

Ngôn ngữ lập trình C

Bài 6. Hàm & Các khai báo

Soạn bởi: TS. Nguyễn Bá Ngọc

Nội dung

- Khái niệm hàm
- Định nghĩa và khai báo hàm
- Biểu thức gọi hàm
- Các thuộc tính của biến

Nội dung

- Khái niệm hàm
- Định nghĩa và khai báo hàm
- Biểu thức gọi hàm
- Các thuộc tính của biến

Khái niệm hàm

- Hàm trong NNLT là tính năng cho phép đóng gói một phần mã nguồn thành 1 đơn vị cú pháp có thể được sử dụng nhiều lần ở nhiều nơi mà không cần lặp lại mã nguồn.
- Với hàm chúng ta có thể chia nhỏ một công việc lớn thành nhiều công việc nhỏ hơn có thể được thực hiện bởi nhiều người.
- Hàm cũng ẩn các chi tiết triển khai, người lập trình có thể sử dụng hàm như một thao tác bậc cao, chỉ quan tâm tới các dữ liệu đầu vào và các kết quả đầu ra.
- => Hàm là một tính năng quan trọng của bất kỳ NNLT nào.

Khái niệm hàm trong NNLT có một số đặc điểm chung nhưng trong tổng thể thì khác với khái niệm hàm trong toán học

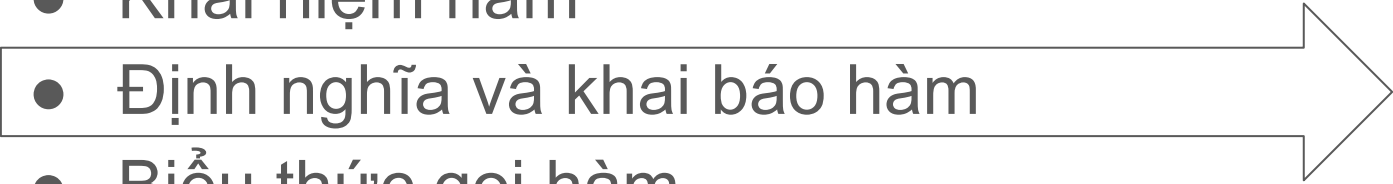
Tóm tắt các nội dung đã học về hàm

Có thể nói chúng ta đã sử dụng hàm ngay từ những chương trình đầu tiên:

- Hàm main như chúng ta đã viết trong các chương trình là một hàm hoàn chỉnh. Tuy nhiên main là 1 hàm đặc biệt:
 - Được quy ước là điểm bắt đầu thực hiện chương trình.
 - Không được khai báo inline hoặc static.
- Chúng ta đã sử dụng một số hàm trong thư viện chuẩn:
 - **printf** - Hàm xuất dữ liệu theo định dạng.
 - **scanf** - Hàm nhập dữ liệu theo định dạng.
 - **sqrt, sqrtf** - Các hàm tính căn bậc 2.

Bài giảng này sẽ tiếp tục cung cấp các chi tiết về cách định nghĩa hàm, khai báo hàm và gọi hàm.

Nội dung

- Khái niệm hàm
 - Định nghĩa và khai báo hàm
 - Biểu thức gọi hàm
 - Các thuộc tính của biến
- 

Ví dụ 6.1. Định nghĩa hàm

Chương trình này sử dụng hàm để quy đổi độ dài được đo bằng inch sang cm.

```
vd6-1.c x
6  #include <stdio.h>
7
8  #define INCH1 2.54
9
10 double inch_to_cm(double x) {
11     return x * INCH1;
12 }
13
14 int main() {
15     double x;
16     printf("Nhập độ dài (inch): ");
17     scanf("%lf", &x);
18     printf("%.2f in = %.2f cm\n", x, inch_to_cm(x));
19     return 0;
20 }
```

Nguyên mẫu hàm

```
bangoc:$gcc -o prog vd6-1.c
bangoc:$./prog
Nhập độ dài (inch): 5
5.00 in = 12.70 cm
bangoc:$
```

Định nghĩa hàm `inch_to_cm`

- Trả về giá trị có kiểu `double`

- Nhận 1 tham số có kiểu `double`

Đóng gói và ẩn cách chuyển đổi inch => cm

Câu lệnh `return` trả về giá trị cho biểu thức gọi hàm

Biểu thức gọi hàm với đối số `x`

Hàm `inch_to_cm` thuộc **kiểu hàm trả về (giá trị kiểu) `double`**

Sơ lược định nghĩa & khai báo hàm

NNLT C tách rời 2 khái niệm định nghĩa hàm và khai báo hàm.

- Chúng ta chỉ cần khai báo hàm trước khi sử dụng. Định nghĩa hàm có thể xuất hiện sau, hoặc trong 1 đơn vị biên dịch khác. Tuy nhiên sử dụng hàm không được định nghĩa sẽ phát sinh lỗi ghép nối (*tiến trình biên dịch trong **Bài 1***).
- Cấu trúc mã nguồn:
 - Thông thường các khai báo hàm được đặt trong các tệp .h và được chèn vào các đơn vị biên dịch để sử dụng.
 - Các định nghĩa hàm thường được cung cấp 1 lần và được đặt trong các tệp .c để tránh định nghĩa lại.
- Hàm (thường) được định nghĩa và khai báo trong phạm vi tệp và có phạm vi đóng gói là tập con của phạm vi tệp.
(tham khảo định nghĩa hàm trong hàm với mở rộng GNU)

Cấu trúc cơ bản của hàm

Ở dạng cơ bản hàm được định nghĩa theo cấu trúc:

kiểu-trả-về **tên-hàm**(*các-khai-báo-tham-số_{opt}*) **câu-lệnh-gộp**

- *kiểu-trả-về*: Hàm có thể trả về kiểu số và nhiều kiểu khác do người dùng tự định nghĩa (sẽ học sau):
 - Nếu hàm không trả về giá trị thì kiểu trả về được để là **void**.
 - (Tính năng mặc định kiểu trả về là *int*, ví dụ `main()`, trong phiên bản cũ có thể không còn được hỗ trợ trong tương lai).
- **tên-hàm**: Là định danh có thể được sử dụng để gọi hàm.
- *các-khai-báo-tham-số_{opt}*: Các tham số (nếu có) sẽ có phạm vi khối và được sử dụng trong câu-lệnh-gộp/thân hàm.
- **câu-lệnh-gộp**: Thân hàm, được thực hiện khi hàm được gọi và sau khi giá trị của các tham số được thiết lập.

Khai báo các tham số

Hàm có thể nhận một danh sách tham số hữu hạn. Trong trường hợp như vậy chúng ta có 2 cách khai báo:

- Cách khai báo thông dụng (hiện nay) là khai báo các tham số theo hình thức danh sách kiểu, khai báo từng tham số:
 - Ví dụ: `void f(int a, int b, int c) { /* ... */ } // OK`
 - `void f(int a, b, c) { /* ... */ } // NOK-Cần khai báo từng tham số`
 - Các đối số được ép kiểu thành kiểu của tham số.
 - Được coi là phong cách C hiện đại.
- Các tham số cũng có thể được khai báo theo hình thức danh sách định danh (có thể gặp trong mã nguồn cũ):
 - Ví dụ: `void f(a, b, c) { /* ... */ } // Được mặc định có kiểu int`
 - `void f(a, b) double a, b; { /* .. */ } // Khai báo sau`
 - Không ép kiểu các đối số.
 - *(Có thể không còn được hỗ trợ trong tương lai).*

Khai báo các tham số₍₂₎

Hàm cũng có thể không nhận tham số, và cũng có thể nhận một số lượng tham số tùy ý.

- Trường hợp hàm không nhận tham số:
 - Trong định nghĩa hàm chúng ta có thể để trống danh sách khai báo tham số.
 - Lưu ý về sự đa nghĩa của danh sách trống:
 - Trong định nghĩa: Danh sách trống = Không nhận tham số
 - Trong khai báo: Danh sách trống = Chưa biết số lượng tham số
 - Chúng ta cũng có thể sử dụng từ khóa **void** để mô tả hàm không có tham số trong cả khai báo và định nghĩa.
- Trường hợp hàm nhận số lượng tham số tùy ý:
 - Ví dụ: `extern int printf (const char *format, ...);`
 - *(Tham khảo thêm)*

Các thuộc tính của hàm

Để đáp ứng các yêu cầu phát triển phần mềm, định nghĩa hàm ở dạng cơ bản có thể được mở rộng với các thuộc tính.

- Trong các chương trình được tạo thành từ nhiều đơn vị biên dịch, hàm có thể được khai báo với thuộc tính liên kết:
 - **Ngoại/extern** (mặc định): Định nghĩa trong 1 đơn vị biên dịch nhưng có thể được sử dụng trong nhiều đơn vị biên dịch.
 - **Nội/static**: Giới hạn chỉ sử dụng hàm trong đơn vị biên dịch.
- Gọi hàm là thao tác phức tạp, để tối ưu hóa hàm có thể được khai báo với thuộc tính inline:
 - **inline**: Có ý nghĩa như 1 chỉ dẫn cho trình biên dịch tìm cách tối ưu hóa các gọi hàm, ví dụ thay thế biểu thức gọi hàm bằng mã nguồn được sinh tự động dựa trên định nghĩa hàm.
 - *(Tham khảo thêm các quy tắc áp dụng cho hàm inline).*
- Các thành phần khai báo: Kiểu trả về và các thuộc tính kết hợp lại được gọi là **đặc-tả-khai-báo**.

Khai báo hàm & nguyên mẫu hàm

Khai báo hàm bao gồm các thành phần như trong định nghĩa hàm trừ phần thân hàm. Cấu trúc tổng quát:

đặc-tả-khai-báo **tên-hàm**(*các-khai-báo-tham-số_{opt}*);

- Định nghĩa hàm mặc định chứa 1 khai báo của hàm được định nghĩa.
- Khai báo hàm như cấu trúc tổng quát đã nêu không phải là thành phần của định nghĩa hàm nào.
- Khai báo hàm có mô tả kiểu của các tham số được gọi là nguyên mẫu hàm.
 - Nguyên mẫu hàm được coi là phong cách C hiện đại và được sử dụng phổ biến hiện nay.
 - Ví dụ: `int f(int, int);` // Nhận 2 tham số int, trả về int.
 - `int f(void);` // Không nhận tham số, trả về int
 - `int f();` // - Không phải nguyên mẫu

Ví dụ 6.2. Định nghĩa và khai báo hàm

Chúng ta tái cấu trúc ví dụ 6.1, chuyển định nghĩa hàm `inch_to_cm` sang 1 đơn vị biên dịch khác.

```
vd6-2a.c
6  #include <stdio.h>
7
8  double inch_to_cm(double x);
9
10 int main() {
11     double x;
12     printf("Nhập độ dài (inch): ");
13     scanf("%lf", &x);
14     printf("%.2f in = %.2f cm\n", x, inch_to_cm(x));
15     return 0;
16 }
```

Khai báo và cũng là nguyên mẫu của hàm `inch_to_cm`

Nguyên mẫu hàm = Khai báo hàm với mô tả kiểu của tham số.

Phát sinh lỗi ở pha ghép nối nếu không liệt kê `vd6-2b.c`

```
bangoc:$gcc -o prog vd6-2a.c vd6-2b.c
bangoc:$./prog
Nhập độ dài (inch): 3
3.00 in = 7.62 cm
bangoc:$
```


```
vd6-2b.c
1  #define INCH1 2.54
2
3  double inch_to_cm(double x) {
4      return x * INCH1;
5  }
```

Hàm được mặc định có liên kết ngoại (extern) và có thể được sử dụng (gắn kết với khai báo) trong đơn vị biên dịch khác.

Tính tương thích của các khai báo

- Do thiết kế tách rời khai báo và định nghĩa, trong 1 chương trình C có thể có nhiều khai báo của 1 hàm, quy chuẩn C không yêu cầu các khai báo của 1 hàm phải giống nhau tuyệt đối, nhưng phải tương thích.
- Khái niệm tương thích của các khai báo là một khái niệm phức tạp với nhiều trường hợp (*tham khảo thêm khi cần*).
- Trong bài giảng này chúng ta xét 1 số trường hợp đơn giản và thường gặp:
 - Các khai báo **giống hệt nhau** là các khai báo tương thích.
 - Ví dụ, sao chép phần khai báo trong định nghĩa hàm.
 - `double inch_to_cm(double x);` // Tên tham số không quan trọng
 - Chúng ta có thể bỏ qua định danh của các tham số.
 - Ví dụ: `double inch_to_cm (double);`
 - *(double inch_to_cm()); Khai báo với danh sách tham số rỗng là phong cách cũ, có thể không được hỗ trợ trong tương lai).*

Nội dung

- Khái niệm hàm
 - Định nghĩa và khai báo hàm
 - Biểu thức gọi hàm
 - Các thuộc tính của biến
- 

Biểu thức gọi hàm

Cấu trúc của biểu thức gọi hàm:

biểu-thức-hàm(danh-sách-đối-số_{opt})

- *biểu-thức-hàm*: Có giá trị là **con trỏ đến hàm** được gọi (có kiểu con trỏ hàm).
 - Trường hợp đơn giản nhất là **tên hàm**, trong biểu thức gọi hàm **được ép kiểu thành con trỏ** tới hàm tương ứng.
 - Chúng ta cũng có thể gọi hàm bằng con trỏ hàm, không bắt buộc phải sử dụng tên hàm.
- Danh-sách-đối-số_{opt}:
 - Bao gồm các biểu thức được sử dụng để khởi tạo giá trị cho các tham số của hàm;
 - Các đối số được truyền theo giá trị.

Biểu thức gọi hàm: Ép kiểu tự động

- *Quy tắc nâng kiểu mặc định cho đối số:* Nếu đối số có kiểu số nguyên thì áp dụng quy tắc nâng kiểu số nguyên, nếu ngược lại và có kiểu **float** thì được chuyển thành **double**.
- Nếu *biểu-thức-hàm* có kiểu gắn với nguyên mẫu hàm và các tham số được khai báo trong nguyên mẫu thì các đối số tương ứng được ép kiểu như đối với phép gán.
 - Nếu nguyên mẫu hàm có số lượng tham số tùy ý (*tương tự như với hàm **printf***) thì *quy tắc nâng kiểu mặc định cho đối số* được áp dụng cho các tham số không được khai báo.
- Nếu *biểu-thức-hàm* có kiểu không gắn với nguyên mẫu hàm thì *quy tắc nâng kiểu mặc định cho đối số* được áp dụng cho các đối số.
 - (Ví dụ: Sử dụng khai báo với danh sách tham số rỗng.)

Biểu thức gọi hàm: Ý nghĩa và kết quả

- Các đối số được truyền cho hàm **theo giá trị**. Thứ tự thực hiện các biểu thức thành phần là không xác định, tuy nhiên có một điểm tuần tự trước khi thân hàm được thực hiện (*sau khi các giá trị được thiết lập cho các tham số*).
- Nếu *biểu-thức-hàm* có kiểu con trỏ tới hàm trả về giá trị thì kết quả biểu thức gọi hàm được xác định theo câu lệnh **return** trong hàm được gọi.
 - Trong trường hợp hàm được gọi kết thúc mà không có câu lệnh **return** nào được thực hiện thì sử dụng kết quả gọi hàm trong trường hợp này là *hành vi bất định*.
- (*Nếu hàm được gọi không tương thích với kiểu được trỏ tới bởi biểu-thức-hàm thì hành vi là bất định.*)

Trong C một hàm có thể gọi chính nó trực tiếp hoặc gián tiếp, các biểu thức gọi hàm như vậy được gọi là gọi hàm đệ quy.

Ví dụ 6.3a. Truyền tham số cho hàm

*Tham số x là biến địa phương
trong phạm vi hàm inc10*

```
vd6-3a.c x
6 #include <stdio.h>
7
8 void inc10(int x) {
9     x += 10;
10    printf("(Trong inc10) x = %d\n", x);
11 }
12
13 int main() {
14     int x = 100;
15     inc10(x);
16     printf("(sau khi gọi inc10) x = %d\n", x);
17     return 0;
18 }
```

Giá trị của x được truyền cho hàm inc10.

```
bangoc:$gcc -o pa vd6-3a.c
bangoc:$./pa
(Trong inc10) x = 110
(sau khi gọi inc10) x = 100
bangoc:$
```

Thực hiện hàm inc10 không làm thay đổi giá trị biến x trong hàm main trong trường hợp này.

Ví dụ 6.3b. Truyền tham số cho hàm

Tham số x là con trỏ, đồng thời vẫn là biến địa phương trong phạm vi hàm inc10

```
vd6-3b.c x
6 #include <stdio.h>
7
8 void inc10(int *x) {
9     *x += 10;
10    printf("(Trong inc10) *x = %d\n", *x);
11 }
12
13 int main() {
14     int x = 100;
15     inc10(&x);
16     printf("(sau khi gọi inc10) x = %d\n", x);
17     return 0;
18 }
```

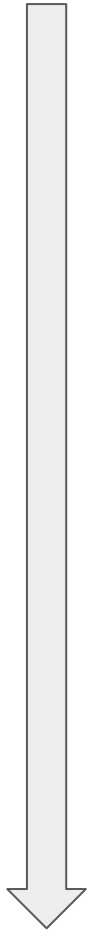
```
bangoc:$gcc -o prog vd6-3b.c
bangoc:$./prog
(Trong inc10) *x = 110
(sau khi gọi inc10) x = 110
bangoc:$
```

Giá trị của biểu thức &x và là địa chỉ của biến x được truyền cho hàm inc10.

Thực hiện hàm inc10 làm thay đổi giá trị của biến x trong trường hợp này.

Thứ tự ưu tiên và chiều thực hiện chuỗi toán tử

Ưu tiên cao
(thực hiện trước)



Ưu tiên thấp
(thực hiện sau)

Tên toán tử	Ký hiệu	Thứ tự
Gọi hàm Tăng 1, giảm 1 (hậu tố)	() f(3, 5) ++, -- (x++, x--)	Trái -> Phải
Tăng, giảm 1 (tiền tố) Toán tử dấu (tiền tố) Phủ định lô-gic Toán tử sizeof	++, -- (++x, --x) +, - (+x, -x) ! (!e) sizeof (sizeof(int))	Phải -> Trái
Ép kiểu	() (double)5	Phải -> Trái
Nhân, Chia Phần dư	*, / (x * y, x / y) % (x % y)	Trái -> Phải
Cộng Trừ	+ (x + y) - (x - y)	Trái -> Phải
Dịch sang trái Dịch sang phải	<< (x << y) >> (x >> y)	Trái -> Phải
Quan hệ so sánh	<, >, <=, >=	Trái -> Phải
Quan hệ bằng	==, !=	Trái -> Phải
AND theo bit	& (x & y)	Trái -> Phải
XOR theo bit	^ (x ^ y)	Trái -> Phải
OR theo bit	(x y)	Trái -> Phải
AND lô-gic	&&	Trái -> Phải
OR lô-gic		Trái -> Phải
Lựa chọn	?:	Phải -> Trái (Rẽ nhánh)
Gán đơn giản Gán kết hợp	= *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	Phải -> Trái

Kiểu hàm

Chúng ta xét 1 số trường hợp đơn giản

- Như đã biết đối với khai báo biến kiểu số:
 - Ví dụ `int x; // Khai báo biến x có kiểu int`
- Đối với khai báo hàm:
 - Ví dụ `int f(); // Khai báo hàm f thuộc kiểu hàm trả về int.`
- Chúng ta có thể định nghĩa kiểu hàm trên cơ sở khai báo hàm bằng từ khóa **typedef**:
 - Ví dụ: `int f(); typedef int F(); // F là kiểu hàm tương đương với kiểu hàm được xác định cho f .`
 - Kiểu hàm thường được định nghĩa để hỗ trợ khai báo con trỏ hàm. Ví dụ, `F *fp;`

Với đặc điểm trả về giá trị, kiểu hàm được coi là 1 kiểu suy diễn từ kiểu dữ liệu của giá trị mà hàm trả về.

Con trỏ hàm

Giả sử `T f(/*...*/);` khai báo một hàm `f`, chúng ta có:

- `T (*pf) (/* ... */);` khai báo `pf` là con trỏ hàm có thể trỏ tới `f`.
 - Chúng ta có thể thiết lập con trỏ `pf` trỏ tới `f` bằng phép gán, ví dụ, `pf = &f;` hoặc đơn giản là `pf = f;` // Ép kiểu tự động.
- **`typedef T ft(/*...*/);`** khai báo `ft` là một kiểu hàm tương đương với kiểu được xác định cho hàm `f`.
- Chúng ta cũng có thể khai báo con trỏ hàm dựa trên kiểu hàm được định nghĩa
 - `ft *pf2;` khai báo `pf2` là con trỏ hàm, `pf2` tương đương với `pf`, và có thể trỏ tới `f`.
 - `pf2 = f;` thiết lập `pf2` cùng trỏ tới `f`.
- Chúng ta có thể sử dụng con trỏ hàm để gọi hàm
 - Ví dụ, `pf(....);` gọi hàm `f` giống như `f(....);`

Ví dụ 6.4. Minh họa con trỏ hàm

```
vd6-4.c
6  #include <stdio.h>
7
8  double average(double a, double b) {
9      return (a + b) / 2;
10 }
11 double max(double a, double b) {
12     return a > b ? a : b;
13 }
14
15 double (*op)(double, double);
16 double (*pf)(int, int) = max;
17 int main() {
18     double a, b;
19     printf("Nhập 2 số thực: ");
20     scanf("%lf%lf", &a, &b);
21     op = average;
22     printf("Trung bình cộng = %lf\n", op(a, b)); OK - Gọi hàm average
23     op = max;
24     printf("Max = %lf\n", op(a, b)); OK - Gọi hàm max
25     printf("Max = %lf\n", pf(a, b));
26     return 0;
27 }
```

bangoc:\$gcc -o prog vd6-4.c
vd6-4.c:16:26: warning: initialization from incompatible pointer type [-Wincompatible-pointer-types]
double (*pf)(int, int) = max;
 ^~~

bangoc:\$./prog
Nhập 2 số thực: 3 8
Trung bình cộng = 5.500000
Max = 8.000000
Max = 3.000000
bangoc:\$

NOK - Hành vi bất định, gọi hàm với khai báo không tương thích

Ví dụ 6.5a. Minh họa gọi đệ quy

Chương trình này gọi đệ quy hàm recursion và in ra giá trị biến n - cho biết chiều sâu đệ quy. Chương trình ngừng hoạt động khi ngăn xếp gọi hàm bị tràn (lỗi Segmentation fault).

```
vd6-5a.c x
6  #include <stdio.h>
7
8  void recursion(long n) {
9      printf("n = %ld\n", n);
10     recursion(n+1);
11 }
12
13 int main() {
14     recursion(1);
15     return 0;
16 }
```

```
n = 261727
n = 261728
n = 261729
n = 261730
n = 261731
Segmentation fault (core dumped)
bangoc:$
```

Kết quả xuất ra màn hình có thể tương tự như trong hình (phụ thuộc vào môi trường thực thi).

Ví dụ 6.5b. Minh họa đệ quy trong chuỗi gọi hàm

Hàm a và hàm b được gọi đệ quy trong chu trình gọi hàm.

```
vd6-5b.c
6  #include <stdio.h>
7
8  void a(long a);
9  void b(long b);
10
11 void a(long v) {
12     printf("v (a) = %ld\n", v);
13     b(v + 1);
14 }
15
16 void b(long v) {
17     printf("v (b) = %ld\n", v);
18     a(v + 1);
19 }
20
21 int main() {
22     a(1);
23     return 0;
24 }
```

```
v (a) = 261875
v (b) = 261876
v (a) = 261877
v (b) = 261878
Segmentation fault (core dumped)
bangoc:$
```

Kết quả xuất ra màn hình có thể tương tự như trong hình (phụ thuộc vào môi trường thực thi).

Chương trình có thể ngừng hoạt động do phát sinh lỗi tràn ngăn xếp bộ nhớ gọi hàm.

Ví dụ 6.6. Hành vi bất định với giá trị trả về

```
bangoc:$gcc -o prog vd6-6.c -Wall
vd6-6.c: In function 'average':
vd6-6.c:11:10: warning: unused variable 'avg' [-Wunused-variable]
    double avg = (a + b) / 2;
           ^~~
vd6-6.c:13:1: warning: control reaches end of non-void function [-Wreturn-type]
}
^
bangoc:$./prog
Nhập 2 số thực: 3 8
Trung bình = 5.500000
```

Trình biên dịch đưa ra cảnh báo & kết quả thu được đúng 1 cách bí ẩn trong trường hợp này.

```
vd6-6.c
8 #include <stdio.h>
9
10 double average(double a, double b) {
11     double avg = (a + b) / 2;
12     /* Quên return avg; */
13 }
14
15 int main() {
16     double x, y;
17     printf("Nhập 2 số thực: ");
18     scanf("%lf%lf", &x, &y);
19     printf("Trung bình = %f\n", (x + y)/2);
20     return 0;
21 }
```

Nội dung

- Khái niệm hàm
- Định nghĩa và khai báo hàm
- Biểu thức gọi hàm
- Các thuộc tính của biến

Giới thiệu

Như đã học trong **Bài 2**, có thể coi biến như một vùng nhớ được đặt tên và định kiểu.

- Tính đến thời điểm này chúng ta đã khai báo biến theo định dạng cơ bản
 - *Kiểu-dữ-liệu* tên-biến;
 - ví dụ, `int x; // Khai báo biến x có kiểu int`
 - Hoặc *kiểu-dữ-liệu* tên-biến = *biểu-thức*;
 - ví dụ, `int x = 101; // Khai báo biến x có kiểu int và khởi tạo = 101`
- Theo yêu cầu phát triển phần mềm, định dạng khai báo biến cơ bản có thể được mở rộng với các thuộc tính. Trong bài giảng này chúng ta sẽ học về:
 - Liên kết;
 - Phân lớp lưu trữ & thời gian lưu trữ;
 - Giới hạn đọc/ghi.

Thuộc tính liên kết của biến

- Biến có thể không có liên kết, hoặc có liên kết ngoại hoặc có liên kết nội (các trường hợp này loại trừ lẫn nhau). Liên kết được xác định mặc định theo vị trí khai báo hoặc theo đặc tả khai báo:
 - Trong phạm vi tệp mặc định có liên kết ngoại;
 - Trong phạm vi tệp với từ khóa **static** có liên kết nội;
 - Trong phạm vi khối mặc định không có liên kết;
 - Trong phạm vi khối với từ khóa **extern** có liên kết ngoại.
- Tương tự như hàm, biến cũng có thể có liên kết ngoại, vì vậy có thể được khai báo nhiều lần và có thể sử dụng 1 định nghĩa biến trong nhiều đơn vị biên dịch khác nhau.
 - Liên kết là 1 cơ sở để xác định các khai báo có phải là của cùng 1 biến hay không.
 - *(Tham khảo thêm về các quy tắc phân giải)*

Ví dụ 6.7. Liên kết của biến

```
vd6-7a.c
6 #include <stdio.h>
7
8 int x; // Khai báo
9 int z = 3; // Định nghĩa
10 void inc_all();
11 int main() {
12     extern int y; // Khai báo
13     printf("%d %d %d\n", x, y, z);
14     inc_all();
15     printf("%d %d %d\n", x, y, z);
16     return 0;
17 }
```

```
bangoc:$gcc -o prog vd6-7a.c vd6-7b.c
bangoc:$./prog
101 102 3
111 112 3
bangoc:$
```

x được khai báo trong phạm vi tệp có liên kết ngoại, được liên kết với x trong đơn vị biên dịch vd6-7b.c, có giá trị khởi tạo = 101.

z có liên kết ngoại nhưng không được liên kết với z trong vd6-8b.c (có phạm vi liên kết nội).

```
vd6-7b.c
1 int x = 101; // Định nghĩa
2 int y = 102;
3 static int z = 103;
4 void inc_all() {
5     x += 10; y += 10; z += 10;
6 }
```

y được khai báo trong phạm vi khối nhưng có liên kết ngoại (do extern) và được liên kết với y trong vd6-8b.c, khởi tạo = 102.

Sẽ phát sinh lỗi biên dịch (định nghĩa lại) nếu z có liên kết ngoại.

NNLT C có phạm vi đóng gói là đơn vị biên dịch. Nên hạn chế sử dụng biến có phạm vi tệp (nếu có thể).

Phân lớp lưu trữ & thời gian lưu trữ

- Thời gian lưu trữ: Khoảng thời gian bộ nhớ được cấp phát và duy trì cho biến, được xác định theo phân lớp lưu trữ.
 - Các biến có liên kết (ngoại hoặc nội): Được khởi tạo 1 lần duy nhất trước khi chương trình bắt đầu được thực hiện và được lưu trong suốt thời gian thực hiện chương trình. Chúng ta gọi mô hình lưu trữ này là phân lớp lưu trữ **tĩnh/static**.
 - Biến không có liên kết: Mặc định được khởi tạo mỗi khi khối chứa nó được thực hiện và chỉ tồn tại trong thời gian khối được thực hiện, bộ nhớ được cấp phát cho biến được giải phóng tự động ngay sau khi kết thúc khối. Chúng ta gọi mô hình lưu trữ này là phân lớp lưu trữ **tự động/auto**.
 - Biến không có liên kết được khai báo với từ khóa **static** thuộc phân lớp lưu trữ tĩnh.
 - Trong C bộ nhớ động do người dùng tự quản lý, có thể yêu cầu cấp phát/giải phóng ở bất kỳ thời điểm nào (*học sau*).

Ví dụ 6.8. Lưu trữ cố định và lưu trữ tự động

```
vd6-8.c
6  #include <stdio.h>
7
8  int x;
9  static int y;
10 void f1() {
11     static int z = 10;
12     printf("z (f1) = %d\n", z++);
13 }
14 void f2() {
15     int z = 10;
16     printf("z (f2) = %d\n", z++);
17 }
18 int main() {
19     for (int i = 0; i < 3; ++i) {
20         f1();
21         f2();
22     }
23     return 0;
24 }
```

x, y, và z (do static) thuộc phân lớp lưu trữ cố định.

Trong đó x có liên kết ngoại, y có liên kết nội và z không có liên kết.

z trong f1 chỉ được khởi tạo 1 lần và tồn tại trong suốt thời gian chương trình hoạt động

z trong f2 thuộc phân lớp lưu trữ tự động, được khởi tạo mỗi khi hàm f2 được thực hiện và chỉ tồn tại trong thời gian hàm f2 được thực hiện.

giá trị z trong f1 được lưu lại, còn giá trị z trong f2 thì không

```
bangoc:$gcc -o prog vd6-8.c
bangoc:$./prog
z (f1) = 10
z (f2) = 10
z (f1) = 11
z (f2) = 10
z (f1) = 12
z (f2) = 10
bangoc:$
```

Giới hạn đọc/ghi kiểu

Thông thường chúng ta có thể ghi/lưu giá trị và đọc giá trị với biến được khai báo mặc định.

- Trong 1 số trường hợp (do cách triển khai hoặc tối ưu hóa) chúng ta cần giới hạn các đặc điểm đọc/ghi với biến (thường thay đổi lô-gic ghi giá trị).
- Được thực hiện bằng cách bổ xung thuộc tính cho kiểu. Điển hình như bổ xung thuộc tính **const** cho kiểu của biến.
- *Tham khảo thêm về **volatile**, **restrict***
 - volatile mô tả vùng nhớ có thể thay đổi giá trị, thường được sử dụng cho lập trình ở tầng thấp (ví dụ: `const volatile int x`).
 - restrict (C99) chỉ được sử dụng với con trỏ và thường được sử dụng cho các mục đích tối ưu hóa một số trường hợp xử lý với con trỏ.
 - (không học trong học phần này)

Biến chỉ đọc

- Nếu `T x;` khai báo biến `x` (mặc định đọc và ghi) thì `const T x;` khai báo `x` là một biến chỉ đọc (read only).
 - Ví dụ: `int x;` // có thể đọc và ghi với `x`
 - `const int x;` // chỉ có thể đọc giá trị với `x`
 - Biến chỉ đọc có địa chỉ và đầy đủ các thuộc tính liên kết, thời gian lưu trữ và phạm vi tương tự như biến thường.
- Thuộc tính `const` có ý nghĩa giống như ghi chú người lập trình sẽ không thay đổi giá trị của biến sau khi khởi tạo.
 - Không thể gán giá trị cho biến chỉ đọc (gán giá trị cho biến chỉ đọc là lỗi biên dịch). Vì vậy biến chỉ đọc phải được khởi tạo trong định nghĩa, ví dụ: `const int x = 101;`
 - Gián tiếp thay đổi giá trị của biến chỉ đọc, ví dụ qua con trỏ là hành vi bất định.
 - Biến chỉ đọc có thể được tạo trong phân vùng nhớ chỉ đọc,
 - hoặc trong phân vùng nhớ có thể thay đổi giá trị.

Ví dụ 6.10. Biến chỉ đọc

```
vd6-10.c
6 #include <stdio.h>
7
8 const int x = 100;
9 int main() {
10     // x = 200; // Lỗi biên dịch
11     const int y = 200;
12     int *py = (int*)&y;
13     *py = 202; // (Có thể) y == 202
14     printf("y = %d\n", y);
15     int *px = (int*)&x;
16     *px = 101; // Có thể lỗi
17     return 0;
18 }
```

Phát sinh lỗi khi thực hiện nếu x được cấp phát trong phân vùng nhớ chỉ đọc.

```
bangoc:$gcc -o prog vd6-10.c
bangoc:$./prog
y = 202
Segmentation fault (core dumped)
bangoc:$
```

x có liên kết ngoại và thuộc phân lớp lưu trữ cố định, vì vậy thường được cấp phát trong phân vùng bộ nhớ chỉ đọc.

y không có liên kết và thuộc phân lớp lưu trữ tự động, vì vậy thường được cấp phát trong cùng phân vùng bộ nhớ có thể thay đổi.

Có thể lấy địa chỉ của biến chỉ đọc.

*Thay đổi giá trị của biến chỉ đọc là **hành vi bất định**: Có thể dẫn đến giá trị của đối tượng được cập nhật, hoặc không được cập nhật (bỏ qua khi tối ưu hóa), hoặc sự cố (core dump/crash) v.v.*

Đặt tên cho hằng giá trị

- Đặt tên cho hằng giá trị và sử dụng tên trong các mô tả có thể làm mã nguồn dễ đọc và dễ bảo trì hơn.
 - Ví dụ đối với số π , chúng ta có thể sử dụng tên PI thay cho giá trị 3.1415... trong các biểu thức.
- Chúng ta có 2 lựa chọn: Sử dụng macro thay thế cho hằng giá trị hoặc sử dụng biến chỉ đọc để lưu hằng giá trị. Lựa chọn macro hay biến chỉ đọc thường là sự sáng tạo của người lập trình với một số lưu ý:
 - Macro được thay thế bằng giá trị thực ở pha biên dịch.
 - Biến chỉ đọc được cấp phát bộ nhớ, được khởi tạo khi chạy chương trình, và có thể có các thuộc tính giống như biến.
 - Macro thường được sử dụng cho các hằng giá trị số, ví dụ như: kích thước mảng tĩnh, độ dài chuỗi ký tự, số π , v.v.
 - Chúng ta có thể tạo biến chỉ đọc của bất kỳ kiểu nào.

Ví dụ 6.11. Đặt tên hằng

```
vd6-11.c x
6 #include <stdio.h>
7
8 #define PI 3.14159265358979323846
9 const char *colors[] = {"red", "black"};
10 int main() {
11     double r = 1.0;
12     printf("S = %f\n", PI * r * r);
13     int c = 0;
14     printf("Color = %s\n", colors[c]);
15     return 0;
16 }
```

```
bangoc:$gcc -o prog vd6-11.c
bangoc:$./prog
S = 3.141593
Color = red
bangoc:$
```

PI là macro, trong dòng 12 được thay thế bằng giá trị 3.141... trong bước tiền xử lý.

colors là mảng của hằng chuỗi ký tự (*chi tiết được cung cấp sau*).

Khởi tạo giá trị cho biến

- Các biến thuộc phân lớp lưu trữ tĩnh được mặc định khởi tạo giá trị = 0 tương ứng với kiểu của biến.
- Các biến thuộc phân lớp lưu trữ tự động không được khởi tạo giá trị mặc định, có thể nhận bất kỳ giá trị nào như đang có trong vùng nhớ được cấp phát.
- Chúng ta có thể chủ động khởi tạo giá trị cho biến:
 - Mỗi biến kiểu số có thể được khởi tạo giá trị bằng giá trị của 1 biểu thức khởi tạo (sau khi ép kiểu nếu cần).
 - *(Chi tiết về cách khởi tạo biến thuộc các kiểu dữ liệu khác sẽ được cung cấp sau cùng với kiểu dữ liệu).*
- Trong phạm vi tệp một biến có thể được khai báo nhiều lần (*nếu các khai báo tương thích*), nhưng chỉ được có tối đa 1 khai báo có khởi tạo giá trị và đó đồng thời cũng là định nghĩa của biến.

Ví dụ 6.12. Khởi tạo giá trị cho biến

```
vd6-12.c x
6  #include <stdio.h>
7
8  int x; // Được khởi tạo = 0
9  int main() {
10     int y; // Không được khởi tạo
11     do {
12         static int z; // = 0
13         printf("Trước: x = %d, "
14             "y = %d z = %d\n",
15             x, y, z);
16         scanf("%d%d%d", &x, &y, &z);
17         int sum = 0;
18         sum = x + y + z;
19         printf("Sum = %d\n", sum);
20     } while (0);
21     return 0;
22 }
```

Biến x và z (do static) thuộc phân lớp lưu trữ tĩnh vì vậy được mặc định khởi tạo giá trị = 0;

Biến y thuộc phân lớp lưu trữ tự động vì vậy không được khởi tạo giá trị mặc định.

Biến sum thuộc phân lớp lưu trữ tự động và được chủ động khởi tạo = 0.

```
bangoc:$gcc -o prog vd6-12.c
bangoc:$./prog
Trước: x = 0, y = -452390240 z = 0
1 2 3
Sum = 6
bangoc:$
```

