

# Con trỏ

Lập trình nâng cao

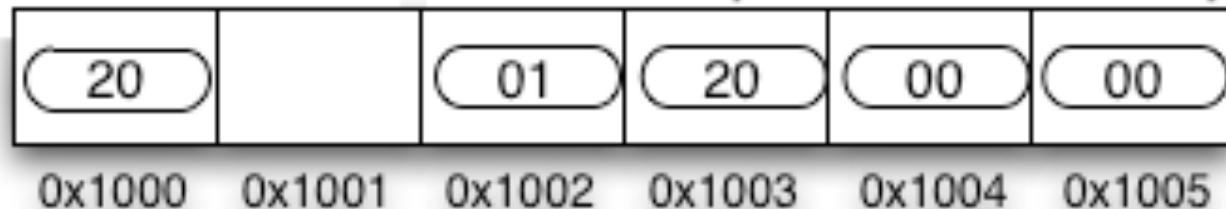
# Outline

- Cơ chế bộ nhớ
- Cách sử dụng
- Cơ chế truyền tham số
  - Truyền bằng con trỏ - Pass-by-pointer
- Lỗi thường gặp
- Các phép toán
  - Đổi kiểu, +, -, ++, --
- Con trỏ và mảng

# Cơ chế bộ nhớ

- Con trỏ là một biến
  - Nó có một địa chỉ và lưu một giá trị
  - Nhưng giá trị của nó được hiểu là địa chỉ bộ nhớ.
- `X x;      // biến kiểu X`
- `X* p;      // biến kiểu con trỏ tới giá trị kiểu X`
- Kích thước của con trỏ không phụ thuộc kiểu dữ liệu nó trỏ tới.

`char a = 0x20      char *pX = 0x2001;`



# Gán giá trị cho con trỏ

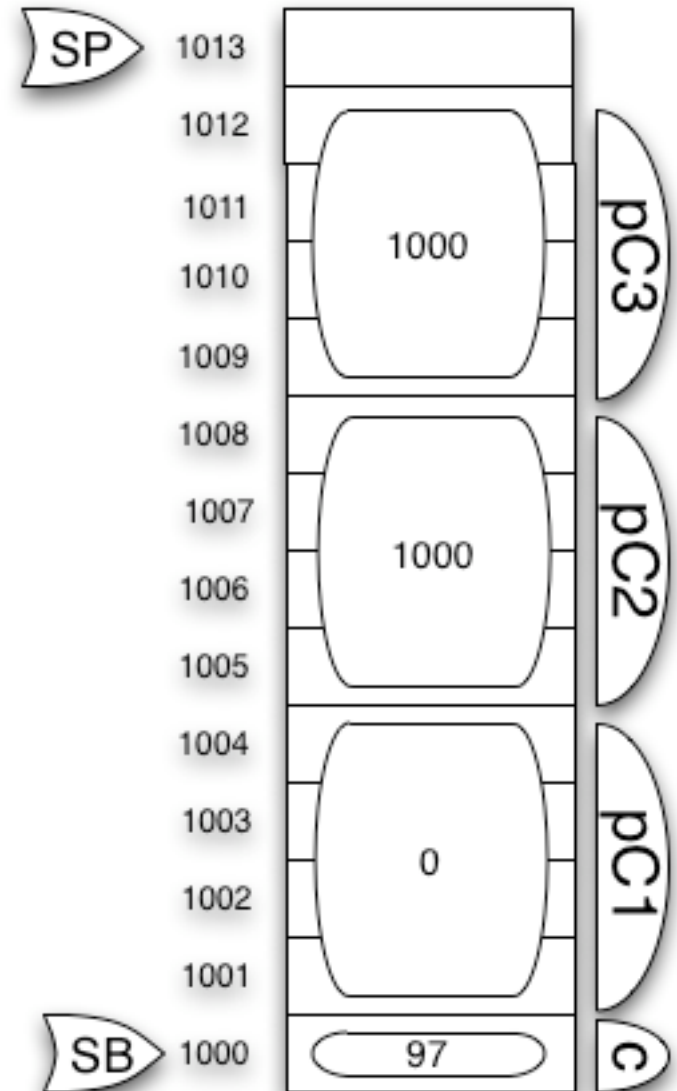
```
void foo()  
{  
    char c, *pC1, *pC2, *pC3;  
    c = 'a';  
    pC1 = NULL;  
    pC2 = &c;  
    pC3 = pC2;  
}
```

Gán giá trị số

Gán địa chỉ của biến

Gán giá trị con trỏ khác

Gán địa chỉ của hàm (ngoài chương trình)



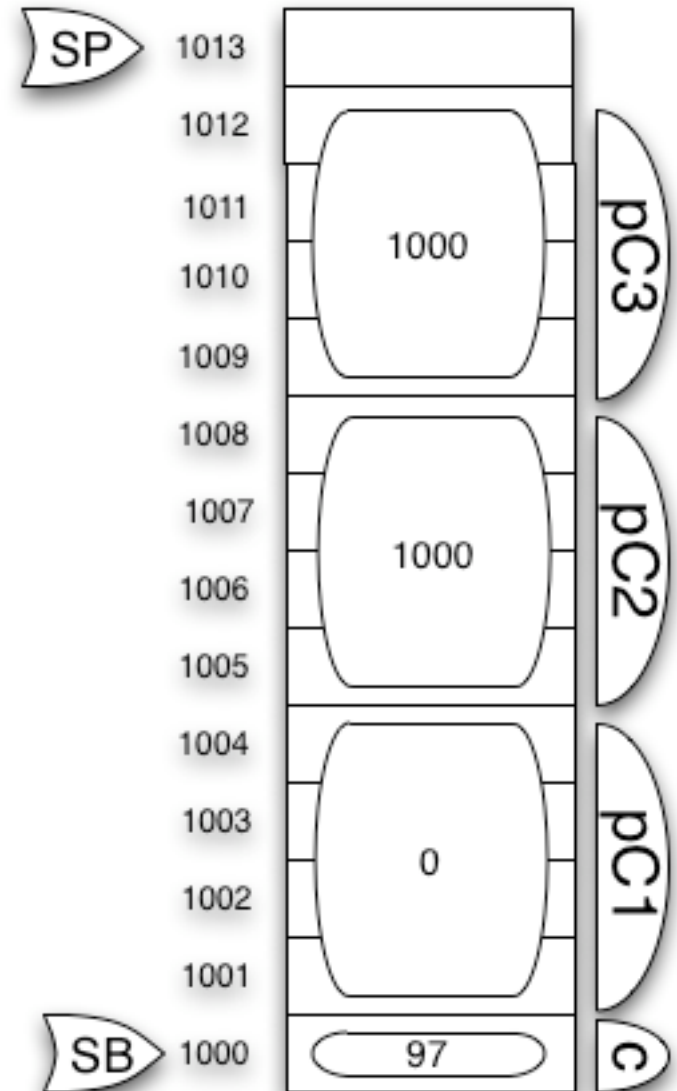
# Dereferencing

## Lấy giá trị biến con trỏ trỏ tới

```
void foo()  
{  
    char c, *pC1, *pC2, *pC3;  
    c = 'a';  
    pC1 = NULL;  
    pC2 = &c;  
    pC3 = pC2;  
}
```

Nếu pX là con trỏ thì (\*pX) truy nhập vùng nhớ pX trỏ tới.

- (\*pC1) tương đương với c
- c tương đương với (\*(&c))



# Dereferencing - Ví dụ

```
int dereferencing_test(int myX)
{
    int* pX = &myX;
    myX += 1;
    myX += *pX;
    (*pX) += 2;
    return *pX;
}
```

Có thể dùng (\*pX) tương tự  
như dùng biến mà pX trỏ tới

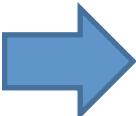
- Đọc giá trị
- Ghi giá trị mới
- Trả về giá trị

```
int main()
{
    printf("test %d is %d", 0, dereferencing_test(0));
}
```

# pass-by-pointer

```
void swap(int* px, int* py) {  
    int c;  
    c = *px;  
    *px = *py;  
    *py = c;  
}
```

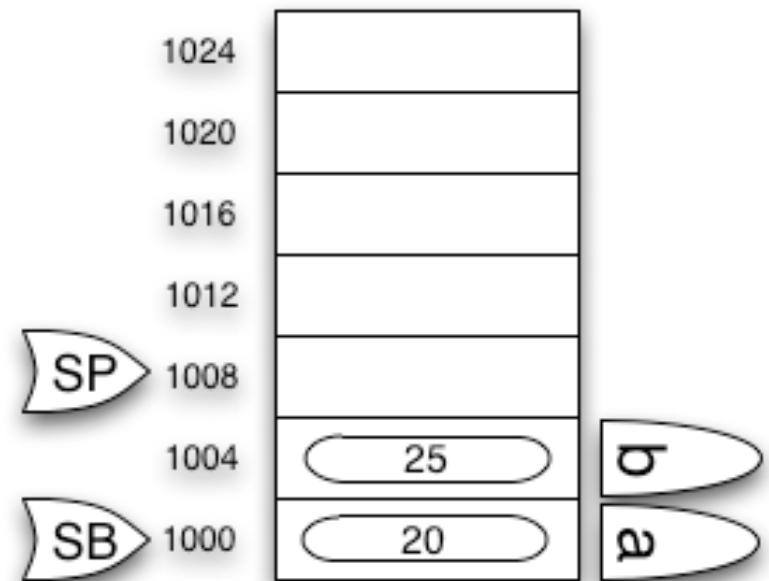
```
int main() {  
    int a = 20;  
    int b = 25;
```



```
    swap(&a, &b);  
    cout << a << ", " << b;
```

```
    return 0
```

```
}
```



# pass-by-pointer

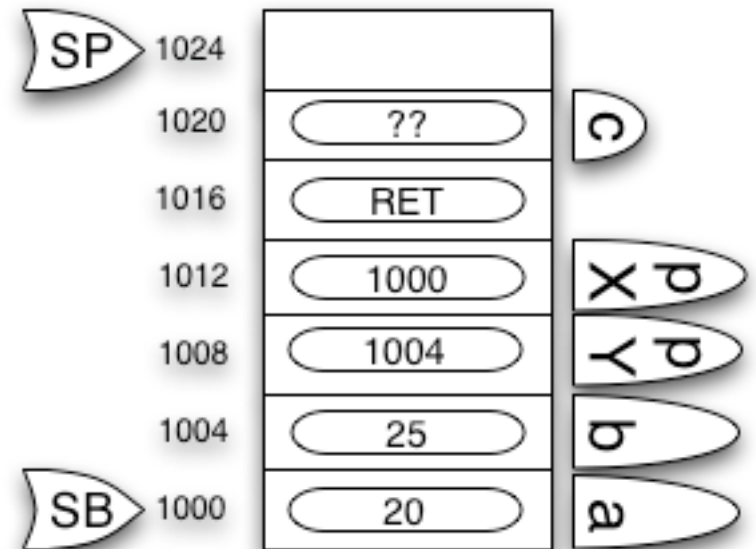
void swap(int\* px, int\* py) {  
 int c;  
 c = \*px;  
 \*px = \*py;  
 \*py = c;  
}

int main() {  
 int a = 20;  
 int b = 25;

swap(&a, &b);  
 cout << a << ", " << b;

return 0

}



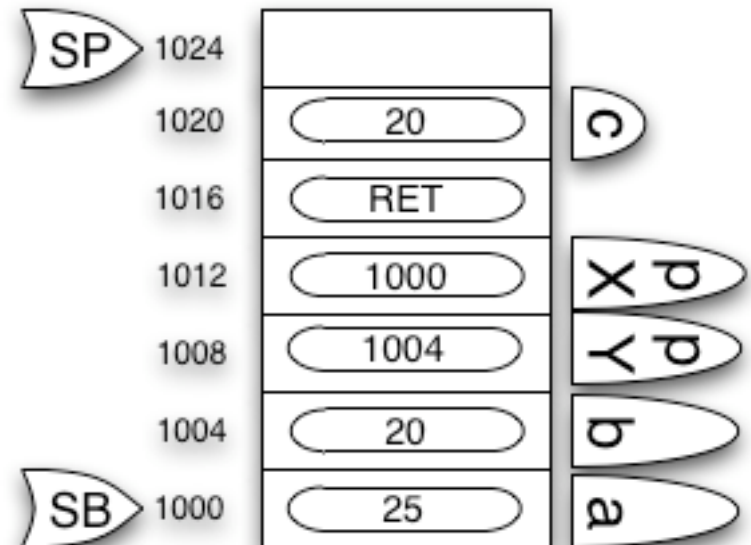


# pass-by-pointer

```
void swap(int* px, int* py) {  
    int c;  
    c = *px;  
    *px = *py;  
    *py = c;  
}
```



```
int main() {  
    int a = 20;  
    int b = 25;  
  
    swap(&a, &b);  
    cout << a << ", " << b;  
  
    return 0;  
}
```



# pass-by-pointer

```
void swap(int* px, int* py) {  
    int c;  
    c = *px;  
    *px = *py;  
    *py = c;  
}
```

```
int main() {  
    int a = 20;  
    int b = 25;
```

➔

```
    swap(&a, &b);  
    cout << a << ", " << b;
```

```
    return 0
```

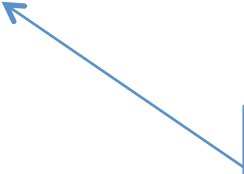
```
}
```



# pass-by-pointer

```
void swap(int* px, int* py) {  
    int c;  
    c = *px;  
    *px = *py;  
    *py = c;  
}
```

Tham số là con trỏ



```
int main() {  
    int a = 20;  
    int b = 25;  
  
    swap(&a, &b);  
    cout << a << ", " << b;  
  
    return 0;  
}
```

Đối số là địa chỉ



# Lỗi thường gặp – con trỏ chưa khởi tạo

- Con trỏ chưa khởi tạo có thể chứa dữ liệu rác – địa chỉ ngẫu nhiên
- Truy nhập chúng dẫn đến các lỗi ghi đè dữ liệu, ghi vào vùng cấm ghi....segmentation faults, v.v..

```
int main (int argc, const char * argv[])  
{  
    int *pX;  
    printf("%d\n", pX);  
    printf("%d\n", *pX);  
    *pX = 0;  
    return 0;  
}
```

# Lỗi thường gặp: truy nhập con trỏ null

- Tương đương truy nhập địa chỉ 0 trong bộ nhớ

```
int main (int argc, const char * argv[])
{
    int *pX = NULL;
    printf("%d\n", pX);
    printf("%d\n", *pX);
    *pX = 0;
    return 0;
}
```

# Lỗi thường gặp: dangling references

- dangling reference: truy nhập tới vùng nhớ không còn hợp lệ
- Ví dụ: trả về con trỏ tới biến địa phương

```
int* weird_sum(int a, int b) {  
    int c;  
    c = a + b;  
    return &c;  
}
```

- Lời khuyên: đừng giữ con trỏ tới biến có phạm vi nhỏ hơn chính biến con trỏ đó.

```
int main (int argc, const char * argv[])
{
    int a = 20, b = 25, *pG;
    {
        int g;
        pG = &g;
        g = gcd(a,b);
    }

    {
        int temp = 100;
        printf("temp is %d\n", temp);
    }

    printf("GCD(%d,%d)=%d\n", a, b, *pG);
    return 0;
}
```

```
int main (int argc, const char * argv[])
{
    {
        int temp1;
        printf("%d\n", &temp1);
    }

    {
        int temp2;
        printf("%d\n" , &temp2);
    }
    return 0;
}
```

```
Coldfront:TestProj udekel$ gcc -O0 main.c
```

```
Coldfront:TestProj udekel$ ./a.out
-1073743780
-1073743784
```

```
Coldfront:TestProj udekel$ gcc -O1 main.c
```

```
Coldfront:TestProj udekel$ ./a.out
-1073743796
-1073743796
```



# Đổi kiểu

```
char a = 'a';  
char* p1 = &a;  
int* p2 = (int*)p1;  
*p2 = 'b';
```

- Rủi ro, không khuyến khích
- Trình biên dịch cảnh báo
- Phải đổi kiểu là dấu hiệu của thiết kế tồi

# void\*

- Kiểu con trỏ trỏ đến loại dữ liệu không xác định kiểu.
- Lập trình viên tự chịu trách nhiệm ép kiểu

# Hằng con trỏ

- Đọc từ phải sang trái

`const int* p1 = &a;`                    `// con trỏ tới hằng int`

`int* const p2 = &b;`                    `// hằng con trỏ`

`const int* const p3 = &c;`    `// hằng con trỏ tới hằng int`

# Hằng con trỏ

```
int a = 20, b = 25, c=30;  
const int* pA = &a;  
int* const pB = &b;  
const int* const pC = &c;
```

```
*pA += 1;
```

✗ error: assignment of read-only location

```
*pB += 1;
```

```
*pC += 1;
```

✗ error: assignment of read-only location

```
pA = NULL;
```

```
pB = NULL;
```

✗ error: assignment of read-only variable 'pB'

```
pC = NULL;
```

✗ error: assignment of read-only variable 'pC'

# Quy tắc lập trình an toàn

- Khóa tất cả những gì có thể khóa
- Gắn const vào tất cả những gì không nên bị sửa giá trị.

# Con trỏ tới con trỏ

```
void pointersToPointers()  
{
```

```
    int a = 10, b = 20, c = 30, sum = 0;  
    int *pA = &a, *pB = &b, *pInt;
```



```
    int **ppInt = &pInt;
```

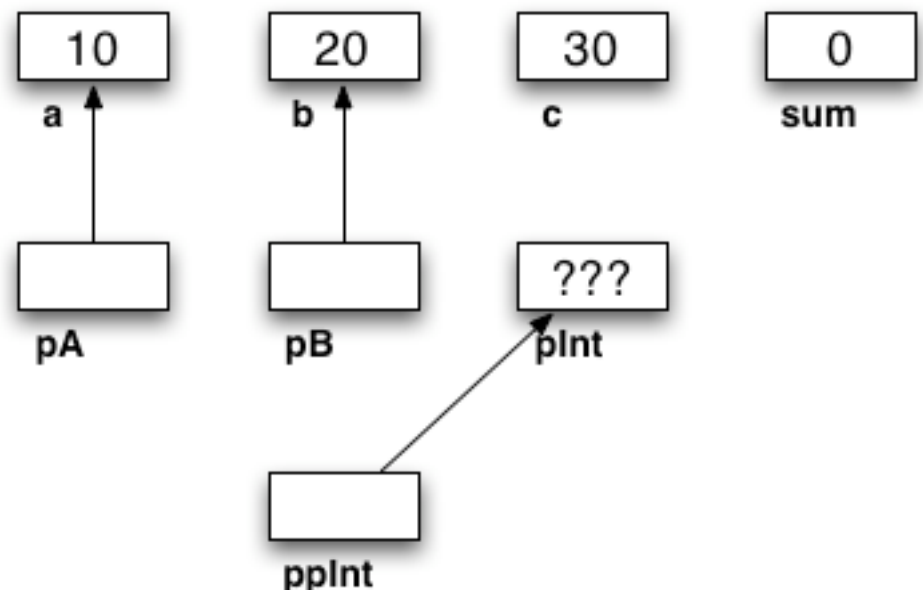
```
    (*ppInt)=pA;  
    sum += (*ppInt);
```

```
    (*ppInt)=pB;  
    sum += (*pInt);
```

```
    *ppInt = &c;  
    sum += (**ppInt);
```

```
    printf("Sum is %d\n", sum);
```

```
}
```



```
void pointersToPointers()
{
    int a = 10, b = 20, c = 30, sum = 0;
    int *pA = &a, *pB = &b, *pInt;

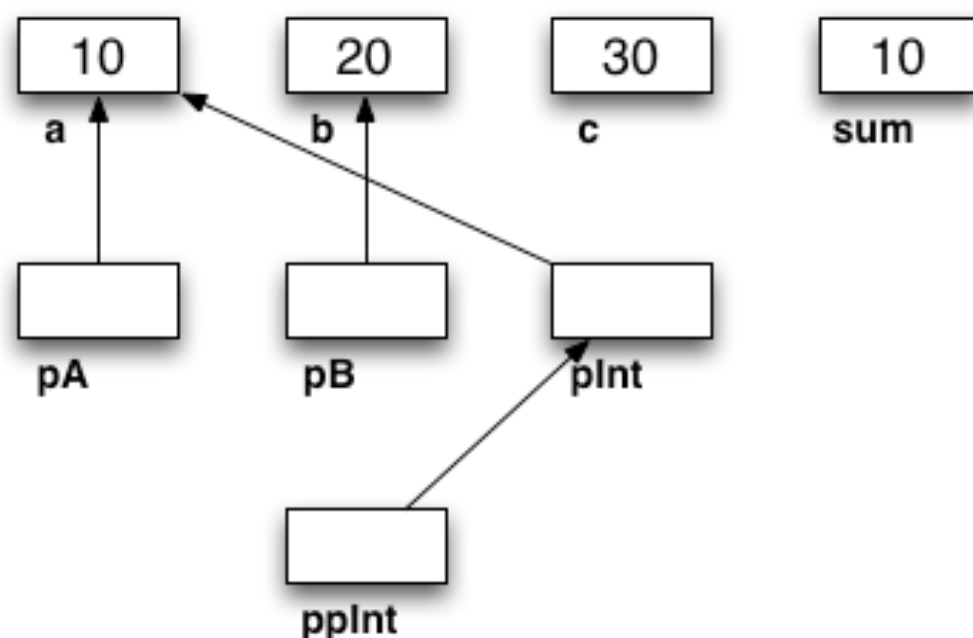
    int **ppInt = &pInt;

    (*ppInt)=pA;
    sum += (**ppInt);

    (*ppInt)=pB;
    sum += (*pInt);

    *ppInt = &c;
    sum += (**ppInt);

    printf("Sum is %d\n", sum);
}
```



```

void pointersToPointers()
{
    int a = 10, b = 20, c = 30, sum = 0;
    int *pA = &a, *pB = &b, *pInt;

    int **ppInt = &pInt;

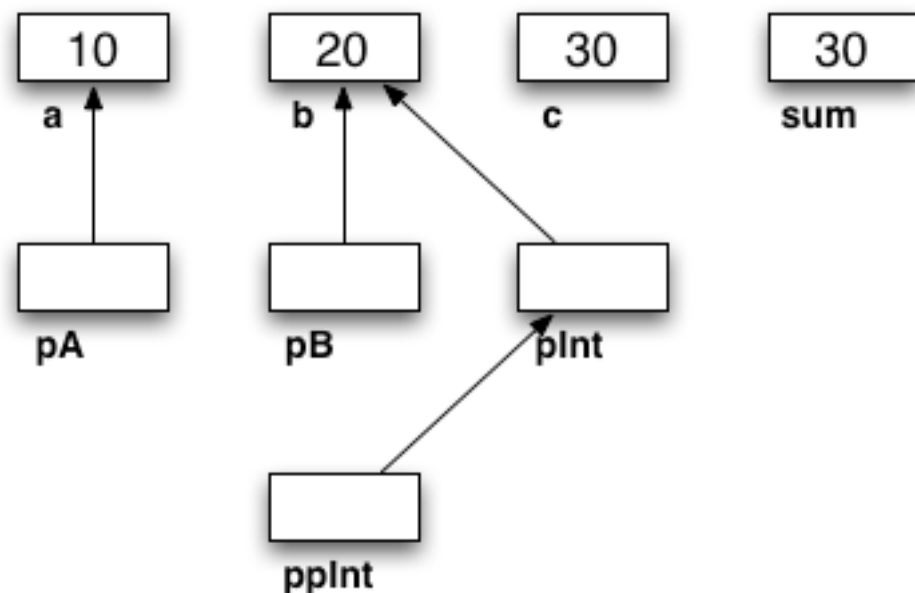
    (*ppInt)=pA;
    sum += (**ppInt);

    (*ppInt)=pB;
    sum += (*pInt);

    *ppInt = &c;
    sum += (**ppInt);

    printf("Sum is %d\n", sum);
}

```





```

void pointersToPointers()
{
    int a = 10, b = 20, c = 30, sum = 0;
    int *pA = &a, *pB = &b, *pInt;

    int **ppInt = &pInt;

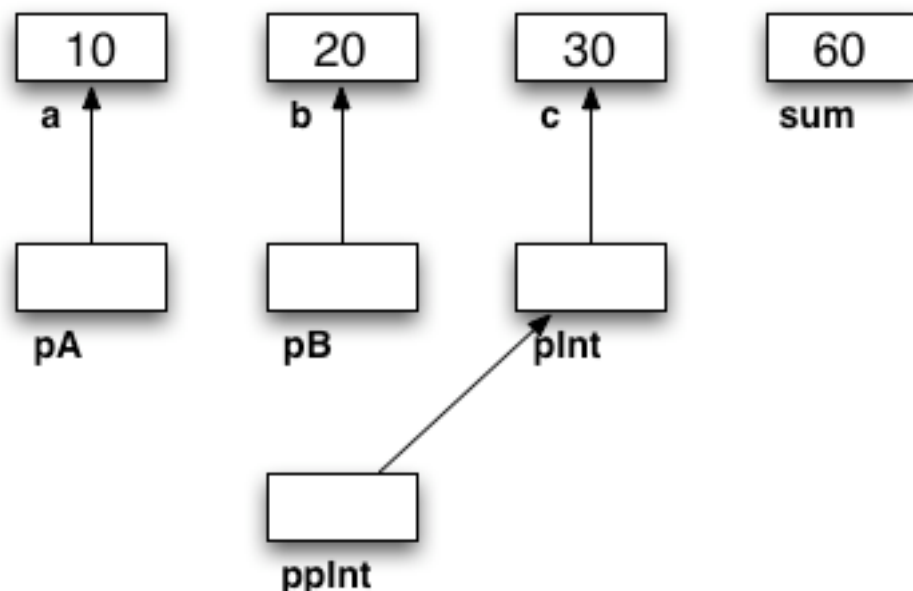
    (*ppInt)=pA;
    sum += (**ppInt);

    (*ppInt)=pB;
    sum += (*pInt);

    *ppInt = &c;
    sum += (**ppInt);

    printf("Sum is %d\n", sum);
}

```



# Luyện tập lần bước trong bộ nhớ

Xếp lần lượt địa chỉ a,b,c,sum, pa, pb...  
theo địa chỉ tăng dần trong stack  
bắt đầu từ 0x1000 (con trỏ 32bit)

Khi con trỏ chạy đến vị trí trong hình  
tính tất cả các biểu thức sau (nếu hợp lệ)

&sum, sum, \*sum, \*\*sum  
&(a+1), a+1, \*(a+1), \*\*(a+1)  
&pa, pa, \*pa, \*\*pa  
&(pa+1), pa+1, \*(pa+1), \*\*(pa+1)  
&plnt, plnt, \*plnt, \*\*plnt  
&(plnt+1), plnt+1, \*(plnt+1),  
\*\*(pplnt+1)

```
void pointersToPointers()
{
    int a = 10, b = 20, c = 30, sum = 0;
    int *pA = &a, *pB = &b, *pInt;

    int **ppInt = &pInt;

    (*ppInt)=pA;
    sum += (**ppInt);

    (*ppInt)=pB;
    sum += (*pInt);

    *ppInt = &c;
    sum += (**ppInt);

    printf("Sum is %d\n", sum);
}
```

# Luyện tập lần bước trong bộ nhớ

a:10	1000
b:20	1004
c:30	1008
sum:60	100c
pa:0x1000	1010
pb:0x1004	1014
pInt:0x1008	1018
ppInt:0x1018	101c

&(pInt+1) → &(0x101c) không hợp lệ

\*\* (ppInt+1) → \*\* (0x101c)  
→ \*(0x1018)  
→ 0x1008

```
void pointersToPointers()
{
    int a = 10, b = 20, c = 30, sum = 0;
    int *pA = &a, *pB = &b, *pInt;

    int **ppInt = &pInt;

    (*ppInt)=pA;
    sum += (**ppInt);

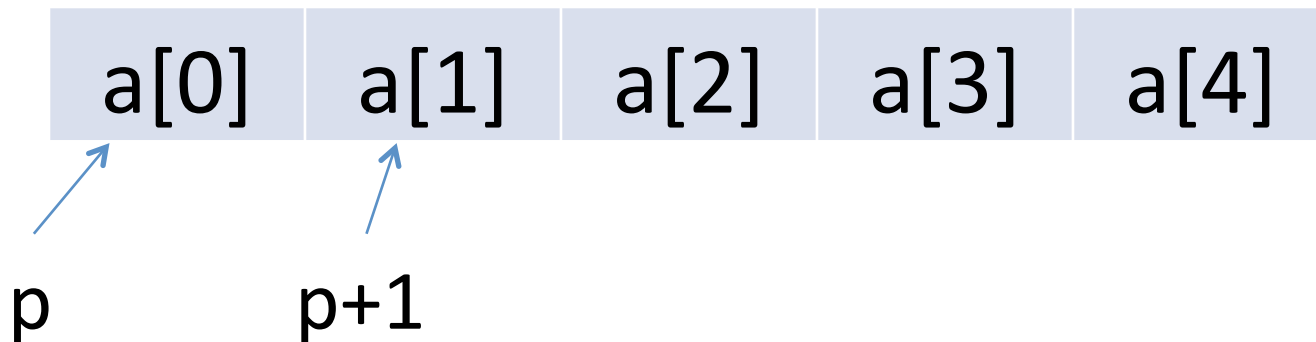
    (*ppInt)=pB;
    sum += (*pInt);

    *ppInt = &c;
    sum += (**ppInt);

    printf("Sum is %d\n", sum);
}
```

# Con trỏ và mảng

- `int a[5];`
- `int* p = a;`



# Con trỏ và mảng

- Các đoạn code tương đương

```
int score[N] = ...
```

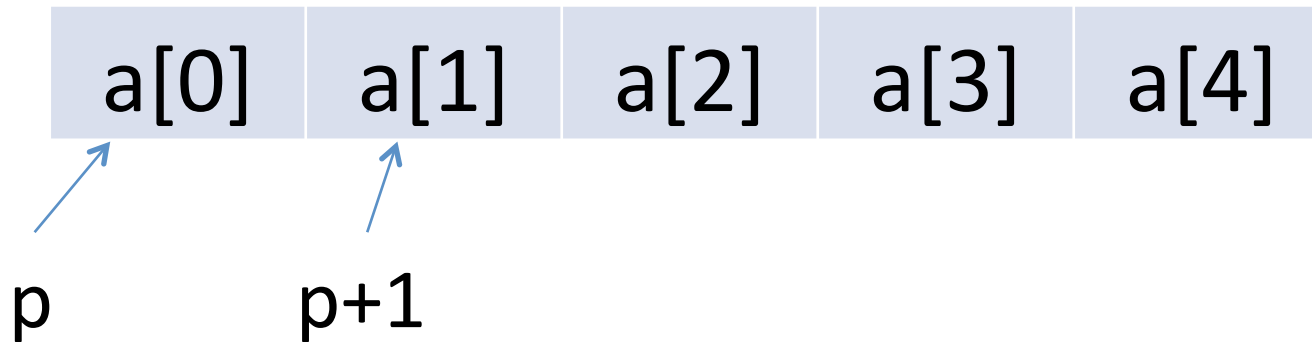
```
for (int i = 0; i < N; i++)  
    cout << score[i] << " ";
```

```
for (int i = 0, int* p = score; i < N; i++)  
    cout << *(p+i) << " ";
```

```
for (int *ptr = &score[0]; ptr <= &score[N-1]; ptr++)  
    cout << *ptr << " ";
```

```
for (int *ptr = score, int* end = &score[N-1]; ptr <= end; ptr++)  
    cout << *ptr << " ";
```

# Các phép toán với con trỏ



- `==`, `!=`, `>`, `<` so sánh địa chỉ lưu bởi hai con trỏ
- `++`, `--`, `+`, `-`, `+=`, `-=` với một số nguyên làm thay đổi giá trị con trỏ một khoảng bằng số nguyên đó nhân với kích thước của kiểu dữ liệu.

```
int main (int argc, const char * argv[])
{
    int i;
    char* p;
    char str[50] = "Hello World!";

    for(i=0; str[i]!='\0'; ++i)
    {
        if(isupper(str[i])) str[i]=tolower(str[i]);
        else if(islower(str[i])) str[i]=toupper(str[i]);
    }
    printf("Reversed string is %s\n",str);

    for(p=&str[0]; (*p)!='\0'; ++p)
    {
        if(isupper(*p)) *p=tolower(*p);
        else if(islower(*p)) *p=toupper(*p);
    }
    printf("Original string is %s\n",str);
    return 0;
}
```

```
git add main.cpp
```

```
main(2, {"add", "main.cpp"})
```

```
C:\>domin.exe 3 4 2
```

```
main(int argc, const char* argv[])
```

```
argc <- 3
```

```
Argv <- {"3", "4", "2"};
```

```
argv[0] là một c-string
```



# “Biến” mảng

- Biến mảng thường được xem như con trỏ tới phần tử đầu tiên
- Không phải con trỏ
- Truyền được vào hàm nhận tham số là con trỏ
- Không sửa giá trị được
- sizeof trả về kích thước mảng

```
char str[50] = "Hello World!";  
  
char *p1 = str;  
  
char *p2 = &str[0];  
  
if(p1==p2) printf("EQUAL!\n");  
  
if(p1[5]==' ') *(p2+5)='-';  
  
p1=NULL;  
p2=NULL;  
str=NULL;
```

✗ error: incompatible types in assignment

# Con trỏ và C-string

- Đọc lại các hàm xử lý xâu trong thư viện `<cstring>`, trong đó chủ yếu dùng tham số dạng con trỏ. Ví dụ:
  - `strcpy(char * desc, char* source)`
  - `strncpy(int length, char * desc, char* source)`

Cho đoạn lệnh sau:

```
int a[3]={2, 3}; int* p=a; int** pp=&p;
```

**Câu 1:** Biết địa chỉ của a, p, pp lần lượt là 0x100, 0x110, 0x114.

Với mỗi biểu thức dưới đây, hỏi:

1. nó có hợp lệ (compiler chấp nhận) hay không, nếu có thì giá trị của nó là gì, nếu không thì giải thích lí do;
2. nó có truy nhập vùng bộ nhớ không hợp lệ hay không?

&a, a, (\*a), (\*\*a), &p, p, (\*p), (\*\*p),

&(a+1), (a+1), \*(a+1), \*\*(a+1), &a[1], a[1], \*(&a[1]),

&(p+2), (p+2), \*(p+2), \*\*(p+2), &p[2], p[2], \*(&p[2]),

\*(&p[0]) + \*(a+1),

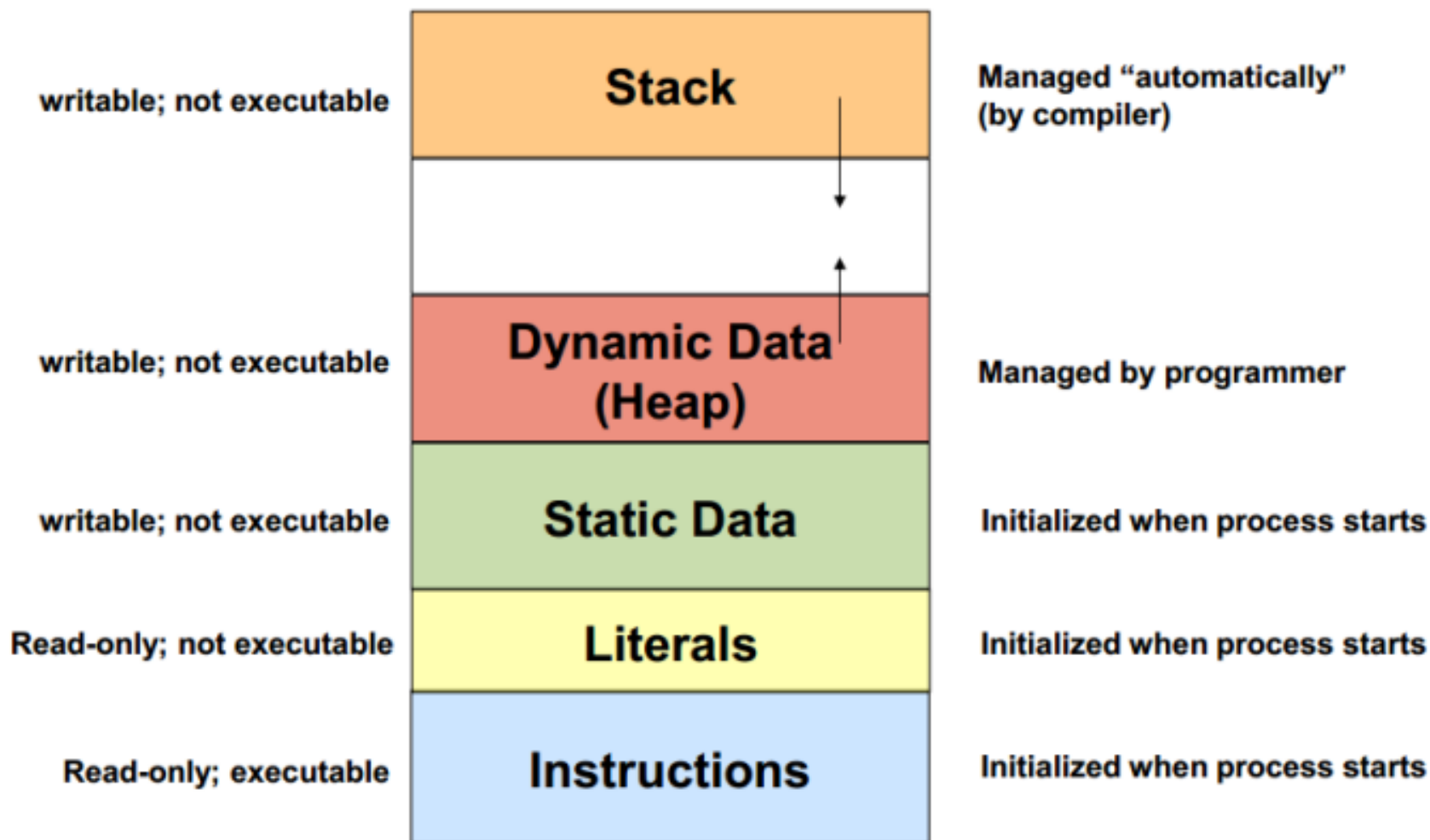
&pp, pp, (\*pp), (\*\*pp), (pp+1), \*pp + 1,

\*(pp+1), \*(\*pp + 1), \*(pp+1) + \*(p+2) , \*pp + \*(p+2)

**Câu 2:** Chỉ dùng biến pp, hãy viết lệnh tương đương a[2] = 10;

Bộ nhớ động

# Heap – nơi đặt dữ liệu động



# cấp phát biến trong bộ nhớ động

- Được cấp phát trong vùng bộ nhớ heap

```
int* p = new int;    // cấp phát một biến int  
int* arr = new int[10]; // cấp phát mảng
```

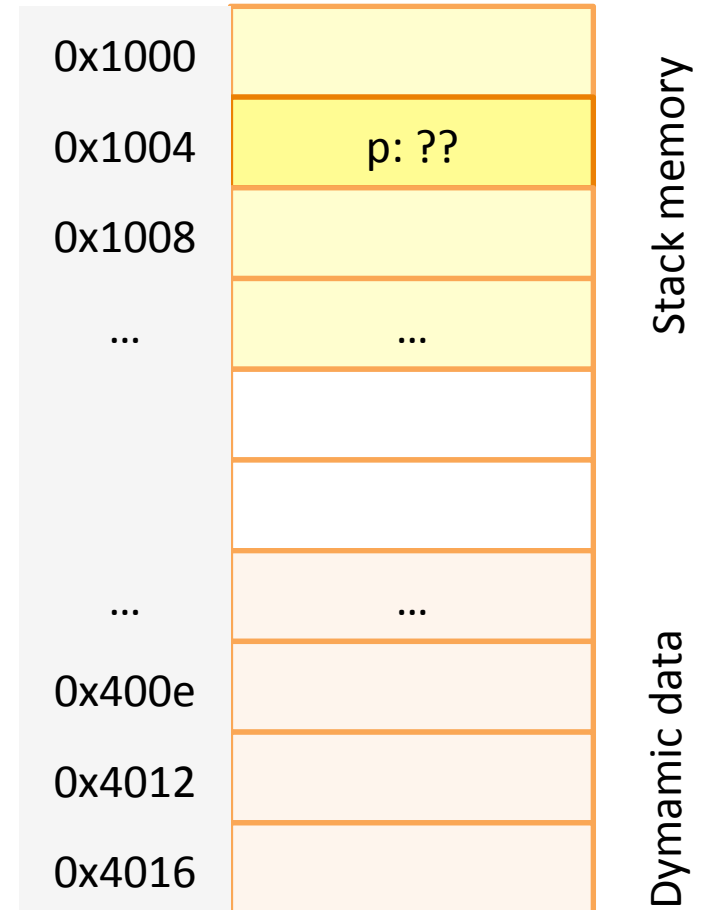
- Toán tử **new**
  - Cấp phát một vùng nhớ kiểu int trong heap và lưu địa chỉ của vùng nhớ đó tại p.
  - p nhận giá trị 0 (NULL) nếu không cấp phát thành công (chẳng hạn vì thiếu bộ nhớ). **Lập trình viên cần kiểm tra.**

# Trình tự cấp phát động

```
int* p = new int;
```



1. Khai báo biến p

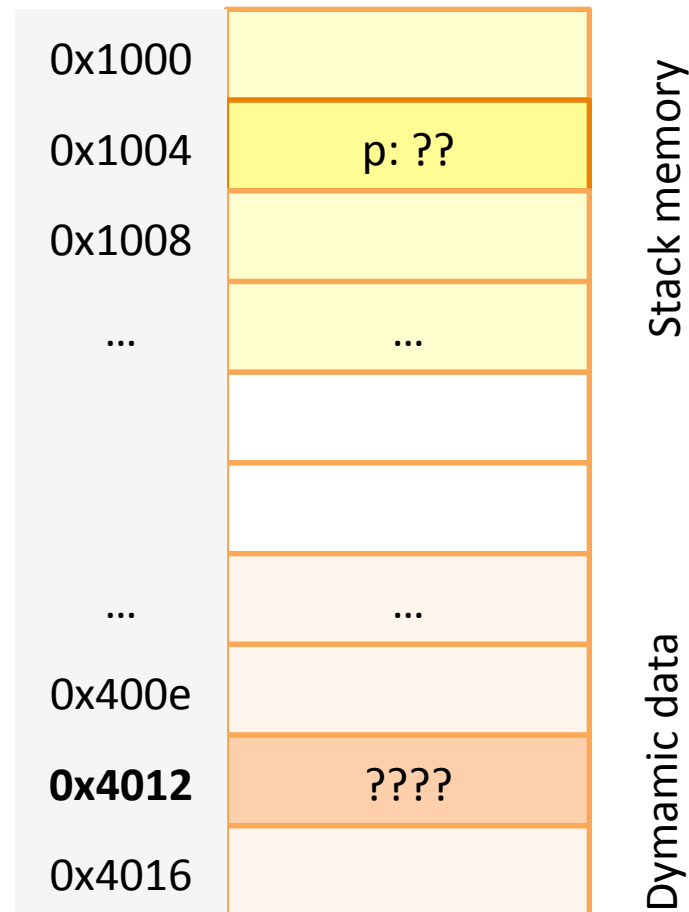


# Trình tự cấp phát động

```
int* p = new int;
```



1. Khai báo biến p
2. Tính biểu thức new int
  - Cấp phát 1 ô nhớ int
  - Lấy địa chỉ ô nhớ đó



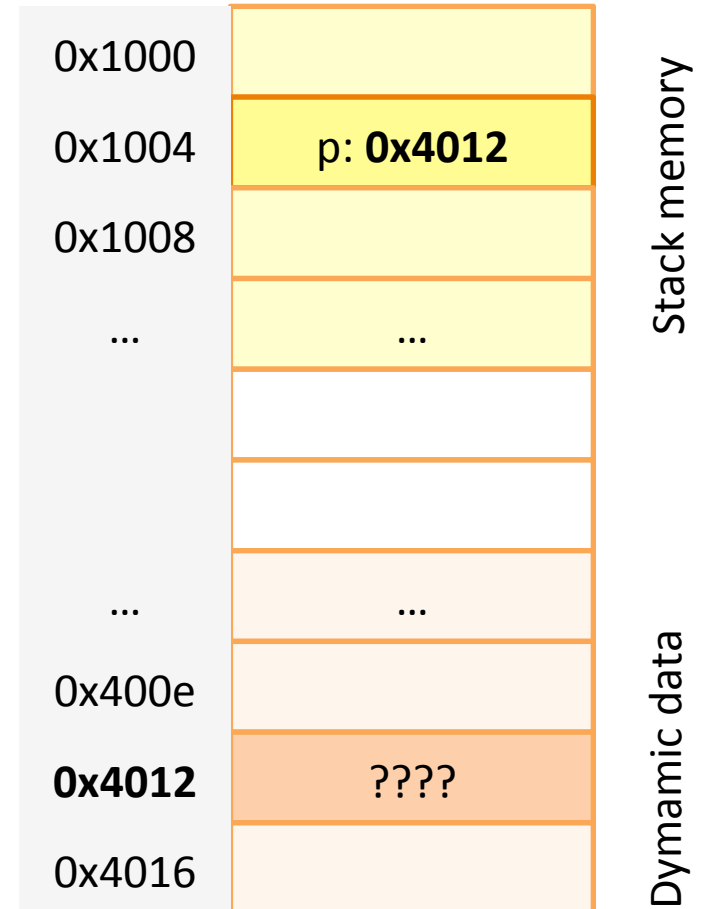


# Trình tự cấp phát động

```
int* p = new int;
```



1. Khai báo biến p
2. Tính biểu thức new int
  - Cấp phát 1 ô nhớ int
  - Lấy địa chỉ ô nhớ đó
3. Thực hiện phép gán

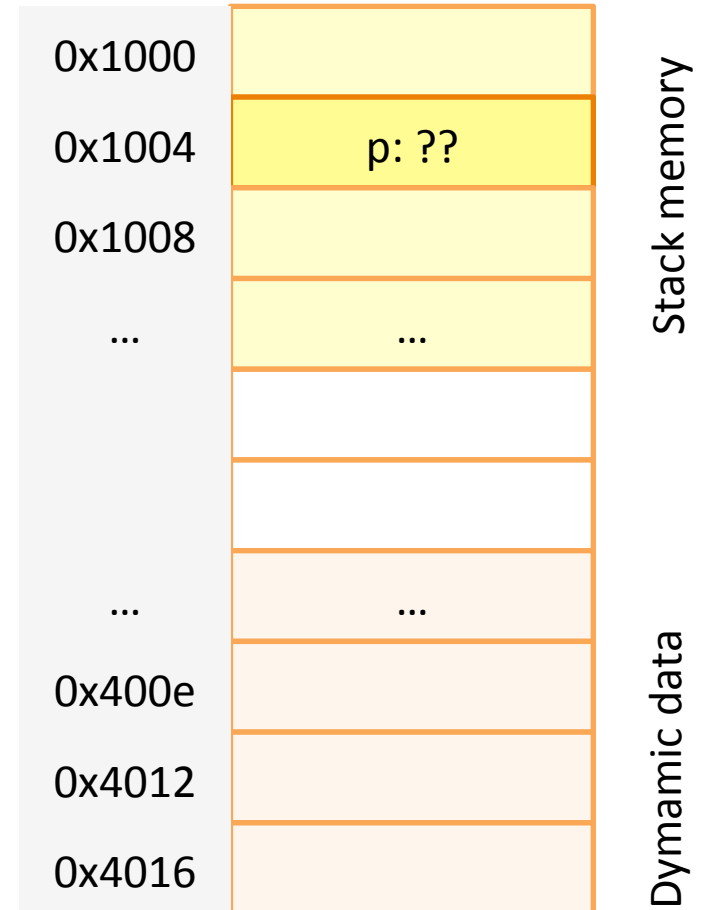


# Trình tự cấp phát động

```
int* p = new int[3];
```



1. Khai báo biến p

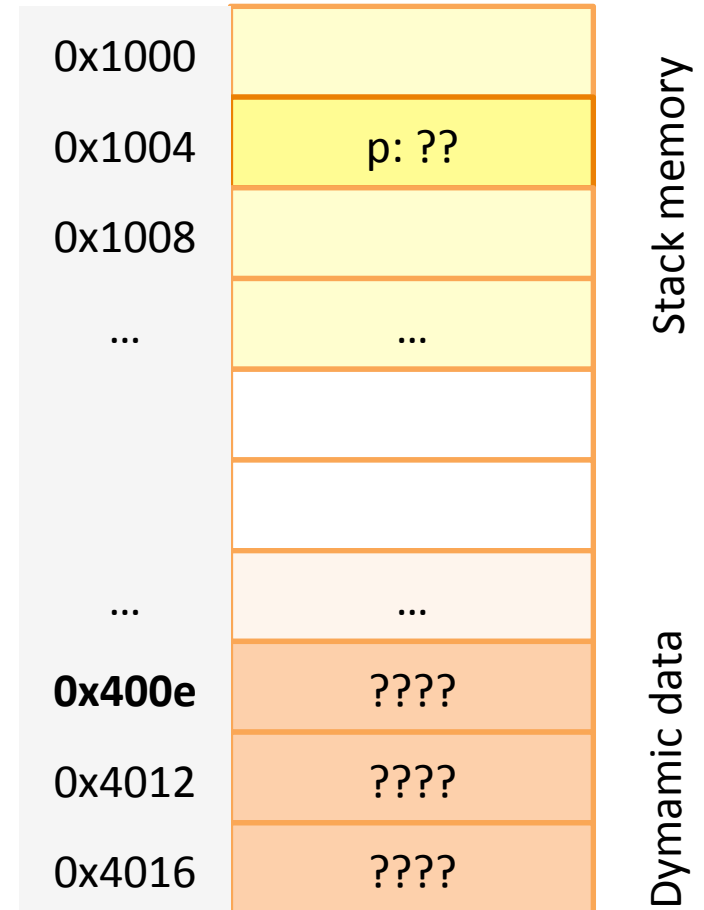


# Trình tự cấp phát động

```
int* p = new int[3];
```



1. Khai báo biến p
2. Tính biểu thức new int[3]
  - Cấp phát chuỗi 3 ô nhớ int
  - Lấy địa chỉ ô nhớ đầu tiên



# Trình tự cấp phát động

```
int* p = new int[3];
```



1. Khai báo biến p
2. Tính biểu thức new int[3]
  - Cấp phát chuỗi 3 ô nhớ int
  - Lấy địa chỉ ô nhớ đầu tiên
3. Thực hiện phép gán

0x1000		Stack memory
0x1004	p: 0x400e	
0x1008		
...	...	
		Dynamic data
...	...	
0x400e	????	
0x4012	????	
0x4016	????	

# Thu hồi vùng dữ liệu động

```
int* p = new int;  
// sử dụng p ...  
delete p;
```

```
int* arr = new int[10];  
// sử dụng arr ...  
delete [] arr;
```

- Thu hồi bằng toán tử **delete**
- Sau khi thu hồi, trình biên dịch có thể tái sử dụng, cấp phát cho lần new khác.
- Vùng nhớ chưa được thu hồi sẽ không thể được sử dụng cho biến khác → lập trình viên cần thu hồi sau khi không còn sử dụng nữa.

# Lỗi thường gặp: thất thoát bộ nhớ

```
ptr1 = new int;  
ptr2 = new int;  
ptr1 = ptr2;
```

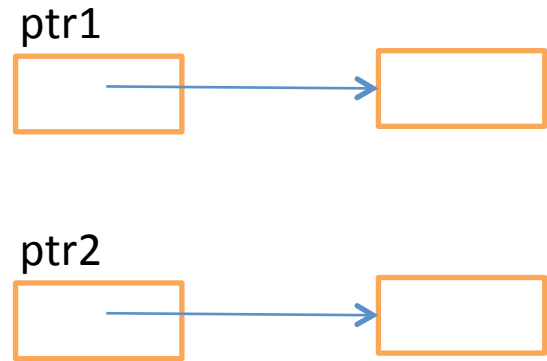
# Lỗi thường gặp: thất thoát bộ nhớ

```
ptr1 = new int;
```

```
ptr2 = new int;
```

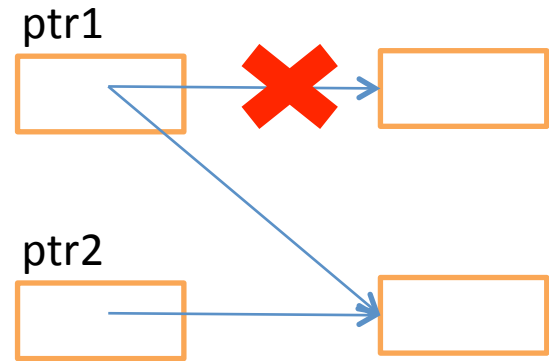
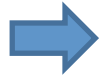
➔ 

```
ptr1 = ptr2;
```



# Lỗi thường gặp: thất thoát bộ nhớ

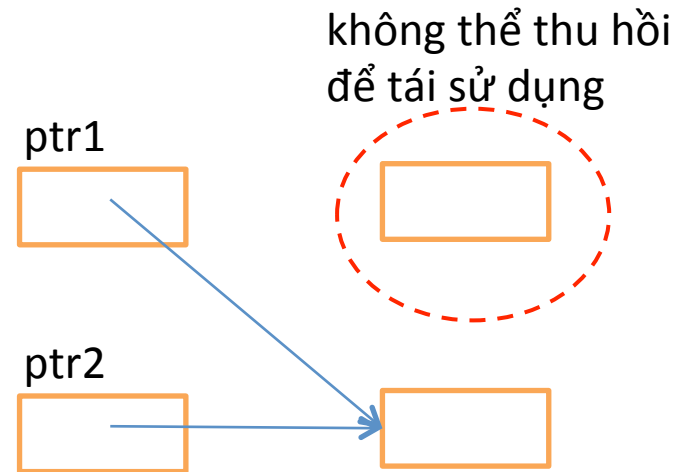
```
ptr1 = new int;  
ptr2 = new int;  
ptr1 = ptr2;
```





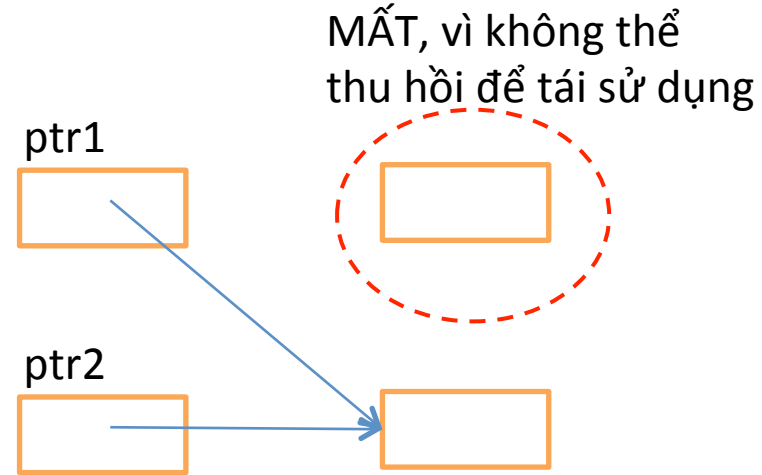
# Lỗi thường gặp: thất thoát bộ nhớ

```
ptr1 = new int;  
ptr2 = new int;  
ptr1 = ptr2;
```



# Lỗi thường gặp: thất thoát bộ nhớ

```
ptr1 = new int;  
ptr2 = new int;  
ptr1 = ptr2;
```



**Không dùng nữa thì phải giải phóng bộ nhớ  
càng sớm càng tốt!!!**

# Lỗi thường gặp: giải phóng quá sớm

- Đừng giải phóng bộ nhớ quá sớm, khi vẫn còn con trỏ trỏ tới vùng bộ nhớ đó.

```
int* p = new int;  
int* p2 = p;  
*p = 10;
```

```
delete p;
```

```
cout << *p2;
```

# Lỗi thường gặp: giải phóng quá sớm

- Đừng giải phóng bộ nhớ quá sớm, khi vẫn còn con trỏ trỏ tới vùng bộ nhớ đó.

```
int* p = new int;  
int* p2 = p;  
*p = 10;
```

```
delete p;
```

```
cout << *p2;
```

Giải phóng p làm cho  
p2 trở thành con trỏ  
vào vùng nhớ không  
còn hiệu lực

# Các lỗi khác

- Giải phóng vùng bộ nhớ đã được giải phóng
  - Từ hai con trỏ cùng trỏ vào một nơi
- Delete con trỏ trỏ tới giữa một vùng bộ nhớ động
- Giải phóng vùng bộ nhớ của biến địa phương.

# Lời khuyên

- Phải rất cẩn thận khi sử dụng con trỏ
- Để chương trình đỡ lỗi, nếu tránh được con trỏ thì nên tránh
  - Dùng tham chiếu
- Nếu không tránh được thì sử dụng công cụ phân tích mã nguồn để giúp phát hiện lỗi sử dụng con trỏ và địa chỉ bộ nhớ.