Ngôn ngữ lập trình

Bài 9:
Đệ Quy

Giảng viên: Lê Nguyễn Tuấn Thành

Email: thanhlnt@tlu.edu.vn

Bộ Môn Công Nghệ Phần Mềm – Khoa CNTT

Trường Đại Học Thủy Lợi

Nội dung

- Đệ quy với hàm void
  - Truy vết lời gọi đệ quy
  - Đệ quy vô hạn (infinite recursion), tràn (overflows)
- Đệ quy với hàm trả về giá trị
  - Hàm Power()
- Suy nghĩ theo kiểu đệ quy
  - Kỹ thuật thiết kế đệ quy
  - Tìm kiếm nhị phân

Bài giảng có sử dụng hình vẽ trong cuốn sách "*Absolute C++*. W. Savitch, Addison Wesley, 2002"

Minh họa Đệ Quy

Giới thiệu về đệ quy (recursion)

- Một hàm gọi chính nó
  - Trong định nghĩa của hàm đó, có lời gọi đến chính hàm đó
- C++ cho phép đệ quy
  - Giống như phần lớn ngôn ngữ lập trình bậc cao
  - Có thể là một kỹ thuật lập trình hữu ích
  - Có những giới hạn

Đệ quy với hàm void

- Chia để trị (Devide and Conquer)
  - Kỹ thuật thiết kế cơ bản
  - Chia các tác vụ lớn thành các tác vụ con
- Tác vụ con có thể là phiên bản nhỏ hơn của tác vụ gốc!
  - Khi đó gọi là đệ quy

Ví dụ Đệ quy với hàm void

- Xem xét tác vụ sau:
  - Tìm kiếm một giá trị trong danh sách
    - Tác vụ con 1: tìm kiếm nửa đầu của danh sách
    - Tác vụ con 2: tìm kiếm nửa sau của danh sách
- Các tác vụ con là phiên bản nhỏ hơn của tác vụ gốc!
- Khi điều này xảy ra, hàm đệ quy có thể được sử dụng

Đệ quy với hàm void: số theo chiều dọc

- Tác vụ: hiển thị các chữ số của một số nguyên theo chiều dọc, mỗi số một dòng
- Ví dụ lời gọi hàm *writeVertical(1234);* sẽ có kết quả:

  1
  2
  3
  4

số theo chiều dọc
định nghĩa hàm đệ quy

- Chia vấn đề thành 2 trường hợp
- Trường hợp đơn giản/cơ sở: if n<10
  - Đơn giản in số *n* ra màn hình
- Trường hợp đệ quy: if n>=10, có 2 tác vụ con:
  1. Hiển thị theo chiều dọc tất cả chữ số trừ chữ số cuối cùng
  2. Hiển thị chữ số cuối cùng
- Ví dụ: với tham số 1234
  - Tác vụ con 1: hiển thị 1,2,3 theo chiều dọc
  - Tác vụ con 2: hiển thị chữ số 4

Định nghĩa hàm writeVertical()

- Xét các trường hợp ở slide trước

```
void writeVertical(int n)
{
    if (n < 10)                    //Trường hợp cơ sở
        cout << n << endl;
    else
    {                              // Bước đệ quy
        writeVertical(n/10);
        cout << (n%10) << endl;
    }
}
```

Truy vết hàm writeVertical()

- Ví dụ lời gọi: *writeVertical(123);*

  → *writeVertical(12); (123/10)*

  → *writeVertical(1); (12/10)*

  → *cout << 1 << endl;*

  cout << 2 << endl;

  cout << 3 << endl;
- Mũi tên chỉ định tác vụ hàm thực hiện
- Chú ý hai lời gọi đầu tiên: gọi lại cùng hàm (đệ quy)
- Lời gọi cuối cùng hiển thị (1) và "kết thúc"

Đệ quy – một cái nhìn gần hơn

- Máy tính lưu vết các lời gọi đệ quy
  - Dừng hàm hiện tại
  - Phải biết kết quả của lời gọi đệ quy mới trước khi tiến hành xử lý (proceeding)
  - Lưu trữ mọi thông tin cần thiết cho lời gọi hiện tại
    - Để sử dụng sau
  - Tiến hành với đánh giá lời gọi đệ quy mới
  - Khi lời gọi đó hoàn thiện, trả lại cho tính toán bên ngoài ("outer" computation)

Đệ quy – bức tranh lớn

- Tổng quan về hàm đệ quy thành công:
  - Một hoặc nhiều trường hợp khi hàm hoàn thiện những nhiệm vụ của nó bằng cách:
    - Tạo một hoặc nhiều lời gọi đệ quy để giải quyết những phiên bản nhỏ hơn của tác vụ gốc
    - Được gọi là "các trường hợp đệ quy"
  - Một hoặc nhiều trường hợp khi hàm hoàn thiện tác vụ của nó mà không dùng lời gọi đệ quy
    - Được gọi là "trường hợp cơ sở" hoặc trường hợp dừng

Đệ quy vô hạn

- Trường hợp cơ sở cuối cùng phải được gọi
- Nếu không → đệ quy vô hạn
  - Lời gọi đệ quy không bao giờ dừng!
- Nhớ lại ví dụ: writeVertical()
  - Trường hợp cơ sở xảy ra khi xét đến số chỉ có 1 chữ số
  - Khi đó sự đệ quy sẽ dừng

Ví dụ đệ quy vô hạn

- Xét một định nghĩa hàm thay thế

```
void newWriteVertical(int n)
{
        newWriteVertical(n/10);
        cout << (n%10) << endl;
}
```

- Dường như hàm này là đầy đủ
- Thiếu "trường hợp cơ sở"!
- Đệ quy không bao giờ dừng

Ngăn xếp cho đệ quy

- Một ngăn xếp (stack)
  - Một cấu trúc bộ nhớ chuyên biệt
  - Giống ngăn xếp giấy
    - Đặt tờ mới trên đầu ngăn xếp
    - Lấy ra khi cần thiết từ đầu ngăn xếp
  - Được gọi là cấu trúc bộ nhớ "vào sau/ra trước" (last-in/first-out)
- Đệ quy sử dụng ngăn xếp
  - Mỗi lời gọi đệ quy được đặt trên ngăn xếp
  - Khi một lời gọi hoàn thành, nó sẽ được loại bỏ khỏi ngăn xếp

Tràn ngăn xếp (stack overflow)

- Kích thước của ngăn xếp là giới hạn
  - Bộ nhớ có hạn
- Chuỗi dài của lời gọi đệ quy tiếp tục được thêm vào ngăn xếp
  - Tất cả được thêm vào trước khi gặp trường hợp cơ sở (sẽ khiến lời gọi bị loại bỏ khỏi ngăn xếp)
- Nếu ngăn xếp cố gắng phát triển vượt quá giới hạn
  - Lỗi tràn ngăn xếp
- Đệ quy vô hạn luôn tạo ra lỗi này

Đệ quy so với cấu trúc lặp

- Đệ quy không phải luôn luôn cần thiết
- Thậm chí không cho phép trong một số ngôn ngữ
- Mọi tác vụ được hoàn thành với đệ quy có thể được làm mà không cần sử dụng đệ quy
  - Khử đệ quy (nonrecursive):gọi là cấu trúc lặp, sử dụng vòng lặp
- Đệ quy:
  - Chạy chậm, sử dụng nhiều bộ nhớ
  - Ít code hơn, giải pháp ngắn gọn hơn (elegent solution)

Hàm đệ quy trả về một giá trị

- Đệ quy không chỉ giới hạn với các hàm void
- Có thể trả về giá trị của bất kỳ kiểu nào
- Cùng một kỹ thuật:
  1. Một hoặc nhiều trường hợp có giá trị trả về được tính toán bởi lời gọi đệ quy: những vấn đề nhỏ hơn
  2. Một hoặc nhiều trường hợp có giá trị trả về được tính toán mà không có lời gọi đệ quy: trường hợp cơ sở

Trả về một giá trị
ví dụ đệ quy: hàm lũy thừa (Power)

- Nhớ lại hàm được định nghĩa trước pow():

  result = pow(2.0,3.0);

  - Trả về kết quả 2.0 lũy thừa 3 (8.0)
  - Nhận 2 tham số kiểu double
  - Trả về giá trị kiểu double
- Hãy viết theo cách đệ quy
  - Cho ví dụ đơn giản

Định nghĩa hàm power()

```
int power(int x, int n)
{
        if (n<0)
        {
                cout << "Illegal argument";
                exit(1);
        }
        if (n>0)
                return (power(x, n-1)*x);
        else
                return (1);
}
```

Gọi hàm power() (1/2)

- Ví dụ lời gọi hàm:
- *power(2, 0);*

  → *trả về giá trị 1*
- *power(2, 1);*
  → *trả về (power(2, 0) * 2);*
  → *trả về 1*
- Giá trị 1 được nhân với 2 và trả về cho lời gọi gốc

Gọi hàm power() (2/2)

- Ví dụ lớn hơn
- *power(2,3);*

  → *power(2,2)*2*

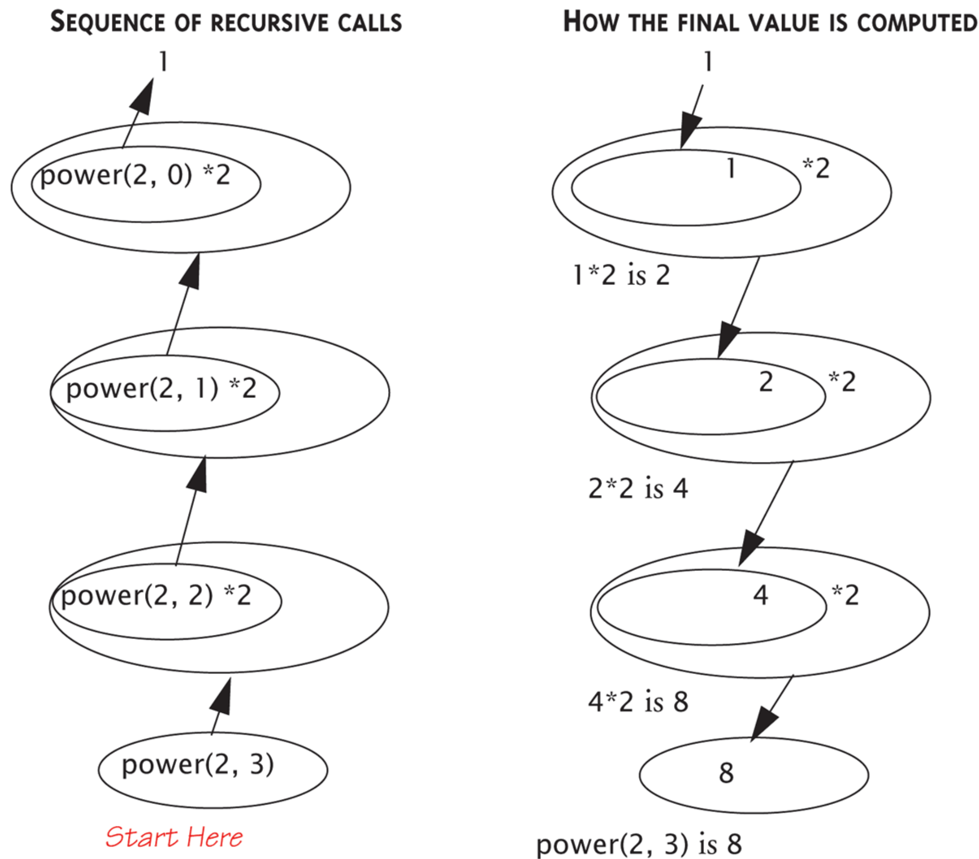
  → *power(2,1)*2*

  → *power(2,0)*2*

  → *1*

  - Đi đến trường hợp cơ sở
  - Dừng đệ quy
  - Giá trị được trả lại ngược trên ngăn xếp

Lưu vết hàm power()
đánh giá lời gọi hàm đệ quy power(2,3)



Suy nghĩ theo cách đệ quy

- Bỏ qua chi tiết
  - Quên đi cách ngăn xếp hoạt động
  - Quên đi những tính toán bị treo (suspended)
  - Đây là một nguyên tắc "trừu tượng" ("abstraction" principle)!
  - Và cũng là nguyên tắc đóng gói (encapsulation principle)!
- Hãy để máy tính thực hiện chi tiết ("bookkeeping")
  - Lập trình viên chỉ nghĩ về "bức tranh lớn" (big picture)

Suy nghĩ theo cách đệ quy: hàm power

- Xét lại hàm power()
- Định nghĩa đệ quy của hàm power():

  power(x, n)

  Trả về giá trị:

  *power(x, n – 1) * x*

  - Chỉ cần đảm bảo "công thức" đúng
  - Và đảm bảo tính đến trường hợp cơ sở

Kỹ thuật thiết kế đệ quy

- Không lưu vết toàn bộ chuỗi đệ quy
- Chỉ kiểm tra 3 thuộc tính:
    1. Không đệ quy vô hạn
    2. Các trường hợp dừng trả về giá trị đúng
    3. Các trường hợp đệ quy trả về giá trị đúng

Kiểm tra thiết kế đệ quy: hàm power()

- Kiểm tra hàm power() với 3 thuộc tính
  1. Không đệ quy vô hạn
     - Tham số thứ 2 giảm đi 1 mỗi lần gọi hàm
     - Cuối cùng phải gặp trường hợp cơ sở là 1
  2. Các trường hợp dừng trả về giá trị đúng
     - power(x,0) là trường hợp cơ sở
     - Trả về giá trị 1, đúng với x^0
  3. Các trường hợp đệ quy trả về giá trị đúng
     - Với n > 1, power(x,n) trả về power(x,n-1)*x
     - Giá trị này đúng

Tìm kiếm nhị phân

- Hàm đệ quy để tìm kiếm trong mảng
  - Quyết định xem liệu một phần tử có ở trong mảng không và nếu có:
  - Phần tử đó ở vị trí nào trong mảng
- Giả sử mảng đã được sắp xếp
- Chia danh sách thành 2 phần
  - Quyết định xem liệu phần tử đó ở nửa thứ nhất hay nửa thứ hai
  - Sau đó lại bắt đầu tìm kiếm trên nửa đó -> một cách làm theo kiểu đệ quy!

Mã giả cho tìm kiếm nhị phân

Display 13.5 Pseudocode for Binary Search

```
int a[Some_Size_Value];
```

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
//Precondition:
//a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
```

TO LOCATE THE VALUE KEY:

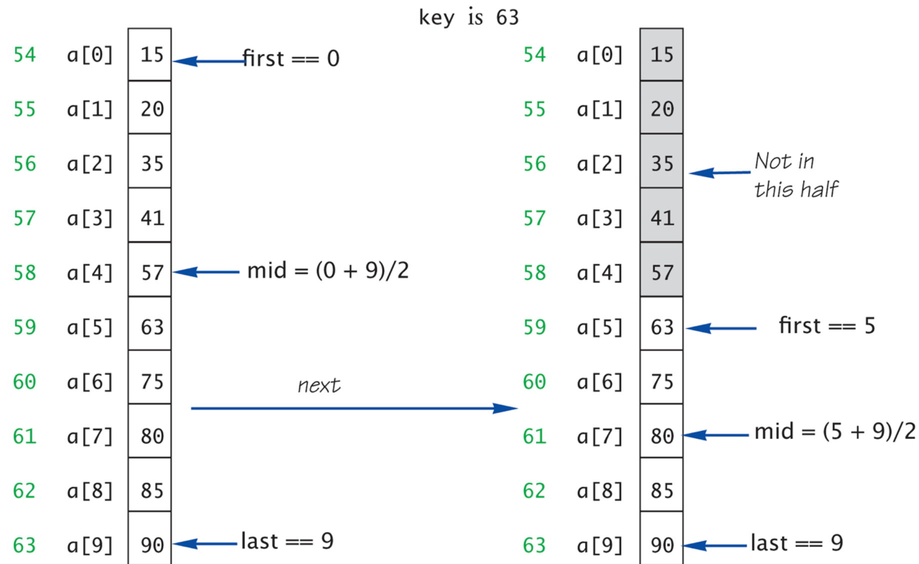
```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
 }
```

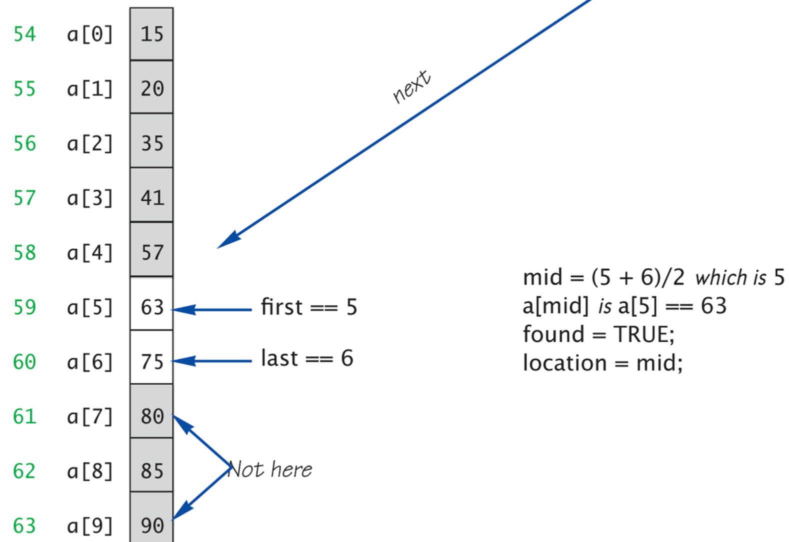
Kiểm tra sự đệ quy

- Kiểm tra tìm kiếm nhị phân theo các tiêu chí:
  1. Không đệ quy vô hạn
     - Mỗi lời gọi tăng *first* hoặc giảm *last*
     - Cuối cùng *first* sẽ lớn hơn *last*
  2. Các trường hợp dừng thực hiện hành động đúng
     - Nếu first > last → không có phần tử nào ở giữa chúng, do đó key không thể ở đó!
     - Nếu key == a[mid] → tìm được đúng!
  3. Các trường hợp đệ quy thực hiện hành động đúng
     - Nếu key < a[mid] → key ở trong nửa đầu tiên – lời gọi đúng
     - Nếu key > a[mid] → key ở trong nửa thứ hai – lời gọi đúng

kiếm nhị phân

Display 13.7 **Execution of the Function search**





mid = (5 + 6)/2 *which is* 5
a[mid] *is* a[5] == 63
found = TRUE;
location = mid;

Hiệu quả của tìm kiếm nhị phân

- Cực kỳ nhanh, khi so sánh với tìm kiếm tuần tự
- Một nửa của mảng bị loại bỏ tại thời điểm đầu tiên
  - Sau đó là 1/4, tiếp theo là 1/8, etc.
  - Về bản chất loại bỏ một nửa với mỗi lời gọi
- Ví dụ: một mảng có 100 phần tử
  - Tìm kiếm nhị phân không bao giờ cần đến nhiều hơn 7 so sánh!
  - Mức độ hiệu quả theo logarit (log n)

Những giải pháp đệ quy

- Chú ý rằng thuật toán tìm kiếm nhị phân thực sự giải quyết vấn đề "tổng quan hơn"
  - Mục tiêu ban đầu: thiết kế hàm để tìm kiếm trong toàn bộ mảng
  - Hàm của chúng ta: cho phép tìm kiếm bất kỳ đoạn con nào của mảng
    - Bằng cách chỉ định hai ranh giới *first* và *last*
- Rất phổ biến khi thiết kế hàm đệ quy

Tóm tắt

- Giảm một vấn đề thành các trường hợp nhỏ hơn của cùng một vấn đề -> giải pháp đệ quy
- Thuật toán đệ quy có 2 trường hợp
  - Trường hợp cơ sở/dừng
  - Trường hợp đệ quy
- Đảm bảo không có đệ quy vô hạn
- Sử dụng các tiêu chí để quyết định xem đệ quy đúng
  - Ba thuộc tính quan trọng
- Thường giải quyết vấn đề "tổng quát hơn"

Giáo trình Tham khảo

- **Giáo trình chính: W. Savitch, *Absolute* C++, Addison Wesley, 2002**
- Tham khảo:
  - A. Ford and T. Teorey, *Practical Debugging in C++*, Prentice Hall, 2002
  - Nguyễn Thanh Thủy, *Kĩ thuật lập trình C++*, NXB Khoa học và Kĩ Thuật, 2006