Ngôn ngữ lập trình C++

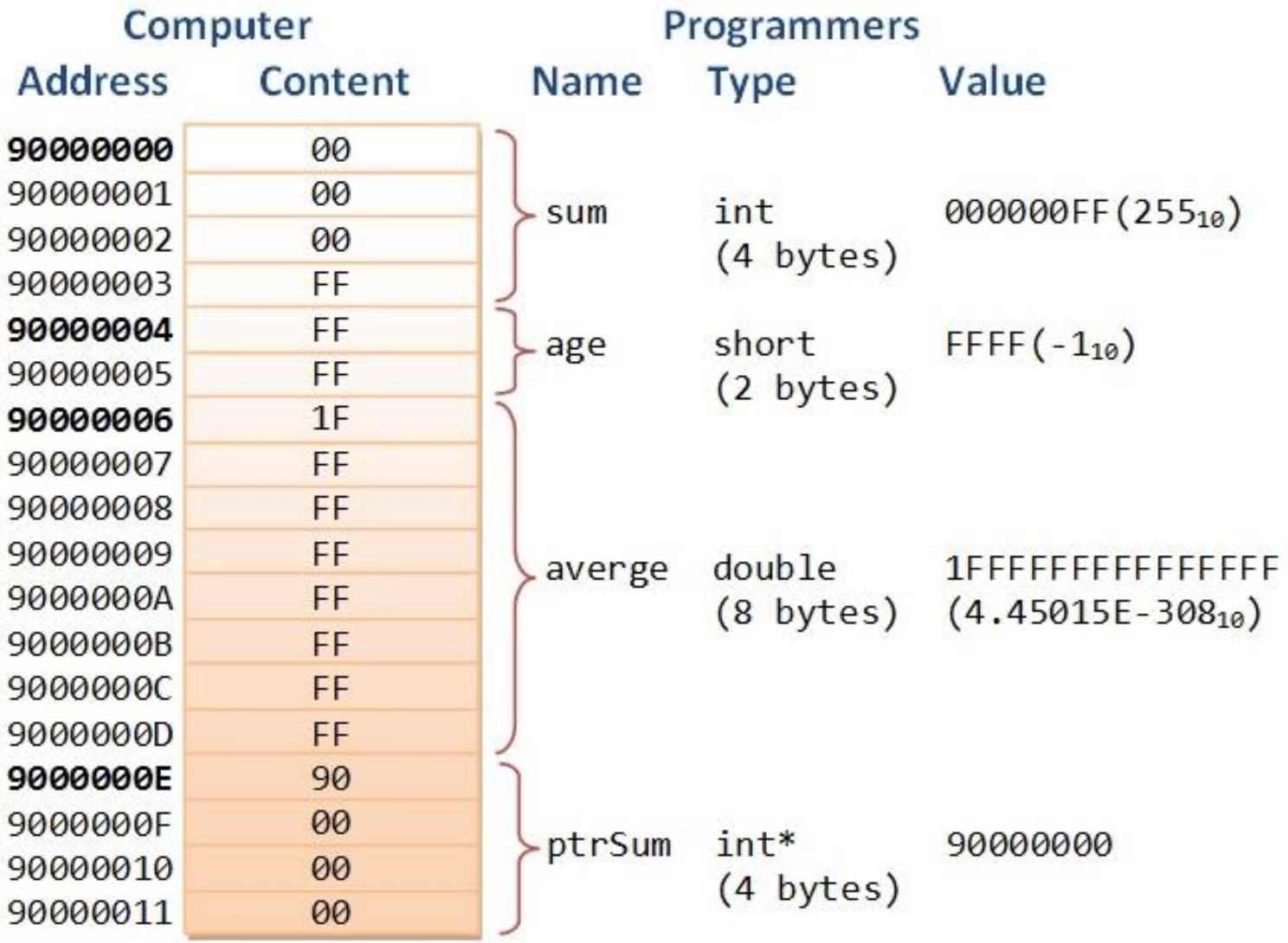
Chương 4.
CON TRỎ TRONG C++

4.1 Giới thiệu về con trỏ

A. Biến và địa chỉ của biến:

- Biến là một ô nhớ / vùng nhớ được hệ điều hành cấp phát cho chương trình nhằm để lưu trữ giá trị vào bên trong vùng nhớ đó.
- Để truy xuất đến giá trị mà biến đang nắm giữ, chương trình cần tìm đến vùng nhớ (địa chỉ) của biến để đọc giá trị bên trong vùng nhớ đó.
- Khi thao tác với các biến thông thường, người lập trình không cần quan tâm đến địa chỉ vùng nhớ của biến.
- Khi cần truy xuất giá trị của biến, người lập trình chỉ cần gọi định danh (tên biến)

A. Biến và địa chỉ của biến:



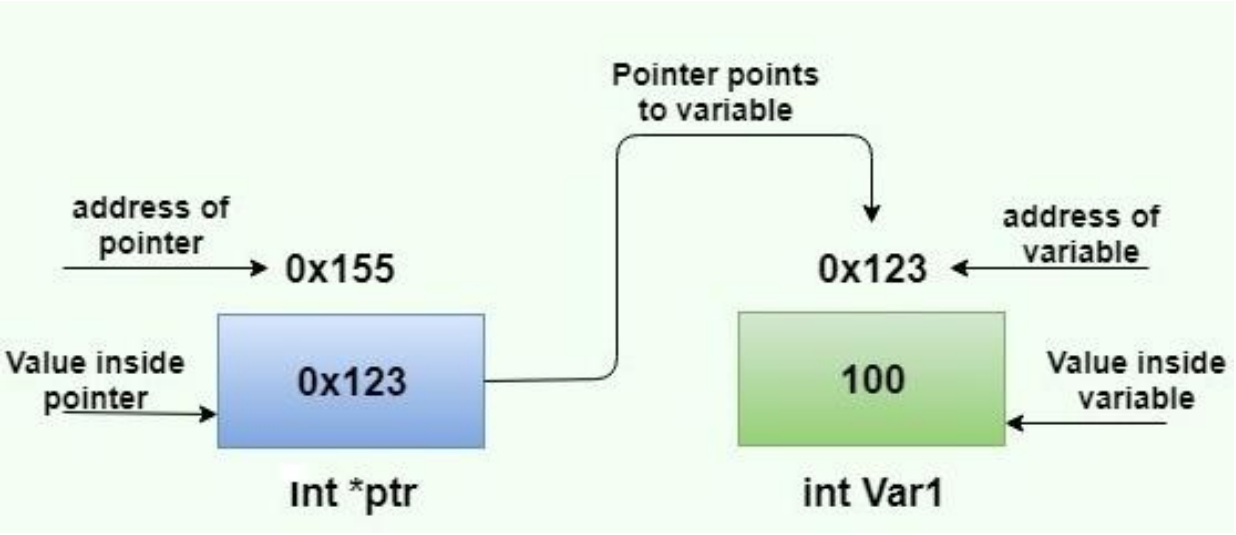
B. Con trỏ

- **Con trỏ (pointer)** là một biến chứa một địa chỉ bộ nhớ của một biến khác.
  - Khai báo: **kiểu dữ liệu *tên_biến_contrỏ;**
  - Kiểu dữ liệu của con trỏ dùng để xác định kiểu dữ liệu của biến mà nó trỏ đến.
  - Ví dụ: khai báo biến con trỏ kiểu int, float, double

```
int* p;   // p trỏ đến một biến kiểu int
float * pf; // pf trỏ đến một biến kiểu float
double *pdb; // pdb trỏ đến một biến kiểu double
```

  - Khai báo nhiều con trỏ cùng một kiểu

```
int  *p, *q, *u; // p, q, u là 3 con trỏ int
```



B. Con trỏ

- Lấy địa chỉ của biến (toán tử &-reference): **&ten_bien**
- Cần trỏ đến biến nào, ta gán địa chỉ của biến cho con trỏ.
- Ví dụ: int a=10, *ip;

  ip = &a;
- Qua biến trỏ, có thể truy nhập giá trị của đối tượng được trỏ (toán tử * - dereference)
- Cú pháp: ***biến_trỏ**
- Ví dụ: cout << “a=“ << *ip;
- Nếu ip trỏ đến a thì *ip có thể thay thế cho biến a trong các phép toán và lệnh.
- Ví dụ: *ip=20;

  cout << “a=“ << a; // xuất ra a=20
- Ví dụ: gioithieu_pointer.cpp

B. Con trỏ

- Bản thân con trỏ cũng có địa chỉ. Lấy địa chỉ của con trỏ: &biến_trỏ
- Độ lớn của con trỏ:
  - Với hệ điều hành 32bit: 4 Byte
  - Với hệ điều hành 64bit: 8 Byte
- Lấy độ lớn của con trỏ: sizeof(biến_trỏ)

C. Con trỏ NULL

- Tương tự như các biến thông thường, **con trỏ không được khởi tạo khi khai báo**. Nếu con trỏ không được khởi tạo một giá trị hay được gán với 1 địa chỉ, giá trị trong biến con trỏ là 1 giá trị vô định (rác).

$\Rightarrow$Khi chưa trỏ vào đâu thì nên khởi gán con trỏ bằng NULL.

$\Rightarrow$**Con trỏ NULL trong C++** là một hằng với giá trị = 0 được định nghĩa trong một vài thư viện chuẩn, gồm **iostream**.

```
int *ptr = 0;    // ptr là con trỏ NULL
```

Hoặc:

```
int *ptr = NULL;
```

- Nếu một con trỏ chứa giá trị 0 được xem như là không trỏ tới bất cứ thứ gì
- Ví dụ: controNull.cpp

D. Con trỏ void

- Con trỏ kiểu **void** là một con trỏ không định kiểu, có thể trỏ đến bất kỳ đối tượng nào (với bất kỳ kiểu dữ liệu nào) có địa chỉ cụ thể.
- Cách khai báo: void *tên_biếntrỏ;
- Ví dụ:

```
double  x; int  y; char  z;
void* ptr;
ptr = &x; //Hợp lệ! ptr chứa địa chỉ của số thực x.
ptr = &y; //Hợp lệ! ptr chứa địa chỉ của số nguyên y;
ptr = &z; //Hợp lệ! ptr chứa địa chỉ của ký tự z.
```

D. Con trỏ void

- Con trỏ **void** không xác định được kiểu dữ liệu của vùng nhớ mà nó trỏ tới $\Rightarrow$ không thể truy xuất trực tiếp nội dung thông qua toán tử *.
- Cần phải được ép kiểu một cách rõ ràng sang con trỏ có kiểu dữ liệu khác trước khi sử dụng toán tử * cho vùng nhớ mà con trỏ void đang giữ.
- Ví dụ:

```
int value = 5;
void *vPtr = &value;
int *iPtr = static_cast<int *> (vPtr); // (int*)vPtr
cout << *iPtr << endl;
/* Chỉ có thể sử dụng toán tử * lên iPtr chứ không thể sử dụng cho con trỏ vPtr.  Xem controvoid.cpp*/
```

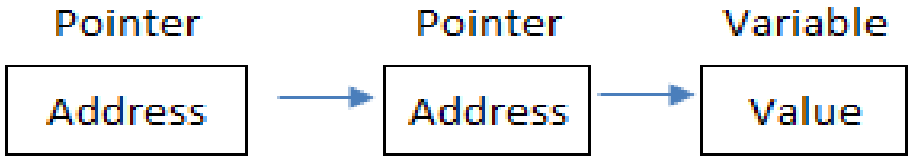
D. Con trỏ void

- Mục đích sử dụng con trỏ void:
  - Khi mà dữ liệu bên trong vùng nhớ đó không quan trọng. Ví dụ copy dữ liệu từ vùng nhớ này sang vùng nhớ khác mà không cần quan tâm định dạng của chúng.
  - Làm tham số của hàm khi đối của hàm là con trỏ có kiểu dữ liệu bất kỳ.
- Việc giải phóng một vùng nhớ trên Heap bằng tên con trỏ kiểu void cũng có thể gây ra lỗi vì HĐH không tính được kích thước vùng nhớ cần thu hồi là bao nhiêu.

$\Rightarrow$Tránh sử dụng con trỏ kiểu void trừ những lúc thực sự cần thiết để tránh gây ra những sai sót không đáng có cho chương trình.

E. Con trỏ trỏ đến con trỏ

- Thông thường, một con trỏ chứa địa chỉ của một biến. Khi định nghĩa một con trỏ (pointer) trỏ tới một con trỏ, con trỏ đầu tiên chứa địa chỉ của con trỏ thứ hai, con trỏ thứ 2 trỏ tới vị trí chứa giá trị.



- Con trỏ tới một con trỏ phải được khai báo bởi việc đặt thêm một dấu sao (*) ở trước tên của nó.
- Ví dụ: int **ptr;
- Nếu muốn lấy giá trị của biến của con trỏ mà nó đang trỏ tới phải dùng hai lần toán tử *:
- Ví dụ: int *p = new int(2021);
  - Lần 1: Lấy địa chỉ của con trỏ mà nó (***ptr***) trỏ tới: int **ptr = &p;
  - Lần 2: Lấy giá trị của biến có địa chỉ mà con trỏ p trỏ tới:

    cout << **ptr;

    // được 2021 do ***ptr*** lấy giá trị lưu trữ tại địa chỉ mà ***p*** đang giữ
- Ví dụ: ptrtoptr.cpp

F. Con trỏ cấu trúc

- Một biến struct sẽ bao gồm tập hợp các trường dữ liệu mà địa chỉ của biến struct sẽ là địa chỉ của trường dữ liệu khai báo đầu tiên trong struct.
- Khai báo con trỏ tới cấu trúc: **Kiểu_cấutrúc *tên_biếntrỏ;**
- Mục đích sử dụng:
  - Làm tham số trong hàm,
  - Đặc biệt có ý nghĩa khi xây dựng các cấu trúc dữ liệu động như danh sách liên kết, ngăn xếp, hàng đợi, cây,... $\Rightarrow$ **cấu trúc tự trỏ**.
- Truy nhập vào thành phần của cấu trúc:
  - Cách 1 (thông dụng): biến_trỏ-> tên_thành_phần
  - Cách 2: (* biến_trỏ).tên_thành_phần
- Ví dụ: controcautruc1(2).cpp

G. Các phép toán với con trỏ

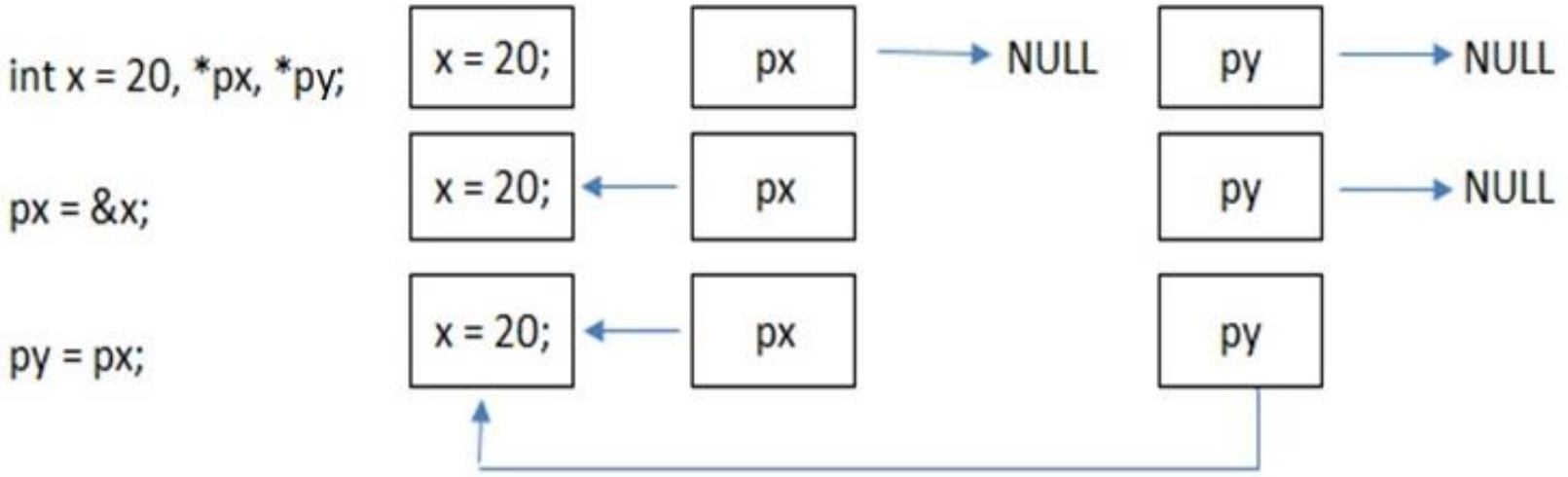
- Các phép toán số học: +,-,++,--:
  - Cộng hoặc trừ với một số nguyên n trả về 1 con trỏ cùng kiểu, là địa chỉ mới trỏ tới một đối tượng khác nằm cách đối tượng đang bị trỏ n phần tử.
  - Ví dụ:

```
short* sp; float* fp;
//Giả sử địa chỉ sp trỏ tới là 200, fp là 300;
sp++; // sp sẽ trỏ tới địa chỉ 202;
fp +=5; // fp sẽ trỏ tới địa chỉ 300 + 5*4 = 320
```

  - Trừ 2 con trỏ cùng kiểu cho lại khoảng cách (số phần tử) giữa 2 con trỏ (ví dụ với mảng)
- Các phép toán so sánh <, >, <=, >=, ==, !=
- Ví dụ: ptr_op.cpp

G. Các phép toán với con trỏ

- Phép gán:
  - Là phép gán **địa chỉ của biến**, không gán **giá trị của biến** do con trỏ trỏ tới.
  - Hai con trỏ phải cùng kiểu.
  - Trong trường hợp hai con trỏ khác kiểu, phải sử dụng các phương thức ép kiểu.
- Ví dụ:



4.2 Con trỏ và cấp phát động

A. Biến tĩnh và biến động

- Biến tĩnh hay biến được cấp phát tĩnh là biến được khai báo bằng cú pháp khai báo biến, có tên và được cấp phát một vùng nhớ cố định trước khi sử dụng.
- Vùng nhớ cố định là vùng nhớ luôn tồn tại khi chương trình thực thi, không thể được xóa đi (tức trả lại cho hệ điều hành) hoặc thay đổi kích thước (đối với mảng), sau khi kết thúc chương trình vùng nhớ đó sẽ được tự động trả lại cho hệ điều hành.

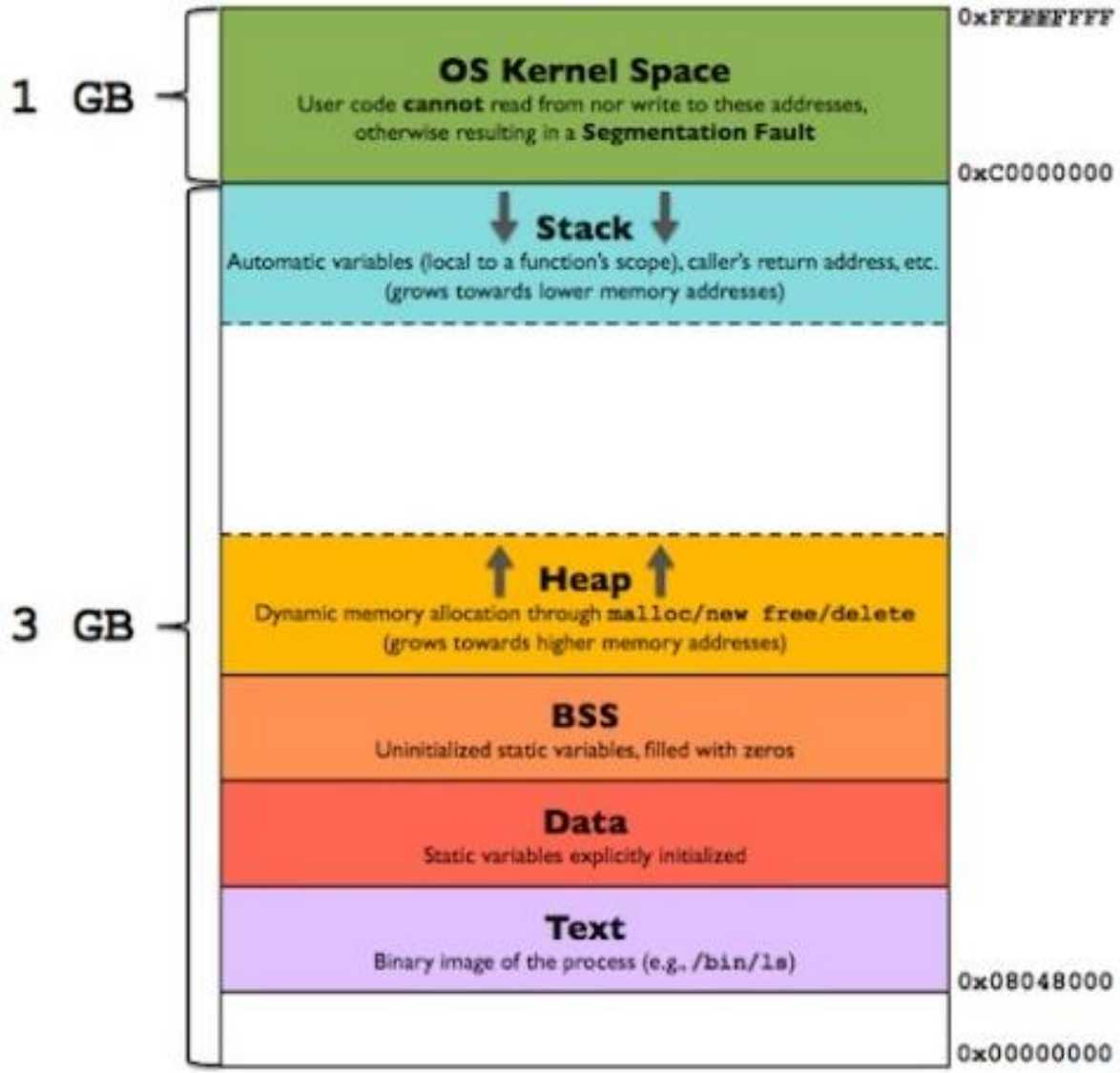
$\Rightarrow$ Gây chiếm dụng bộ nhớ nếu không có nhu cầu sử dụng biến nữa, hoặc không thể thay đổi kích thước nếu dữ liệu vượt quá kích thước lưu trữ của biến (đối với mảng).

$\Rightarrow$ **Sử dụng biến động**.

- Biến động hay biến được cấp phát động là biến được cấp phát một vùng nhớ trong bộ nhớ RAM, không được liên kết với tên biến do đó nó không có tên mà chỉ là một vùng nhớ.

$\Rightarrow$ **Việc quản lý biến động được thực hiện qua con trỏ.**

Bản đồ cấp phát bộ nhớ cho chương trình (HĐH32bit)



Ví dụ: VD_memory_layout.cpp

B. Con trỏ và cấp phát động

- Cấp phát vùng nhớ cho một biến động:

  new kiểu_dữ_liệu;

- Ví dụ: new int;

  new float;

- Nếu thành công, toán tử new sẽ trả về một con trỏ trỏ tới địa chỉ của vùng nhớ mới.
- Do biến động không có tên nên được quản lý bằng con trỏ $\Rightarrow$ có thể gán địa chỉ của biến động cho con trỏ :

  int *ptr = new int; // *con trỏ ptr trỏ tới biến động kiểu int đã được tạo*

- Cũng có thể khởi tạo giá trị ngay khi khai báo:
- Ví dụ: int *ptr = new int(2021);
- Có thể thao tác trên biến động vừa cấp phát thông qua con trỏ:
- Ví dụ: *ptr = 1989;

B. Con trỏ và cấp phát động

- Cấp phát động là yêu cầu cấp phát một vùng nhớ $\Rightarrow$ sẽ có thể xảy ra trường hợp không đủ bộ nhớ để cấp phát $\Rightarrow$ toán tử **new** sẽ trả về con trỏ NULL.
- Có thể kiểm tra:

```
int* myPtr = new int;
if    (myPtr != NULL)  //hoặc  if (myPtr)
      cout << “Cap phat thanh cong!";
else
      cout << “Khong cap phat duoc!";
```

- Con trỏ NULL chỉ ra rằng con trỏ không trỏ đến đâu cả.

B. Con trỏ và cấp phát động

- Sau khi đã sử dụng xong, dữ liệu trong vùng nhớ của biến động cần được xóa đi và trả lại cho HĐH $\Rightarrow$ Rất quan trọng tránh việc vùng nhớ tồn tại nhưng HĐH không được sử dụng do đã được cấp phát cho chương trình $\Rightarrow$ Rò rỉ bộ nhớ!
- Hầu hết các HĐH hiện đại quản lý việc cấp phát bộ nhớ một cách triệt để $\Rightarrow$ Mỗi khi chương trình kết thúc bộ nhớ đã được cấp phát sẽ được thu hồi lại, tuy nhiên dữ liệu trong vùng nhớ đó không được xóa $\Rightarrow$ Cũng dẫn đến rò rỉ bộ nhớ!

$\Rightarrow$ ***Cần xóa và giải phóng biến động mỗi khi kết thúc chương trình hoặc sử dụng xong.***

- Cú pháp : **delete tên_biếncontrỏ;**
- Ví dụ: delete ptr;
- Sau khi xóa, vùng nhớ đã được xóa dữ liệu và trả lại cho HĐH quản lý, tuy nhiên, con trỏ mà đang trỏ đến vùng nhớ đó vẫn đang chứa địa chỉ $\Rightarrow$ Sử dụng con trỏ này sẽ gây ra hậu quả không mong muốn do biến động nó trỏ tới không còn tồn tại $\Rightarrow$ Đặt con trỏ = NULL.
- Ví dụ: ptr=NULL;

4.3 Con trỏ và mảng

A. Con trỏ và mảng 1 chiều

- Mảng 1 chiều là 1 con trỏ chứa địa chỉ của phần tử đầu tiên.
- Con trỏ và mảng trong C/C++ có liên hệ chặt chẽ với nhau, có thể thay thế cho nhau trong một số trường hợp.
- Có thể dùng toán tử * cho tên_mảng nhưng không được phép sửa đổi giá trị vì tên_mảng là hằng số con trỏ trỏ tới phần tử đầu tiên của mảng.
- Ví dụ:

```
int arr[10];
*(arr + 2) = 150; //arr[2]=150;
```

- Ví dụ: contro_mang1.cpp

B. Con trỏ mảng

- Có thể dùng con trỏ để truy nhập mảng dữ liệu 1 chiều
- Cú pháp: **biến_contrỏ = tên_mảng;**
- Ví dụ:

  int a[10];

  int *p = a; // hay int *p; p=a; **Phép gán hợp lệ**
- Sử dụng con trỏ để truy nhập giá trị phần tử mảng

  p[0] ⇔ a[0]

  p[1] ⇔ a[1]

  …

  p[9] ⇔ a[9]

B. Con trỏ mảng

- Có thể thực hiện phép + và phép – với biến kiểu con trỏ để truy nhập vào các phần tử của mảng.

***(biến_contrỏ + chỉ số)**

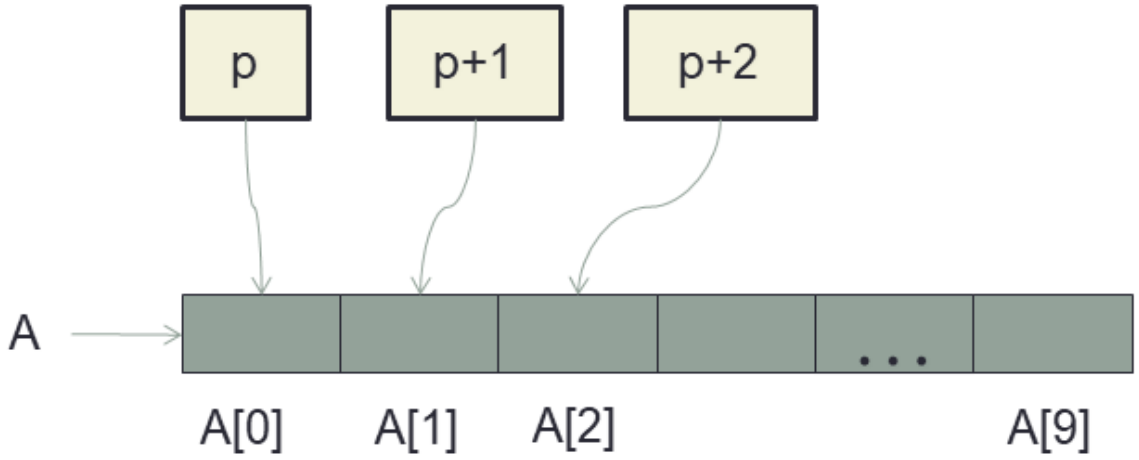
tương đương với: **biến_contrỏ[chỉ số]**

- Ví dụ:

```
int A[10];  int  *p  = A; // p trỏ đến A[0]
```

p + 1 trỏ đến A[1]

p + 2 trỏ đến A[2]



- Ví dụ: contro_mang2.cpp

C. Kỹ thuật tạo mảng động

- Mảng động là mảng dữ liệu được cấp phát bộ nhớ sau khi chương trình đã chạy.
- Trái ngược với mảng tĩnh, kích thước của mảng động không được biết trước.
- Kỹ thuật tạo mảng động trên C++ dựa trên
  - **Biến con trỏ**
  - Toán tử cấp phát bộ nhớ : **new**
  - Toán tử giải phóng bộ nhớ : **delete**

C. Kỹ thuật tạo mảng động

- Cấp phát bộ nhớ bằng từ khóa ***new***:
  - Cú pháp: **new kiểu_dữliệu[số phần tử]**
    - Kiểu dữ liệu: kiểu dữ liệu của phần tử trong mảng
    - Số phần tử: Số phần tử của mảng, có thể là số phần tử tức thời được biết tại thời điểm cấp phát.
  - Ví dụ:

    ```
    int N;
    cin >> N;
    int * pv= new int[N];
    ```
- Giải phóng vùng nhớ được cấp phát bằng từ khóa ***delete:***
  - Cú pháp: **delete[] tên_mảngđộng**
  - Toán tử [] dùng để báo với hệ điều hành vùng nhớ đã được cấp phát không dùng cho 1 biến đơn.
  - Ví dụ: delete[] pv;
- Ví dụ: contro_mang3.cpp

C. Kỹ thuật tạo mảng động

- Tác dụng của mảng động:
  - Mảng tĩnh chiếm giữ bộ nhớ cho đến khi chương trình kết thúc → không hiệu quả.
  - Mảng động linh hoạt hơn, khi nào không cần dùng đến có thể thu hồi lại vùng bộ nhớ đã cấp.
  - Sử dụng trong trường hợp số phần tử của mảng không biết trước hoặc không phải hằng số, hạn chế các ô nhớ không dùng đến do khai báo quá nhiều số phần tử cũng như thiếu chỗ nếu khai báo số phần tử thấp hơn thực tế.
  - Vùng nhớ Heap dành cho cấp phát động lớn hơn nhiều vùng nhớ dành cho cấp phát tĩnh. (Ví dụ: ptrapp.cpp)

D. Mảng các con trỏ

- Khai báo:

kiểu_dữliệu *tênmảng_contrỏ[số phần tử];

- Ứng dụng:
  - Trỏ đến một mảng mà các phần tử đòi hỏi dung lượng bộ nhớ lớn.
  - Trỏ tới mảng nhiều chiều.
- Ví dụ: Quản lý hồ sơ sinh viên một trường:

```
struct  sinhvien{
int masv;
string hoten, ngaysinh, diachi, email;
int dienthoai;
string khoa, lop;
int sotc;
float dtb;};
sinhvien sv[2000]; //Khai báo tĩnh sẽ tốn nhiều bộ nhớ.
```

$\Rightarrow$ Sử dụng mảng con trỏ: sinhvien *ptr_sv[2000];

$\Rightarrow$ Ví dụ: mangcontro_cautruc1.cpp

E. Con trỏ và mảng nhiều chiều

- Mảng tĩnh:

  kiểu_dữ_liệu tên_mảng[số hàng][số cột]

- Mảng động:

1) Mảng các con trỏ
2) Con trỏ trỏ đến con trỏ

E. Con trỏ và mảng nhiều chiều

- Mảng động:

1) Mảng các con trỏ:

- Khai báo:

```
kiểu_dữ_liệu  *tên_mảng[số_hàng];
```

- Cấp phát bộ nhớ:

```
for (int i=0;   i<số_hàng;   i++)
    tên_mảng[chỉ_số_hàng] = new kiểu_dữ_liệu[số_cột];
```

- Ví dụ:

```
int *a[10];
for (int i=0;   i<10;   i++)
    a[i] = new int[5];
```

- Giải phóng vùng nhớ đã được cấp phát:

```
for (int i=0;  i<số_hàng;  i++)
{   delete[] tên_mảng[chỉ số hàng];
    tên_mảng[chỉ số hàng]=NULL; }
```

- Ví dụ: contro_mang2chieu1.cpp

E. Con trỏ và mảng nhiều chiều

- Mảng động:

2) Con trỏ trỏ đến con trỏ:

- Khai báo:

kiểu_dữ_liệu **tên_mảng;

- Cấp phát bộ nhớ (2 lần):

```
tên_mảng = new kiểu_dữ_liệu*[số_hàng];
for (int i=0;   i<số_hàng;   i++)
    tên_mảng[chỉ_số_hàng] = new kiểu_dữ_liệu[số_cột];
```

- Ví dụ:

```
int **b=new int*[10];
for (int i=0;   i<10;   i++)
    b[i] = new int[5];
```

- Giải phóng vùng nhớ đã được cấp phát (2 lần):

```
for (int i=0;  i<số_hàng;  i++)
    delete[] tên_mảng[chỉ số hàng];
delete[] tên_mảng;
tên_mảng=NULL;
```

- Ví dụ: contro_mang2chieu2.cpp, (mangcontro_cautruc2.cpp)

E. Con trỏ và mảng nhiều chiều

- Truy nhập vào phần tử mảng động:
  - Theo chỉ số:

    Tên_contrỏ[chỉsố_hàng][chỉsố_cột]
  - Theo con trỏ:

    ((ten_contrỏ + chỉsố_hàng) + chỉsố_cột)

4.4 Con trỏ và hàm

- Khai báo hàm:

Kiểu_dữ_liệu_trả về **tên_hàm**(*danh sách khai báo tham số hình thức*)

$\Rightarrow$Con trỏ là tham số/đối của hàm

$\Rightarrow$Kiểu dữ liệu trả về là con trỏ

A. Con trỏ là tham số/đối của hàm

- Truyền tham số cho hàm:

1) Truyền theo trị
2) Truyền theo biến:
   - Dùng tham chiếu: &tham_số

   Ví dụ: swap_reference.cpp
   - Dùng con trỏ: *tham_số

   Ví dụ: swap_pointer.cpp

B. Con trỏ và kiểu trả về của hàm

- Các kiểu trả về của hàm:

1) Giá trị:

```
kiểu_dữ _liệu    tên_hàm(danh sách tham số){
…
return giá_trị; // hoặc biểu_thức}
```

2) Tham chiếu
3) Con trỏ

B. Con trỏ và kiểu trả về của hàm

2) **Tham chiếu:** Giá trị trả về là biến không phải hằng hay biểu thức.

kiểu_dữ _liệu& tên_hàm(danh sách tham số)

Chú ý:

- **Không** trả về các tham chiếu của **biến cục bộ** do biến cục bộ **bị hủy** ngay sau khi ra khỏi hàm $\Rightarrow$ hàm trả về một tham chiếu đến vùng nhớ rác.
- Cách giải quyết: Cấp phát động hoặc biến cục bộ là static.
- Vùng nhớ cho biến static tồn tại theo vòng đời của chương trình và giá trị của biến được lưu lại để có thể sử dụng qua các lần gọi hàm tiếp theo.
- Ví dụ: thamchieuham1.cpp

Ứng dụng của hàm trả về tham chiếu:

- Hàm trả về tham chiếu có thể được sử dụng ở phần trái của một lệnh gán.
- Khi trả về các đối số hàm được truyền bằng tham chiếu.
- Khi hàm trả về một **kiểu cấu trúc (struct)** hoặc **lớp (class)** phức tạp.
- Khi trả về một phần tử từ một mảng được truyền vào hàm.
- Ví dụ: thamchieuham2.cpp

B. Con trỏ và kiểu trả về của hàm

3) **Con trỏ:** Kiểu dữ liệu hàm trả về là địa chỉ.

kiểu_dữ _liệu* tên_hàm(danh sách tham số)

- Chú ý: Tương tự như tham chiếu
- Ứng dụng: Khi trả về một giá trị mảng hoặc con trỏ.
- Ví dụ: controham1(2,3,4).cpp

4.5 Standard Template Library (STL)

- Standard Template Library (STL) là một tập hợp các Template trong C++ để cung cấp các lớp và các hàm được tạo theo khuôn mẫu cho mục đích lập trình tổng quát.
- Bộ thư viện thực hiện toàn bộ các công việc vào ra dữ liệu (iostream), quản lý mảng (vector), thực hiện hầu hết các tính năng của các cấu trúc dữ liệu cơ bản (stack, queue, map, set...).
- STL còn bao gồm các thuật toán cơ bản: tìm min, max, tính tổng, sắp xếp (với nhiều thuật toán khác nhau), thay thế các phần tử, tìm kiếm (tìm kiếm thường và tìm kiếm nhị phân), trộn... được cung cấp dưới dạng template nên việc lập trình luôn thể hiện tính khái quát hóa cao.