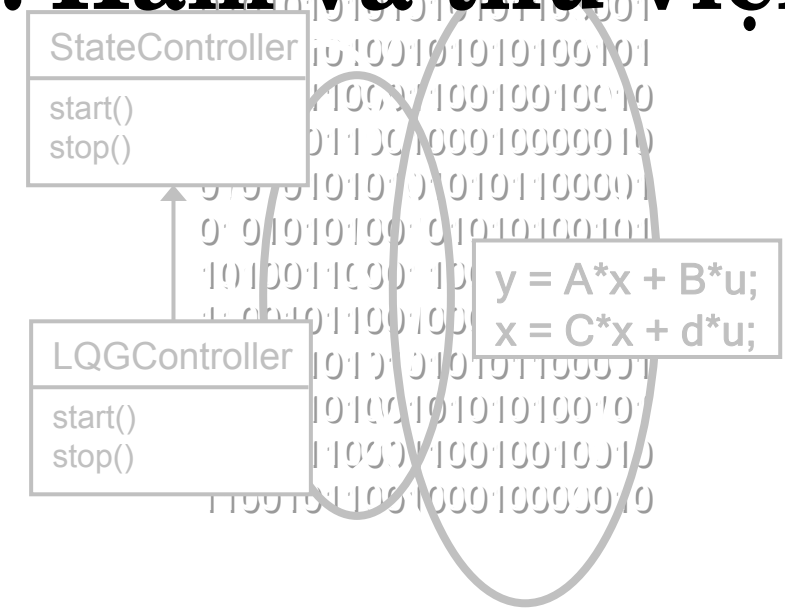
Kỹ thuật lập trình

Phần II: Lập trình có cấu trúc

Chương 3: Hàm và thư viện



Nội dung chương 3

3.1 Hàm và lập trình hướng hàm

3.2 Khai báo, định nghĩa hàm

3.3 Truyền tham số và trả về kết quả

3.4 Thiết kế hàm và thư viện

3.5 Thư viện chuẩn ANSI-C

3.6 Làm việc với tệp tin sử dụng thư viện C++

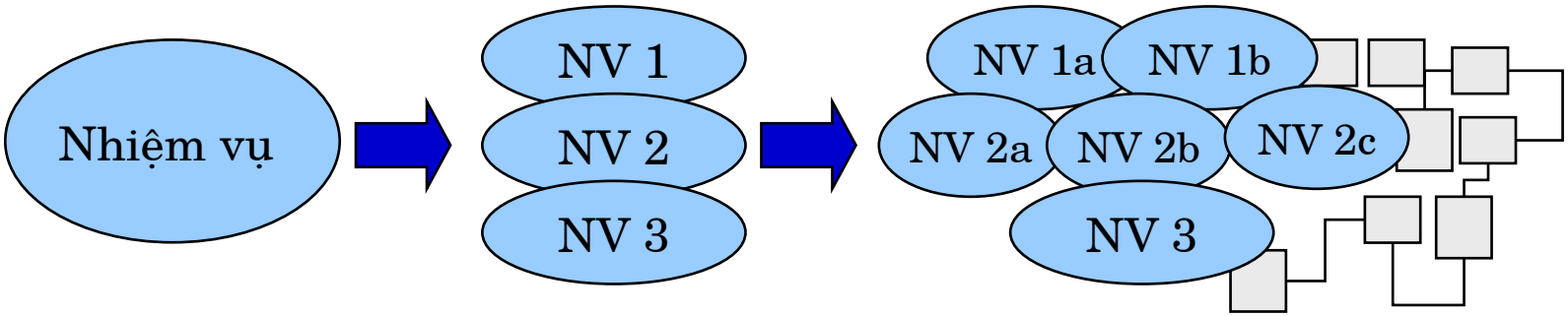
3.7 Nạp chồng tên hàm C++

3.8 Hàm inline trong C++

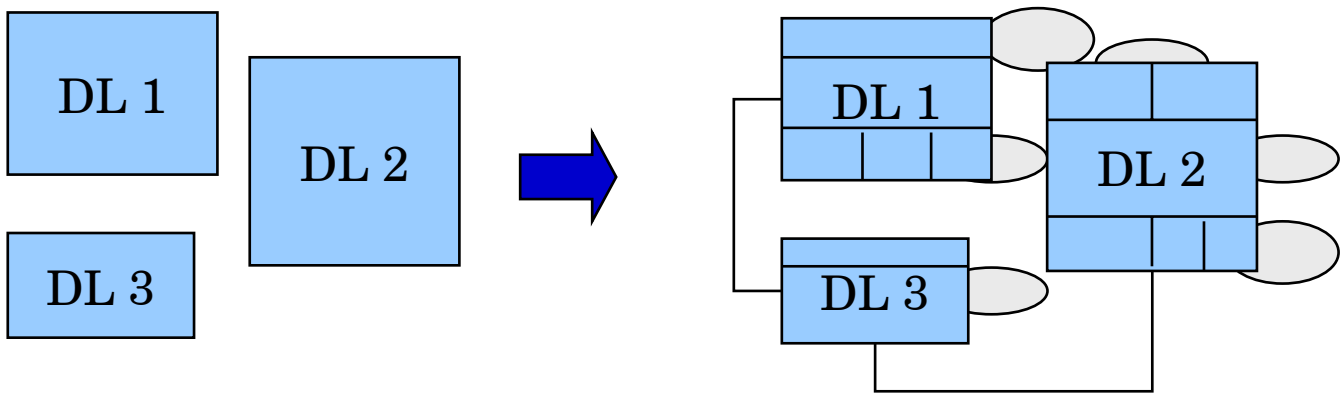
3.1 Hàm và lập trình hướng hàm

Lập trình có cấu trúc có thể dựa trên một trong hai phương pháp:

- Lập trình hướng hàm (*function-oriented*), còn gọi là hướng nhiệm vụ (*task-oriented*), hướng thủ tục (*procedure-oriented*)



- Lập trình hướng dữ liệu (*data-oriented*)



Hàm là gì?

- Tiếng Anh: function -> hàm, chức năng
- Một đơn vị tổ chức chương trình, một đoạn mã chương trình có cấu trúc để thực hiện một **chức năng** nhất định, có **giá trị sử dụng lại**
- Các hàm có quan hệ với nhau thông qua lời gọi, các biến tham số (đầu vào, đầu ra) và giá trị trả về
- Cách thực hiện cụ thể một hàm phụ thuộc nhiều vào dữ kiện (tham số, đối số của hàm):
  - Thông thường, kết quả thực hiện hàm mỗi lần đều giống nhau nếu các tham số đầu vào như nhau
  - Một hàm không có tham số thì giá trị sử dụng lại rất thấp
- Trong C/C++: Không phân biệt giữa thủ tục và hàm, cả đoạn mã chương trình chính cũng là hàm

Ví dụ phân tích

- Yêu cầu bài toán: Tính tổng một dãy số nguyên (liên tục) trong phạm vi do người sử dụng nhập. In kết quả ra màn hình.
- Các nhiệm vụ:
  - Nhập số nguyên thứ nhất:
    - Yêu cầu người sử dụng nhập
    - Nhập số vào một biến
  - Nhập số nguyên thứ hai
    - Yêu cầu người sử dụng nhập
    - Nhập số vào một biến
  - Tính tổng với vòng lặp
  - Hiển thị kết quả ra màn hình

Phương án 4 trong 1

```
#include <iostream.h>
void main() {
   int a, b;
   char c;
   do {
      cout << "Enter the first integer number: ";
      cin >> a;
      cout << "Enter the second integer number: ";
      cin >> b;
      int Total = 0;
      for (int i = a; i <= b; ++i)
        Total += i;
      cout << "The sum from " << a << " to " << b
           << " is " << Total << endl;
      cout << "Do you want to continue? (Y/N):";
      cin  >> c;
   } while (c == 'y' || c == 'Y');
}
```

Phương án phân hoạch hàm (1)

```
#include <iostream.h>

int  ReadInt();
int  SumInt(int,int);
void WriteResult(int a, int b, int kq);

void main() {
   char c;
   do {
      int a = ReadInt();
      int b = ReadInt();
      int T = SumInt(a,b);
      WriteResult(a,b,T);
      cout << "Do you want to continue? (Y/N):";
      cin  >> c;
   } while (c == 'y' || c == 'Y');
}
```

Phương án phân hoạch hàm (1)

```
int ReadInt() {
   cout << "Enter an integer number: ";
   int N;
   cin >> N;
   return N;
}

int SumInt(int a, int b) {
   int Total = 0;
   for (int i = a; i <= b; ++i)
      Total += i;
   return Total;
}

void WriteResult(int a, int b, int kq) {
   cout << "The sum from " << a << " to " << b
        << " is " << kq << endl;
}
```

Không có tham số,
Giá trị sử dụng lại?

OK,
Không thể tốt hơn!

Quá nhiều tham số,
Hiệu năng?

Phương án phân hoạch hàm (1)

- Chương trình dễ đọc hơn => dễ phát hiện lỗi
- Chương trình dễ mở rộng hơn
- Hàm SumInt có thể sử dụng lại tốt
- Mã nguồn dài hơn
- Mã chạy lớn hơn
- Chạy chậm hơn

➲ Không phải cứ phân hoạch thành nhiều hàm là tốt, mà vấn đề nằm ở cách phân hoạch và thiết kế hàm làm sao cho **tối ưu**!

Phương án phân hoạch hàm (2)

```
#include <iostream.h>

int  ReadInt(const char*);
int  SumInt(int,int);

void main() {
   char c;
   do {
      int a = ReadInt("Enter the first integer number :");
      int b = ReadInt("Enter the second integer number:");
      cout << "The sum from " << a << " to " << b
           << " is " << SumInt(a,b) << endl;
      cout << "Do you want to continue? (Y/N):";
      cin  >> c;
   } while (c == 'y' || c == 'Y');
}
```

Phương án phân hoạch hàm (2)

```
int ReadInt(const char* userPrompt) {
   cout << userPrompt;
   int N;
   cin >> N;
   return N;
}
int SumInt(int a, int b) {
   int Total = 0;
   for (int i = a; i <= b; ++i)
      Total += i;
   return Total;
}
```

OK,
Đã tốt hơn!

3.2 Khai báo và định nghĩa hàm

- Định nghĩa hàm: tạo mã thực thi hàm

Kiểu trả về Tên hàm Tham biến (hình thức)

```
int SumInt(int a, int b) {
    int Total = 0;
    for (int i = a; i <= b; ++i)
        Total += i;
    return Total;
}
```

- Khai báo hàm thuần túy: không tạo mã hàm

```
int SumInt(int a, int b);
```

Kiểu trả về Tên hàm Kiểu tham biến

- Tại sao và khi nào cần khai báo hàm?

Khai báo hàm và lời gọi hàm

- Ý nghĩa của khai báo hàm:
  - Khi cần sử dụng hàm (gọi hàm)
  - Trình biên dịch cần lời khai báo hàm để kiểm tra lời gọi hàm đúng hay sai về cú pháp, về số lượng các tham số, kiểu các tham số và cách sử dụng giá trị trả về.

    ```
    int SumInt(int a, int b);
    ```
  - Có thể khai báo hàm độc lập với việc định nghĩa hàm (tất nhiên phải đảm bảo nhất quán)
- Gọi hàm: yêu cầu thực thi mã hàm với tham số thực tế (tham trị)

```
int x = 5;
int k = SumInt(x, 10);
```

Tên hàm Tham số (gọi hàm)

Khi biên dịch chưa cần phải có định nghĩa hàm, nhưng phải có khai báo hàm!

Khai báo hàm C/C++ ở đâu?

- Ở phạm vi toàn cục (ngoài bất cứ hàm nào)
- Một hàm phải được khai báo trước lời gọi đầu tiên trong một tệp tin mã nguồn
- Nếu sử dụng nhiều hàm thì sẽ cần rất nhiều dòng mã khai báo (mất công viết, dễ sai và mã chương trình lớn lên?):
  - Nếu người xây dựng hàm (định nghĩa hàm) đưa sẵn tất cả phần khai báo vào trong một tệp tin => **Header file** (*.h, *.hx,...) thì người sử dụng chỉ cần bổ sung dòng lệnh `#include <filename>`
  - Mã chương trình không lớn lên, bởi khai báo không sinh mã!
- Một hàm có thể khai báo nhiều lần tùy ý!

Định nghĩa hàm ở đâu?

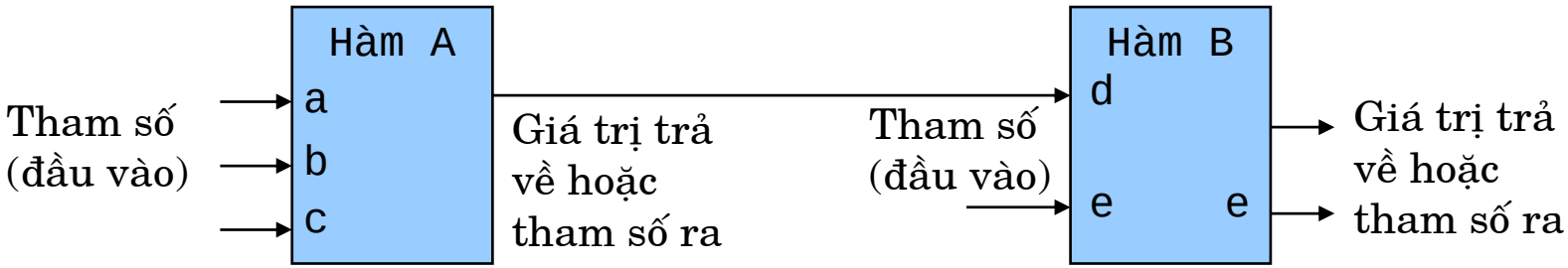
- Ở phạm vi toàn cục (ngoài bất cứ hàm nào)
- Có thể định nghĩa trong cùng tệp tin với mã chương trình chính, hoặc tách ra một tệp tin riêng. Trong Visual C++:

  `*.c` => C compiler,

  `*.cpp` => C++ compiler
- Một hàm đã có lời gọi thì phải được định nghĩa chính xác 1 lần trong toàn bộ (dự án) chương trình, trước khi gọi trình liên kết (lệnh Build trong Visual C++)
- Đưa tệp tin mã nguồn vào dự án, không nên:
  `#include "xxx.cpp"`
- Một hàm có được định nghĩa bằng C, C++, hợp ngữ hoặc bằng một ngôn ngữ khác và dùng trong C/C++ => Sử dụng hàm không cần mã nguồn!
- Một thư viện cho C/C++ bao gồm:
  - Header file (thường đuôi `*.h`, `*.hxx`, ..., nhưng không bắt buộc)
  - Tệp tin mã nguồn (`*.c`, `*.cpp`, `*.cxx`, ...) hoặc mã đích (`*.obj`, `*.o`, `*.lib`, `*.dll`, ...)

3.3 Truyền tham số và trả về kết quả

- Truyền tham số và trả về kết quả là phương pháp cơ bản để tổ chức quan hệ giữa các hàm (giữa các chức năng trong hệ thống)



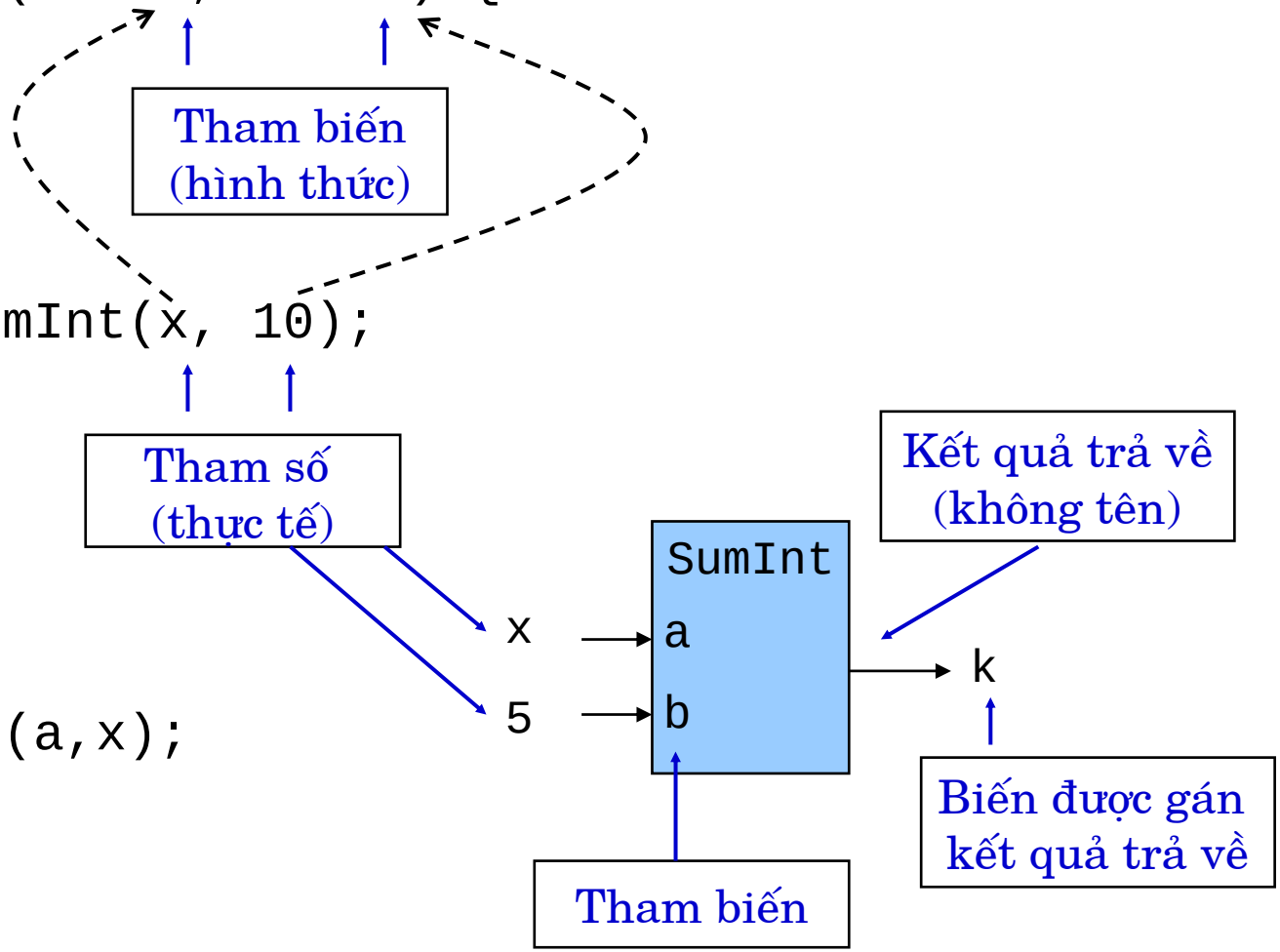
- Ngoài ra, còn có các cách khác:
  - Sử dụng biến toàn cục: nói chung là không nên!
  - Sử dụng các tệp tin, streams: dù sao vẫn phải sử dụng tham số để nói rõ tệp tin nào, streams nào
  - Các cơ chế giao tiếp hệ thống khác (phụ thuộc vào hệ điều hành, nền tảng và giao thức truyền thông) => nói chung vẫn cần các tham số bổ sung
- Truyền tham số & trả về kết quả là một vấn đề cốt lõi trong xây dựng và sử dụng hàm, một trong những yếu tố ảnh hưởng quyết định tới chất lượng phần mềm!

Tham biến hình thức và tham số thực tế

```
int SumInt(int a, int b) {
    ...
}

int x = 5;
int k = SumInt(x, 10);
...

int a = 2;
k = SumInt(a,x);
```



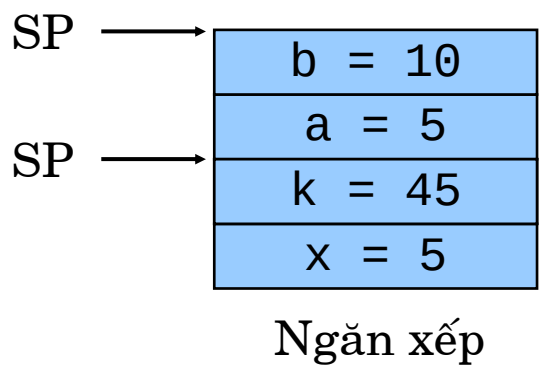
3.3.1 Truyền giá trị

```
int SumInt(int, int);

// Function call
void main() {
    int x = 5;
    int k = SumInt(x, 10);
    ...
}

// Function definition
int SumInt(int a, int b) {
    ...


}
```



Thử ví dụ đọc từ bàn phím

```
#include <iostream.h>
void ReadInt(const char* userPrompt, int N) {
   cout << userPrompt;
   cin >> N;
}

void main() {
   int x = 5;
   ReadInt("Input an integer number:", x);
   cout << "Now x is " << x;
   ...
}
```

- Kết quả: x không hề thay đổi sau đó.

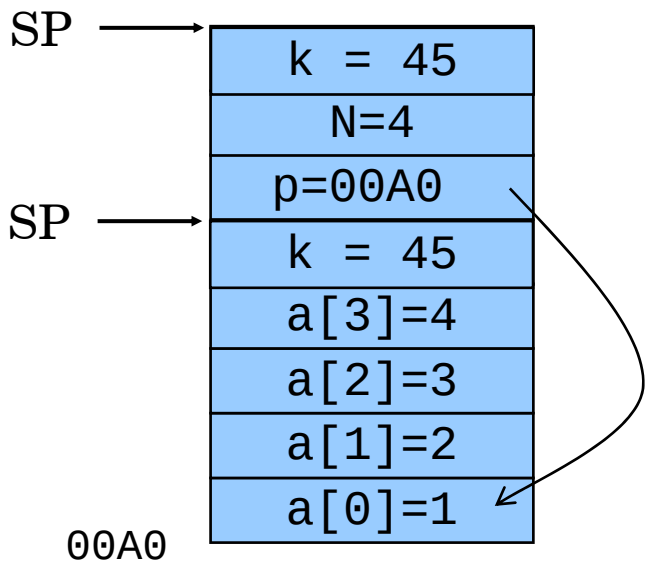
Truyền giá trị

- Truyền giá trị là cách thông thường trong C
- Tham biến chỉ nhận được bản sao của biến đầu vào (tham số thực tế)
- Thay đổi tham biến chỉ làm thay đổi vùng nhớ cục bộ, không làm thay đổi biến đầu vào
- Tham biến chỉ có thể mang tham số đầu vào, không chứa được kết quả (tham số ra)
- Truyền giá trị khá an toàn, tránh được một số hiệu ứng phụ
- Truyền giá trị trong nhiều trường hợp kém hiệu quả do mất công sao chép dữ liệu

3.3.2 Truyền địa chỉ

```
int SumInt(int* p, int N);
    // Function call
void main() {
    int a[] = {1, 2, 3, 4};
    int k = SumInt(a,4);
    ...
}

// Function definition
int SumInt(int* p, int N) {
    int *p2 = p + N, k = 0;
    while (p < p2)
        k += *p++;
    return k;
}
```



Truyền mảng tham số?

```
int SumInt(int p[4], int N);
    // Function call
void main() {
    int a[] = {1, 2, 3, 4};
    int k = SumInt(a,4);
    ...
}

// Function definition
int SumInt(int p[4], int N) {
    int *p2 = p + N, k = 0;
    while (p < p2)
          k += *p++;
    return k;
}
```

Bản chất như trong ví dụ trước: Truyền địa chỉ!

Thử lại ví dụ đọc từ bàn phím

```
#include <iostream.h>
void ReadInt(const char* userPrompt, int* pN) {
   cout << userPrompt;
   cin >> *pN;
}

void main() {
   int x = 5;
   ReadInt("Input an integer number:", &x);
   cout << "Now x is " << x;
   ...
}
```

- Kết quả: x thay đổi giá trị sau đó (cũng là lý do tại sao hàm scanf() lại yêu cầu kiểu tham biến là con trỏ!)

Khi nào sử dụng truyền địa chỉ?

- Khi cần thay đổi "biến đầu vào" (truy nhập trực tiếp vào ô nhớ, không qua bản sao)
- Khi kích cỡ kiểu dữ liệu lớn => tránh sao chép dữ liệu vào ngăn xếp
- Truyền tham số là một mảng => bắt buộc truyền địa chỉ

- Lưu ý: Sử dụng con trỏ để truyền địa chỉ của vùng nhớ dữ liệu đầu vào. Bản thân con trỏ có thể thay đổi được trong hàm nhưng địa chỉ vùng nhớ không thay đổi (nội dung của vùng nhớ đó thay đổi được): xem ví dụ biến p trong hàm SumInt trang 21.

3.3.3 Truyền tham chiếu (C++)

```
#include <iostream.h>
void ReadInt(const char* userPrompt, int& N) {
   cout << userPrompt;
   cin >> N;
}

void main() {
   int x = 5;
   ReadInt("Input an integer number:", x);
   cout << "Now x is " << x;
   ...
}
```

- Kết quả: x thay đổi giá trị sau đó

Thử ví dụ hàm swap

```
#include <iostream.h>
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}

void main() {
   int x = 5, y = 10;
   swap(x,y);
   cout << "Now x is " << x << ", y is " << y;
   ...
}
```

Khi nào sử dụng truyền tham chiếu?

- Chỉ trong C++
- Khi cần thay đổi "biến đầu vào" (truy nhập trực tiếp vào ô nhớ, không qua bản sao)
- Một tham biến tham chiếu có thể đóng vai trò là đầu ra (chứa kết quả), hoặc có thể vừa là đầu vào và đầu ra
- Khi kích cỡ kiểu dữ liệu lớn => tránh sao chép dữ liệu vào ngăn xếp, ví dụ:

```
void copyData(const Student& sv1, Student& sv2) {
    sv2.birthday = sv1.birthday;
    ...
}
```

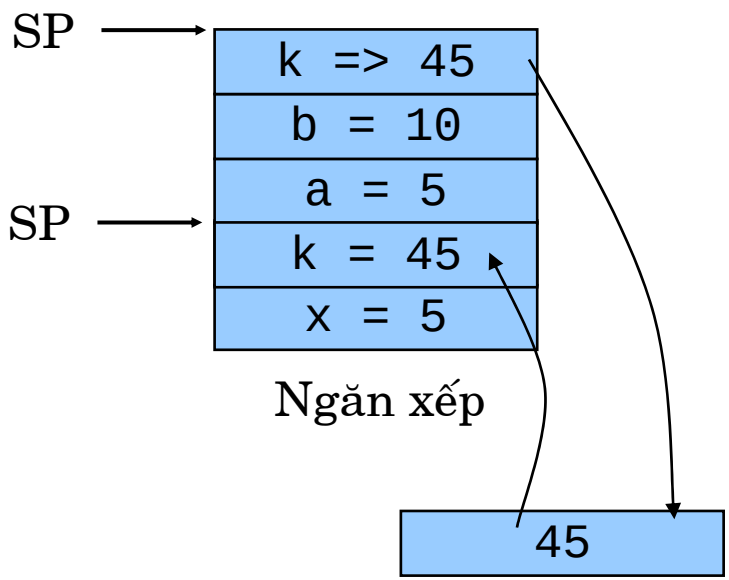
3.3.4 Kiểu trả về

- Kiểu trả về: gần như tùy ý, chỉ không thể trả về trực tiếp một mảng
- Về nguyên tắc, có thể trả về kiểu:
  - Giá trị
  - Con trỏ
  - Tham chiếu
- Tuy nhiên, cần rất thận trọng với trả về địa chỉ hoặc tham chiếu:
  - Không bao giờ trả về con trỏ hoặc tham chiếu vào biến cục bộ
  - Không bao giờ trả về con trỏ hoặc tham chiếu vào tham biến truyền qua giá trị
- Với người lập trình ít có kinh nghiệm: chỉ nên trả về kiểu giá trị

Cơ chế trả về

```
int SumInt(int a, int b) {
    int k = 0;
    for (int i=a; i <= b; ++i)
          k +=i;
    return k;
}

void main() {
    int x = 5, k = 0;
    k = SumInt(x,10);
    ...
}
```



Trả về con trỏ

- Viết hàm trả về địa chỉ của phần tử lớn nhất trong một mảng:

```
int* FindMax(int* p, int n) {
   int *pMax = p;
   int *p2 = p + n;
   while (p < p2) {
       if (*p > *pMax)
              pMax = p;
       ++p;
   }
   return pMax;
}
void main() {
   int s[5] = { 1, 2, 3, 4, 5};
   int *p = FindMax(s,5);
}
```

Lý do trả về con trỏ hoặc tham chiếu

- Tương tự như lý do truyền địa chỉ hoặc truyền tham chiếu:
  - Tránh sao chép dữ liệu lớn không cần thiết
  - Để có thể truy cập trực tiếp và thay đổi giá trị đầu ra
- Có thể trả về con trỏ hoặc tham chiếu vào đâu?
  - Vào biến toàn cục
  - Vào tham số truyền cho hàm qua địa chỉ hoặc qua tham chiếu
  - Nói chung: vào vùng nhớ mà còn tiếp tục tồn tại sau khi kết thúc hàm
- Con trỏ lại phức tạp thêm một chút?

Phản ví dụ: trả về con trỏ

```
int* f(int* p, int n) {
   int Max = *p;
   int *p2 = p + n;
   while (p < p2) {
       if (*p > Max)
             Max = *p;
       ++p;
   }
   return &Max;
}
void main() {
   int s[5] = { 1, 2, 3, 4, 5};
   int *p = FindMax(s,5);  // get invalid address
}
```

Các ví dụ nghiên cứu: Đúng / sai?

```
int* f1(int a) {
   ...
   return &a;
}
int& f2(int &a) {
   ...
   return a;
}
int f3(int &a) {
   ...
   return a;
}
int* f4(int *pa) {
   ...
   return pa;
}
```

```
int f5(int *pa) {
   ...
   return *pa;
}
int& f6(int *pa) {
   ...
   return *pa;
}
int& f7(int a) {
   ...
   return a;
}
int *pa;
int* f8() {
   ...
   return pa;
}
```

3.4 Thiết kế hàm và thư viện

- Viết một chương trình chạy tốt đã khó, viết một thư viện hàm tốt còn khó hơn!
- Một thư viện hàm định nghĩa:
  - một tập hợp các hàm (có liên quan theo một chủ đề chức năng)
  - những kiểu dữ liệu sử dụng trong các hàm
  - một số biến toàn cục (rất hạn chế)
- Một thư viện hàm tốt cần phải:
  - Thực hiện những chức năng hữu ích
  - Đơn giản, dễ sử dụng
  - Hiệu suất và độ tin cậy cao
  - Trọn vẹn, nhất quán và đồng bộ

Thiết kế hàm

- Phân tích yêu cầu:
  - Làm rõ các dữ kiện (đầu vào) và kết quả (đầu ra)
  - Tìm ra các chức năng cần thực hiện
- Đặt tên hàm: ngắn gọn, ý nghĩa xác đáng, tự miêu tả
  - Hàm chỉ hành động: Chọn tên hàm là một động từ kết hợp với kiểu đối tượng chủ thể, ví dụ `printVector, displayMatrix, addComplex, sortEventQueue, filterAnalogSignal`,...
  - Hàm truy nhập thuộc tính: Có thể chọn là động từ hoặc danh từ kết hợp kiểu đối tượng chủ thể, ví dụ length, size, numberOfColums, getMatrixElem, putShapeColor
  - Trong C++ nhiều hàm có thể giống tên (nạp chồng tên hàm), có thể chọn tên ngắn, ví dụ sort, print, display, add, putColor, getColor => nguyên tắc đa hình/đa xạ theo quan điểm hướng đối tượng
  - Trong C++ còn có thể định nghĩa hàm toán tử để có thể sử dụng các ký hiệu toán tử định nghĩa sẵn như *, /, +, - thay cho lời gọi hàm.

- Chọn tham số đầu vào (=> tham biến)
  - Đặc tả ý nghĩa: Thể hiện rõ vai trò tham số
  - Đặt tên: Ngắn gọn, tự mô tả
  - Chọn kiểu: Kiểu nhỏ nhất mà đủ biểu diễn
  - Chọn cách truyền tham số: cân nhắc giữa truyền giá trị hay truyền địa chỉ/tham chiếu vào kiểu hằng
- Chọn tham số đầu ra (=> tham biến truyền qua địa chỉ/qua tham chiếu hoặc sử dụng giá trị trả về)
  - Đặc tả ý nghĩa, đặt tên, chọn kiểu tương tự như tham số đầu vào
- Định nghĩa bổ sung các kiểu dữ liệu mới như cần thiết
- Mô tả rõ tiền trạng (*pre-condition*): điều kiện biên cho các tham số đầu vào và các điều kiện ngoại cảnh cho việc gọi hàm
- Mô tả rõ hậu trạng (*post-condition*): tác động của việc sử dụng hàm tới ngoại cảnh, các thao tác bắt buộc sau này,...
- Thiết kế thân hàm dựa vào các chức năng đã phân tích, sử dụng lưu đồ thuật toán với các cấu trúc điều kiện/rẽ nhánh (kể cả vòng lặp) => có thể phân chia thành các hàm con nếu cần

Ví dụ minh họa: Tìm số nguyên tố

Bài toán: Xây dựng hàm tìm N số nguyên tố đầu tiên!

- Phân tích:
  - Dữ kiện: N - số số nguyên tố đầu tiên cần tìm
  - Kết quả: Một dãy N số nguyên tố đầu tiên
  - Các chức năng cần thực hiện:
    - Nhập dữ liệu? KHÔNG!
    - Kiểm tra dữ kiện vào (N)? Có/không (Nếu kiểm tra mà N nhỏ hơn 0 thì hàm làm gì?)
    - Cho biết k số nguyên tố đầu tiên, xác định số nguyên tố tiếp theo
    - Lưu trữ kết quả mỗi lần tìm ra vào một cấu trúc dữ liệu phù hợp (dãy số cần tìm)
    - In kết quả ra màn hình? KHÔNG!

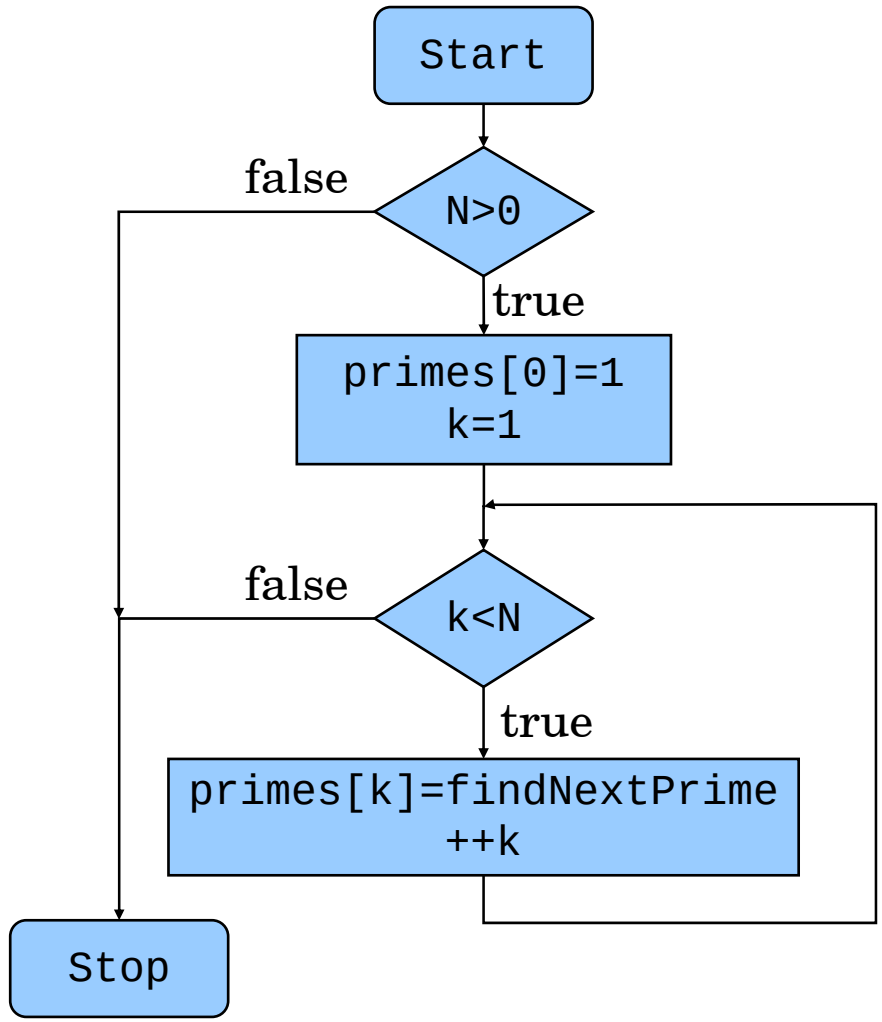
- Đặt tên hàm: `findPrimeSequence`
- Tham số vào: 1
  - Ý nghĩa: số các số nguyên tố cần tìm
  - Tên: `N`
  - Kiểu: số nguyên đủ lớn (`int/long`)
  - Truyền tham số: qua giá trị
- Tham số ra: 1
  - Ý nghĩa: dãy N số nguyên tố đầu tiên tính từ 1
  - Giá trị trả về hay tham biến? **Tham biến!**
  - Tên: `primes`
  - Kiểu: mảng số nguyên (của `int/long`)
  - Truyền tham số: qua địa chỉ (`int*` hoặc `long*`)
- Tiền trạng:
  - Tham số `N` phải là số không âm (có nên chọn kiểu `unsigned`?)
  - `primes` phải mang địa chỉ của mảng số nguyên có ít nhất N phần tử
- Hậu trạng: không có gì đặc biệt

- Khai báo hàm:

```
void findPrimeSequence(int N, int* primes);
```

- Thiết kế thân hàm
  - Lưu đồ thuật toán như hình vẽ
  - Phân chia, bổ sung một hàm mới: `findNextPrime`
- Lặp lại qui trình thiết kế hàm cho `findNextPrime`

  (Bài tập về nhà!)



3.5 Thư viện chuẩn ANSI-C

- Thư viện vào/ra (nhập/xuất) `<stdio.h>`
- Xử lý ký tự và chuỗi ký tự `<string.h>, <ctype.h>`
- Thư viện hàm toán `<math.h>, <float.h>`
- Thời gian, ngày tháng `<time.h>, <locale.h>`
- Cấp phát bộ nhớ động `<stdlib.h>`
- Các hàm ký tự rộng `<wchar.h>,<wctype.h>`
- Các hàm khác `<stdlib.h>, ...`

3.6 Làm việc với tệp tin trong C++

```
#include <iostream.h>
#include <fstream.h>
```

- Khai báo một biến:

```
ifstream fin;  // input
ofstream fout;        // output
fstream  fio;  // input and output
```

- Mở/tạo một tệp tin:

```
fin.open("file1.txt");
fout.open("file2.dat");
fio.open("file3.inf");
```

- Kết hợp khai báo biến và mở/tạo một tệp tin

```
ifstream fin("file1.txt"); // input
ofstream fout("file2.inf");// output
fstream  fio("file3.dat"); // input and output
```

- Ghi dữ liệu ra tệp tin
  - Tương tự như sử dụng `cout`
  - Tệp tin có thể chứa dữ liệu kiểu hỗn hợp, ví dụ:

```
fout << "Nguyen Van A" << endl;
fout << 21 << endl << false;
```

- Đọc dữ liệu từ một tệp tin
  - Tương tự như sử dụng `cin`

```
char name[32];
int  age, married;
fin.getline(name,32);
fin >> age >> married;
```

- Đóng một tệp tin:
  - Tự động khi kết thúc phạm vi { },
  - Hoặc gọi hàm thành viên `close()`:

```
fin.close();
fout.close();
fio.close();
```

Ví dụ: làm việc với tệp tin

```
#include <iostream.h>
#include <fstream.h>
void main() {
   {
     ofstream fout("file1.dat");// output
     fout << "Nguyen Van A" << endl << 21 << endl << false;
   }
   {
     ifstream fin("file1.dat"); // input
     char name[32];
     int  age;
     int  married;
     fin.getline(name,32);
     fin >> age >> married;
     cout << "Name:\t" << name << endl;
     cout << "Age:\t" << age << endl;
     cout << "Married:" << (married ? "Yes" : "No");
   }
   char c;
   cin >> c;
}
```

3.7 Nạp chồng tên hàm trong C++

- Trong C++ có thể xây dựng nhiều hàm có cùng tên, ví dụ:

```
int    max(int a, int b);
double max(double a, double b);
double max(double a, double b, double c);
double max(double *seq, int n);
```

- Mục đích của **nạp chồng tên hàm** là:
  - Đơn giản hóa cho người xây dựng hàm trong việc chọn tên (thay vì `maxInt, maxDouble, maxDouble3, maxDoubleSequence,...`)
  - Đơn giản hóa cho người sử dụng hàm, chỉ cần nhớ 1 tên quen thuộc thay cho nhiều tên phức tạp

Ví dụ: định nghĩa các hàm max()

```
int max(int a, int b) {                              // (1)
   return (a > b)? a : b;
}
double max(double a, double b) {                     // (2)
   return (a > b)? a : b;
}
double max(double a, double b, double c); {          // (3)
   if (a < b) a = b;
   if (a < c) a = c;
   return a;
}
double max(double *seq, int n) {                     // (4)
   int i = 0, kq = seq[0];
   while (i < n) {
     if (kq < seq[i])kq = seq[i];
     ++i;
   }
   return kq;
}
```

Ví dụ: sử dụng các hàm max()

```
int max(int a, int b);                          // (1)
double max(double a, double b);                 // (2)
double max(double a, double b, double c);       // (3)
double max(double *seq, int n);                 // (4)

void main() {
   int k = max(5,7);                  // call (1)
   double d = max(5.0,7.0);// call (2)
   double a[] = {1,2,3,4,5,6};
   d = max(d, a[1], a[2]);            // call (3)
   d = max(a, 5);                     // call (4)
   d = max(5,7);                      // ?
   d = max(d, 5);                     // ?
}
```

➲ Đẩy trách nhiệm kiểm tra và tìm hàm phù hợp cho compiler!

Một số qui tắc về nạp chồng tên hàm

- Các hàm cùng tên được **định nghĩa cùng trong một file/ trong một thư viện** hoặc **sử dụng trong cùng một chương trình** phải khác nhau ít nhất về:
  - Số lượng các tham số, hoặc
  - Kiểu của ít nhất một tham số (`int` khác `short`, `const int` khác `int`, `int` khác `int&`, `...`)

  ➲ Không thể chỉ khác nhau ở kiểu trả về
- Tại sao vậy?
  - Compiler cần có cơ sở để quyết định gọi hàm nào
  - Dựa vào cú pháp trong lời gọi (số lượng và kiểu các tham số thực tế) compiler sẽ chọn hàm có cú pháp phù hợp nhất
  - Khi cần compiler có thể tự động chuyển đổi kiểu theo chiều hướng **hợp lý nhất** (vd `short=>int, int => double`)

3.8 Hàm inline trong C++

- Vấn đề: Hàm tiện dụng, nhưng nhiều khi hiệu suất không cao, đặc biệt khi mã thực thi hàm ngắn
  - Các thủ tục như nhớ lại trạng thái chương trình, cấp phát bộ nhớ ngăn xếp, sao chép tham số, sao chép giá trị trả về, khôi phục trạng thái chương trình **mất nhiều thời gian**
  - Nếu mã thực thi hàm ngắn thì **sự tiện dụng không bõ so với sự lãng phí thời gian**
- Giải pháp trong C: Sử dụng macro, ví dụ

```
#define max(a,b)  a>b?a:b
```

  - Vấn đề: Macro do tiền xử lý chạy (preprocessor), không có kiểm tra kiểu, không có phân biệt ngữ cảnh => gây ra các hiệu ứng phụ không mong muốn

    Ví dụ dòng lệnh `l=max(k*5-2,l);`

    sẽ được thay thế bằng `l=k*5-2>k?k*5-2:l;  // OOPS!`
  - Những cách giải quyết như thêm dấu ngoặc chỉ làm mã khó đọc, không khắc phục triệt để các nhược điểm

Giải pháp hàm inline trong C++

- Điều duy nhất cần làm là thêm từ khóa `inline` vào đầu dòng khai báo và định nghĩa hàm

```
inline int max(int a, int b) {
    return (a > b)? a : b;
}
```

- Hàm inline khác gì hàm bình thường:
  - "Hàm inline" thực chất không phải là một hàm!
  - Khi gọi hàm thì lời gọi hàm được **thay thế một cách thông minh bởi mã nguồn định nghĩa hàm,** không thực hiện các thủ tục gọi hàm

Ví dụ:

```
l=max(k*5-2,l);
```

Được thay thế bằng các dòng lệnh kiểu như:

```
int x=k*5-2;  // biến tạm trung gian
l=(x>l)?x:l;  // OK
```

Khi nào nên dùng hàm inline

- Ưu điểm của hàm inline:
  - Tiện dụng như hàm bình thường
  - Hiệu suất như viết thẳng mã, không gọi hàm
  - Tin cậy, an toàn hơn nhiều so với sử dụng Macro
- Nhược điểm của hàm inline:
  - Nếu gọi hàm nhiều lần trong chương trình, mã chương trình có thể lớn lên nhiều (mã thực hiện hàm xuất hiện nhiều lần trong chương trình)
  - Mã định nghĩa hàm phải để mở => đưa trong header file
- Lựa chọn xây dựng và sử dụng hàm inline khi:
  - Mã định nghĩa hàm nhỏ (một vài dòng lệnh, không chứa vòng lặp)
  - Yêu cầu về **tốc độ** đặt ra trước **dung lượng bộ nhớ**

Bài tập về nhà

- Xây dựng hàm tìm N số nguyên tố đầu tiên
  - Hoàn thiện thiết kế hàm
  - Định nghĩa hàm
- Viết chương trình minh họa cách sử dụng