

Ngôn ngữ lập trình C++

Chương 3 – Hàm

Chương 3 - Hàm

Đề mục

- 3.1 Giới thiệu
- 3.2 Các thành phần của chương trình C++
- 3.3 Các hàm trong thư viện toán học
- 3.4 Hàm
- 3.5 Định nghĩa hàm (Function Definition)
- 3.6 Nguyên mẫu hàm (Function Prototype)
- 3.7 Header File
- 3.8 Sinh số ngẫu nhiên
- 3.9 Ví dụ: Trò chơi may rủi và Giới thiệu về kiểu enum
- 3.10 Các kiểu lưu trữ (Storage Class)
- 3.11 Các quy tắc phạm vi (Scope Rule)
- 3.12 đệ quy (Recursion)
- 3.13 Ví dụ sử dụng đệ quy: chuỗi Fibonacci
- 3.14 So sánh đệ quy và Vòng lặp
- 3.15 Hàm với danh sách đối số rỗng

Chương 3 - Hàm

Đề mục

- 3.16 Hàm Inline
- 3.17 Tham chiếu và tham số là tham chiếu
- 3.18 Đối số mặc định
- 3.19 Toán tử phạm vi đơn (Unary Scope Resolution Operator)
- 3.20 Chồng hàm (Function Overloading)
- 3.21 Khuôn mẫu hàm (Function Templates)

3.1 Giới thiệu

- Chia để trị - Divide and conquer
 - Xây dựng một chương trình từ các thành phần (component) nhỏ hơn
 - Quản lý từng thành phần dễ quản lý hơn quản lý chương trình ban đầu

3.2 Các thành phần của chương trình C++

- Các module: các hàm(function) và lớp(class)
- Các chương trình sử dụng các module mới và đóng gói sẵn (“prepackaged”)
 - Mới: các hàm và lớp do lập trình viên tự định nghĩa
 - Đóng gói sẵn: các hàm và lớp từ thư viện chuẩn
- lời gọi hàm - function call
 - tên hàm và các thông tin (các đối số - arguments) mà nó cần
- định nghĩa hàm - function definition
 - chỉ viết một lần
 - được che khỏi các hàm khác
- tương tự
 - Một ông chủ (hàm gọi - the calling function or caller) đề nghị một công nhân (hàm được gọi - the called function) thực hiện một nhiệm vụ và trả lại (báo cáo lại) kết quả khi nhiệm vụ hoàn thành.

3.3 Các hàm trong thư viện toán học

- Thực hiện các tính toán toán học thông thường
 - Include header file **<cmath>** (hoặc **<math.h>**)
- Cách gọi hàm
 - tên_hàm(đối_số); hoặc
 - tên_hàm(đối_số_1, đối_số_2, ...);
- Ví dụ

```
cout << sqrt( 900.0 );
```

 - Mọi hàm trong thư viện toán đều trả về giá trị kiểu **double**
- các đối số (argument) cho hàm có thể là
 - hằng - Constants
 - **sqrt(4);**
 - biến - Variables
 - **sqrt(x);**
 - biểu thức - Expressions
 - **sqrt(sqrt(x));**
 - **sqrt(3 - 6x);**

Method	Description	Example
ceil(x)	làm tròn x tới số nguyên nhỏ nhất không nhỏ hơn x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	cos của x (lưu ý: x tính theo độ)	cos(0.0) is 1.0
exp(x)	hàm mũ: e mũ x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
fabs(x)	giá trị tuyệt đối của x	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	làm tròn x xuống số nguyên lớn nhất không lớn hơn x	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	phần dư của x/y , tính bằng kiểu số thực	fmod(13.657, 2.333) is 1.992
log(x)	loga tự nhiên của x (cơ số e)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	loga cơ số 10 của x	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	x mũ y	pow(2, 7) is 128 pow(9, .5) is 3
sin(x)	sin x (lưu ý: x tính theo radian)	sin(0.0) is 0
sqrt(x)	căn bậc hai của x	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	tang x (lưu ý: x tính theo radian)	tan(0.0) is 0

Fig. 3.2 Math library functions.

3.4 Hàm - function

- Chương trình con
 - Module hóa một chương trình
 - khả năng tái sử dụng phần mềm – Software reusability
 - gọi hàm nhiều lần
- Các biến địa phương – Local variables
 - khai báo trong hàm nào thì chỉ được biết đến bên trong hàm đó
 - biến được khai báo bên trong định hàm là biến địa phương
- Các tham số – Parameters
 - là các biến địa phương với giá trị được truyền vào hàm khi hàm được gọi
 - cung cấp thông tin về bên ngoài hàm

fig03_03.cpp
(1 of 2)

```
2 // Creating and using a programmer-defined function.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int square( int );    // function prototype
9
10 int main()
11 {
12     // loop 10 times and calculate and output
13     // square of x each time
14     for ( int x = 1; x <= 10; x++ )
15         cout << square( x ) << " ";    // function call
16
17     cout << endl;
18
19     return 0;    // indicates successful termination
20
21 } // end main
22
23 // square function definition returns square of an integer
24 int square( int y )    // y is a copy of argument to function
25 {
26     return y * y;    // returns square of
27
28 } // end function square
```

Function prototype: chỉ rõ kiểu dữ liệu của đối số và giá trị trả về. **square** cần một số **int**, và trả về **int**.

Cặp ngoặc () dùng khi gọi hàm.
Khi chạy xong, hàm trả kết quả.

1 4 9 16 25 36 49 64 81 100

Định nghĩa hàm **square**. **y** là một bản sao của đối số được truyền vào. Hàm trả về **y * y**, hoặc **y** bình phương.

```
1 // Fig. 3.4: fig03_04.cpp
2 // Finding the maximum of three floating-point numbers.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 double maximum( double, double, double ); // function prototype
10
11 int main()
12 {
13     double number1;
14     double number2;
15     double number3;
16
17     cout << "Enter three floating-point numbers: ";
18     cin >> number1 >> number2 >> number3;
19
20     // number1, number2 and number3 are arguments to
21     // the maximum function call
22     cout << "Maximum is: "
23         << maximum( number1, number2, number3 ) << endl;
24
25     return 0; // indicates successful termination
```

Hàm **maximum** lấy 3 tham số
(cả 3 là **double**) và trả về
một **double**.

```

26
27 } // end main
28
29 // function maximum definition;
30 // x, y and z are parameters
31 double maximum( double x, double y, double z )
32 {
33     double max = x;    // assume x is largest
34
35     if ( y > max )      // if y is larger,
36         max = y;        // assign y to max
37
38     if ( z > max )      // if z is larger,
39         max = z;        // assign z to max
40
41     return max;        // max is largest value
42
43 } // end function maximum

```

dấu phẩy phân tách các tham số.

fig03_04.cpp

(2 of 2)

fig03_04.cpp
output (1 of 1)

Enter three floating-point numbers: 99.32 37.3 27.1928

Maximum is: 99.32

Enter three floating-point numbers: 1.1 3.333 2.22

Maximum is: 3.333

Enter three floating-point numbers: 27.9 14.31 88.99

3.4 Hàm

- Nguyên mẫu hàm - Function prototype
 - Cho trình biên dịch biết kiểu dữ liệu của đối số và kiểu giá trị trả về của hàm
int square(int);
 - Hàm lấy một giá trị **int** và trả về một giá trị **int**
 - Sẽ giới thiệu kỹ hơn sau
- Gọi hàm
square(x) ;
 - Cặp ngoặc đơn là toán tử dùng để gọi hàm
 - Truyền đối số x
 - Hàm nhận được bản sao các đối số cho riêng mình
 - Sau khi kết thúc, hàm trả kết quả về cho nơi gọi hàm

3.5 Định nghĩa hàm – function definition

- định nghĩa hàm

```
return-value-type function-name ( parameter-list )  
{  
    declarations and statements  
}
```

- danh sách tham số – Parameter list

- dấu phẩy tách các tham số

- mỗi tham số cần cho biết kiểu dữ liệu của tham số đó

- Nếu không có đối số, sử dụng **void** hoặc để trống

- giá trị trả về – Return-value-type

- kiểu của giá trị trả về (sử dụng **void** nếu không trả về giá trị gì)

3.5 Định nghĩa hàm

Ví dụ về hàm

```
int square( int y )
{
    return y * y;
}

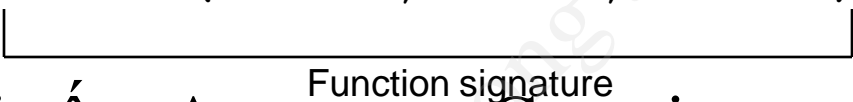
int main()
{
    ...
    cout << square(x);
    ...
}
```

- Từ khóa **return**
 - trả dữ liệu về, và trả điều khiển lại cho nơi gọi (caller)
 - nếu không trả về, sử dụng **return**;
 - hàm kết thúc khi chạy đến ngoặc phải (})
 - điều khiển cũng được trả về cho nơi gọi
- Không thể định nghĩa một hàm bên trong một hàm khác

3.6 Nguyên mẫu hàm - Function Prototype

- Function prototype bao gồm
 - Tên hàm
 - Các tham số (số lượng và kiểu dữ liệu)
 - Kiểu trả về (**void** nếu không trả về giá trị gì)
- Function prototype chỉ cần đến nếu định nghĩa hàm đặt sau lời gọi hàm (function call)
- Prototype phải khớp với định nghĩa hàm
 - Function prototype
`double maximum(double, double, double);`
 - Function definition
`double maximum(double x, double y, double z)
{
...
}`

3.6 Function Prototype

- Chữ ký của hàm - Function signature
 - Phần prototype chứa tên và các tham số của hàm
 - `double maximum(double, double, double);`

- Ép kiểu đối số – Argument Coercion
 - Ép các đối số thành các kiểu dữ liệu thích hợp
 - đổi `int` (4) thành `double` (4.0)
`cout << sqrt(4)`
 - các quy tắc biến đổi
 - các đối số thường được tự động đổi kiểu
 - đổi từ `double` sang `int` có thể làm tròn dữ liệu
 - 3.4 thành 3
 - các kiểu hỗn hợp được nâng lên kiểu cao nhất
 - `int * double`

3.6 Function Prototype

Data types	
<code>long double</code>	
<code>double</code>	
<code>float</code>	
<code>unsigned long int</code>	(synonymous with <code>unsigned long</code>)
<code>long int</code>	(synonymous with <code>long</code>)
<code>unsigned int</code>	(synonymous with <code>unsigned</code>)
<code>int</code>	
<code>unsigned short int</code>	(synonymous with <code>unsigned short</code>)
<code>short int</code>	(synonymous with <code>short</code>)
<code>unsigned char</code>	
<code>char</code>	
<code>bool</code>	(<code>false</code> becomes 0, <code>true</code> becomes 1)
Fig . 3.5 Promotion hierarchy for built-in data types.	

3.7 Header File

- Các file header chứa
 - các function prototype
 - định nghĩa của các kiểu dữ liệu và các hằng
- Các file header kết thúc bằng .h
 - các file header do lập trình viên định nghĩa
`#include "myheader.h"`
- Các file header của thư viện
 - `#include <cmath>`
 - chú ý:
 - <cmath> tương đương với <math.h> (kiểu cũ, trước ANSI C++)
 - <iostream> tương đương với <iostream.h> (kiểu cũ, trước ANSI C++)

3.8 Sinh số ngẫu nhiên

Random Number Generation

- Hàm **rand** (thuộc **<cstdlib>**)
 - **i = rand();**
 - Sinh một số nguyên không âm trong đoạn từ 0 đến RAND_MAX (thường là 32767)
- Lấy tỷ lệ và dịch (scaling and shifting)
 - phép đồng dư (lấy số dư) – Modulus (remainder) operator: %
 - **10 % 3** bằng **1**
 - **x % y** nằm giữa **0** và **y - 1**
 - Ví dụ
 - i = rand() % 6 + 1;**
 - **"Rand() % 6"** sinh một số trong khoảng từ 0 đến 5 (lấy tỷ lệ)
 - **" + 1"** tạo khoảng từ 1 đến 6 (dịch)
 - Tiếp theo: chương trình thả súc sắc

fig03_07.cpp

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

```

1  // Fig. 3.7: fig03_07.cpp
2  // Shifted, scaled integers produced by 1 + rand() % 6.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>    // contains function prototype for rand
13
14 int main()
15 {
16     // loop 20 times
17     for ( int counter = 1; counter <= 20; counter++ )
18
19         // pick random number from 1 to 6 and output it
20         cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22         // if counter divisible by 5, begin new line of output
23         if ( counter % 5 == 0 )
24             cout << endl;
25
26     } // end for structure
27
28     return 0;    // indicates successful termination
29
30 } // end main

```

Kết quả của **rand()** được lấy tỷ lệ và dịch thành một số trong khoảng từ 1 đến 6.

3.8 Sinh số ngẫu nhiên

- Tiếp theo
 - Chương trình biểu diễn phân bố (distribution) của hàm **rand()**
 - Giả lập 6000 lần thả súc sắc
 - In số lượng các giá trị 1, 2, 3, v.v.... thả được
 - số lượng đếm được của mỗi giá trị phải xấp xỉ 1000

fig03_08.cpp
(1 of 3)

```
1  // Fig. 3.8: fig03_08.cpp
2  // Roll a six-sided die 6000 times.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>    // contains function prototype for rand
13
14 int main()
15 {
16     int frequency1 = 0;
17     int frequency2 = 0;
18     int frequency3 = 0;
19     int frequency4 = 0;
20     int frequency5 = 0;
21     int frequency6 = 0;
22     int face;    // represents one roll of the die
23
```

fig03_08.cpp**(2 of 3)**

```
24 // loop 6000 times and summarize results
25 for ( int roll = 1; roll <= 6000; roll++ ) {
26     face = 1 + rand() % 6; // random number from 1 to 6
27
28     // determine face value and increment appropriate counter
29     switch ( face ) {
30
31         case 1: // rolled 1
32             ++frequency1;
33             break;
34
35         case 2: // rolled 2
36             ++frequency2;
37             break;
38
39         case 3: // rolled 3
40             ++frequency3;
41             break;
42
43         case 4: // rolled 4
44             ++frequency4;
45             break;
46
47         case 5: // rolled 5
48             ++frequency5;
49             break;
```

fig03_08.cpp
(3 of 3)

```
50
51     case 6:          // rolled 6
52         ++frequency6;
53         break;
54
55     default:         // invalid value
56         cout << "Program should never get here!";
57
58 } // end switch
59
60 } // end for
61
62 // display results in tabular format
63 cout << "Face" << setw( 13 ) << "Frequency"
64      << "\n   1" << setw( 13 ) << frequency1
65      << "\n   2" << setw( 13 ) << frequency2
66      << "\n   3" << setw( 13 ) << frequency3
67      << "\n   4" << setw( 13 ) << frequency4
68      << "\n   5" << setw( 13 ) << frequency5
69      << "\n   6" << setw( 13 ) << frequency6 << endl;
70
71 return 0; // indicates successful termination
72
73 } // end main
```

Trường hợp mặc định được xét đến, ngay cả khi nó không bao giờ xảy ra. Đây là một nét của phong cách lập trình tốt

Face	Frequency
1	1003
2	1017
3	983
4	994
5	1004
6	999

3.8 Sinh số ngẫu nhiên

- Gọi `rand()` lặp đi lặp lại
 - cho kết quả là cùng một chuỗi số
- Các số giả ngẫu nhiên (pseudorandom numbers)
 - chuỗi các số "ngẫu nhiên" được định sẵn
 - chương trình chạy lần nào cũng sinh cùng một chuỗi
- Để được các chuỗi ngẫu nhiên khác nhau
 - Cung cấp một giá trị hạt giống
 - điểm xuất phát cho việc sinh chuỗi ngẫu nhiên
 - hạt giống giống nhau sẽ cho cùng một chuỗi ngẫu nhiên
 - **`srand (seed) ;`**
 - **`<cstdlib>`**
 - sử dụng trước **`rand ()`** để đặt hạt giống

fig03_09.cpp
(1 of 2)

```
1 // Fig. 3.9: fig03_09.cpp
2 // Randomizing die-rolling program.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 // contains prototypes for functions srand and rand
14 #include <cstdlib>
15
16 // main function begins program execution
17 int main()
18 {
19     unsigned seed;
20
21     cout << "Enter seed: ";
22     cin >> seed;
23     srand( seed ); // seed random number generator
24 }
```

Đặt hạt giống bằng
srand().

fig03_09.cpp
(2 of 2)

fig03_09.cpp
output (1 of 1)

```
25 // loop 10 times
26 for ( int counter = 1; counter <= 10; counter++ ) {
27
28     // pick random number from 1 to 6 and output it
29     cout << setw( 10 ) << ( 1 + rand() % 6 );
30
31     // if counter divisible by 5, begin new line of output
32     if ( counter % 5 == 0 )
33         cout << endl;
34
35 } // end for
36
37 return 0; // indicate successful termination
38
39 } // end main
```

rand() sinh cùng một chuỗi
ngẫu nhiên nếu dùng cùng
một hạt giống

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

Enter seed: 432

4	6	3	1	6
3	1	5	4	2

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

3.8 Sinh số ngẫu nhiên

- Có thể sử dụng thời gian hiện tại để làm hạt giống
 - không cần phải đặt hạt giống mỗi lần sinh 1 số ngẫu nhiên
 - `srand(time(0));`
 - `time(0);`
 - `<ctime>`
 - trả về thời gian hiện tại, tính bằng giây
- Tổng quát về định và lấy tỷ lệ
 - $Number = shiftingValue + rand() \% scalingFactor$
 - `shiftingValue` = số đầu tiên của khoảng mong muốn
 - `scalingFactor` = độ rộng của khoảng mong muốn

3.9 Ví dụ: Trò chơi may rủi và Giới thiệu về kiểu enum

- Kiểu liệt kê - Enumeration

- tập hợp các số tự nhiên được đặt tên

enum *typeName* { *constant1*, *constant2*... } ;

- Các hằng số là các số nguyên bắt đầu từ 0 (mặc định), tăng dần, mỗi lần thêm 1 đơn vị.
- Các hằng phải có tên riêng
- Không thể gán giá trị kiểu nguyên cho biến kiểu liệt kê
 - Phải dùng một giá trị thuộc cùng kiểu liệt kê đã được định nghĩa

- Ví dụ

```
enum Status {CONTINUE, WON, LOST};
```

```
enum Foo {Zero, One, Two};
```

```
Status enumVar;
```

```
enumVar = WON; // cannot do enumVar = 1 or enumVar=One
```

3.9 Ví dụ: Trò chơi may rủi và Giới thiệu về kiểu enum

- Các hằng kiểu liệt kê có thể có giá trị đặt trước
`enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC } ;`
 - bắt đầu tại 1, tăng dần mỗi lần thêm 1
- Tiếp theo: giả lập trò chơi gieo súc sắc
 - Gieo 2 con súc sắc, được kết quả là tổng hai giá trị gieo được
 - 7 hoặc 11 tại lần gieo đầu tiên: người chơi thắng
 - 2, 3, hoặc 12 tại lần gieo đầu tiên: người chơi thua
 - 4, 5, 6, 8, 9, 10
 - giá trị gieo được trở thành "điểm" (point) của người chơi
 - người chơi phải gieo được số điểm của mình trước khi gieo được 7 để thắng cuộc

fig03_10.cpp
(1 of 5)

```
1 // Fig. 3.10: fig03_10.cpp
2 // Craps.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // contains function prototypes for functions srand and rand
9 #include <cstdlib>
10
11 #include <ctime> // contains prototype for function time
12
13 int rollDice( void ); // function prototype
14
15 int main()
16 {
17     // enumeration constants represent game status
18     enum Status { CONTINUE, WON, LOST };
19
20     int sum;
21     int myPoint;
22
23     Status gameStatus; // can contain CONTINUE, WON or LOST
24
```

Hàm gieo 2 con súc sắc và trả về kết quả là 1 giá trị kiểu **int**.

Kiểu liệt kê để ghi trạng thái của ván chơi hiện tại.

fig03_10.cpp
(2 of 5)

```
25 // randomize random number generator using current time
26 srand( time( 0 ) );
27
28 sum = rollDice(); // first roll of the dice
29
30 // determine game status and point based on sum of dice
31 switch ( sum ) {
32
33     // win on first roll
34     case 7:
35     case 11:
36         gameStatus = WON;
37         break;
38
39     // lose on first roll
40     case 2:
41     case 3:
42     case 12:
43         gameStatus = LOST;
44         break;
45
```

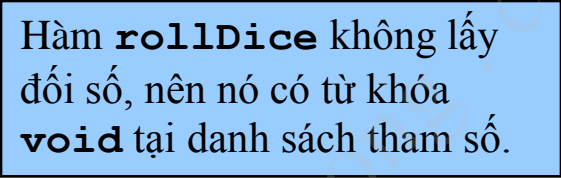
lệnh **switch** quyết định kết
cục ván chơi, dựa vào kết quả
gieo súc sắc.

fig03_10.cpp
(3 of 5)

```
46     // remember point
47     default:
48         gameStatus = CONTINUE;
49         myPoint = sum;
50         cout << "Point is " << myPoint << endl;
51         break;                // optional
52
53 } // end switch
54
55 // while game not complete ...
56 while ( gameStatus == CONTINUE ) {
57     sum = rollDice();        // roll dice again
58
59     // determine game status
60     if ( sum == myPoint )    // win by making point
61         gameStatus = WON;
62     else
63         if ( sum == 7 )      // lose by rolling 7
64             gameStatus = LOST;
65
66 } // end while
67
```

fig03_10.cpp
(4 of 5)

```
68 // display won or lost message
69 if ( gameStatus == WON )
70     cout << "Player wins" << endl;
71 else
72     cout << "Player loses" << endl;
73
74 return 0; // indicates successful termination
75
76 } // end main
77
78 // roll dice, calculate sum and display results
79 int rollDice( void )
80 {
81     int die1;
82     int die2;
83     int workSum;
84
85     die1 = 1 + rand() % 6; // pick random die1 value
86     die2 = 1 + rand() % 6; // pick random die2 value
87     workSum = die1 + die2; // sum die1 and die2
88
```



Hàm **rollDice** không lấy
đối số, nên nó có từ khóa
void tại danh sách tham số.

```

89 // display results of this roll
90 cout << "Player rolled " << die1 << " + " << die2
91     << " = " << workSum << endl;
92
93 return workSum;           // return sum of dice
94
95 } // end function rollDice

```

fig03_10.cpp
(5 of 5)

fig03_10.cpp
output (1 of 2)

Player rolled 2 + 5 = 7
Player wins

Player rolled 6 + 6 = 12
Player loses

Player rolled 3 + 3 = 6
Point is 6

Player rolled 5 + 3 = 8

Player rolled 4 + 5 = 9

Player rolled 2 + 1 = 3

Player rolled 1 + 5 = 6

Player wins

Player rolled 1 + 3 = 4

Point is 4

Player rolled 4 + 6 = 10

Player rolled 2 + 4 = 6

Player rolled 6 + 4 = 10

Player rolled 2 + 3 = 5

Player rolled 2 + 4 = 6

Player rolled 1 + 1 = 2

Player rolled 4 + 4 = 8

Player rolled 4 + 3 = 7

Player loses

3.10 Các kiểu lưu trữ – Storage Classes

- biến có các thuộc tính
 - đã biết: tên, kiểu, kích thước, giá trị
 - kiểu lưu trữ – Storage class
 - biến tồn tại bao lâu trong bộ nhớ
 - Phạm vi – Scope
 - biến có thể được sử dụng tại những nơi nào trong chương trình
 - Liên kết – Linkage
 - Đối với những chương trình gồm nhiều file (multiple-file program) – (xem chương 6), những file nào có thể sử dụng biến đó

3.10 Các kiểu lưu trữ – Storage Classes

- loại biến tự động – Automatic storage class
 - biến được tạo khi chương trình chạy vào một khối chương trình (block)
 - và bị hủy bỏ khi chương trình ra khỏi block
 - Chỉ có các biến địa phương của các hàm mới có thể là biến tự động
 - mặc định là tự động
 - từ khóa **auto** dùng để khai báo biến tự động
 - từ khóa **register**
 - gợi ý đặt biến vào thanh ghi tốc độ cao
 - có lợi cho các biến thường xuyên được sử dụng (con đếm vòng lặp)
 - Thường là không cần thiết, trình biên dịch tự tối ưu hóa
 - Chỉ dùng một trong hai từ **register** hoặc **auto**.
 - **register int counter = 1;**

3.10 Các kiểu lưu trữ

- loại biến tĩnh – Static storage class
 - Biến tồn tại trong suốt chương trình
 - Có thể không phải nơi nào cũng dùng được, do áp dụng quy tắc phạm vi (scope rules)
- từ khóa **static**
 - dành cho biến địa phương bên trong hàm
 - giữ giá trị giữa các lần gọi hàm
 - chỉ được biết đến trong hàm của biến đó
- từ khóa **extern**
 - mặc định với các biến/hàm toàn cục (global variables/functions)
 - toàn cục: được định nghĩa bên ngoài các hàm
 - được biết đến tại mọi hàm nằm sau biến đó

3.11 Các quy tắc phạm vi – Scope Rules

- Phạm vi – Scope
 - Phạm vi của một định danh (tên) là phần chương trình nơi có thể sử dụng định danh đó
- Phạm vi file – File scope
 - được định nghĩa bên ngoài một hàm và được biết đến tại mọi hàm trong file
 - các biến toàn cục (global variable), định nghĩa và prototype của các hàm.
- Phạm vi hàm – Function scope
 - chỉ có thể được dùng đến bên trong hàm chứa định nghĩa
 - Chỉ áp dụng cho các nhãn (label), ví dụ: các định danh đi kèm một dấu hai chấm (**case :**)

3.11 Các quy tắc phạm vi

- Phạm vi khối – Block scope
 - Bắt đầu tại nơi khai báo, kết thúc tại ngoặc phải }
 - chỉ có thể được dùng trong khoảng này
 - Các biến địa phương, các tham số hàm
 - các biến **static** cũng có phạm vi khối
 - loại lưu trữ độc lập với phạm vi
- Function-prototype scope
 - danh sách tham số của function prototype
 - không bắt buộc phải chỉ rõ các tên trong prototype
 - Trình biên dịch bỏ qua
 - Trong một prototype, mỗi tên chỉ được dùng một lần

fig03_12.cpp
(1 of 5)

```
1 // Fig. 3.12: fig03_12.cpp
2 // A scoping example.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void useLocal( void );
9 void useStaticLocal( void );
10 void useGlobal( void ); // function prototype
11
12 int x = 1; // global variable
13
14 int main()
15 {
16     int x = 5; // local variable to main
17
18     cout << "local x in main's outer scope is " << x << endl;
19
20     { // start new scope
21
22         int x = 7;
23
24         cout << "local x in main's inner scope is " << x << endl;
25
26     } // end new scope
```

được khai báo bên ngoài hàm;
là biến toàn cục với phạm vi file.

Biến địa phương với phạm vi hàm.

Tạo một khối, cho **x** phạm vi
khối. Khi khối kết thúc, **x** sẽ
bị hủy bỏ.

```
27     cout << "local x in main's outer scope is " << x << endl;
28
29
30     useLocal();           // useLocal has local x
31     useStaticLocal();     // useStaticLocal has static local x
32     useGlobal();          // useGlobal uses global x
33     useLocal();           // useLocal reinitializes its local x
34     useStaticLocal();     // static local x retains its prior value
35     useGlobal();          // global x also retains its value
36
37     cout << "\nlocal x in main is " << x << endl;
38
39     return 0;             // indicates successful termination
40
41 } // end main
42
```

```
43 // useLocal reinitializes local variable x during each call
44 void useLocal( void )
45 {
46     int x = 25; // initialized each time useLocal is called
47
48     cout << endl << "local x is
49         << " on entering useLo
50     ++x;
51     cout << "local x is " << x
52         << " on exiting useLocal" << endl;
53
54 } // end function useLocal
55
```

Biến tự động (biến địa phương của hàm). Biến này sẽ bị hủy khi hàm kết thúc, và được khởi tạo lại khi hàm bắt đầu.

fig03_12.cpp
(3 of 5)

fig03_12.cpp
(4 of 5)

```
56 // useStaticLocal initializes static local variable x only the
57 // first time the function is called; value of x is saved
58 // between calls to this function
59 void useStaticLocal( void )
60 {
61     // initialized only first time useStaticLocal is called
62     static int x = 50;
63
64     cout << endl << "local static x is " << x
65         << " on entering useStaticLocal" << endl;
66     ++x;
67     cout << "local static x is " << x
68         << " on exiting useStaticLocal" << endl;
69
70 } // end function useStaticLocal
71
```

Biến tĩnh địa phương của hàm; nó được khởi tạo đúng một lần và giữ nguyên giá trị giữa các lần gọi hàm.

```
72 // useGlobal modifies global variable x during each call
```

```
73 void useGlobal( void )
```

```
74 {
```

```
75     cout << endl << "global x is " << x
```

```
76         << " on entering useGlobal" << endl;
```

```
77     x *= 10;
```

```
78     cout << "global x is " << x
```

```
79         << " on exiting useGlobal" << endl;
```

```
80
```

```
81 } // end function useGlobal
```

Hàm này không khai báo biến nào. Nó sử dụng biến toàn cục **x** đã được khai báo tại đầu chương trình.

fig03_12.cpp

(5 of 5)

fig03_12.cpp

output (1 of 2)

```
local x in main's outer scope is 5
```

```
local x in main's inner scope is 7
```

```
local x in main's outer scope is 5
```

```
local x is 25 on entering useLocal
```

```
local x is 26 on exiting useLocal
```

```
local static x is 50 on entering useStaticLocal
```

```
local static x is 51 on exiting useStaticLocal
```

```
global x is 1 on entering useGlobal
```

```
global x is 10 on exiting useGlobal
```

local x is 25 on entering useLocal

local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal

local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal

global x is 100 on exiting useGlobal

local x in main is 5

fig03_12.cpp
output (2 of 2)

3.12 Đệ quy – Recursion

- Các hàm đệ quy – Recursive functions
 - các hàm tự gọi chính mình
 - chỉ giải quyết một trường hợp cơ bản (base case)
- Nếu không phải trường hợp cơ bản
 - Chia bài toán thành các bài toán nhỏ hơn
 - Gọi bản sao mới của hàm để giải quyết vấn đề nhỏ hơn (gọi đệ quy (recursive call) hoặc bước đệ quy(recursive step))
 - hội tụ dần dần về trường hợp cơ bản
 - hàm gọi chính nó tại lệnh return
 - Cuối cùng, trường hợp cơ bản được giải quyết
 - câu trả lời đi ngược lên, giải quyết toàn bộ bài toán

3.12 Độ quy

- Ví dụ: tính giai thừa (factorial)

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

- Quan hệ đệ quy ($n! = n * (n - 1)!$)

$$5! = 5 * 4!$$

$$4! = 4 * 3! \dots$$

- Trường hợp cơ bản ($1! = 0! = 1$)

fig03_14.cpp
(1 of 2)

```
1 // Fig. 3.14: fig03_14.cpp
2 // Recursive factorial function.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 unsigned long factorial( unsigned long ); // function prototype
13
14 int main()
15 {
16     // Loop 10 times. During each iteration, calculate
17     // factorial( i ) and display result.
18     for ( int i = 0; i <= 10; i++ )
19         cout << setw( 2 ) << i << "! = "
20             << factorial( i ) << endl;
21
22     return 0; // indicates successful termination
23
24 } // end main
```

Kiểu dữ liệu **unsigned long** có thể lưu số nguyên trong khoảng từ 0 đến 4 tỷ.

```
25
26 // recursive definition of function factorial
27 unsigned long factorial( unsigned long number )
28 {
29     // base case
30     if ( number <= 1 )
31         return 1;
32
33     // recursive step
34     else
35         return number * factorial( number - 1 );
36
37 } // end function factorial
```

Trường hợp cơ bản xảy ra khi ta có 0! hoặc 1!.

Mọi trường hợp khác phải được chia nhỏ (bước đệ qui).

fig03_14.cpp
(2 of 2)

fig03_14.cpp
output (1 of 1)

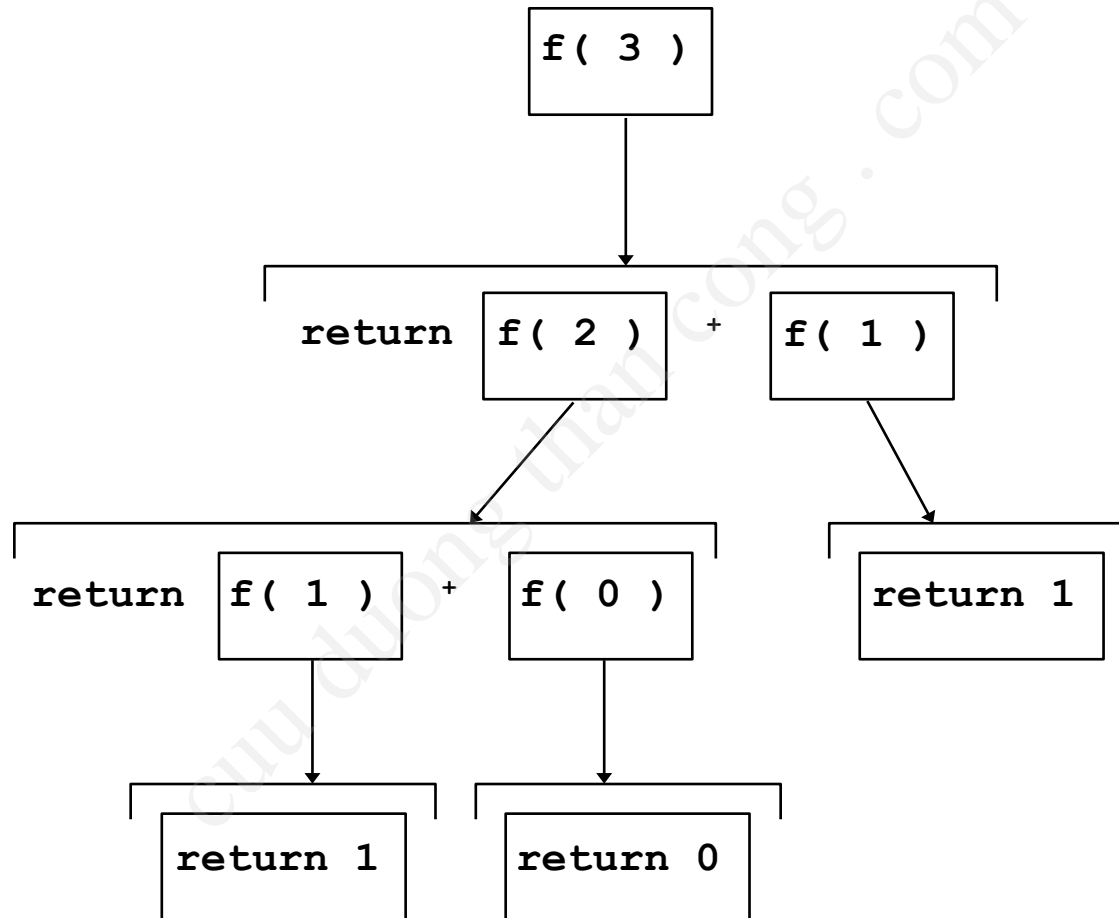
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

3.13 Ví dụ sử dụng đệ quy: chuỗi Fibonacci

- Chuỗi Fibonacci: 0, 1, 1, 2, 3, 5, 8...
 - Mỗi số là tổng của hai số đứng liền trước
 - Ví dụ một công thức đệ quy:
 - $fib(n) = fib(n-1) + fib(n-2)$
- Mã C++ cho hàm Fibonacci

```
long fibonacci( long n )  
{  
    if ( n == 0 || n == 1 )    // base case  
        return n;  
    else  
        return fibonacci( n - 1 ) +  
               fibonacci( n - 2 );  
}
```

3.13 Ví dụ sử dụng đệ quy: chuỗi Fibonacci



3.13 Ví dụ sử dụng đệ quy: chuỗi Fibonacci

- Thứ tự thực hiện
 - `return fibonacci(n - 1) + fibonacci(n - 2);`
- Không xác định hàm nào được thực hiện trước
 - C++ không qui định
 - Chỉ có các phép `&&`, `||` và `?:` đảm bảo thứ tự thực hiện từ trái sang phải
- Các lời gọi hàm đệ quy
 - Mỗi tầng đệ quy nhân đôi số lần gọi hàm
 - số thứ 30 cần $2^{30} \sim 4$ tỷ lời gọi hàm
 - Độ phức tạp lũy thừa (Exponential complexity)

fig03_15.cpp
(1 of 2)

Các số Fibonacci tăng rất nhanh và đều là số không âm. Do đó, ta dùng kiểu **unsigned long**.

```
1 // Fig. 3.15: fig03_15.cpp
2 // Recursive fibonacci function.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 unsigned long fibonacci( unsigned long ); // func
10
11 int main()
12 {
13     unsigned long result, number;
14
15     // obtain integer from user
16     cout << "Enter an integer: ";
17     cin >> number;
18
19     // calculate fibonacci value for number input by user
20     result = fibonacci( number );
21
22     // display result
23     cout << "Fibonacci(" << number << ") = " << result << endl;
24
25     return 0; // indicates successful termination
```

```

26
27 } // end main
28
29 // recursive definition of function fibonacci
30 unsigned long fibonacci( unsigned long n )
31 {
32     // base case
33     if ( n == 0 || n == 1 )
34         return n;
35
36     // recursive step
37     else
38         return fibonacci( n - 1 ) + fibonacci( n - 2 );
39
40 } // end function fibonacci

```

fig03_15.cpp

(2 of 2)

fig03_15.cpp

output (1 of 2)

Enter an integer: 0

Fibonacci(0) = 0

Enter an integer: 1

Fibonacci(1) = 1

Enter an integer: 2

Fibonacci(2) = 1

Enter an integer: 3

Fibonacci(3) = 2

Enter an integer: 4

Fibonacci(4) = 3

Enter an integer: 5

Fibonacci(5) = 5

Enter an integer: 6

Fibonacci(6) = 8

Enter an integer: 10

Fibonacci(10) = 55

Enter an integer: 20

Fibonacci(20) = 6765

Enter an integer: 30

Fibonacci(30) = 832040

Enter an integer: 35

Fibonacci(35) = 9227465

fig03_15.cpp
output (2 of 2)

3.14 So sánh Độ quy và Vòng lặp

- Lặp
 - Vòng lặp (Iteration): lặp tường minh
 - Độ quy: các lời gọi hàm được lặp đi lặp lại
- Kết thúc
 - Vòng lặp: điều kiện lặp thất bại
 - Độ quy: gặp trường hợp cơ bản
- Cả hai đều có thể lặp vô tận
- Cân đối giữa hiệu quả chương trình (vòng lặp) và công nghệ phần mềm tốt (độ quy)
 - vòng lặp chạy nhanh hơn
 - độ quy trong sáng hơn

3.15 Hàm với danh sách tham số rỗng

- Danh sách tham số rỗng
 - dùng từ khóa **void** hoặc để danh sách rỗng
 - dành cho hàm không lấy đối số
 - thí dụ: hàm **print** không lấy đối số và không trả về giá trị nào
 - `void print() ;`
 - `void print(void) ;`

```
1 // Fig. 3.18: fig03_18.cpp
2 // Functions that take no arguments.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void function1();           // function prototype
9 void function2( void );    // function prototype
10
11 int main()
12 {
13     function1();           // call function1 with no arguments
14     function2();           // call function2 with no arguments
15
16     return 0;              // indicates successful termination
17
18 } // end main
19
```

fig03_18.cpp
(1 of 2)

```
20 // function1 uses an empty parameter list to specify that
21 // the function receives no arguments
22 void function1()
23 {
24     cout << "function1 takes no arguments" << endl;
25
26 } // end function1
27
28 // function2 uses a void parameter list to specify that
29 // the function receives no arguments
30 void function2( void )
31 {
32     cout << "function2 also takes no arguments" << endl;
33
34 } // end function2
```

fig03_18.cpp
(2 of 2)

fig03_18.cpp
output (1 of 1)

```
function1 takes no arguments
function2 also takes no arguments
```

3.16 Hàm Inline

- Hàm inline
 - Từ khóa **inline** đặt trước hàm
 - Yêu cầu trình biên dịch sao chép mã vào chương trình thay cho việc tạo lời gọi hàm
 - Giảm chi phí gọi hàm (function-call overhead)
 - Trình biên dịch có thể bỏ qua **inline**
 - Tốt đối với các hàm nhỏ, hay dùng

- Ví dụ

```
inline double cube( const double s )  
{ return s * s * s; }
```

- **const** cho trình biên dịch biết rằng hàm không sửa đổi **s**
 - được nói đến trong các chương 6-7

```
1 // Fig. 3.19: fig03_19.cpp
2 // Using an inline function to calculate.
3 // the volume of a cube.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 // Definition of inline function cube. Definition of function
11 // appears before function is called, so a function prototype
12 // is not required. First line of function definition acts as
13 // the prototype.
14 inline double cube( const double side )
15 {
16     return side * side * side; // calculate cube
17
18 } // end function cube
19
```

fig03_19.cpp
(1 of 2)

```
20 int main()
21 {
22     cout << "Enter the side length of your cube: ";
23
24     double sideValue;
25
26     cin >> sideValue;
27
28     // calculate cube of sideValue and display result
29     cout << "Volume of cube with side "
30         << sideValue << " is " << cube( sideValue ) << endl;
31
32     return 0; // indicates successful termination
33
34 } // end main
```

fig03_19.cpp
(2 of 2)

fig03_19.cpp
output (1 of 1)

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

3.17 Tham chiếu và Tham số là tham chiếu

- Các tham chiếu là các biệt danh (alias) của các biến khác

- chỉ tới cùng một biến
- có thể được dùng bên trong một hàm

```
int count = 1; // khai báo biến nguyên count
int &cRef = count; // tạo cRef là một biệt danh của count
++cRef; // tăng count (sử dụng biệt danh của count)
```

- Các tham chiếu phải được khởi tạo khi khai báo
 - Nếu không, trình biên dịch báo lỗi
 - Tham chiếu lạc (Dangling reference)
 - tham chiếu tới biến không xác định


```
1 // Fig. 3.21: fig03_21.cpp
2 // References must be initialized.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int x = 3;
11
12     // y refers to (is an alias for) x
13     int &y = x;
14
15     cout << "x = " << x << endl << "y = " << y << endl;
16     y = 7;
17     cout << "x = " << x << endl << "y = " << y << endl;
18
19     return 0; // indicates successful termination
20
21 } // end main
```

y được khai báo là một tham chiếu tới x.

fig03_21.cpp
(1 of 1)

fig03_21.cpp
output (1 of 1)

```
x = 3
y = 3
x = 7
y = 7
```

fig03_22.cpp
(1 of 1)

fig03_22.cpp
(1 of 1)

Lỗi biên dịch – tham chiếu không được khởi tạo.

```
1  // Fig. 3.22: fig03_22.cpp
2  // References must be initialized.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     int x = 3;
11     int &y;    // Error: y must be initialized
12
13     cout << "x = " << x << endl << "y = " << y << endl;
14     y = 7;
15     cout << "x = " << x << endl << "y = " << y << endl;
16
17     return 0;  // indicates successful termination
18
19 } // end main
```

Borland C++ command-line compiler error message:

Error E2304 Fig03_22.cpp 11: Reference variable 'y' must be
initialized- in function main()

Microsoft Visual C++ compiler error message:

D:\cpphttp4_examples\ch03\Fig03_22.cpp(11) : error C2530: 'y' :
references must be initialized

3.17 Tham chiếu và Tham số là tham chiếu

- Gọi bằng giá trị - Call by value
 - Bản sao của dữ liệu được truyền cho hàm
 - Thay đổi đối với bản sao không ảnh hưởng tới dữ liệu gốc
 - Ngăn chặn các hiệu ứng phụ không mong muốn
- Gọi bằng tham chiếu - Call by reference
 - Hàm có thể truy nhập trực tiếp tới dữ liệu gốc
 - Các thay đổi thể hiện tại dữ liệu gốc

3.17 Tham chiếu và Tham số là tham chiếu

- Tham số tham chiếu - Reference parameter
 - Ý nghĩa: Là biệt danh (alias) của biến được truyền vào lời gọi hàm
 - 'truyền tham số bằng tham chiếu' hay 'truyền tham chiếu'
 - Cú pháp: Đặt ký hiệu **&** sau kiểu dữ liệu tại prototype của hàm
 - **void myFunction(int &data)**
 - có nghĩa “**data** là một tham chiếu tới một biến kiểu **int**”
 - dạng của lời gọi hàm không thay đổi
 - tuy nhiên dữ liệu gốc khi được truyền bằng tham chiếu có thể bị sửa đổi
- Con trỏ (chương 5)
 - Một cách truyền tham chiếu khác

fig03_20.cpp
(1 of 2)

```
1 // Fig. 3.20: fig03_20.cpp
2 // Comparing pass-by-value and pass-by-reference
3 // with references.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 int squareByValue( int ); // func
10 void squareByReference( int & ); // function prototype
11
12 int main()
13 {
14     int x = 2;
15     int z = 4;
16
17     // demonstrate squareByValue
18     cout << "x = " << x << " before squareByValue\n";
19     cout << "Value returned by squareByValue: "
20         << squareByValue( x ) << endl;
21     cout << "x = " << x << " after squareByValue\n" << endl;
22
```

Lưu ý ký hiệu & có nghĩa
truyền tham chiếu (pass-by-
reference).

```
23 // demonstrate squareByReference
24 cout << "z = " << z << " before squareByReference" << endl;
25 squareByReference( z );
26 cout << "z = " << z << " after squareByReference" << endl;
27
28 return 0; // indicates successful termination
29 } // end main
30
31 // squareByValue multiplies number by itself, stores the
32 // result in number and returns the new value of number
33 int squareByValue( int number )
34 {
35     return number *= number; // caller's argument not modified
36
37 } // end function squareByValue
38
39 // squareByReference multiplies numberRef by itself and
40 // stores the result in the variable to which numberRef
41 // refers in function main
42 void squareByReference( int &numberRef )
43 {
44     numberRef *= numberRef; // caller's argument modified
45
46 } // end function squareByReference
```

thay đổi **number**, nhưng đổi số gốc
(**x**) không bị thay đổi.

thay đổi **numberRef**, một biệt danh
của đổi số gốc. Do đó, **z** bị thay đổi.

```
x = 2 before squareByValue  
Value returned by squareByValue: 4  
x = 2 after squareByValue  
  
z = 4 before squareByReference  
z = 16 after squareByReference
```

fig03_20.cpp
output (1 of 1)

3.18 Các đối số mặc định

- Lời gọi hàm với các tham số được bỏ qua
 - Nếu không đủ số tham số, các vị trí ở bên phải nhất sẽ được nhận giá trị mặc định của chúng
 - Các giá trị mặc định
 - Có thể là hằng, biến toàn cục, hoặc các lời gọi hàm
- Đặt các giá trị mặc định tại function prototype

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

 - **myFunction(3)**
 - **x = 3**, **y** và **z** nhận giá trị mặc định (bên phải nhất)
 - **myFunction(3, 5)**
 - **x = 3**, **y = 5** còn **z** nhận giá trị mặc định

fig03_23.cpp
(1 of 2)

```
1 // Fig. 3.23: fig03_23.cpp
2 // Using default arguments.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // function prototype that specifies default arguments
9 int boxVolume( int length = 1, int width = 1, int height = 1 );
10
11 int main()
12 {
13     // no arguments--use default values for all dimensions
14     cout << "The default box volume is: " << boxVolume();
15
16     // specify length; default width and height
17     cout << "\n\nThe volume of a box with length 10,\n"
18         << "width 1 and height 1 is: " << boxVolume( 10 );
19
20     // specify length and width; default height
21     cout << "\n\nThe volume of a box with length 10,\n"
22         << "width 5 and height 1 is: " << boxVolume( 10, 5 );
23 }
```

Các giá trị mặc định được đặt trong function prototype.

Các lời gọi hàm thiếu một số đối số – Các đối số bên phải nhất nhận giá trị mặc định.

```

24 // specify all arguments
25 cout << "\n\nThe volume of a box with length 10,\n"
26     << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
27     << endl;
28
29 return 0; // indicates successful termination
30
31 } // end main
32
33 // function boxVolume calculates the volume of a box
34 int boxVolume( int length, int width, int height )
35 {
36     return length * width * height;
37
38 } // end function boxVolume

```

fig03_23.cpp
(2 of 2)

fig03_23.cpp
output (1 of 1)

The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100

3.19 Toán tử phạm vi đơn

- Toán tử phạm vi đơn (::)

Unitary Scope Resolution Operator

- Dùng để truy nhập biến toàn cục nếu biến địa phương có cùng tên
- Không cần thiết nếu các tên biến khác nhau
- Cách dùng :: **tên_biến**
y = ::x + 3;
- Nên tránh dùng các tên giống nhau cho các biến địa phương và toàn cục

fig03_24.cpp
(1 of 2)

```
1  // Fig. 3.24: fig03_24.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setprecision;
11
12 // define global constant PI
13 const double PI = 3.14159265358979;
14
15 int main()
16 {
17     // define local constant PI
18     const float PI = static_cast< float >( ::PI );
19
20     // display values of local and global PI constants
21     cout << setprecision( 20 )
22         << "   Local float value of PI = " << PI
23         << "\nGlobal double value of PI = " << ::PI << endl;
24
25     return 0; // indicates successful termination
```

Truy nhập **PI** toàn cục với **::PI**.

Chuyển đổi **PI** toàn cục thành một giá trị **float** cho **PI** địa phương. Ví dụ này cho thấy sự khác nhau giữa **float** và **double**.

26

27 } // end main

Borland C++ command-line compiler output:

Local float value of PI = 3.141592741012573242

Global double value of PI = 3.141592653589790007

Microsoft Visual C++ compiler output:

Local float value of PI = 3.1415927410125732

Global double value of PI = 3.14159265358979

fig03_24.cpp

(2 of 2)

fig03_24.cpp

output (1 of 1)

3.20 Chồng hàm

- Chồng hàm - Function overloading
 - Các hàm có cùng tên nhưng khác nhau về tham số
 - Nên thực hiện các nhiệm vụ tương tự
 - ví dụ, hàm tính bình phương cho **int** và hàm tính bình phương cho **float**

```
int square( int x) {return x * x;}  
float square(float x) { return x * x; }
```
- Các hàm chồng phân biệt nhau bởi chữ ký
 - Dựa vào tên và kiểu tham số (xét cả thứ tự)
 - Trình biên dịch đảm bảo gọi đúng hàm chồng được yêu cầu

fig03_25.cpp
(1 of 2)

Các hàm chồng có cùng tên
nhưng có tham số khác nhau

```
1 // Fig. 3.25: fig03_25.cpp
2 // Using overloaded functions.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // function square for int values
9 int square( int x )
10 {
11     cout << "Called square with int argument: " << x << endl;
12     return x * x;
13 }
14 // end int version of function square
15
16 // function square for double values
17 double square( double y )
18 {
19     cout << "Called square with double argument: " << y << endl;
20     return y * y;
21 }
22 // end double version of function square
23
```

```
24 int main()
25 {
26     int intResult = square( 7 );           // calls int version
27     double doubleResult = square( 7.5 ); // calls double version
28
29     cout << "\nThe square of integer 7 is " << intResult
30          << "\nThe square of double 7.5 is " << doubleResult
31          << endl;
32
33     return 0; // indicates successful termination
34
35 } // end main
```

fig03_25.cpp
(2 of 2)

fig03_25.cpp
output (1 of 1)

Tùy theo đối số được truyền vào (**int** hoặc **double**) để gọi hàm thích hợp.

Called square with int argument: 7
Called square with double argument: 7.5

The square of integer 7 is 49
The square of double 7.5 is 56.25

3.21 Khuôn mẫu hàm - Function Template

- Cách ngắn gọn để tạo các hàm chồng
 - Sinh các hàm riêng biệt cho các kiểu dữ liệu khác nhau
- Cú pháp
 - Bắt đầu bằng từ khóa **template**
 - các tham số kiểu hình thức trong cặp ngoặc **<>**
 - **typename** hoặc **class** (đồng nghĩa) đặt trước mỗi tham số kiểu
 - là đại diện cho các kiểu cài sẵn (ví dụ **int**) hoặc các kiểu dữ liệu người dùng
 - chỉ ra các kiểu dữ liệu cho đối số hàm, giá trị trả về, biến địa phương
 - Hàm được định nghĩa như bình thường, ngoại trừ việc sử dụng các kiểu hình thức

3.21 Khuôn mẫu hàm

- Ví dụ

```
template < typename T > // or template< class T >
T square( T value1 )
{
    return value1 * value1;
}
```

- **T** là một kiểu hình thức, được dùng làm tham số kiểu
 - hàm trên trả về giá trị thuộc cùng kiểu với tham số
- Tại lời gọi hàm, T được thay bằng kiểu dữ liệu thực
 - Nếu là **int**, mọi **T** trở thành **int**

```
int x;
int y = square(x);
```

fig03_27.cpp
(1 of 3)

```
1 // Fig. 3.27: fig03_27.cpp
2 // Using a function template.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 // definition of function template maximum
10 template < class T > // or template < typename T >
11 T maximum( T value1, T value2, T value3 )
12 {
13     T max = value1;
14
15     if ( value2 > max )
16         max = value2;
17
18     if ( value3 > max )
19         max = value3;
20
21     return max;
22
23 } // end function template maximum
24
```

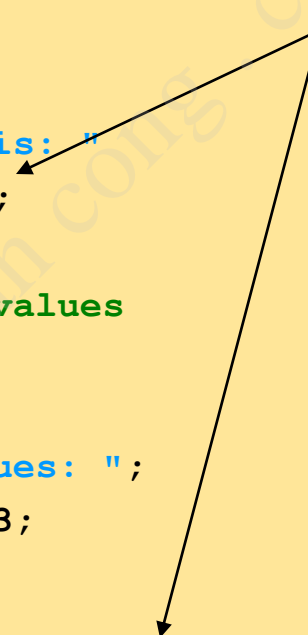
Tham số kiểu hình thức **T** là đại diện của kiểu dữ liệu được kiểm tra trong hàm **maximum**.

maximum mong đợi mọi tham số đều thuộc cùng một kiểu dữ liệu.

fig03_27.cpp
(2 of 3)

```
25 int main()
26 {
27     // demonstrate maximum with int values
28     int int1, int2, int3;
29
30     cout << "Input three integer values: ";
31     cin >> int1 >> int2 >> int3;
32
33     // invoke int version of maximum
34     cout << "The maximum integer value is: "
35           << maximum( int1, int2, int3 );
36
37     // demonstrate maximum with double values
38     double double1, double2, double3;
39
40     cout << "\n\nInput three double values: ";
41     cin >> double1 >> double2 >> double3;
42
43     // invoke double version of maximum
44     cout << "The maximum double value is: "
45           << maximum( double1, double2, double3 );
46
```

maximum được gọi với nhiều kiểu dữ liệu.



```

47 // demonstrate maximum with char values
48 char char1, char2, char3;
49
50 cout << "\n\nInput three characters: ";
51 cin >> char1 >> char2 >> char3;
52
53 // invoke char version of maximum
54 cout << "The maximum character value is: "
55     << maximum( char1, char2, char3 )
56     << endl;
57
58 return 0; // indicates successful termination
59
60 } // end main

```

fig03_27.cpp
(3 of 3)

fig03_27.cpp
output (1 of 1)

Input three integer values: 1 2 3

The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1

The maximum double value is: 3.3

Input three characters: A C B

The maximum character value is: C

3.21 Khuôn mẫu hàm

- Ví dụ

```
template < typename T, typename U >
T square( T value1, U value2 )
{
    std::cout << value2 << ":" << value1;
    return value1 * value1;
}

int x;
char c = 'a';
int y = square(x, c );
```

- char[] a = “skjhdjsdfh”;
- float z = square(2.3, a);