Bài 46: CẤP PHÁT ĐỘNG TRONG C++ (DYNAMIC MEMORY ALLOCATION).

Xem bài học trên website để ủng hộ Kteam: Cấp phát động trong C++ (Dynamic memory allocation).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage How Kteam nhé!

Dẫn nhập

Ở bài học trước, mình đã chia sẻ cho các bạn về sự liên quan giữa CÁC PHÉP TOÁN TRÊN CON TRỎ & CHỈ MỤC MẢNG TRONG C++. Kiến thức khá quan trọng về con trỏ và mảng mà bạn cần nắm.

Hôm nay, chúng ta sẽ cùng tìm hiểu về **Cấp phát động trong C++ (Dynamic memory allocation)**.

Nội dung

Để đọc hiểu bài này tốt nhất các bạn nên có kiến thức cơ bản về:

- BIẾN TRONG C++ (Variables)
- BIẾN CỤC BỘ TRONG C++ (Local variables)
- BIẾN TOÀN CỤC TRONG C++ (Global variables)
- BIẾN TĨNH TRONG C++ (Static variables)
- CON TRỎ TRONG C++ (Pointer)

Trong bài ta sẽ cùng tìm hiểu các vấn đề:

- Cấp phát bộ nhớ trong C++
- Cấp phát động trong C++

Cấp phát bộ nhớ trong C++

Ngôn ngữ C++ hỗ trợ **ba loại cấp phát bộ nhớ** cơ bản, hai loại trong số đó bạn đã được học ở những bài học trước:

1. **Cấp phát bộ nhớ tĩnh (Static memory allocation)**:

- Xảy ra trên các **biến tĩnh** và **biến toàn cục**.
- Vùng nhớ của các loại biến này được **cấp phát một lần** khi chương trình bắt đầu chạy và vẫn **tồn tại trong suốt thời gian tồn tại của chương trình**.
- Kích thước của biến/mảng phải được biết tại thời điểm biên dịch chương trình.

2. **Cấp phát bộ nhớ tự động (Automatic memory allocation)**:

- Xảy ra trên các **tham số hàm** và **biến cục bộ**.
- Vùng nhớ của các loại biến này được cấp phát khi chương trình **đi vào khối lệnh** và được giải phóng khi **khối lệnh bị thoát**.
- Kích thước của biến/mảng phải được biết tại thời điểm biên dịch chương trình.

3. **Cấp phát bộ nhớ động (Dynamic memory allocation)** sẽ được nói đến trong bài học này.

Trong hầu hết các trường hợp, **cấp phát bộ nhớ tĩnh** và **tự động** có thể đáp ứng tốt các yêu cầu của chương trình. Tuy nhiên, ta cùng xem ví dụ bên dưới:

Ví dụ: Chúng ta cần sử dụng một chuỗi để lưu tên của người dùng, nhưng chúng ta không biết tên của họ dài bao nhiêu cho đến khi họ nhập tên. Hoặc chúng ta cần lưu trữ danh sách nhân viên trong một công ty, nhưng chúng ta không biết trước được công ty đó sẽ có bao nhiêu nhân viên.

Đối với **cấp phát bộ nhớ tĩnh** và **tự động**, kích thước của biến/mảng phải được **biết tại thời điểm biên dịch chương trình**. Vì vậy, điều tốt nhất chúng ta có thể làm là cố gắng **đoán một kích thước tối đa** của các biến đó:

```
char name[30]; // tên không quá 30 ký tự
Staff staff[500]; // công ty không quá 500 nhân viên
```

Khuyết điểm của cách khai báo trên:

1. Gây lãng phí bộ nhớ nếu các biến không thực sự sử dụng hết kích thước khi khai báo. Ví dụ: nếu công ty chỉ có 100 nhân viên, chúng ta có 400 vùng nhớ nhân viên không được sử dụng tới.
2. Thứ hai, hầu hết **các biến thông thường (bao gồm mảng tĩnh)** được cấp phát trong một phần bộ nhớ gọi là **ngăn xếp (stack)**. Kích thước **bộ nhớ stack** cho một chương trình khá nhỏ (khoảng **1Mb** với Visual Studio), nếu yêu cầu cấp phát vùng nhớ vượt quá con số này, **chương trình sẽ bị đóng** bởi hệ điều hành với lỗi stack overflow.

```
char byte[1000000 * 2]; // khoảng 2Mb bộ nhớ => lỗi stack overflow
```

3. Thứ ba, điều gì xảy ra nếu công ty có 600 nhân viên, trong khi mảng staff chỉ có 500 phần tử. Lúc này, chương trình sẽ bị giới hạn bởi kích thước được khai báo ban đầu.

Để giải quyết những hạn chế trên, **cấp phát bộ nhớ động** được ra đời.

Cấp phát động trong C++

Cấp phát bộ nhớ động (Dynamic memory allocation) là cách để yêu cầu bộ nhớ từ hệ điều hành **khi cần thiết** (thời điểm chương trình đang chạy). **Cấp phát bộ nhớ động** sử dụng vùng nhớ được quản lý bởi hệ điều hành được gọi là **heap**. Ngày nay, bộ nhớ **heap** có thể có kích thước **gigabyte**.

Cấp phát động cho các biến đơn (Dynamically allocating single variables)

Để cấp phát động cho một biến, ta sử dụng toán tử **new**:

```
new int; // cấp phát động một số nguyên (kiểu dữ liệu có thể thay đổi)
```

Trong ví dụ trên, chương trình yêu cầu cấp phát vùng nhớ của một số nguyên từ hệ điều hành. Toán tử **new** tạo đối tượng sử dụng vùng nhớ đó và sau đó trả về một con trỏ chứa địa chỉ của vùng nhớ đã được cấp phát.

Thông thường, để truy cập vào vùng nhớ được cấp phát, chúng ta dùng **con trỏ** để **lưu giữ địa chỉ** được trả về bởi toán tử **new**:

```
// cấp phát động một số nguyên và gán địa chỉ cho con trỏ ptr nắm giữ
int *ptr = new int;
```

Sau đó, chúng ta có thể thao tác trên vùng nhớ vừa được cấp phát thông qua con trỏ:

```
int *ptr = new int;
*ptr = 10; // gán 10 cho vùng nhớ vừa được cấp phát
```

Khi cấp phát động cho một biến, bạn có thể cùng lúc khởi tạo giá trị cho nó:

```
int *ptr1 = new int(10);
int *ptr2 = new int{ 20 };
```

Xóa các biến đơn (Deleting single variables)

Khi chúng ta không còn sử dụng một biến được cấp phát động, chúng ta cần trao quyền quản lý vùng nhớ đó lại cho hệ điều hành. Đối với các **biến đơn (không phải mảng)**, điều này được thực hiện thông qua toán tử **delete**:

```
int *ptr = new int;

delete ptr; // trả lại vùng nhớ ptr đang trỏ đến cho hệ điều hành
ptr = nullptr; // gán ptr thành con trỏ null
```

Toán tử **delete** không **thực sự xóa** bất cứ điều gì. Nó chỉ đơn giản là **trao lại quyền sử dụng vùng nhớ** được cấp phát cho hệ điều hành. Sau đó, hệ điều hành được **tự do gán lại vùng nhớ** đó cho một ứng dụng khác (hoặc ứng dụng này).

> **Chú ý:** Mặc dù câu lệnh "**delete ptr**" giống như việc xóa một biến, nhưng thực tế không phải! Biến con trỏ **ptr** vẫn có thể sử dụng như trước và **có thể được gán một giá trị mới** giống như bất kỳ biến nào khác.

Con trỏ lơ lửng (Dangling pointers)

Thông thường, khi **delete** một con trỏ, vùng nhớ được **trả lại cho hệ điều hành** sẽ **chứa cùng giá trị** mà nó có trước đó. Lúc này, con trỏ đang trỏ sang một vùng nhớ chưa được cấp phát (hệ điều hành quản lý).

Con trỏ trỏ đến vùng nhớ **chưa được cấp phát** gọi là một **con trỏ lơ lửng (Dangling pointers)**. **Truy cập vào vùng nhớ (dereferencing pointer)** hoặc **xóa** một **con trỏ lơ lửng** sẽ dẫn đến lỗi **undefined behavior**.

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

int main()
{
	// cấp phát động một số nguyên và gán địa chỉ cho con trỏ ptr nắm giữ
	int *ptr = new int;
	*ptr = 10; // gán 10 vào vùng nhớ được cấp phát

	// giải phóng vùng nhớ cho hệ điều hành, ptr đang là con trỏ lơ lửng
	delete ptr;

	// truy cập vùng nhớ ptr đang trỏ tới => lỗi undefined behavior
	cout << *ptr;

// giải phóng vùng nhớ con trỏ đã được giải phóng trước đó => lỗi undefined behavior
delete ptr;

	return 0;
}
```

Việc giải phóng **một vùng nhớ** cũng có thể tạo ra **nhiều** con trỏ lơ lửng (dangling pointers).

Ví dụ:

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

int main()
{
	int *ptr = new int; // cấp phát động một số nguyên
	int *otherPtr = ptr; // otherPtr và ptr đang cùng trỏ đến một vùng nhớ

	// giải phóng vùng nhớ cho hệ điều hành, ptr và otherPtr đang là con trỏ lơ lửng
	delete ptr;

	ptr = nullptr; // ptr đang là con trỏ null

	// tuy nhiên, otherPtr vẫn là con trỏ lơ lửng!

	return 0;
}
```

Chú ý:

- **Tránh** sử dụng nhiều con trỏ trỏ vào **cùng một vùng nhớ**.
- Khi **xóa** một con trỏ, nếu chương trình không ra khỏi phạm vi của con trỏ ngay sau đó, hãy **gán con trỏ** thành **0** (hoặc **nullptr** trong C++ 11).

Con trỏ null và cấp phát động

Chúng ta đã được biết về **con trỏ null** trong bài CON TRỎ NULL TRONG C++ (NULL pointers).

Con trỏ null đặc biệt hữu ích trong cấp phát bộ nhớ động. Trong cấp phát bộ nhớ động, một **con trỏ null** có ý nghĩa "**không có vùng nhớ nào được cấp phát cho con trỏ này**".

```
#include <iostream>

int main()
{
	int *ptr = new int;
	*ptr = 10;

	delete ptr;
	ptr = nullptr;

	// kiểm tra con trỏ trước khi cấp phát
	// nếu ptr null (chưa được cấp phát), cấp phát ptr
	if (!ptr)
		ptr = new int;
*ptr = 20;

	// không cần kiểm tra con trỏ khi xóa
	// Nếu ptr không null, biến được cấp phát động sẽ bị xóa.
	// Nếu ptr là null, không có gì sẽ xảy ra.
	delete ptr;

	return 0;
}
```

Rò rỉ bộ nhớ trong C++ (Memory leaks)

Chúng ta cùng xem hàm bên dưới:

Ví dụ 1:

```
void doSomething()
{
	int *ptr = new int;
}
```

Trong hàm **doSomething()** cấp phát động một số nguyên, nhưng không sử dụng toán tử **delete** để giải phóng vùng nhớ đó. Vì con trỏ **tuân theo tất cả các quy tắc** giống như các biến thông thường, khi hàm kết thúc, **ptr** sẽ bị hủy. Mặt khác, **ptr** là biến duy nhất giữ địa chỉ của số nguyên được cấp phát động. Nghĩa là chương trình đã "**mất**" địa chỉ của bộ nhớ được cấp phát động trong hàm **doSomething()**. Kết quả là chương trình **không thể giải phóng vùng nhớ** được cấp phát động.

Vấn đề trên được gọi là **rò rỉ bộ nhớ (memory leaks)**. **Rò rỉ bộ nhớ** xảy ra khi chương **mất địa chỉ** của một số vùng nhớ được cấp phát động trước khi giải phóng nó cho hệ điều hành.

Khi rò rỉ bộ nhớ, chương trình của bạn **không thể xóa** bộ nhớ được cấp phát động, bởi vì chương trình **không còn nắm giữ địa chỉ** vùng nhớ đó. **Hệ điều hành** cũng **không thể sử dụng** vùng nhớ này, vì vùng nhớ đó vẫn nằm trong quyền sử dụng của chương trình.

Các chương trình gặp vấn đề rò rỉ bộ nhớ nghiêm trọng có thể lấy hết bộ nhớ có sẵn, làm cho hệ điều hành chạy chậm hoặc thậm chí bị crash. Chỉ sau khi chương trình tắt, hệ điều hành mới có thể dọn dẹp và "đòi lại" tất cả vùng nhớ bị rò rỉ.

Một số trường hợp khác có thể gây rò rỉ bộ nhớ trong C++:

Ví dụ 2: Con trỏ giữ địa chỉ của bộ nhớ được cấp phát động được gán một giá trị khác gây rò rỉ bộ nhớ.

```
int value = 10;
int *ptr = new int; // cấp phát vùng nhớ
ptr = &value; // địa chỉ vùng nhớ cấp phát trước đó bị mất, rò rỉ bộ nhớ
```

Ví dụ 3: Cấp phát vùng nhớ liên tục nhiều lần

```
int *ptr = new int;
ptr = new int; // địa chỉ vùng nhớ cấp phát trước đó bị mất, rò rỉ bộ nhớ
```

Để khắc phục vấn đề **rò rỉ bộ nhớ (memory leaks)** trong C++, chúng ta cần giải phóng vùng nhớ khi **ra khỏi phạm vi** con trỏ (ví dụ 1), hoặc **trước khi gán** (ví dụ 2), **cấp phát** một con trỏ (ví dụ 3).

Kết luận

Qua bài học này, bạn đã nắm được cách Cấp phát động trong C++ (Dynamic memory allocation). Với kỹ thuật này, bạn có thể tự do sử dụng bộ nhớ hệ thống một cách không giới hạn (giới hạn phần cứng) trong chương trình.

> **Lưu ý** rằng khi sử dụng cấp phát động, bạn cần nắm rõ những kiến thức cơ bản về cấp phát và giải phóng vùng nhớ trong bài viết này để tránh rò rỉ bộ nhớ, cũng như những vấn đề về vùng nhớ khác.

Trong bài tiếp theo, mình sẽ giới thiệu cho các bạn cách Cấp phát mảng động (Dynamically allocating arrays).

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thử thách – Không ngại khó**".