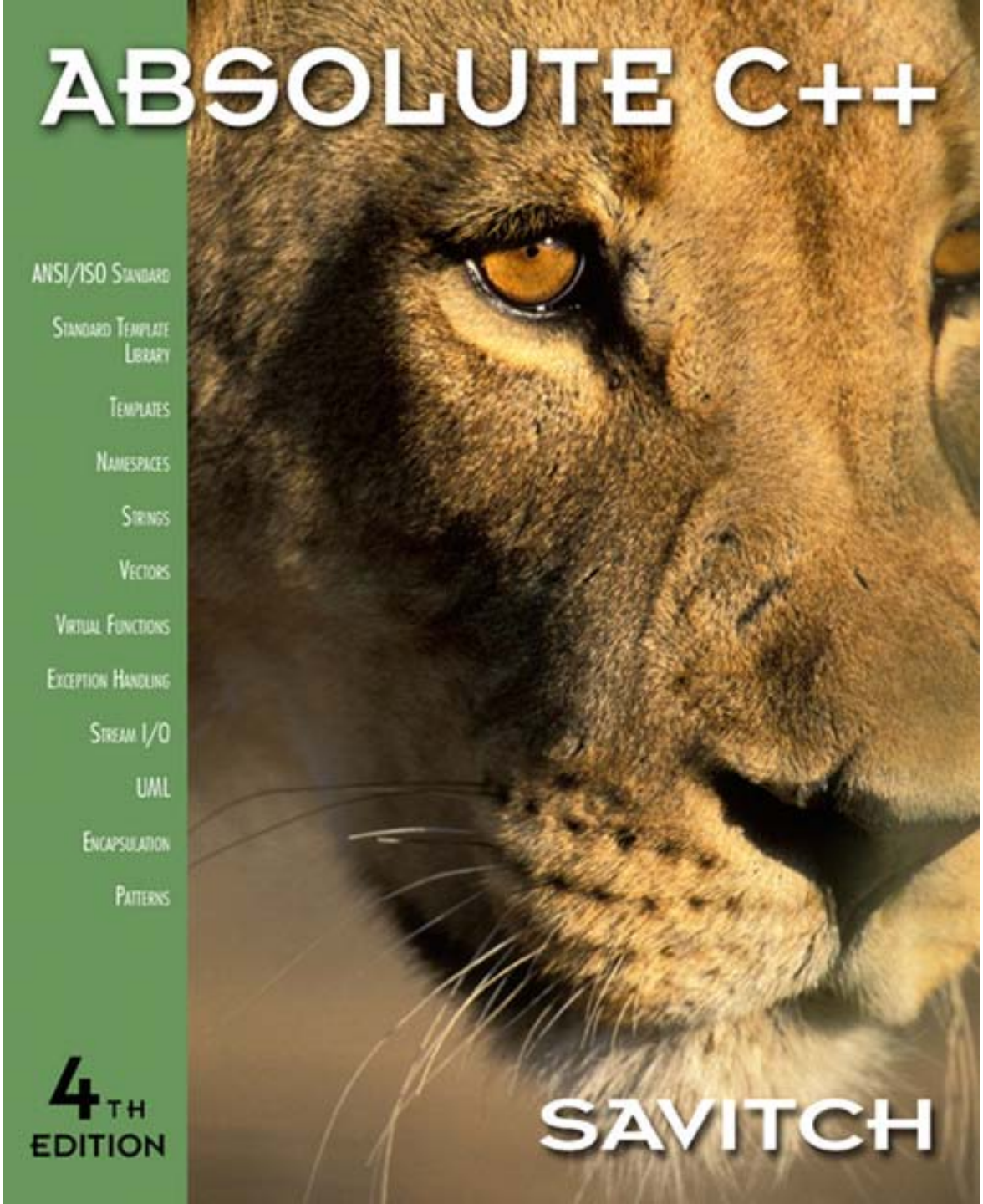
Bài 10: Con trỏ và Mảng động

Giảng viên: Hoàng Thị Điệp
Khoa Công nghệ Thông tin – ĐH Công Nghệ



Chapter 10

Pointers and Dynamic Arrays

PEARSON Addison Wesley

Mục tiêu bài học

- Con trỏ
  - Biến con trỏ
  - Quản lý bộ nhớ
- Mảng động
  - Tạo và sử dụng
  - Số học con trỏ
- Lớp, con trỏ, mảng động
  - Sử dụng con trỏ this
  - Hàm hủy, hàm kiến tạo sao chép

Giới thiệu con trỏ

- Định nghĩa con trỏ:
  - Địa chỉ nhớ của một biến
- Nhắc lại: bộ nhớ được chia thành
  - Các vùng nhớ đánh số
  - Địa chỉ được dùng như tên của biến
- Trước bài này ta đã sử dụng con trỏ!
  - Tham số truyền bằng tham chiếu
    - Địa chỉ của đối số thực sự sẽ được truyền vào hàm

Biến con trỏ

- Con trỏ được định kiểu
  - Có thể lưu con trỏ trong biến
  - Không phải biến int, double, ...
    - mà là con trỏ tới int, double, …
- Ví dụ:
  double *p;
  - Khai báo p là biến kiểu “con trỏ tới double”
  - Nó có thể lưu giá trị con trỏ tới biến double
    - Không lưu được con trỏ tới các kiểu khác!

Khai báo biến con trỏ

- Khai báo biến con trỏ như những kiểu có sẵn
  - Thêm "*" trước tên biến
  - Tạo ra "con trỏ" tới kiểu đó
- "*" phải nằm trước mỗi biến
- int *p1, *p2, v1, v2;
  - p1, p2 lưu con trỏ tới biến int
  - v1, v2 là biến int thông thường

Địa chỉ và giá trị số

- Con trỏ là một địa chỉ
- Địa chỉ là một số nguyên
- Con trỏ không phải là một số nguyên!
- C++ bắt buộc sử dụng con trỏ như địa chỉ
  - Không thể dùng nó như giá trị số
  - Mặc dù nó thực chất là một giá trị số

Trỏ

- Về mặt thuật ngữ
  - Ta tập trung vào tới việc trỏ chứ không phải bản thân địa chỉ
  - Biến con trỏ trỏ tới biến thường
  - Bỏ qua bàn luận về địa chỉ
- Khiến việc trực quan hóa rõ ràng hơn
  - “Thấy" tham chiếu tới vùng nhớ
    - Mũi tên

Trỏ …

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - Chỉ định con trỏ p1 trỏ tới biến int v1
- Toán tử &
  - Xác định địa chỉ của biến
- Cách đọc:
  - "p1 bằng địa chỉ của v1"
  - Hoặc "p1 trỏ tới v1"

Trỏ …

- Ví dụ:
  ```
  int *p1, *p2, v1, v2;
  p1 = &v1;
  ```
- Có 2 cách để làm việc với v1:
  - Dùng chính biến v1:
    ```
    cout << v1;
    ```
  - Thông qua con trỏ p1:
    ```
    cout << *p1;
    ```
- Toán tử dereference *
  - Dịch Anh-Việt: giải tham chiếu / tham chiếu ngược / khử tham chiếu
  - Sẽ giải tham chiếu cho biến con trỏ
  - Nghĩa là “Lấy dữ liệu mà p1 trỏ tới”

Ví dụ: Trỏ

- Xét đoạn mã:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

- Kết quả:

```
42
42
```

- *p1 và v1 là một biến

Toán tử &

- Là toán tử lấy địa chỉ
- Cũng dùng để chỉ định tham số truyền bằng tham chiếu
  - Không phải là trùng hợp ngẫu nhiên!
  - Nhắc lại: truyền tham chiếu thực chất truyền "địa chỉ của" đối số thực sự vào hàm
- 2 cách dùng toán tử này có liên hệ mật thiết

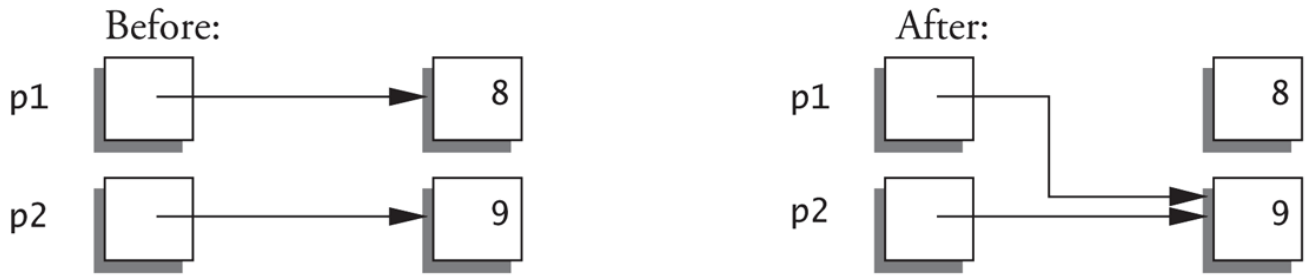
Gán con trỏ

- Có thể gán biến con trỏ:
  int *p1, *p2;
  p2 = p1;
  - Gán một con trỏ cho con trỏ khác
  - “Chỉ định p2 trỏ tới nơi mà p1 đang trỏ tới"
- Dễ bị lẫn với:
  *p2 = *p1;
  - Gán “giá trị trỏ bởi p1” cho “giá trị trỏ bởi p2”

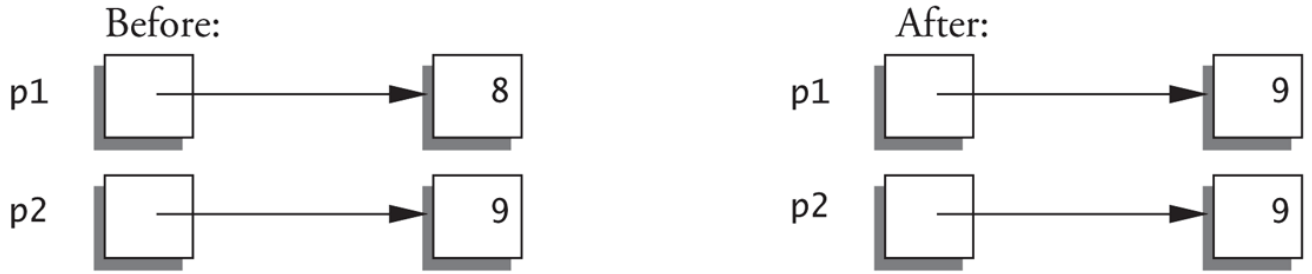
Minh họa phép gán con trỏ:
Display 10.1 Dùng phép gán trên các biến con trỏ

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```



Toán tử new

- Vì con trỏ có thể tham chiếu tới biến…
  - Không thực sự cần phải có định danh chuẩn cho biến đó
- Có thể cấp phát động cho biến
  - Toán tử new tạo ra biến
    - Không có định danh cho nó
    - Chỉ có một con trỏ
- p1 = new int;
  - Tạo ra một biến "không tên" và gán p1 trỏ tới nó
  - Có thể làm việc với biến thông qua *p1
    - Dùng như biến thường

Ví dụ thao thác cơ bản trên con trỏ: **Display 10.2** Thao tác cơ bản trên con trỏ (1/2)

Display 10.2 Basic Pointer Manipulations

```
1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;


5   int main( )
6   {
7       int *p1, *p2;


8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;


13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
```

Ví dụ thao thác cơ bản trên con trỏ:
Display 10.2 Thao tác cơ bản trên con trỏ (2/2)

```
16        p1 = new int;
17        *p1 = 88;
18        cout << "*p1 == " << *p1 << endl;
19        cout << "*p2 == " << *p2 << endl;

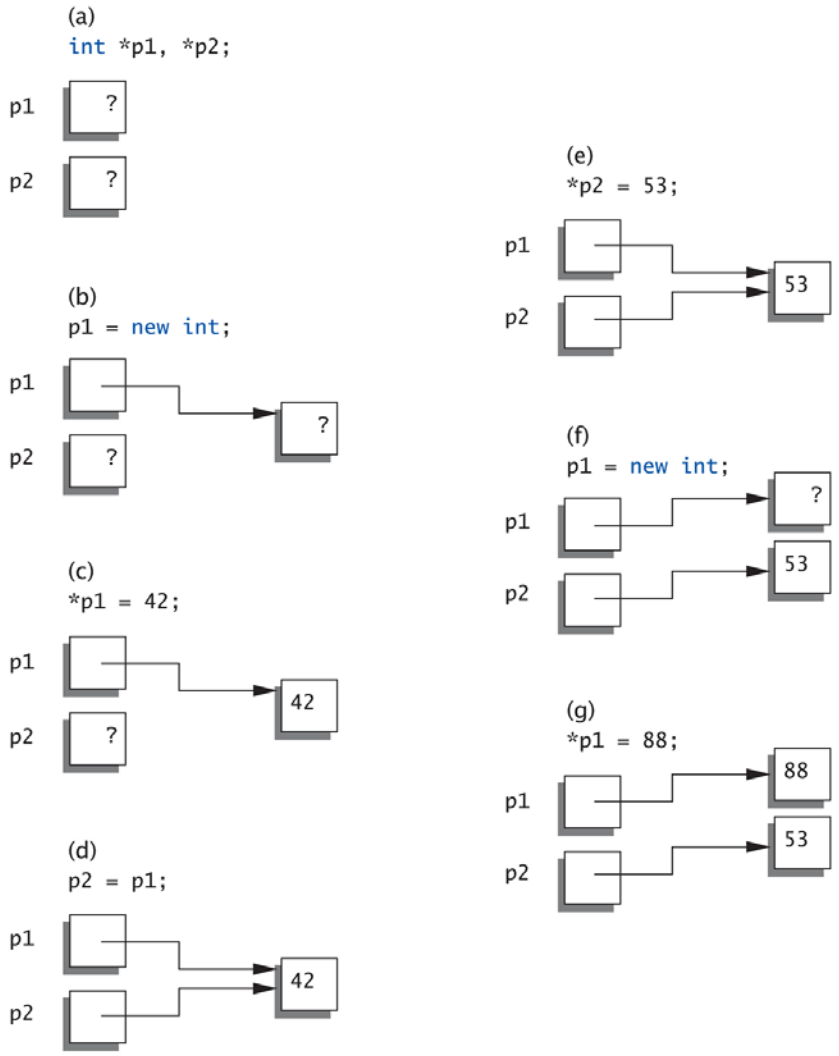

20        cout << "Hope you got the point of this example!\n";
21        return 0;
22    }
```

SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Hình minh họa thao tác cơ bản trên con trỏ: **Display 10.3** Giải thích Display 10.2

Display 10.3 Explanation of Display 10.2



Bàn thêm về toán tử new

- Tạo ra một biến cấp phát động mới
- Trả về con trỏ tới biến mới này
- Nếu kiểu của nó định nghĩa bởi lớp:
  - Hàm kiến tạo sẽ được gọi
  - Có thể gọi hàm kiến tạo khác khi có đối số khởi tạo:
    ```
    MyClass *mcPtr;
    mcPtr = new MyClass(32.0, 17);
    ```
- Vẫn có thể khởi tạo kiểu cơ bản:
  ```
  int *n;
  n = new int(17);          //Khởi tạo *n bằng 17
  ```

Con trỏ và hàm

- Con trỏ là kiểu dữ liệu hoàn chỉnh
  - Có thể dùng nó như các kiểu khác
- Nó có thể là tham số của hàm
- Có thể là kiểu trả về của hàm
- Ví dụ:
  int* findOtherPointer(int* p);
  - Hàm này khai báo:
    - Có tham số kiểu con trỏ trỏ tới int
    - Trả về biến con trỏ trỏ tới int

Quản lý bộ nhớ

- Heap
  - Còn được gọi là "freestore"
  - Dành riêng cho các biến cấp phát động
  - Tất cả các biến cấp phát động mới đều được đặt trong freestore
    - Nếu nhiều quá → chiếm toàn bộ vùng nhớ freestore
- Nếu freestore đã “đầy”, các phép toán new sau đó sẽ thất bại.

Kiểm tra new thành công

- Trình biên dịch cũ:
  - Kiểm tra xem lời gọi new có trả về null hay không*:*

```
int *p;
p = new int;
if (p == NULL)
{
    cout << "Error: Khong du bo nho.\n";
    exit(1);
}
```

  - Với đoạn mã trên, nếu new thành công, chương trình sẽ tiếp tục

new thành công trong trình biên dịch mới

- Trình biên dịch mới hơn:
  - Nếu phép toán new thất bại:
    - Chương trình sẽ tự động kết thúc
    - Sinh thông báo lỗi
- Bổ sung kiểm tra NULL vẫn là kĩ năng thực hành tốt

Kích thước của freestore

- Thay đổi tùy cài đặt
- Thường là lớn
  - Hầu hết các chương trình không dùng hết vùng nhớ này
- Quản lý bộ nhớ
  - Vẫn là kĩ năng thực hành tốt
  - Nguyên lý bắt buộc của kĩ nghệ phần mềm
  - Bộ nhớ luôn hữu hạn
    - Dù có nhiều tới đâu

Toán tử delete

- Giải phóng bộ nhớ động
  - Khi không còn cần nữa
  - Trả lại vùng nhớ cho freestore
  - Ví dụ:

```
int *p;
p = new int(5);
… // Xử lý gì đó …
delete p;
```

  - Giải phóng bộ nhớ động trỏ bởi con trỏ p
    - Thực chất là hủy vùng nhớ

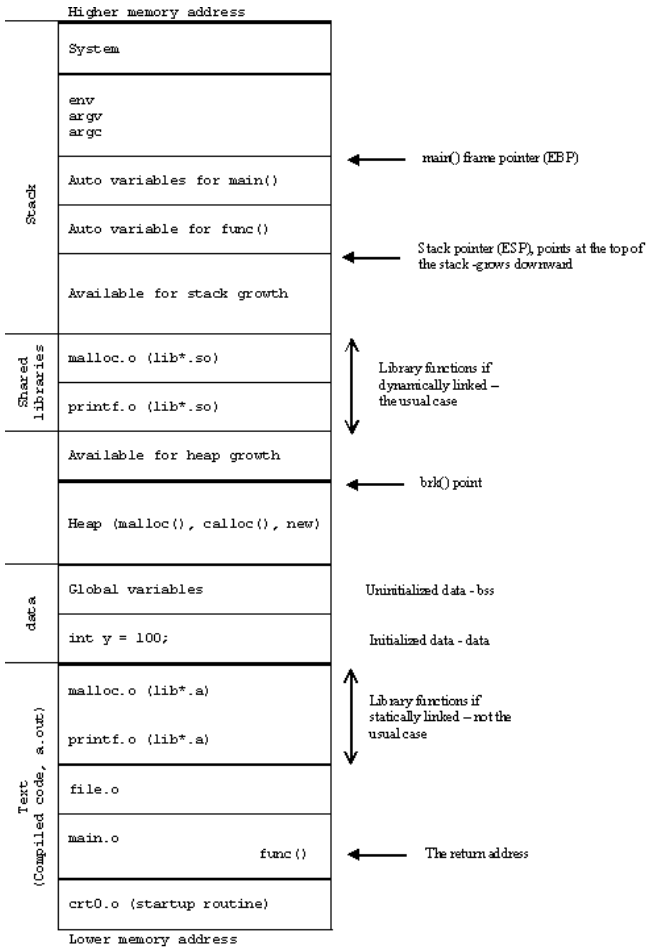
Con trỏ lạc

- delete p;
  - Hủy vùng nhớ động
  - Nhưng p vẫn trỏ tới đó!
    - Gọi là “con trỏ lạc”
  - Nếu sau đó giải tham chiếu trên p (*p)
    - Kết quả không lường trước được!
    - Thường là nguy hiểm!
- Hãy tránh con trỏ lạc
  - Gán con trỏ bằng NULL sau khi delete:
    delete p;
    p = NULL;

Biến cấp phát động và biến tự động

- Biến cấp phát động
  - Sinh ra bởi toán tử new
  - Sinh ra và hủy đi khi chương trình đang chạy
- Biến cục bộ
  - Khai báo bên trong định nghĩa hàm
  - Không động
    - Sinh ra khi hàm được gọi
    - Hủy đi khi hàm kết thúc
  - Thường gọi là biến tự động
    - Các thuộc tính được trình biên dịch quản lý tự động

Biến C++ và Quản lý bộ nhớ



Biến C++ và Quản lý bộ nhớ

Định nghĩa kiểu dữ liệu con trỏ

- Có thể đặt tên cho kiểu dữ liệu con trỏ
- Để có thể khai báo biến con trỏ như các biến khác
  - Loại bỏ * trong khai báo con trỏ
- typedef int* IntPtr;
  - Định nghĩa một tên khác cho kiểu dữ liệu con trỏ
  - Các khai báo sau:
    IntPtr p;
    int *p;
    - là tương đương

Lỗi thường gặp:
Tham số con trỏ truyền giá trị

- Hoạt động tinh vi và phiền hà
  - Nếu hàm biến đổi tham số con trỏ thì chỉ bản sao của nó bị biến đổi
- Ví dụ minh họa …

Ví dụ tham số con trỏ truyền giá trị:
Display 10.4 Tham số con trỏ truyền giá trị (1/2)

Display 10.4 A Call-by-Value Pointer Parameter

```
1   //Program to demonstrate the way call-by-value parameters
2   //behave with pointer arguments.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;

7   typedef int* IntPointer;

8   void sneaky(IntPointer temp);

9   int main( )
10  {
11      IntPointer p;

12      p = new int;
13      *p = 77;
14      cout << "Before call to function *p == "
15           << *p << endl;
```

Ví dụ tham số con trỏ truyền giá trị: **Display 10.4** Tham số con trỏ truyền giá trị (2/2)

```
16        sneaky(p);

17        cout << "After call to function *p == "
18             << *p << endl;

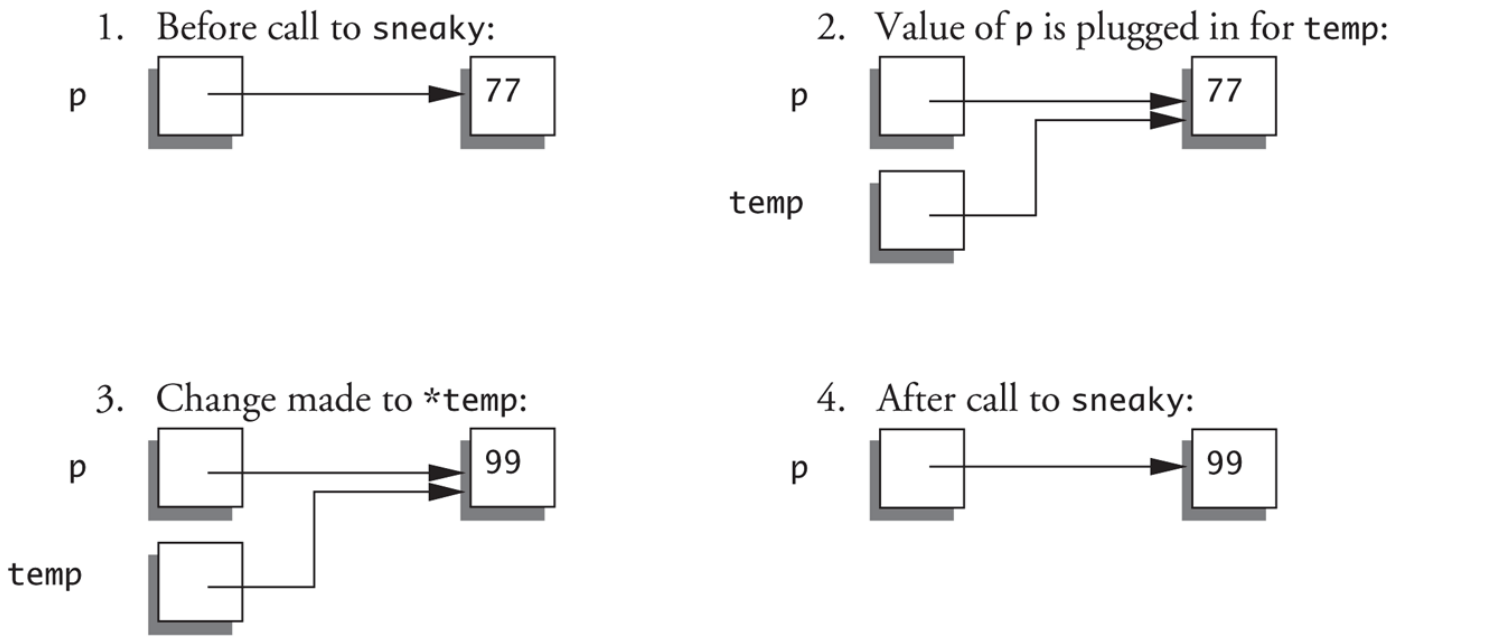
19        return 0;
20   }
21   void sneaky(IntPointer temp)
22   {
23        *temp = 99;
24        cout << "Inside function call *temp == "
25             << *temp << endl;
26   }
```

SAMPLE DIALOGUE

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

Hình minh họa tham số con trỏ truyền giá trị: **Display 10.5** Lời gọi tới sneaky(p);

Display 10.5 The Function Call `sneaky(p);`



Mảng động

- Biến mảng
  - Thực ra là biến con trỏ
- Mảng chuẩn
  - Kích thước cố định
- Mảng động
  - Kích thước không xác định ở thời điểm lập trình
  - Mà xác định khi chạy chương trình

Biến mảng

- Nhắc lại: mảng lưu trong các ô nhớ liên tiếp
  - Biến mảng tham chiếu tới phần tử đầu tiên
  - Suy ra biến mảng là một kiểu biến con trỏ!
- Ví dụ:

```
int a[10];
int * p;
```

  - a và p đều là biến con trỏ!

Biến mảng → Con trỏ

- Ví dụ trước:
  ```
  int a[10];
  typedef int* IntPtr;
  IntPtr p;
  ```
- a và p là các biến con trỏ
  - Có thể thực hiện gán:
    p = a; // Hợp lệ
    - p bây giờ sẽ trỏ tới nơi a trỏ
      - Tức là tới phần tử đầu tiên của mảng a
  - a = p; // Không hợp lệ
    - Con trỏ mảng là con trỏ hằng!

Biến mảng → Con trỏ

- Biến mảng
  int a[10];
- Không chỉ là một biến con trỏ
  - Nó có kiểu "const int *"
  - Mảng đã được cấp phát trong bộ nhớ
  - Biến a phải luôn trỏ tới đó!
    - Không thay đổi được
- Ngược với con trỏ thường
  - Có thể và thường biến đổi

Mảng động

- Hạn chế của mảng chuẩn
  - Phải chỉ định kích thước
  - Thực tế ta có thể không biết kích thước trước khi chạy chương trình!
- Phải "ước lượng" kích thước tối đa có thể cần đến
  - Đôi khi ổn, đôi khi không
  - Lãng phí bộ nhớ
- Mảng động
  - Có thể dãn hay co khi cần

Tạo mảng động

- Rất đơn giản!
- Dùng toán tử new
  - Cấp phát động cho biến con trỏ
  - Sau đó dùng nó như mảng chuẩn
- Ví dụ:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    // kích thước trong cặp ngoặc vuông
```

  - Tạo biến mảng cấp phát động d có 10 phần tử, kiểu cơ sở là double

Xóa mảng động

- Cấp phát động khi chạy chương trình
  - thì nên được hủy khi chạy chương trình
- Thao tác xóa rất đơn giản. Ví dụ:

```
d = new double[10];
… //Processing
delete [] d;
```

  - Giải phóng tất cả vùng nhớ của mảng động này
  - Cặp ngoặc vuông báo hiệu có mảng
  - Nhắc lại: d vẫn trỏ tới vùng nhớ đó!
    - Sau khi delete, cần gán d = NULL;

Hàm trả về một mảng

- Ta không được phép trả về kiểu mảng trong hàm
- Ví dụ:
  int [] someFunction(); // Không hợp lệ!
- Có thể thay bằng trả về con trỏ tới mảng có cùng kiểu cơ sở:
  int* someFunction(); // Hợp lệ!

Số học con trỏ

- Có thể thực hiện các phép toán số học trên con trỏ
  - Số học "địa chỉ"
- Ví dụ:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

  - d chứa địa chỉ của d[0]
  - d + 1 bằng địa chỉ của d[1]
  - d + 2 bằng địa chỉ của d[2]
    - Tương đương với địa chỉ của các phần tử

Cách khác để thao tác mảng

- Dùng số học con trỏ
- Duyệt mảng mà không dùng toán tử [] truy cập chỉ số:

```
for (int i = 0; i < arraySize; i++)
        cout << *(d + i) << " " ;
```

- Tương đương:

```
for (int i = 0; i < arraySize; i++)
        cout << d[i] << " " ;
```

- Chỉ có phép cộng/trừ trên con trỏ
  - Không có nhân, chia
- Có thể tự tăng ++ và tự giảm -- con trỏ

Mảng động nhiều chiều

- Là mảng của mảng
- Sử dụng định nghĩa kiểu con trỏ giúp hiểu rõ hơn:
  ```
  typedef int* IntArrayPtr;
  IntArrayPtr *m = new IntArrayPtr[3];
  ```
  - Tạo ra mảng 3 con trỏ
  - Sau đó biến mỗi con trỏ này thành mảng 4 biến int
- ```
  for (int i = 0; i < 3; i++)
      m[i] = new int[4];
  ```
  - Kết quả là mảng động 3 x 4
  ```

Lớp

- Toán tử ->
  - Kí hiệu viết tắt
- Kết hợp toán tử giải tham chiếu * và toán tử chấm
- Chỉ định thành viên của lớp được trỏ bởi con trỏ cho trước
- Ví dụ:

```
MyClass *p;
p = new MyClass;
p->grade = "A";  // Tương đương với:
(*p).grade = "A";
```

Con trỏ this

- Trong định nghĩa hàm thành viên đôi khi ta cần tham chiếu đến chính đối tượng đang gọi tới nó
- Dùng con trỏ this có sẵn
  - Tự động trỏ tới đối tượng đang gọi:

```
class Simple
{
public:
   void showStuff() const;
private:
   int stuff;
};
```

- Có 2 cách truy cập cho hàm thành viên:

```
cout << stuff;
cout << this->stuff;
```

Nạp chồng toán tử gán

- Toán tử gán trả về tham chiếu
  - Để có thể có nhiều phép gán nối nhau
  - Ví dụ: a = b = c;
    - Gán a và b bằng c
- Toán tử phải trả về cùng kiểu vế trái nó
  - Để có thể gán nối nhau
  - Con trỏ this sẽ hữu ích trong trường hợp này

Nạp chồng toán tử gán

- Nhắc lại: Toán tử gán phải là thành viên của lớp
  - Nó có 1 tham số
  - Vế trái là đối tượng gọi tới toán tử
    s1 = s2;
    - Có thể nghĩ nó là: s1.=(s2);
- s1 = s2 = s3;
  - Cần có (s1 = s2) = s3;
  - Do đó (s1 = s2) phải trả về đối tượng có kiểu như s1
    - Rồi truyền "= s3" vào;

StringClass

```
class StringClass
{
public:
    ...
    void someProcessing( );
    ...
    StringClass& operator=(const StringClass& rtSide);
    ...
private:
    char *a;//Dynamic array for characters in the string
    int capacity;//size of dynamic array a
    int length;//Number of characters in a
};
```

Định nghĩa toán tử = nạp chồng

- Ví dụ cho lớp StringClass:

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
        if (this == &rtSide)          // nếu vế phải giống hệt vế trái
                return *this;
        else
        {
                capacity = rtSide.length;
                length = rtSide.length;
                delete [] a;
                a = new char[capacity];
                for (int i = 0; i < length; i++)
                        a[i] = rtSide.a[i];
                return *this;
        }
}
```

Sao chép nông và sâu

- Sao chép nông
  - Phép gán chỉ sao chép nội dung của các biến thành viên
  - Phép gán mặc định và hàm kiến tạo sao chép mặc định
- Sao chép sâu
  - Khi liên quan tới con trỏ và cấp phát động
  - Phải giải tham chiếu biến con trỏ rồi sao chép vùng dữ liệu được con trỏ trỏ tới
  - Hãy tự nạp chồng toán tử gán và hàm kiến tạo sao chép nếu gặp trường hợp này!

Hàm hủy

- Các biến cấp phát động
  - Không biến mất nếu không được delete tường minh
- Nếu con trỏ là dữ liệu thành viên private
  - Chúng cấp phát động dữ liệu thực
    - Trong hàm kiến tạo
  - Phải có cách nào đó để giải phóng vùng nhớ khi đối tượng bị hủy
- Câu trả lời: Viết hàm hủy.

Hàm hủy

- Ngược lại với hàm kiến tạo
  - Được gọi tự động khi đối tượng ra ngoài phạm vi hoạt động
  - Phiên bản mặc định chỉ xóa các biến thường, không xóa các biến động
- Định nghĩa như hàm kiến tạo, thêm dấu ngã ~
  -
```
MyClass::~MyClass()
{
  //Thực hiện công tác dọn dẹp
}
```

Hàm kiến tạo sao chép

- Tự động gọi khi:
  1. Khai báo đối tượng thuộc lớp đồng thời khởi tạo nó bằng đối tượng khác
  2. Khi hàm trả về đối tượng thuộc lớp
  3. Khi đối số có kiểu của lớp được truyền giá trị vào hàm
- Cần bản sao tạm thời của đối tượng
  - Hàm kiến tạo sao chép sinh ra nó
- Hàm kiến tạo sao chép mặc định
  - Giống phép gán mặc định, nó chỉ sao chép trực tiếp các dữ liệu thành viên
- Có dữ liệu con trỏ → hãy tự viết hàm kiến tạo sao chép

Tóm tắt 1

- Con trỏ là địa chỉ nhớ
  - Cho ta cách tham chiếu gián tiếp tới biến
- Biến động
  - Được tạo và hủy khi chạy chương trình
- Freestore
  - Vùng nhớ cho biến động
- Mảng cấp phát động
  - Có kích thước chỉ định khi chạy chương trình

Tóm tắt 2

- Hàm hủy
  - Là hàm thành viên đặc biệt của lớp
  - Tự động hủy đối tượng
- Hàm kiến tạo sao chép
  - Là hàm thành viên một đối số
  - Được gọi tự động khi cần bản sao tạm thời
- Toán tử gán
  - Cần được nạp chồng dưới dạng hàm thành viên
  - Trả về tham chiếu để có thể gọi theo chuỗi

Chuẩn bị bài tới

- Đọc chương 12 giáo trình: Đọc/ghi trên luồng và tệp