HÀM (P.2)

Kỹ thuật Lập trình (CO1027)

Ngày 3 tháng 4 năm 2021

ThS. Trần Ngọc Bảo Duy

Khoa Khoa học và Kỹ thuật Máy tính

Trường Đại học Bách Khoa, ĐHQG-HCM

Tổng quan

ThS.
Trần Ngọc Bảo Duy

TẦM VỰC CỦA BIẾN

ThS.
Trần Ngọc Bảo Duy

Khái niệm về khối và tầm vực

Định nghĩa

1. Một **khối mã lệnh** (block of code) là một loạt các lệnh có thứ tự nằm trong hai cặp ngoặc nhọn {...}.
2. **Tầm vực của danh hiệu** (scope of identifier) là đoạn chương trình mà ở nơi đó danh hiệu đó có thể được sử dụng.
3. **Tầm vực của danh hiệu** được bắt đầu ngay sau khi khai báo danh hiệu đó và kết thúc khi khối lệnh "trong cùng" mà nơi đó có chứa khai báo danh hiệu.
4. Có thể khai báo các danh hiệu giống nhau tại những khối lệnh khác nhau trong cùng một chương trình.

ThS.
Trần Ngọc Bảo Duy

Khái niệm về khối và tầm vực

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int value1 = 15;
6      cout << value1 << endl;
7
8      if (value1 > 7) {
9          int value2 = 9;
10         cout << value1 + value2 << endl;
11     }
12
13     cout << value2 << endl;
14     return 0;
15 }
```

Tầm vực của biến `value1` được xác định là bất kỳ nơi nào trong hàm `main` trước khi khối lệnh của hàm `main` được kết thúc (ở đây là dòng **15**).

ThS.
Trần Ngọc Bảo Duy

Khái niệm về khối và tầm vực

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int value1 = 15;
6      cout << value1 << endl;
7
8      if (value1 > 7) {
9          int value2 = 9;
10         cout << value1 + value2 << endl;
11     }
12
13     cout << value2 << endl;
14     return 0;
15 }
```

Tầm vực của biến `value2` được xác định là trong thân `if` (bắt đầu tại dòng **8**) trước khi khối lệnh của lệnh `if` được kết thúc (ở đây là dòng **11**).

ThS.
Trần Ngọc Bảo Duy

Khái niệm về khối và tầm vực

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int value1 = 15;
6      cout << value1 << endl;
7
8      if (value1 > 7) {
9          int value2 = 9;
10         cout << value1 + value2 << endl;
11     }
12
13     cout << value2 << endl;
14     return 0;
15 }
```

Việc truy cập của `value1` ở dòng số 10 là hợp lệ.

ThS.
Trần Ngọc Bảo Duy

Khái niệm về khối và tầm vực

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int value1 = 15;
6      cout << value1 << endl;
7
8      if (value1 > 7) {
9          int value2 = 9;
10         cout << value1 + value2 << endl;
11     }
12
13     cout << value2 << endl;
14     return 0;
15 }
```

Việc truy cập của `value2` ở dòng số 13 là không hợp lệ.

ThS.
Trần Ngọc Bảo Duy

Tầm vực của biến

1. **Tầm vực khối** (block scope): là những biến được khai báo nằm trong các khối lệnh như thân của các cấu trúc rẽ nhánh, cấu trúc lặp, ...

```
1  int main() {
2      int x; cin >> x;
3      if (x > 10) {
4          int y; cin >> y;
5          cout << x + y << endl;
6      }
7
8      {
9          int z = 0; cin >> z;
10         cout << x + z;
11     }
12 }
```


ThS.
Trần Ngọc Bảo Duy

Tầm vực của biến

2. **Tầm vực địa phương** (local scope): là những biến được khai báo nằm trong thân hàm hoặc như một tham số trong danh sách tham số của hàm.

```
1 void swap(int a, int b) {
2       int t = a;
3       a = b;
4       b = t;
5 }
```


ThS.
Trần Ngọc Bảo Duy

Tầm vực của biến

❸ **Tầm vực toàn cục** (global scope): là những biến được khai báo nằm ngoài tất cả các hàm của chương trình, kể cả hàm `main`.

```
1  int g = 20;
2
3  int main() {
4      int x = 0; cin >> x;
5      if (x > 10) {
6          cout << g * x;
7      }
8
9      return 0;
10 }
```


ThS.
Trần Ngọc Bảo Duy

Toán tử phân giải tầm vực

- Khi một biến cục bộ cùng tên với một biến toàn cục, mọi sử dụng đến tên đó đều trong tầm vực của biến cục bộ đều tham chiếu đến biến cục bộ.

ThS.
Trần Ngọc Bảo Duy

Toán tử phân giải tầm vực

- Khi một biến cục bộ cùng tên với một biến toàn cục, mọi sử dụng đến tên đó đều trong tầm vực của biến cục bộ đều tham chiếu đến biến cục bộ.
- Trong trường hợp đó, muốn sử dụng biến toàn cục, ta phải sử dụng toán tử phần giải tầm vực (scope resolution scope) :: ngay trước tên biến.

ThS.
Trần Ngọc Bảo Duy

Toán tử phân giải tầm vực

- Khi một biến cục bộ cùng tên với một biến toàn cục, mọi sử dụng đến tên đó đều trong tầm vực của biến cục bộ đều tham chiếu đến biến cục bộ.
- Trong trường hợp đó, muốn sử dụng biến toàn cục, ta phải sử dụng toán tử phần giải tầm vực (scope resolution scope) :: ngay trước tên biến.
- Toán tử :: sẽ chỉ cho trình biên dịch biết là ta đang sử dụng biến toàn cục.

ThS.
Trần Ngọc Bảo Duy

Toán tử phân giải tầm vực

```
1  #include <iostream>
2  using namespace std;
3
4  float number = 24.8;
5
6  int main() {
7      float number = 48.2;
8      cout << number << "␣" << ::number << endl;
9      return 0;
10 }
```

Chương trình in ra:

```
48.2 24.8
```


ThS.
Trần Ngọc Bảo Duy

ThS.
Trần Ngọc Bảo Duy

LỚP LƯU TRỮ CỦA BIẾN

Lớp lưu trữ của biến

Định nghĩa

- Lớp lưu trữ của biến (variable storage class) là khái niệm dùng để chỉ thời gian sống (lifetime) của một biến.
- Có 4 lớp lưu trữ: `auto`, `static`, `extern` và `register` tương ứng với các từ khóa dùng để khai báo.
- Để khai báo lớp lưu trữ, ta đặt các từ khóa trên trước các khai báo biến.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ của biến

Định nghĩa

- Lớp lưu trữ của biến (variable storage class) là khái niệm dùng để chỉ thời gian sống (lifetime) của một biến.
- Có 4 lớp lưu trữ: `auto`, `static`, `extern` và `register` tương ứng với các từ khóa dùng để khai báo.
- Để khai báo lớp lưu trữ, ta đặt các từ khóa trên trước các khai báo biến.

Ví dụ

```
1 auto int num;
2 static int miles;
3 register int dist;
4 extern float price;
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `auto`

- Lớp lưu trữ `auto` chỉ đến các biến chỉ tồn tại đúng bằng thời gian sống của chính khối lệnh (như hàm) chứa nó.
- Một biến không chỉ rõ lớp lưu trữ thì nó được xếp vào lớp lưu trữ `auto`.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ auto

- Lớp lưu trữ auto chỉ đến các biến chỉ tồn tại đúng bằng thời gian sống của chính khối lệnh (như hàm) chứa nó.
- Một biến không chỉ rõ lớp lưu trữ thì nó được xếp vào lớp lưu trữ auto.

```
1  #include <iostream>
2  using namespace std;
3
4  void test();
5
6  int main() {
7      int count;
8      for (count = 1; count <= 3; count++)
9          test();
10     return 0;
11 }
12
13 void test() {
14     int num = 0;
15     cout << num++ << endl;
16     return;
17 }
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `register`

- Từ khóa lớp lưu trữ `register` được sử dụng để định nghĩa các biến cục bộ mà nên được lưu giữ trong một thanh ghi thay vì bộ nhớ chính.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `register`

- Từ khóa lớp lưu trữ `register` được sử dụng để định nghĩa các biến cục bộ mà nên được lưu giữ trong một thanh ghi thay vì bộ nhớ chính.
- Các biến `register` có thời gian sống như các biến `auto`.

```
1 register int time;
2 register double difference;
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `register`

- Từ khóa lớp lưu trữ `register` được sử dụng để định nghĩa các biến cục bộ mà nên được lưu giữ trong một thanh ghi thay vì bộ nhớ chính.
- Các biến `register` có thời gian sống như các biến `auto`.

```
1 register int time;
2 register double difference;
```

- Chỉ được dùng cho các biến yêu cầu truy cập nhanh như các biến đếm (counters).

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `static`

- Lớp lưu trữ `static` trong C/C++ nói với trình biên dịch để giữ một biến cục bộ tồn tại trong toàn bộ thời gian sống của chương trình thay vì tạo và hủy biến mỗi lần nó vào và ra khỏi phạm vi biến.
- Các biến cục bộ `static` cho phép nó duy trì giá trị giữa các lần gọi hàm.
- Tuy nhiên, các biến cục bộ như vậy sẽ không được truy xuất ngoài khối lệnh chứa nó.
- Lớp lưu trữ static cũng có thể được áp dụng cho các biến toàn cục (global). Khi áp dụng cho biến toàn cục, nó nói với trình biên dịch rằng, phạm vi của biến toàn cục bị giới hạn trong tập tin mà nó được khai báo.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ static

```
1  int funct(int);
2  int main()
3  {
4      int c, v;
5      for(c = 1; c <= 10; c++){
6          v = funct(c);
7          cout << c << ' ' << v << endl;
8      }
9      return 0;
10 }
11
12 int funct( int x)
13 {
14     int sum = 100;
15     sum += x;
16     return sum;
17 }
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `static`

Kết quả của đoạn chương trình trên là:

```
1  101
2  102
3  103
4  104
5  105
6  106
7  107
8  108
9  109
10  110
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `static`

Kết quả của đoạn chương trình trên là:

```
1  101
2  102
3  103
4  104
5  105
6  106
7  107
8  108
9  109
10  110
```

Tác dụng tăng biến `sum` trong hàm `funct` sẽ bị mất khi điều khiển trả về cho hàm `main`.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ static

```
1  int funct(int);
2  int main()
3  {
4      int c, v;
5      for(c = 1; c <= 10; c++){
6          v = funct(c);
7          cout << c << ' ' << v << endl;
8      }
9      return 0;
10 }
11
12 int funct( int x)
13 {
14     static int sum = 100;
15     sum += x;
16     return sum;
17 }
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `static`

Kết quả của đoạn chương trình trên là:

```
1  101
2  103
3  106
4  110
5  115
6  121
7  128
8  136
9  145
10  155
```


ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `static`

Kết quả của đoạn chương trình trên là:

```
1  101
2  103
3  106
4  110
5  115
6  121
7  128
8  136
9  145
10  155
```

1. Việc khởi tạo biến `static` sẽ được thực hiện khi chương trình được dịch. Tại thời điểm dịch, biến `static` được tạo ra và nhận giá trị khởi tạo.
2. Tất cả các biến `static` được thiết lập giá trị 0 trong trường hợp không mang giá trị khởi tạo.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ `extern`

- Lớp lưu trữ `extern` trong C/C++ được dùng để cung cấp một tham chiếu của một biến toàn cục được nhìn thấy bởi **TẤT CẢ** các file chương trình.
- Khi sử dụng `extern`, biến không thể được khởi tạo, khi nó trỏ tới tên biến tại một vị trí lớp lưu trữ mà đã được định nghĩa trước đó.

ThS.
Trần Ngọc Bảo Duy

Lớp lưu trữ extern

Một chương trình được dịch từ 2 hai file:

1. File extern1.cpp:

```
1 #include <iostream>
2 int count;
3 extern void test();
4 int main()
5 {
6     count = 5;
7     test();
8     return 0;
9 }
```

2. File extern2.cpp:

```
1 #include <iostream>
2 extern int count;
3 void test(void)
4 {
5     cout << "count = " << count << endl;
6 }
```

ĐỆ QUY

ThS.
Trần Ngọc Bảo Duy

Đệ quy

Định nghĩa

1. C/C++ cho phép một hàm có khả năng gọi đến chính nó. Hàm như vậy được gọi là hàm tự tham chiếu (seft-referential) hoặc là **hàm đệ quy** (recursive function).
2. Trong một số bài toán, sẽ dễ hơn và tự nhiên hơn khi định nghĩa bài toán bằng cách dựa trên chính bài toán của nó.
3. Đệ quy sẽ hữu ích cho các bài toán được trình bày bằng dạng thức đơn giản hơn của chính bài toán đó.

ThS.
Trần Ngọc Bảo Duy

Đệ quy

Định nghĩa

1. C/C++ cho phép một hàm có khả năng gọi đến chính nó. Hàm như vậy được gọi là hàm tự tham chiếu (seft-referential) hoặc là **hàm đệ quy** (recursive function).
2. Trong một số bài toán, sẽ dễ hơn và tự nhiên hơn khi định nghĩa bài toán bằng cách dựa trên chính bài toán của nó.
3. Đệ quy sẽ hữu ích cho các bài toán được trình bày bằng dạng thức đơn giản hơn của chính bài toán đó.

Ví dụ: Giai thừa

$$n! = \begin{cases} 1 & \text{nếu } n = 0 \\ n \times (n-1) \times ... \times 2 \times 1 & \text{nếu } n > 0 \end{cases}$$

Sử dụng đệ quy:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

ThS.
Trần Ngọc Bảo Duy

Các loại đệ quy

1. Đệ quy trực tiếp : $A \rightarrow A$
2. Đệ quy gián tiếp : $A \rightarrow B \rightarrow A$

ThS.
Trần Ngọc Bảo Duy

Các thành phần cơ bản của đệ quy

Hai thành phần cơ bản của đệ quy

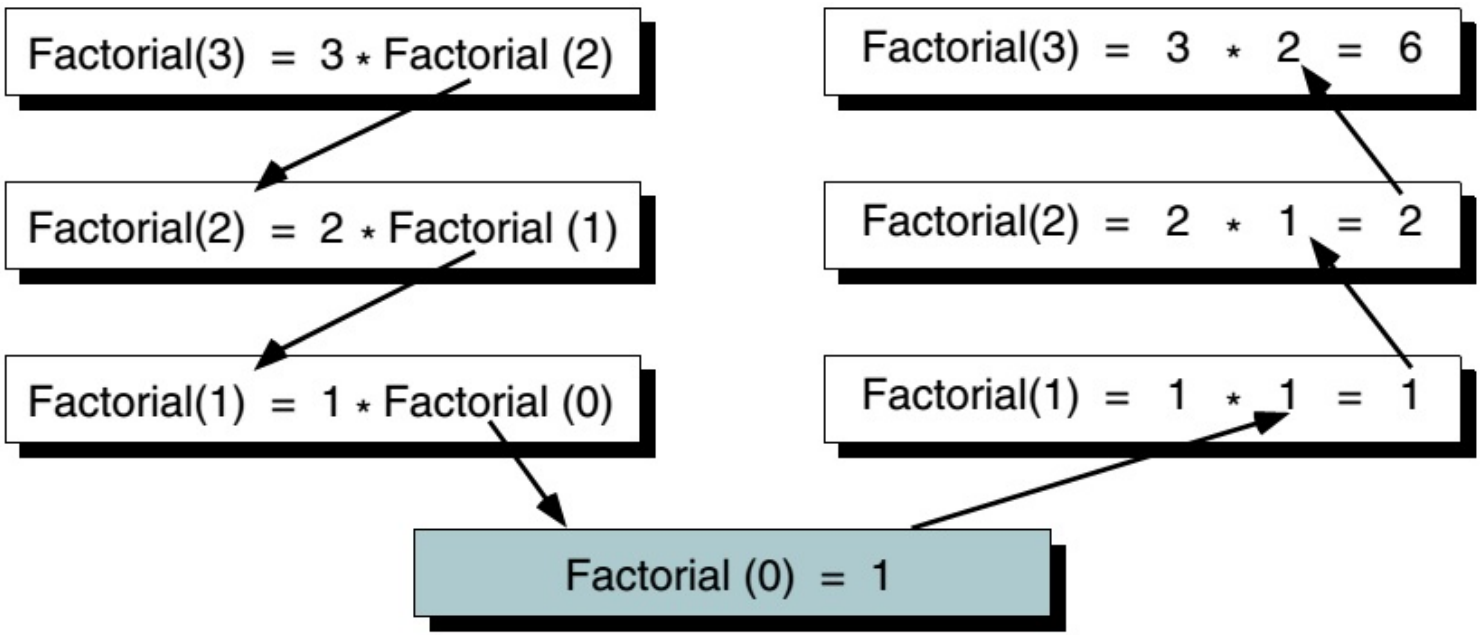
1. Trường hợp cơ sở (i.e. trường hợp dừng)
2. Trường hợp tổng quát (i.e. trường hợp đệ quy)

Ví dụ: Giai thừa

$$n! = \begin{cases} 1 & \text{nếu } n = 0 \quad \text{base} \\ n \times (n-1)! & \text{nếu } n > 0 \quad \text{general} \end{cases}$$

ThS.
Trần Ngọc Bảo Duy

Quy trình giải đệ quy



Hình: Lời gọi 3!

ThS.
Trần Ngọc Bảo Duy

Hiện thực trong C/C++

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4  unsigned long factorial(unsigned long);
5  int main(){
6      for (int i = 0; i <= 10; i++)
7          cout << setw(2) << i << "!␣=␣"
8               << factorial(i) << endl;
9      return 0;
10 }
11 // Recursive definition of function factorial
12 unsigned long factorial(unsigned long number){
13     if (number < 1) // base case
14         return 1;
15     else // recursive case
16         return number * factorial(number - 1);
17 }
```


ThS.
Trần Ngọc Bảo Duy

Nguyên tắc hiện thực đệ quy

1. Quyết định được trường hợp cơ sở (điều kiện dừng).
2. Quyết định được trường hợp đệ quy.
3. Hợp nhất hai trường hợp này vào trong cùng một hàm để hiện thực đệ quy.

ThS.
Trần Ngọc Bảo Duy

Giới hạn của đệ quy

Một **đệ quy** thường chạy chậm hơn một hàm **không đệ quy** tương đương:

- Lời gọi hàm tiêu tốn nhiều thời gian và chi phí hơn sử dụng biến đếm trong vòng lặp.
- Các ứng dụng hiệu năng cao rất hiếm khi sử dụng đệ quy.

ThS.
Trần Ngọc Bảo Duy