập hợp)

- Set là một loại associative containers để lưu trữ các phần tử không bị trùng lặp (unique elements), và các phần tử này chính là các khóa (keys)

- Khi duyệt set theo iterator từ begin đến end, các phần tử của set sẽ tăng dần theo phép toán so sánh.

- Mặc định của set là sử dụng phép toán less, bạn cũng có thể viết lại hàm so sánh theo ý mình.

- Set được thực hiện giống như cây tìm kiếm nhị phân (Binary search tree)

- Khai báo:

```
#include <set>
set <int> s;
set <int, greater<int> > s;
```

Hoặc viết class so sánh theo ý mình:

```
struct cmp{
    bool operator() (int a,int b) {return a<b;}
};
set<int, cmp> myset ;
```



## - Capacity:

- `size()`: trả về kích thước hiện tại của set.
- `empty()`: true nếu set rỗng, và ngược lại.

## - Modifiers:

- `insert()`: Chèn phần tử vào set
- `erase()`: có 2 kiểu xóa: xóa theo iterator, hoặc là xóa theo khóa
- `clear()`: xóa tất cả set
- `swap()`: đổi 2 set cho nhau. ĐPT  $O(n)$ .

## - Operations:

- `find()`: trả về iterator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy iterator trỏ về “end” của set
- `lower_bound()`: trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí “end” của set
- `upper_bound()`: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí “end” của set
- `count()`: trả về số lần xuất hiện của khóa trong container. Nhưng trong set, các phần tử chỉ xuất hiện một lần, nên hàm này có ý nghĩa là sẽ return 1 nếu khóa có trong container, và 0 nếu không có.

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> s;
    set<int>::iterator it;
    s.insert(9);                // s={9}
    s.insert(5);                // s={5,9}
    cout << *s.begin() << endl; //In ra 5
    s.insert(1);                // s={1,5,9}
    cout << *s.begin() << endl;
```

```

it = s.find(5);
if (it == s.end())
    cout << "Khong co trong container" << endl;
else
    cout << "Co trong container" << endl;
s.erase(it); // s={1,9}
s.erase(1); // s={9}
s.insert(3); // s={3,9}
s.insert(4); // s={3,4,9}
it = s.lower_bound(4);
if (it == s.end())
    cout << "Khong co phan tu nao trong set khong be hon 4" << endl;
else
    cout << "Phan tu be nhat khong be hon 4 la " << *it << endl; //
In ra 4
it = s.lower_bound(10);
if (it == s.end())
    cout << "Khong co phan tu nao trong set khong be hon 10" << endl;
else
    cout << "Phan tu be nhat khong be hon 10 la " << *it << endl; //
Khong co ptu nao
it = s.upper_bound(4);
if (it == s.end())
    cout << "Khong co phan tu nao trong set lon hon 4" << endl;
else
    cout << "Phan tu be nhat lon hon 4 la " << *it << endl; // In ra
9
/* Duyet set */
for (it = s.begin(); it != s.end(); it++)
{
    cout << *it << " ";
}
// In ra 3 4 9
cout << endl;
system("pause");
}

```

- Lưu ý: Nếu bạn muốn sử dụng hàm `lower_bound` hay `upper_bound` để tìm phần tử lớn nhất “bé hơn hoặc bằng” hoặc “bé hơn” bạn có thể thay đổi cách so sánh của set để tìm kiếm. Mời bạn xem chương trình sau để rõ hơn:

```
#include <iostream>
#include <set>
#include <vector>

using namespace std;

int main()
{
    set<int, greater<int>> s;
    set<int, greater<int>>::iterator it; // Phép toán so sánh là greater
    s.insert(1);                        // s={1}
    s.insert(2);                        // s={2,1}
    s.insert(4);                        // s={4,2,1}
    s.insert(9);                        // s={9,4,2,1}
    /* Tìm phần tử lớn nhất bé hơn hoặc bằng 5 */
    it = s.lower_bound(5);
    cout << *it << endl; // In ra 4
    /* Tìm phần tử lớn nhất bé hơn 4 */
    it = s.upper_bound(4);
    cout << *it << endl; // In ra 2
    system("pause");
}
```

### 3.3.8 Multiset (Tập hợp)

- Multiset giống như Set nhưng có thể chứa các khóa có giá trị giống nhau.
- Khai báo: giống như set.
- Capacity:
  - `size()`: trả về kích thước hiện tại của multiset.
  - `empty()`: true nếu multiset rỗng, và ngược lại.
- Chỉnh sửa:
  - `insert()`: Chèn phần tử vào set. ĐPT  $O(\log N)$ .
  - `erase()`:

- xóa theo iterator
- xóa theo khóa: xóa tất cả các phần tử bằng khóa trong multiset
- `clear()`: xóa tất cả set.
- `swap()`: đổi 2 set cho nhau.

- Operations:

- `find()`: trả về iterator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy iterator trỏ về “end” của set. Dù trong multiset có nhiều phần tử bằng khóa thì nó cũng chỉ iterator đến một phần tử.
- `lower_bound()`: trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí “end” của set. ĐPT  $O(\log N)$ .
- `upper_bound()`: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí “end” của set.. ĐPT  $O(\log N)$ .
- `count()`: trả về số lần xuất hiện của khóa trong multiset

### 3.3.9 Map (Ánh xạ)

- Map là một loại associative container. Mỗi phần tử của map là sự kết hợp của khóa (key value) và ánh xạ của nó (mapped value). Cũng giống như set, trong map không chứa các khóa mang giá trị giống nhau.

- Trong map, các khóa được sử dụng để xác định giá trị các phần tử. Kiểu của khóa và ánh xạ có thể khác nhau.

- Và cũng giống như set, các phần tử trong map được sắp xếp theo một trình tự nào đó theo cách so sánh.

- Map được cài đặt bằng red-black tree (cây đỏ đen) – một loại cây tìm kiếm nhị phân tự cân bằng. Mỗi phần tử của map lại được cài đặt theo kiểu pair (xem thêm ở thư viện utility)

- Khai báo:

```
#include <map>
...
map <kiểu_dữ_liệu_1, kiểu_dữ_liệu_2>
// kiểu dữ liệu 1 là khóa, kiểu dữ liệu 2 là giá trị của khóa.
```

hoặc sử dụng class so sánh

```
struct cmp{
bool operator() (char a,char b) {return a<b;}
};
.....
map <char,int,cmp> m;
```

Truy cập đến giá trị của các phần tử trong map khi sử dụng iterator: Ví dụ ta đang có một iterator là `it` khai báo cho map thì:

```
(*it).first; // Lấy giá trị của khóa, kiểu_dữ_liệu_1
(*it).second; // Lấy giá trị của giá trị của khóa, kiểu_dữ_liệu_2
(*it) // Lấy giá trị của phần tử mà iterator đang trỏ đến, kiểu pair
it->first; // giống như (*it).first
it->second; // giống như (*it).second
```

- Capacity:

- `size()`: trả về kích thước hiện tại của map.
- `empty()`: true nếu map rỗng, và ngược lại.

- Truy cập tới phần tử:

- `operator [khóa]`: Nếu khóa đã có trong map, thì hàm này sẽ trả về giá trị mà khóa ánh xạ đến. Ngược lại, nếu khóa chưa có trong map, thì khi gọi `[]` nó sẽ thêm vào map khóa đó.

- Chỉnh sửa

- `insert()`: Chèn phần tử vào map. Chú ý: phần tử chèn vào phải ở kiểu “pair”.
- `erase()`:
  - xóa theo iterator ĐPT  $O(\log N)$
  - xóa theo khóa: xóa khóa trong map. ĐPT:  $O(\log N)$ .
- `clear()`: xóa tất cả set. ĐPT  $O(n)$ .
- `swap()`: đổi 2 set cho nhau. ĐPT  $O(n)$ .

- Operations:

- `find()`: trả về iterator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy iterator trả về “end” của map.

- `lower_bound()`: trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa ( dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí “end” của map
- `upper_bound`: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí “end” của map.
- `count` : trả về số lần xuất hiện của khóa trong map.

```
#include <iostream>
#include <map>
#include <vector>

using namespace std;

int main()
{
    map<char, int> m;
    map<char, int>::iterator it;
    m['a'] = 1; // m={{'a',1}}
    m.insert(make_pair('b', 2)); // m={{'a',1};{'b',2}}
    m.insert(pair<char, int>('c', 3)); // m={{'a',1};{'b',2};{'c',3}}
    cout << m['b'] << endl; // In ra 2
    m['b']++; // m={{'a',1};{'b',3};{'c',3}}
    it = m.find('c'); // it point to key 'c'
    cout << it->first << endl; // In ra 'c'
    cout << it->second << endl; // In ra 3
    m['e'] = 100;
    //m={{'a',1};{'b',3};{'c',3};{'e',100}}
    it = m.lower_bound('d'); // it point to 'e'
    cout << it->first << endl; // In ra 'e'
    cout << it->second << endl; // In ra 100
    system("pause");
}
```

### 3.3.10 Multimap (Ảnh xạ)

Giống như map nhưng có thể chứa các phần tử có khóa giống nhau, do đó nó khác map ở chỗ không có operator[].

### 3.4 STL Algorithms (Thư viện thuật toán)

- Khai báo sử dụng: `#include <algorithm>`
- Các hàm trong STL Algorithm khá nhiều nên mình chỉ giới thiệu sơ qua về một số hàm hay sử dụng trong các bài toán.
- Có một lưu ý nhỏ cho các bạn là khi sử dụng các hàm mà thực hiện trong một đoạn phần tử liên tiếp nào đó thì các hàm trong C++ thường có tác dụng trên nửa đoạn [...]. Ví dụ như: bạn muốn hàm  $f$  có tác dụng trong đoạn từ 1 đến  $n$  thì các bạn phải gọi hàm trong đoạn từ 1 đến  $n+1$ .

#### 3.4.1 Min, max

- min: trả về giá trị bé hơn theo phép so sánh (mặc định là phép toán less)  
Ví dụ: `min('a', 'b')` sẽ return 'a', `min(3, 1)` sẽ return 1
- max: ngược lại với hàm min  
Ví dụ: `max('a', 'b')` sẽ return 'b', `max(3, 1)` sẽ return 3
- next\_permutation: hoán vị tiếp theo. Hàm này sẽ return 1 nếu có hoán vị tiếp theo, 0 nếu không có hoán vị tiếp theo.

Ví dụ:

```
// next_permutation
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
    int myints[] = {1, 2, 3};
    cout << "The 3! possible permutations with 3 elements:\n";
    do
    {
        cout << myints[0] << " " << myints[1] << " " << myints[2] <<
endl;
    } while (next_permutation(myints, myints + 3));
    return 0;
}
```

**Kết quả:**

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

- prev\_permutation: ngược lại với next\_permutation

**3.4.2 Sắp xếp**

- sort: sắp xếp đoạn phần tử theo một trình tự nào đó. Mặc định của sort là sử dụng operator <.

- Bạn có thể sử dụng hàm so sánh, hay class so sánh tự định nghĩa để sắp xếp cho linh hoạt.

- Ví dụ:

```
// sort algorithm example
#include <algorithm>
#include <iostream>

using namespace std;

bool myfunction(int i, int j) { return (i < j); }

struct myclass
{
    bool operator()(int i, int j) { return (i > j); }
} myobject;

int main()
{
    int myints[] = {32, 71, 12, 45, 26, 80, 53, 33};

    // using default comparison (operator <):
    sort(myints, myints + 4); //(12 32 45 71)26 80 53 33
```



```

// using function as comp
sort(myints + 4, myints + 8, myfunction); // 12 32 45 71(26 33 53 80)
for (int i = 0; i < 8; i++)
    cout << myints[i] << " ";
cout << endl;

// using object as comp
sort(myints, myints + 8, myobject); //(80 71 53 45 33 32 26 12)
for (int i = 0; i < 8; i++)
    cout << myints[i] << " ";
cout << endl;

system("pause");
return 0;
}

```

### 3.4.3 Tìm kiếm nhị phân

#### 3.4.3.1 *binary\_search*

- Tìm kiếm xem khóa có trong đoạn cần tìm không. Lưu ý: đoạn tìm kiếm phải được sắp xếp theo một trật tự nhất định. Nếu tìm được sẽ return true, ngược lại return false.

- Dạng 1: `binary_search`(vị trí bắt đầu, vị trí kết thúc, khóa);
- Dạng 2: `binary_search`(vị trí bắt đầu, vị trí kết thúc, khóa, phép so sánh);

Ví dụ:

```

// binary_search example
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

bool myfunction(int i, int j) { return (i < j); }

int main()
{
    int myints[] = {1, 2, 3, 4, 5, 4, 3, 2, 1};
    vector<int> v(myints, myints + 9); // 1 2 3 4 5 4 3 2 1
}

```

```

// Sử dụng toán tử so sánh mặc định
sort(v.begin(), v.end());
cout << "looking for a 3... ";
if (binary_search(v.begin(), v.end(), 3))
    cout << "found!\n";
else
    cout << "not found.\n";
// sử dụng hàm so sánh tự định nghĩa:
sort(v.begin(), v.end(), myfunction);
cout << "looking for a 6... ";
if (binary_search(v.begin(), v.end(), 6, myfunction))
    cout << "found!\n";
else
    cout << "not found.\n";
return 0;
}

```

#### 3.4.3.2 *lower\_bound*

- Hàm `lower_bound` và `upper_bound` tương tự như ở trong container set hay map.
- Trả về iterator đến phần tử đầu tiên trong nửa đoạn `[first,last]` mà không bé hơn khóa tìm kiếm.

- Dạng 1: `lower_bound(đầu, cuối, khóa );`
- Dạng 2: `lower_bound(đầu, cuối, khóa, phép toán so sánh của đoạn)`

#### 3.4.3.3 *upper\_bound*

- Trả về iterator đến phần tử đầu tiên trong nửa đoạn `[first,last]` mà lớn hơn khóa tìm kiếm.

- Dạng 1: `upper_bound (đầu, cuối, khóa );`
- Dạng 2: `upper_bound (đầu, cuối, khóa, phép toán so sánh của đoạn)`

### 3.5 Lambda Function

#### 3.5.1 *Định nghĩa*

Lambda expression (có tên khác như: hàm không tên -- anonymous function, là cách viết định nghĩa một hàm trong một ngôn ngữ.

Cách gọi khác: hàm không tên -- anonymous function.

### 3.5.2 Khái quát

Lambda là đối tượng hàm (Function Object) , là operator() của một object. Lambda có thể được dùng thay thế cho std::function và có thể dùng với cả bộ thư viện STL (Các câu sẽ thấy điều này tiện lợi thế nào). Lambda trong C++ có thể capture được các biến local (sẽ trình bày sau). Giá trị trả về của lambda có thể được suy luận kiểu (có nghĩa là dùng được từ khóa auto).

Do lambda là tính năng mới của C++ (C++ 11) nên để dùng tính năng của ngôn ngữ này, ta phải kiếm 1 compiler mới mới, xin xin, hỗ trợ C++ 11

### 3.5.3 Cú pháp

C++ lambda có 4 cú pháp khai báo như sau:

```
[ capture-list ] ( params ) mutable(optional) exception attribute -> ret {
body }      (1)
[ capture-list ] ( params ) -> ret { body }      (2)
[ capture-list ] ( params ) { body }      (3)
[ capture-list ] { body }      (4)
```

### 3.5.4 Ví dụ

#### 3.5.4.1 Lambda đơn giản

Hàm này chả đem lại lợi ích gì cho xã hội, chỉ là 3 cái ngoặc đứng chiếm diện tích. Ấy vậy mà chương trình compile và chạy như thường.

```
int main() {
    [](){};
    return 0;
}
```

#### 3.5.4.2 Lambda đơn giản hơn nữa

```
int main() {
    []{};
    return 0;
}
```

Trong dấu () là đối số của lambda. Ở đây bản thân lambda này không nhận đối số nên có thể bỏ luôn được cả ngoặc.

#### 3.5.4.3 Ý nghĩa của từng cái ngoặc

```
int main() {
    []      // Khai báo lambda
    ()      // Đối số
    {}      // Thân hàm
    ();     // Chạy!
    return 0;
}
```

#### 3.5.4.4 Hello lambda

```
#include<iostream>

int main() {
    []{ std::cout << "Hello Lambda\n"; }();
    return 0;
}
```

Biên dịch và chạy sẽ thấy dòng "Hello Lambda" được xuất ra màn hình.

#### 3.5.4.5 Truyền đối số

```
#include<iostream>

int main() {
    [](const std::string& x) { std::cout << x << std::endl; }("Hello
lambda\n");
    return 0;
}
```

#### 3.5.4.6 Trả về giá trị

```
#include<iostream>

int main() {
    auto x = [](int a) -> int { return a + 5; } (10);
    std::cout << x;
    return 0;
}
```

Kết quả là 15. Ở đây -> int là cách viết diễn đạt là hàm này sẽ trả về kiểu int. auto sẽ tự suy luận kiểu của x.

#### 3.5.4.7 Capture / Capture

Trước khi ví dụ chắc phải giải thích trước thì ví dụ sẽ dễ hiểu hơn. Lambda có đặc điểm là nó là Function Object và truy cập được biến số ngoài scope (Scope tạo bởi dấu {}) của nó. Truy cập một biến được capture có 2 cách:

- Truy cập theo reference.
- Truy cập bằng cách copy (by value).

Trong Lambda, để capture by reference thì viết [&]. Để capture bằng value thì viết [=]. Để không capture gì cả thì để trống [].

```
#include<iostream>

int main() {
    int a = 5;
    [&] { std::cout << a << std::endl; }(); // a sẽ được truyền bằng
reference.
    [=] { std::cout << a << std::endl; }(); // a sẽ được truyền bằng
value.
    return 0;
}
```

#### 3.5.4.8 Ghi đè

Biến số được capture bằng tham chiếu có thể bị ghi đè.

```
#include<iostream>

int main() {
    int a = 5;
    [&] { a = a + 1; std::cout << a << std::endl; }(); // --> In 6
    return 0;
}
```

#### 3.5.4.9 Capture bằng value sẽ báo lỗi khi compile

Lỗi compile

```
#include<iostream>

int main() {
    int a = 5;
    [=] { a = a + 1; std::cout << a << std::endl; }(); // Lỗi
    return 0;
}
```

#### 3.5.4.10 Nếu vẫn muốn sửa giá trị capture bằng value? Dùng mutable

Biến a được truyền bằng giá trị và mutable nên trong lambda giá trị bị thay đổi nhưng khi ra khỏi scope của lambda, mọi việc lại trở lại bình thường.

```
#include<iostream>

int main() {
    int a = 5;
    [=] () mutable { a = a + 1; std::cout << a << std::endl; }(); // In 6
    std::cout << a << std::endl; // In 5
    return 0;
}
```

#### 3.5.4.11 Sự khác nhau giữa reference và mutable value

```
#include<iostream>

int main() {
    int a = 0, b = 0;
    [a, &b] () mutable { a = 1; b = 1; } ();
    std::cout << a << std::endl; // in 0
    std::cout << b << std::endl; // in 1
}
```

#### 3.5.4.12 l) Capture capture những gì?

Capture nếu không chỉ rõ tên biến capture tất cả những gì có thể capture được.

Ví dụ:

- Capture a bằng giá trị và b bằng reference. Không capture c, d

```
#include<iostream>
```

```
int main() {
    int a = 0, b = 0, c = 0, d = 0;
    [a, &b]() mutable {}
}
```

- Capture c, d bằng giá trị, a, b bằng reference.

```
#include<iostream>

int main() {
    int a = 0, b = 0, c = 0, d = 0;
    [=, &a, &b]() mutable {}
}
```

- Capture tất cả

```
#include<iostream>

int main() {
    int a = 0, b = 0, c = 0, d = 0;
    [&]() mutable {} // Capture a, b, c, d bằng reference.
    [=]() mutable {} // Capture a, b, c, d bằng value.
}
```

#### 3.5.4.13 Dùng Lambda và std::function

```
#include <iostream>
#include <string>
#include <functional>

std::function<void()> f() {
    std::string h("Hello WOrld");
    return [=] { std::cout << h << std::endl; };
}

int main()
{
    auto func = f();
    func(); // in Hello WOrld
    f()(); // in Hello WOrld
    getchar();
    return 0;
}
```

}

Hàm `f` khi được chạy `f()` sẽ trả về 1 `std::function` mà có kiểu trả về là `void`. `f()()` sẽ chạy hàm `f`, nhận về 1 hàm anonymous và chạy hàm này.

#### 3.5.4.14 Dùng *lambda* với *stl*

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

struct Sum {
    Sum() { sum = 0; }
    void operator()(int n) { sum += n; }
    int sum;
};

int main()
{
    std::vector<int> nums{ 3, 4, 5, 6, 7, 8 };
    for (auto n : nums)
        std::cout << " " << n;
    std::cout << std::endl;

    Sum s = std::for_each(nums.begin(), nums.end(), Sum());
    std::cout << s.sum << std::endl;

    int sum = 0;
    std::for_each(nums.begin(), nums.end(), [&sum](int i) { sum += i; });
    std::cout << sum << std::endl;

    getchar();
    return 0;
}
```

Để gọi `for_each` theo cách truyền thống ta phải định nghĩa Function Object và hoặc truyền 1 function đã được định nghĩa sẵn vào hàm đấy. Dùng *lambda capture* 1 biến khai báo ở scope cao hơn (biến `sum`), ta giải quyết được bài toán chỉ trong 1 dòng. Kết quả không khác gì cách viết truyền thống.



## CHƯƠNG 4. ĐỀ THI THAM KHẢO

Sau đây là một số đề thi cuối kì của môn Kỹ thuật lập trình C/C++ của Viện Điện tử Viễn thông

### Đề 1

Cho đoạn chương trình sau

```
template <class T>
class Array
{
protected:
    T *data;    // Lưu nội dung các phần tử
    int length; // Kích thước của mảng

public:
    /**
     * Hàm tạo ra mảng rỗng
     */
    Array() : length(0), data(NULL) {}

    /**
     * Hàm lấy nội dung từ mảng v có kích thước size
     */
    Array(int size, const T *v = NULL);

    /**
     * Hàm tạo bản sao
     */
    Array(const Array &arr);

    /**
     * Hàm huỷ đối tượng
     */
    ~Array()
    {
        if (data != NULL)
        {
```

```

        delete[] data;
        data = NULL;
    }
}

/**
 * Kiểm tra mảng rỗng hay không
 */
bool isEmpty();

/**
 * Chiều dài của mảng
 */
int getLength() const { return length; }

/**
 * Xoá phần tử ở đầu mảng vào a
 */
void removeFirst(T &a);

/**
 * @brief Tìm kiếm phần tử k trong danh sách
 * @return Vị trí tìm thấy (trả về -1 nếu không tìm thấy)
 */
int find(T k);

/**
 * Định nghĩa lại toán tử lấy vị trí phần tử của mảng
 */
T &operator[](int index) { return data[index]; }

/**
 * Định nghĩa lại toán tử gán (=)
 */
Array &operator=(const Array &arr);

/**
 * Sắp xếp mảng bằng Insertion Sort
 */

```

```

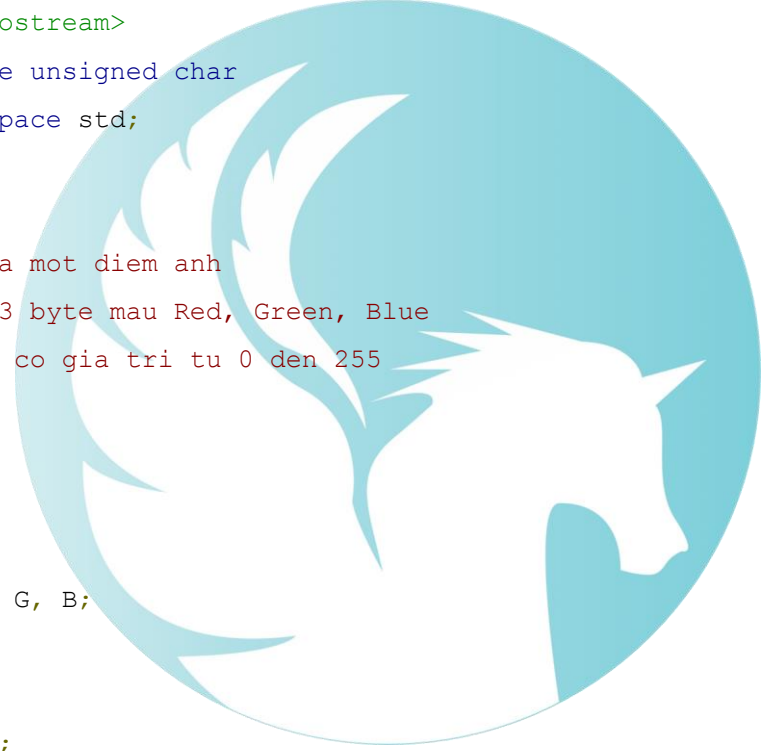
void sort();

/**
 * In ra nội dung
 */
void print();
};

```

Hãy hoàn thành các phương thức còn thiếu trong đoạn code trên để chương trình có thể hoạt động bình thường

## Đề 2



```

#include <iostream>
#define byte unsigned char
using namespace std;

/**
 * Lop mo ta mot diem anh
 * cho boi 3 byte mau Red, Green, Blue
 * moi byte co gia tri tu 0 den 255
 */
class Pixel
{
private:
    byte R, G, B;

public:
    Pixel();
    Pixel(byte r, byte g, byte b);
    Pixel(int value);          // R - byte 0, G - byte 1, B - byte 2
    int GrayScale() const;    // lay muc xam (2R + 3G + B) / 8

    friend Pixel operator+(const Pixel &left, const Pixel &right);
    friend ostream &operator<<(ostream &os, const Pixel &val);
};

/**
 * Lop mo ta mot anh
 * co chieu rong, chieu cao

```

```

* va mang 2 chieu cac diem anh
*/
class Bitmap
{
    int width, height;
    Pixel **data;

    void freeData();
    void createData(int w, int h);

public:
    Bitmap();
    Bitmap(int width, int height);
    Bitmap(const Bitmap &bmp);

    int getWidth() const;
    int getHeight() const;

    /**
     * Lay anh con tu diem (x, y) co kich thuoc w x h
     */
    Bitmap Crop(int x, int y, int w, int h);

    /**
     * Lay tan suat cac muc xam trong anh
     */
    int *Histogram();

    Pixel &operator()(int x, int y);
    Bitmap &operator=(const Bitmap &bmp);
};

void Bitmap::freeData()
{
    if (data != 0)
    {
        for (int i = 0; i < height; i++)
            delete[] data[i];
        delete[] data;
    }
}

```

```

        data = 0;
    }
}

void Bitmap::createData(int w, int h)
{
    width = w;
    height = h;
    data = new Pixel *[height];
    for (int i = 0; i < height; i++)
        data[i] = new Pixel[width];
}

int main()
{
    Pixel p(0, 60, 120);
    cout << p + Pixel(10, 10, 10);
    // cho ra kết quả trên màn hình: (10, 70, 130)

    Bitmap A;
    Bitmap B(100, 100);

    for (int y = 0; y < B.getHeight(); y++)
    {
        for (int x = 0; x < B.getWidth(); x++)
            B(x, y) = Pixel(rand());
    }
    A = B;

    int *hist = A.Histogram();
    for (int i = 0; i < 256; i++)
        cout << hist[i] << endl;

    Bitmap C(B.Crop(20, 20, 40, 40));
    return 0;
}

```

Hoàn thành các hàm thành viên của các lớp để có thể thực hiện đúng các biểu thức của hàm main

**Đề 3**

```

#include <iostream>
#include <string>
using namespace std;

template <class T>
class Node
{
public:
    T Value;
    Node *Next;
    Node(T, v) : Value(v), Next(0) {}
};

template <class T>
class List
{
public:
    Node<T> *head, *tail;
    int count;

    List() : count(0) {}
    ~List(); // Ham huy

    void RemoveAll()
    {
        while (count)
        {
            --count;
            Node<T> *p = head;
            head = p->Next;
            delete p;
            p = 0;
        }
    }

    void Add(T v)
    {
        Node<T> *p = new Node<T>(v);
    }
}

```

```

        if (count++ == 0)
            head = p;
        else
            tail->Next = p;
        tail = p;
    }

    Node<T> *First() const { return head; }
    int Count() const { return count; }

    T *ToArray()
    {
        T *arr = new T[count];
        Node<T> *p = head;
        for (int i = 0; i < count; ++i, p = p->Next)
            arr[i] = p->Value;
        return arr;
    }
};

class Date
{
public:
    int Day, Month, Year;
    Date() : Day(0), Month(0), Year(0) {}
    Date(int, int, int); //Hàm khởi tạo thiết lập các giá trị năm, tháng,
ngay
    friend ostream &operator>>(ostream &, const Date &);
};

class ThiSinh
{
public:
    Date NgaySinh;
    string HoTen;
    double Toan, Ly, Anh;

public:
    ThiSinh() {}

```

```

    ThiSinh(const char *hoTen, const Date ns, double toan, double ly,
double anh);

    double Tong() const; // Tính tổng điểm
    double TB() const;   // Trung bình các điểm
    friend ostream &operator<<(ostream &, const ThiSinh &);
};

class ListThiSinh : public List<ThiSinh *>
{
public:
    ListThiSinh(ThiSinh **arr)
    {
        int i = 0;
        while (arr[i])
            this->Add(arr[i++]);
    }
};

int main()
{
    ThiSinh *arr[] = {
        new ThiSinh("Nguyen Thanh Son", Date(2001, 1, 15), 8, 7, 8),
        new ThiSinh("Tran Hai Nam", Date(2001, 2, 18), 9, 7, 6),
        new ThiSinh("Vu Duy Anh", Date(2001, 5, 1), 7, 10, 8),
        new ThiSinh("Phan Dinh Hai Long", Date(2001, 5, 2), 8, 7, 9),
        new ThiSinh("Nguyen Thi Viet Ha", Date(2001, 4, 7), 5, 8, 8),
        new ThiSinh("Nguyen Duc Nam", Date(2001, 9, 12), 8, 8, 7),
        0};

    ListThiSinh cacThiSinh(arr);
    auto p = cacThiSinh.First();
    int soLuong = cacThiSinh.Count();
    for (int i = 0; i < soLuong; i++, p = p->Next)
        cout << *p->Value << endl;
}

```

1. Hãy bổ sung hoặc hoàn thiện các phương thức của các lớp để khi chạy chương trình được kết quả chính xác dưới đây:

Nguyen Thanh Son	15.01.2001	8	7	8	31	7.75
Tran Hai Nam	18.02.2001	9	7	6	31	7.75



## BCORN – Hỗ trợ học tập sinh viên Bách khoa

Vu Duy Anh	01.05.2001	7	10	8	32	8
Phan Dinh Hai Long	02.05.2001	8	7	9	32	8
Nguyen Thi Viet Ha	07.04.2001	5	8	8	26	6.5
Nguyen Duc Nam	12.09.2001	8	8	7	31	7.75

2. In danh sách thí sinh theo thứ tự tổng điểm giảm dần

3. Thống kê tỷ lệ khá, giỏi (từ 8 điểm trở lên) theo điểm trung bình.



## CHƯƠNG 5. MỘT SỐ PROJECT NHỎ

Yêu cầu:

- Có sử dụng con trỏ
- Có sử dụng đọc từ file và ghi ra file / cổng nối tiếp
- Có dùng hàm
- Có tạo và sử dụng thư viện hàm

### Đề 1:

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- **Điểm số hệ 4** (giả sử hệ số điểm giữa kì, cuối kì luôn là 3 / 7)
- Điểm số dạng chữ
- Kết luận sinh viên nào đạt, không đạt ứng với từng môn (1/0)
- Xuất kết quả ra file .csv

Định dạng file .csv đầu vào như sau:

MSSV, Học kì, Mã học phần, Số Tín chỉ, Điểm QT, Điểm Cuối kì

	A	B	C	D	E	F
1	MSSV	Học kỳ	Mã Học phần	Số TC	Điểm QT	Điểm CK
2	121	1	110	0	7.5	7
3	121	1	122	4	7	8
4	121	1	123	3	9	8.5
5	121	1	146	3	7	8.5

Định dạng file .csv cuối kì như sau: Tất cả các giá trị đều dạng số nguyên hoặc số thực

MSSV, Học kì, Mã học phần, Số Tín chỉ, Điểm QT, Điểm Cuối kì, Điểm Môn học (hệ 10), Điểm môn học (hệ 4), Đạt/không Đạt (1/0)

### Đề 2:

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Nhập mã sinh viên, mã môn, mã kì: tính điểm môn học hệ 4 của từng môn, từng sinh viên, xuất ra màn hình.
- Nhập mã môn 1, và môn 2 từ bàn phím, tính hệ số tương quan giữa 2 môn với nhau dựa trên tập dữ liệu đã cho.
- **Tính hệ số tương quan giữa Điểm Môn học (hệ 4) giữa các môn với nhau** của toàn bộ tập dữ liệu được cho, lưu kết quả vào file .csv. Kết quả lưu là một ma trận đối xứng qua đường chéo chính, giá trị hàng i, cột j là hệ số tương quan của môn i và môn j.

### **Đề 3:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Nhập mã môn học: tính hệ số tương quan giữa Điểm giữa kì và điểm cuối kì của môn đó
- Tính hệ số tương quan của tất cả các môn với nhau. Lưu kết quả vào file .csv. Kết quả là một ma trận 2 cột: cột đầu tiên là mã môn học/học phần, cột thứ 2 là kết quả hệ số tương quan. Chú ý xử lý các trường hợp ngoại lệ: không có điểm giữa kì, ...

### **Đề 4:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Nhập mã sinh viên, mã kì: **tính điểm GPA (hệ 4)** của sinh viên trong kì đó
- Nhập mã kì: tính điểm GPA của tất cả sinh viên trong kì đó, xuất ra dạng file .csv. Ma trận kết quả gồm 2 cột: cột đầu tiên là mã số sinh viên, cột thứ 2 là điểm GPA của kì đó
- Tính GPA của toàn bộ sinh viên ứng với file dữ liệu đầu vào. Xuất dữ liệu ra file dạng csv. Mỗi hàng ứng với một sinh viên. Mỗi cột ứng với 1 kì. Kết quả là một ma trận. Điểm GPA tính ở dạng số hệ 4.
  - cột đầu tiên là mã sinh viên,
  - cột thứ 2 là điểm GPA của kì 1,
  - cột thứ 3 là điểm GPA của kì 2,
  - cột thứ 4 là điểm GPA của kì 3,
  - ...

- cột thứ 8 là điểm GPA của kì 7

### **Đề 5:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Nhập mã số sinh viên và mã kì, lọc bỏ các lần học điểm thấp, chỉ giữ lại lần học có điểm cao nhất tính đến 1 mã kì xác định được nhập,
- Xuất kết quả ra màn hình
- Xuất dữ liệu ra file .csv
- Áp dụng để tạo ma trận kết quả học tập của sinh viên:
  - cột đầu tiên là mã môn học,
  - cột thứ 2 là điểm cao nhất của môn học đó tính đến kì 1,
  - cột thứ 3 là điểm cao nhất của môn đó tính đến kì 2,
  - cột thứ 4 là điểm cao nhất của môn đó tính đến kì 3,
  - ...
  - cột thứ 8 là điểm cao nhất của môn đó tính đến kì 7

Xuất ma trận trên ra file .csv

### **Đề 6:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện để tính CPA của sinh viên. Chương trình cho phép:

- Nhập mã số sinh viên, mã số kì từ bàn phím, trả về CPA tương ứng tính đến hết kì đó.
- Xuất dữ liệu ra file dạng csv. Mỗi hàng ứng với một sinh viên. Mỗi cột ứng với 1 kì. Kết quả là một ma trận. Điểm CPA tính ở dạng số hệ 4.

### **Đề 7:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Viết chương trình cho phép nhập mã môn học, tính histogram của điểm Môn học của môn đó với tất cả dữ liệu được cung cấp trong file. Hiển thị kết quả ra màn hình và lưu vào file csv. Kết quả lưu trong file .csv. Dòng đầu tiên là mã môn học. Từ dòng thứ 2 trở đi là các giá trị để vẽ histogram. Cột đầu tiên là mốc

biên dưới, cột thứ 2 là số phần tử thuộc phạm vi, cột thứ 3 là số phần tử thuộc phạm vi tính bằng phần trăm.

- Mở rộng: tìm thư viện để vẽ histogram và biểu diễn trên màn hình

### **Đề 8:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Nhập mã số sinh viên và số kì, thống kê:
  - Số tín chỉ sinh viên đăng kí trong kì. Xuất ra màn hình và xuất ra file .csv
  - Số môn sinh viên đăng kí trong kì
  - Số môn sinh viên đạt trong kì
  - Số tín chỉ sinh viên đạt trong kì
  - Điểm GPA của sinh viên trong kì
- Xuất ra file kết quả tổng hợp của tất cả các sinh viên trong cơ sở dữ liệu:
  - Cột đầu tiên là mssv
  - Cột thứ 2 là số tín chỉ đăng kí
  - Cột thứ 3 là số tín chỉ đạt
  - Cột thứ 4 là số môn đăng kí
  - Cột thứ 5 là số môn đạt
  - Cột thứ 6 là GPA của kì
- Giải thích các biện pháp xử lí ngoại lệ

### **Đề 9:**

Cho thông tin kết quả học tập của sinh viên trong file .csv. Viết thư viện và dùng thư viện tính:

- Nhập mã học phần. Tính các giá trị thống kê của mã học phần đó từ dữ liệu được cho trong file. Xuất ra màn hình và lưu vào file .csv
- Các giá trị cần tính:
  - Điểm môn học hệ 10
  - Điểm môn học hệ 4
  - Điểm cao nhất
  - Điểm thấp nhất

- Phân bố histogram của điểm theo hệ 4 với bước nhảy 0.5 điểm (tính bằng số lượng và bằng %)
- Tính các hệ số: Median, Q1, Q3, Q3-Q1 (interquartile)  
(<https://www.statisticshowto.com/probability-and-statistics/interquartile-range/#interquartile%20range%20by%20hand>)

### **Đề 10:**

Cho thông tin kết quả học tập của sinh viên và quê quán của sinh viên trong file .csv.

Viết thư viện và dùng thư viện thực hiện các công việc sau:

- Nhập mã số môn, tính điểm cuối kì của môn đó
- Tìm hệ số tương quan giữa nơi sinh với điểm cuối kì của môn đó, lưu vào file .csv. Ma trận kết quả gồm 2 cột: cột đầu là mã nơi sinh, cột thứ 2 là hệ số tương quan của nơi sinh với môn đang xét
- Xây dựng ma trận tính hệ số tương quan giữa nơi sinh với môn đang xét. Mỗi hàng tương ứng với một môn học. Mỗi cột tương ứng với một nơi sinh. Xuất dữ liệu ra dạng file .csv

### **Đề 11:**

Cho thông tin quê quán sinh viên trong file .csv. Viết thư viện và dùng thư viện thực hiện các công việc sau:

- Thống kê số sinh viên theo quê quán từ dữ liệu được cung cấp. Mỗi tương ứng với một mã quê quán. Cột thứ 1 là mã quê quán, cột thứ 2 là số sinh viên, cột thứ 3 là tỉ lệ sinh viên quê đó / tổng số sinh viên được dùng để thống kê tính bằng (%).
- Xuất dữ liệu trên ra file .csv.

## TÀI LIỆU THAM KHẢO

Kipalog (<https://kipalog.com>)

Viblo (<https://viblo.asia>)

Cộng đồng lập trình Việt Nam (<http://diendan.congdongcviet.com>)

GeeksforGeeks (<https://www.geeksforgeeks.org>)

Lập trình không khó (<https://nguyenvanhieu.vn>)

VNOI (<http://vnoi.info>)

Stdio (<https://www.stdio.vn>)

Stackoverflow (<https://stackoverflow.com>)

w3schools (<https://www.w3schools.com>)

