# Chương 0

# Ôn các kiến thức về cú pháp ngôn ngữ VC#

## 0.0 Dẫn nhập

Chương này sẽ tóm tắt lại 1 số kiến thức cơ bản về cú pháp của ngôn ngữ VC# hầu giúp các SV có góc nhìn tổng thể và hệ thống về ngôn ngữ VC#, nhờ đó có nhiều thuận lợi hơn trong việc học các kiến thức của môn học này.

## 0.1 Tổng quát về máy tính và ngôn ngữ VC#

Máy tính số là thiết bị đặc biệt, nó là thiết bị tổng quát hóa, nghĩa là có thể thực hiện nhiều công việc khác nhau. Ta có thể nói máy tính số là thiết bị vạn năng.

Vậy tại 1 thời điểm xác định, máy tính thực hiện công việc gì ? Nó không làm gì cả nếu con người không yêu cầu cụ thể nó.

Làm sao để con người có thể yêu cầu máy tính thực hiện 1 công việc nào đó ? Ta phải viết chương trình giải quyết công việc tương ứng rồi đưa vào máy và nhờ máy chạy dùm.

Viết chương trình là qui trình lớn và dài hạn gồm nhiều bước, trong đó các bước chính yếu là : xác định chính xác các chức năng của chương trình, phân tích cách giải quyết từng chức năng, tìm thuật giải chi tiết để giải quyết từng chức năng, đổi thuật giải chi tiết từ ngôn ngữ đời thường thành ngôn ngữ lập trình cho máy hiểu.

Ngôn ngữ lập trình là ngôn ngữ giao tiếp giữa người và máy. Học ngôn ngữ lập trình cũng giống như học ngôn ngữ tự nhiên, nghĩa là học tuần tự các thành phần của ngôn ngữ từ thấp đến cao như :

- Tập ký tự cơ bản
- Cú pháp xây dựng từ (word). Từ được dùng để đặt tên nhận dạng cho từng phần tử cấu thành chương trình như hằng gợi nhớ, biến, hàm chức năng, class đối tượng,…

- Cú pháp xây dựng biểu thức. Biểu thức (công thức toán học) miêu tả 1 quá trình tính toán tuần tự nhiều phép toán trên nhiều dữ liệu để tạo ra kết quả tính toán.
- Cú pháp xây dựng từng câu lệnh : có 2 loại câu lệnh : lệnh định nghĩa và lệnh thực thi :
  - Lệnh định nghĩa được dùng để định nghĩa và tạo mới phần tử cấu thành phần mềm.
  - Lệnh thực thi miêu tả 1 hành động cụ thể cần phải thực hiện.
- Cú pháp tổ chức 1 hàm chức năng
- Cú pháp tổ chức 1 class chức năng
- Cú pháp tổ chức 1 chương trình.

## 0.2 Tập ký tự cơ bản của ngôn ngữ VC#

Ngôn ngữ VC# hiểu và dùng tập ký tự Unicode. Cụ thể trên Windows, mỗi ký tự Unicode dài 2 byte (16 bit) => có 65536 ký tự Unicode khác nhau trên Windows.

Mặc dù vậy, VC# dùng chủ yếu các ký tự :

- chữ (a-z tiếng Anh), '_',
- ký tự số (0-9),
- khoảng trắng và các dấu ngăn như Tab (gióng cột), CR (quay về đầu dòng), LF (xuống dòng).
- các ký tự đặc biệt để miêu tả phép toán như +, -, *, /, =, !, (, )

Các ký tự khác, nhất là các ký tự có mã > 256 chỉ được dùng trong lệnh chú thích. Các ký tự có dấu tiếng Việt có mã từ 7840-7929.

## 0.3 Extended Backus-Naur Form (EBNF) notation

Ta sẽ dùng qui ước EBNF để miêu tả cú pháp xây dựng các phần tử của ngôn ngữ VC#. Cụ thể ta sẽ dùng các qui ước EBNF sau đây :

- #xN, trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự có mã thập lục phân tương ứng. Thí dụ ta viết #x3e để miêu tả ký tự >.
- [a-zA-Z], [#xN-#xN], trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự thuộc danh sách được liệt kê. Thí dụ ta viết [0-9] để miêu tả 1 ký tự số thập phân từ 0 đến 9.
- [^a-zA-Z], [^#xN-#xN], trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự không thuộc danh sách được liệt kê. Thí dụ ta viết [^0-9] để miêu tả 1 ký tự bất kỳ nhưng không phải là số thập phân từ 0 đến 9.
- [^abc], [^#xN#xN#xN], trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự không thuộc danh sách được liệt kê. Thí dụ ta viết [^<@] để miêu tả 1 ký tự bất kỳ nhưng không phải là < hay @.
- "string". Qui ước này miêu tả chuỗi ký tự có nội dung nằm trong 2 dấu nháy kép. Thí dụ ta viết "DHBK" để miêu tả chuỗi ký tự DHBK.
- 'string'. Qui ước này miêu tả chuỗi ký tự có nội dung nằm trong 2 dấu nháy đơn. Thí dụ ta viết 'DHBK' để miêu tả chuỗi ký tự DHBK.
- (expression). Qui ước này miêu tả kết quả của việc tính biểu thức. Thí dụ (DefStatement | ExeStatement) để miêu tả sự tồn tại của phần tử DefStatement hay ExeStatement.
- A? miêu tả có từ 0 tới 1 lần A. Thí dụ S? miêu tả có từ 0 tới 1 phần tử S.

- A+ miêu tả có từ 1 tới n lần A. Thí dụ S+ miêu tả có từ 1 tới n phần tử S.
- A* miêu tả có từ 0 tới n lần A. Thí dụ S* miêu tả có từ 0 tới n phần tử S.
- A B miêu tả phần tử A rồi tới phần tử B.
- A | B miêu tả chọn lựa A hay B.
- A - B miêu tả chuỗi thỏa A nhưng không thỏa B.
- /* ... */ miêu tả chuỗi chú thích.

## 0.4 Cú pháp định nghĩa tên nhận dạng (Name)

Mỗi phần tử trong chương trình đều được nhận dạng bởi 1 tên nhận dạng riêng biệt. Tên là chuỗi có ít nhất 1 ký tự, ký tự đầu là những ký tự thỏa luật NameStartChar, các ký tự còn lại thỏa luật NameChar. Cú pháp định nghĩa tên của VC# là :

Name ::= NameStartChar (NameChar)*
NameStartChar ::= [a-zA-Z_]
NameChar ::= NameStartChar | [0-9]

Dựa vào cú pháp trên, ta nói tên nhận dạng là 1 chuỗi từ 1 tới nhiều ký tự, ký tự đầu phải là ký tự chữ  hay dấu _, các ký tự còn lại có thể là chữ, số hay dấu _. Độ dài maximum của tên là 255.

Thí dụ System, Console, Writeln...

## 0.5 Cú pháp định nghĩa dấu ngăn (Seperator)

Cú pháp miêu tả các phần tử lớn hơn thường có điểm chung là phần tử lớn gồm tuần tự nhiều phần tử nhỏ hợp lại theo 1 thứ tự xác định.

Thường ta cần từ 1 tới n dấu ngăn nằm giữa các phần tử nhỏ kề nhau để ngăn chúng ra.  Cú pháp miêu tả chuỗi từ 1 đến nhiều ký tự ngăn cách là :

S ::= (#x20 | #x9 | #xD | #xA | Comment)+

Comment ::= InLineComment | OutofLineComment
InLineComment ::= "//" [^#xD#xA]*
OutofLineComment ::= "/*" (Char* - (Char* "*/" Char*)) "*/"

Thí dụ :

```
//đây là chú thích trên 1 dòng
/* còn đây là
chú thích trên nhiều dòng */
```

## 0.6 Cú pháp định nghĩa biểu thức

Ta đã biết trong toán học công thức là phương tiện miêu tả 1 qui trình tính toán nào đó trên các số.

Trong VC++ (hay ngôn ngữ lập trình khác), ta dùng biểu thức để miêu tả qui trình tính toán nào đó trên các dữ liệu  biểu thức cũng giống như công thức toán học, tuy nó tổng quát hơn (xử lý trên nhiều loại dữ liệu khác nhau) và phải tuân theo qui tắc cấu tạo khắt khe hơn công thức toán học.

Để hiểu được biểu thức, ta cần hiểu được các thành phần của nó :

- Các toán hạng : các biến, hằng dữ liệu,...
- Các toán tử tham gia biểu thức : +,-,*,/,...
- Qui tắc kết hợp toán tử và toán hạng để tạo biểu thức.
- Qui trình mà máy dùng để tính trị của biểu thức.
- Kiểu của biểu thức là kiểu của kết quả tính toán biểu thức.

### Các toán hạng :

Biểu thức cơ bản là phần tử nhỏ nhất cấu thành biểu thức bất kỳ. Một trong các phần tử sau được gọi là biểu thức cơ bản :

- Biến, thuộc tính của đối tượng
- Hằng gợi nhớ,
- Giá trị dữ liệu cụ thể thuộc kiểu nào đó (nguyên, thực,..)

- Lời gọi hàm,
- 1 biểu thức được đóng trong 2 dấu ().

Qui trình tạo biểu thức là qui trình lặp đệ qui : ta kết hợp từng toán tử với các toán hạng của nó, rồi đóng trong 2 dấu () để biến nó trở thành biểu thức cơ bản, rồi dùng nó như 1 toán hạng để xây dựng biểu thức lớn hơn và phức tạp hơn...

**Các phép toán :**

Dựa theo số toán hạng tham gia, có 3 loại toán tử thường dùng nhất :

- toán tử 1 ngôi : chỉ cần 1 toán hạng. Ví dụ toán tử '-' để tính phần âm của 1 đại lượng.
- toán tử 2 ngôi : cần dùng 2 toán hạng. Ví dụ toán tử '*' để tính tích của 2 đại lượng.
- toán tử 3 ngôi : cần dùng 3 toán hạng. Ví dụ toán tử 'c?v1:v2' để kiểm tra điều kiện c hầu lấy kết quả v1 hay v2.

VC# thường dùng các ký tự đặc biệt để miêu tả toán tử. Ví dụ :

- toán tử '+' : cộng 2 đại lượng.
- toán tử '-' : trừ đại lượng 2 ra khỏi đại lượng 1.
- toán tử '*' : nhân 2 đại lượng.
- toán tử '/' : chia đại lượng 1 cho đại lượng 2...

Trong vài trường hợp, VC# dùng cùng 1 ký tự đặc biệt để miêu tả nhiều toán tử khác nhau. Trong trường hợp này, ngữ cảnh sẽ được dùng để giải quyết nhằm lẫn.

Ngữ cảnh thường là kiểu của các toán hạng tham gia hoặc do thiếu toán hạng thì toán tử được hiểu là toán tử 1 ngôi.

Thí dụ :

```
-x      // - là phép toán 1 ngôi
a-b // - là phép toán 2 ngôi
```

Trong vài trường hợp khác, VC# dùng cùng chuỗi nhiều ký tự để miêu tả 1 toán tử. Thí dụ :

```
a >= b      // >= là toán tử so sánh lớn hơn hay bằng
a++         // ++ là toán tử tăng 1 đơn vị
a == b      // == là toán tử so sáng bằng (không phải là toán tử gán)
```

**Cú pháp miêu tả các giá trị cụ thể :**

- Giá trị luận lý : true | false
- Giá trị thập phân nguyên : (+|-)? (decdigit)+ (Vd. 125, -548)
- Giá trị thập lục phân nguyên : (+|-)? "0x" (hexdigit)+ (0xFF)
- Giá trị bát phân nguyên : (+|-)? "0" (ocdigit)+ (0577)
- Giá trị nhị phân nguyên : (+|-)? (bidigit)+ "b" (101110b)
- Giá trị thập phân thực :

(+|-)? (decdigit)+ ("." (decdigit)*)? ("E" (+|-)? (decdigit)+)?

3.14159, 0.31459e1,-83.1e-9,...

- Giá trị chuỗi : "Nguyen Van A"

  "\"Nguyen Van A\""

Lưu ý dùng ký tự '\' để thực hiện cơ chế 'escape' dữ liệu hầu giải quyết nhầm lẫn.

## 0.7 Qui trình tính biểu thức :

Một biểu thức có thể chức nhiều phép toán, qui trình tính toán biểu thức như sau : duyệt từ trái sang phải, mỗi lần gặp 1 phép toán (ta gọi là CurrentOp) thì phải nhìn trước toán tử đi ngay sau nó (SuccessorOp), so sánh độ ưu tiên của 2 toán tử và ra quyết định như sau :

- nếu không có SuccessorOp thì tính ngay toán tử CurrentOp (trên 1, 2 hay 3 toán hạng của nó).
- nếu toán tử CurrentOp có độ ưu tiên cao hơn toán tử SuccessorOp thì tính ngay toán tử CurrentOp (trên 1, 2 hay 3 toán hạng của nó).
- nếu toán tử CurrentOp có độ ưu tiên bằng toán tử SuccessorOp và kết hợp trái thì tính ngay toán tử CurrentOp (trên 1, 2 hay 3 toán hạng của nó).
- các trường hợp còn lại thì cố gắng thực hiện toán tử SuccessorOp trước. Việc cố gắng này cũng phải tuân theo các qui định trên,...
- Khi toán tử SussesorOp được thực hiện xong thì toán tử ngay sau SuccessorOp trở thành toán tử đi ngay sau CurrentOp  việc kiểm tra xem CurrentOp có được thực hiện hay không sẽ được lặp lại.

Bảng liệt kê độ ưu tiên của các toán tử từ trên xuống = từ cao xuống thấp :

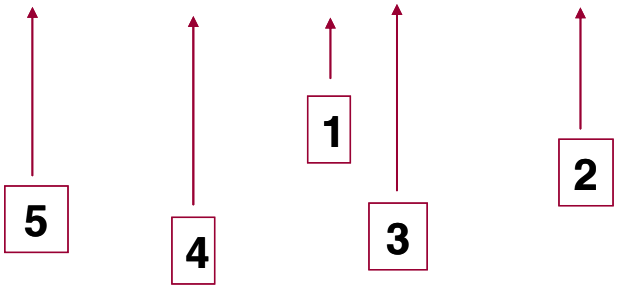
| Operator | Name or Meaning | Associativity |
|---|---|---|
| [ ] | Array subscript | Left to right |
| ( ) | Function call | Left to right |
| ( ) | Conversion | None |
| . | Member selection (object) | Left to right |
| -> | Member selection (pointer) | Left to right |
| ++ | Postfix increment | None |
| -- | Postfix decrement | None |
| new | Allocate object | None |
| typeof | Type of | |
| checked | | |
| unchecked | | |
| ++ | Prefix increment | None |
| -- | Prefix decrement | None |

| | | |
|---|---|---|
| + | Unary plus | None |
| – | Arithmetic negation (unary) | None |
| ! | Logical NOT | None |
| ~ | Bitwise complement | None |
| & | Address of | None |
| sizeof ( ) | Size of type | None |
| typeid( ) | type name | None |
| (type) | Type cast (conversion) | Right to left |
| true | true | None |
| false | false | None |
| * | Multiplication | Left to right |
| / | Division | Left to right |
| % | Remainder (modulus) | Left to right |
| + | Addition | Left to right |
| – | Subtraction | Left to right |
| << | Left shift | Left to right |
| >> | Right shift | Left to right |
| < | Less than | Left to right |
| > | Greater than | Left to right |
| <= | Less than or equal to | Left to right |
| >= | Greater than or equal to | Left to right |
| is | | |
| as | | |
| == | Equality | Left to right |
| != | Inequality | Left to right |
| & | Bitwise AND | Left to right |
| ^ | Bitwise exclusive OR | Left to right |
| \| | Bitwise OR | Left to right |
| && | Logical AND | Left to right |
| \|\| | Logical OR | Left to right |
| e1?e2:e3 | Conditional | Right to left |

| | | |
|---|---|---|
| = | Assignment | Right to left |
| *= | Multiplication assignment | Right to left |
| /= | Division assignment | Right to left |
| %= | Modulus assignment | Right to left |
| += | Addition assignment | Right to left |
| –= | Subtraction assignment | Right to left |
| <<= | Left-shift assignment | Right to left |
| >>= | Right-shift assignment | Right to left |
| &= | Bitwise AND assignment | Right to left |
| \|= | Bitwise inclusive OR assignment | Right to left |
| ^= | Bitwise exclusive OR assignment | Right to left |
| ?? | | |
| , | Comma | Left to right |

Thí dụ :

```
dblDv = dblDv + intpn * d * pow(10,-bytPosDigit);
```




## 0.8 Các lệnh định nghĩa thành phần phần mềm

### Định nghĩa hằng gợi nhớ

Cú pháp định nghĩa hằng gợi nhớ cơ bản :

ConstDef ::= "const" S TName S Name S? "=" S? Expr S? ";"

Thí dụ :

```
const double PI = 3.1416;
```

**Định nghĩa biến cục bộ trong hàm**

Cú pháp định nghĩa biến cục bộ trong hàm :

VarDef ::= TName S Name (S? "=" S? Expr S?)? ";"

Thí dụ :

double epsilon = 0.000001;

**Định nghĩa kiểu người dùng** (học chi tiết trong môn Kỹ thuật lập trình và các chương sau của môn này)

**Định nghĩa hàm hay tác vụ chức năng** (học chi tiết trong môn Kỹ thuật lập trình và các chương sau của môn này)

**Định nghĩa chương trình** (học chi tiết trong môn Kỹ thuật lập trình và các chương sau của môn này)

## 0.9 Các lệnh thực thi

Ta đã biết giải thuật để giải quyết 1 vấn đề nào đó là trình tự các công việc nhỏ hơn, nếu ta thực hiện đúng trình tự các công việc nhỏ hơn này thì sẽ giải quyết được vấn đề lớn.

VC# (hay ngôn ngữ lập trình khác) cung cấp 1 tập các lệnh thực thi, mỗi lệnh thực thi được dùng để miêu tả 1 công việc nhỏ trong 1 giải thuật với ý tưởng chung như sau :

Nếu tồn tại lệnh thực thi miêu tả được công việc nhỏ của giải thuật thì ta dùng lệnh thực thi này để miêu tả nó.

Nếu công việc nhỏ của thuật giải vẫn còn quá phức tạp và không có lệnh thực thi nào miêu tả được thì ta dùng lệnh gọi hàm (function, method) trong đó hàm là trình tự các lệnh thực hiện công việc nhỏ này...

Hầu hết các lệnh thực thi đều có chứa biểu thức và dùng kết quả của biểu thức này để quyết định công việc kế tiếp cần được thực hiện $\Rightarrow$ ta thường gọi các lệnh thực thi là các cấu trúc điều khiển.

Để dễ học, dễ nhớ và dễ dùng, VC# (cũng như các ngôn ngữ khác) chỉ cung cấp 1 số lượng rất nhỏ các lệnh thực thi :

Nhóm lệnh không điều khiển :

- Lệnh gán dữ liệu vào 1 biến.

Nhóm lệnh tạo quyết định :

- Lệnh kiểm tra điều kiện luận lý if ... else ...
- Lệnh kiểm tra điều kiện số học switch

Nhóm lệnh lặp :

- Lệnh lặp : while
- Lệnh lặp : for
- Lệnh lặp : do ... while

Nhóm lệnh gọi hàm :

- Lệnh gọi hàm
- Lệnh thoát khỏi cấu trúc điều khiển : break
- Lệnh thoát khỏi hàm : return

**Lệnh gán :**

Là lệnh được dùng nhiều nhất trong chương trình, chức năng của lệnh này là gán giá trị dữ liệu vào 1 vùng nhớ để lưu trữ hầu sử dụng lại nó sau đó. Cú pháp :

lvar S? "=" S? Expr S? ";"

biểu thức Expr bên phải sẽ được tính để tạo ra kết quả (1 giá trị cụ thể thuộc 1 kiểu cụ thể), giá trị này sẽ được gán vào ô nhớ do lvar qui định. Trước khi gán, VC# sẽ kiểm tra kiểu của 2 phần tử (qui tắc kiểm tra sẽ được trình bày sau).

lvar có thể là biến đơn (intTuoi), phần tử của biến array (matran[2,3]), thuộc tính của đối tượng (rect.dorong).

Thí dụ :

```
x1 = (-b-sqrt(delta))/2/a;
```

**Lệnh kiểm tra điều kiện luận lý if ... else :**

cho phép dựa vào kết quả luận lý (tính được từ 1 biểu thức luận lý) để quyết định thi hành 1 trong 2 nhánh lệnh. Sau khi thực hiện 1 trong 2 nhánh lệnh, chương trình sẽ tiếp tục thi hành lệnh ngay sau lệnh IF. Cú pháp :

"if" S? "(" S? Expr S? ")" S? Statement S?
("else" S Statement)?

Thí dụ :

```
if (delta <0) //báo sai
    System.Console.Writeln ("Phuong trinh vo nghiem");
else { //tính 2 nghiệm
     x1 = (-b-sqrt(delta))/2/a;
     x2 = (-b+sqrt(delta))/2/a;
   }
```

**Lệnh kiểm tra điều kiện số học switch :**

cho phép dựa vào kết quả số học (tính được từ 1 biểu thức số học) để quyết định thi hành 1 trong n nhánh lệnh. Sau khi thực hiện 1 trong n nhánh lệnh, chương trình sẽ tiếp tục thi hành lệnh ngay sau lệnh switch. Cú pháp :

"switch" S? "(" Expr S? ")" S? "{" S?
"case" S expr1 S? ":" S? Statement*
"case" S expr2 S? ":" S? Statement*
...
"case" S exprn S? ":" S? Statement*
("default" S? ":" S? Statement*)?
S? "}"

Thí dụ :

```
switch (diem) {
```

```
case 0 : case 1 : case 2 : case 3 : case 4 :
    Console.Writeln("Quá yếu"); break;
case 5 : case 6 :
    Console.Writeln("Trung bình"); break;
case 7 : case 8 :
    Console.Writeln("Khá"); break;
case 9 : case 10 :
    Console.Writeln("Giỏi"); break;
  }
```

**Lệnh lặp do... while :**

cho phép lặp thực hiện 1 công việc nào đó từ 1 tới n lần theo 1 điều kiện kiểm soát. Cú pháp :

"do" S Statement S? "while" S? "(" S? Expr S? ")" S? ";"

Thí dụ :

```
int i = 1;
long giaithua = 1;
do {
    i = i+1;
    giaithua = giaithua*i;
} while (i < n)
```

hay viết ngắn gọn hơn như sau :

```
int i = 1;
long giaithua = 1;
do giaithua *= (++i);
while (i < n);
```

**Lệnh lặp while :**

cho phép lặp thực hiện 1 công việc nào đó từ 0 tới n lần theo 1 điều kiện kiểm soát. Cú pháp :

"while" S? "(" S? Expr S? ")" S? Statement

Thí dụ :

```
int i = 1;
long giaithua = 1;
while (i < n) {
    i = i+1;
    giaithua = giaithua*i;
}
```

hay viết ngắn gọn hơn như sau :

```
int i = 1;
long giaithua = 1;
while (i < n) giaithua *= (++i);
```

**Lệnh lặp for :**

cho phép lặp thực hiện 1 công việc nào đó từ 0 tới n lần theo 1 điều kiện kiểm soát. Cú pháp :

"for" S? "(" S? init-expr? S? ";" S? cond-expr? ";" S? loop-expr? S? ")" S? Statement

Thí dụ :

```
int i;
long giaithua = 1;
for (i=2; i <=n; i++) {
    giaithua = giaithua*i;
}
```

hay viết ngắn gọn hơn như sau :

```
int i;
long giaithua = 1;
for (i=2; i <=n; i++) giaithua *= i;
```

**Các lệnh lồng nhau :**

Như ta đã thấy trong cú pháp của hầu hết các lệnh VC# đều có chứa thành phần Statement, đây là 1 lệnh thực thi VC# bất kỳ

⇒ ta gọi cú pháp định nghĩa lệnh VC# là đệ qui ⇒ tạo ra các lệnh VC# lồng nhau. Ta gọi cấp ngoài cùng là cấp 1, các lệnh hiện diện trong cú pháp của lệnh cấp 1 được gọi là lệnh cấp 2, các lệnh hiện diện trong cú pháp của lệnh cấp 2 được gọi là lệnh cấp 3,... Để dễ đọc, các lệnh cấp thứ i nên gióng cột nhờ i ký tự Tab.

Ví dụ : đoạn chương trình tính ma trận tổng của 2 ma trận

```
const int N = 100;
double[,] a, b, c;
...
for (i = 0; i <N; i++)    ' duyệt theo hàng
  for (j = 0; j<N; j++)   ' duyệt theo cột
        c[i,j] = a[i,j] + b[i,j];
```

**Vấn đề thoát đột ngột khỏi cấp điều khiển :**

Trong cú pháp của hầu hết các lệnh VC# đều có chứa thành phần Statement mà đa số là phát biểu kép chứa nhiều lệnh khác. Theo trình tự thi hành thông thường, các lệnh bên trong phát biểu kép sẽ được thực thi tuần tự, hết lệnh này đến lệnh khác cho đến lệnh cuối, lúc này thì việc thi hành lệnh cha mới kết thúc. Tuy nhiên trong 1 vài trạng thái thi hành đặc biệt, ta muốn thoát ra khỏi lệnh cha đột ngột chứ không muốn thực thi hết các lệnh con trong danh sách. Để phục vụ yêu cầu này, VC# cung cấp lệnh break với cú pháp đơn giản sau đây :

```
break;
```

Lưu ý lệnh break chỉ cho phép thoát khỏi cấp trong cùng (lệnh chứa lệnh break. Để thoát trực tiếp ra nhiều cấp 1 cách tự do, ta dùng lệnh goto với cú pháp :

```
goto stat_label;  //trong đó stat_label là nhãn của lệnh cần goto đến.
```

### Vấn đề thoát đột ngột khỏi hàm :

Như ta đã biết hàm là danh sách các lệnh thực thi để thực hiện 1 chức năng nào đó. Thông thường thì danh sách lệnh này sẽ được thực hiện từ đầu đến cuối rồi điều khiển sẽ được trả về lệnh gọi hàm này, tuy nhiên ta có quyền trả điều khiển về lệnh gọi hàm bất cứ đâu trong danh sách lệnh của hàm. Cú pháp lệnh trả điều khiển như sau :

"return" S? ";"  // nếu hàm có kiểu trả về là void

"return" S? "(" S? expr S? ")" S? ";"  // nếu hàm có kiểu trả về ≠ void

## 0.10 Kết chương

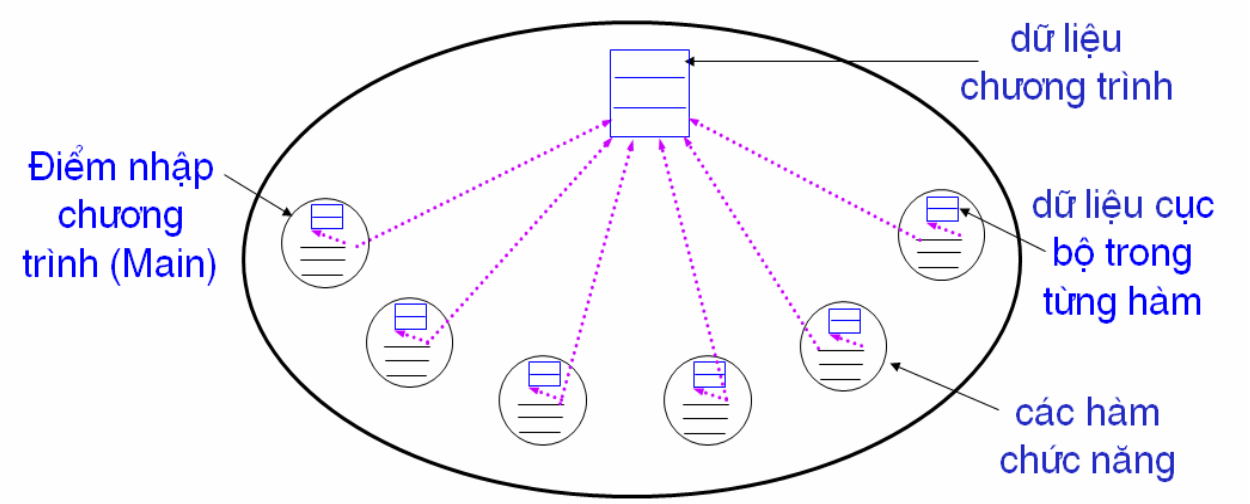
Chương này đã tóm tắt lại 1 số kiến thức cơ bản về cú pháp của ngôn ngữ VC# hầu giúp các SV có góc nhìn tổng thể và hệ thống về ngôn ngữ VC#, nhờ đó có nhiều thuận lợi hơn trong việc học các kiến thức của môn học này.

# Chương 1

# Các kiến thức cơ bản về lập trình C# đã học

## 1.1 Cấu trúc của 1 ứng dụng C# nhỏ

Trong môn kỹ thuật lập trình, chúng ta đã viết được 1 số ứng dụng C# nhỏ và đơn giản. Trong trường hợp này, 1 ứng dụng C# là 1 class gồm nhiều thuộc tính dữ liệu và nhiều hàm chức năng. Chương trình bắt đầu chạy từ hàm Main.




Xem đoạn chương trình giải phương trình bậc 2 ở chế độ text-mode sau đây :

```
using System;
namespace GPTB2 {
   class Program {
      //định nghĩa các biến cần dùng
      static double a, b, c;
      static double delta;
      static double x1, x2;
      //định nghĩa hàm nhập 3 thông số a,b,c của phương trình bậc 2
      static void NhapABC() {
         String buf;
         Console.Write("Nhập a : "); buf= Console.ReadLine();
         a = Double.Parse(buf);
```

```
    Console.Write("Nhập b : "); buf = Console.ReadLine();
    b = Double.Parse(buf);

    Console.Write("Nhập c : "); buf = Console.ReadLine();
    c = Double.Parse(buf);
}
//định nghĩa hàm tính nghiệm của phương trình bậc 2
static void GiaiPT()
{
    //tính biệt số delta của phương trình
    delta = b * b - 4 * a * c;
    if (delta >= 0) //nếu có nghiệm thực
    {
        x1 = (-b + Math.Sqrt(delta)) / 2 / a;
        x2 = (-b - Math.Sqrt(delta)) / 2 / a;
    }
}

//định nghĩa hàm xuất kết quả
static void XuatKetqua()
{
    if (delta < 0)
        //báo vô nghiệm
       Console.WriteLine("Phương trình vô nghiệm");
    else //báo có 2 nghiệm
    {
        Console.WriteLine("Phương trình có 2 nghiệm thực : ");
        Console.WriteLine("X1 = " + x1);
        Console.WriteLine("X2 = " + x2);
    }
}
//định nghĩa chương trình (hàm Main)
static void Main(string[] args)
{
```

```
            NhapABC();          //1. nhập a,b,c
            GiaiPT();                   //2. giải phương trình
            XuatKetqua();       //3. xuất kết quả
            //4. chờ người dùng ấn Enter để đóng cửa sổ Console lại.
            Console.Write("Ấn Enter để dừng chương trình : ");
            Console.Read();
        }
    }  //kết thúc class
} //kết thúc namespace
```

Quan sát cấu trúc của chương trình C# nhỏ phía trên, chúng ta có 1 số nhận xét sau :

1. Dữ liệu chương trình thường rất phong phú, đa dạng về chủng loại → Cơ chế định nghĩa kiểu dữ liệu nào được dùng để đảm bảo người lập trình có thể định nghĩa kiểu riêng mà ứng dụng của họ cần dùng ?
2. Nếu ứng dụng lớn chứa rất nhiều hàm chức năng và phải xử lý rất nhiều dữ liệu thì rất khó quản lý chúng trong 1 class đơn giản → cần 1 cấu trúc phù hợp để quản lý ứng dụng lớn.
3. Chương trình thường phải nhờ các hàm chức năng ở các class khác để hỗ trợ mình. Thí dụ ta đã gọi hàm Read, Write của class Console để nhập/xuất dữ liệu cho chương trình → Cơ chế nhờ vả nào được dùng để đảm bảo các thành phần trong ứng dụng không "quậy phá" nhau?

## 1.2 Kiểu dữ liệu cơ bản định sẵn

Các thuật giải chức năng của chương trình sẽ xử lý dữ liệu. Dữ liệu của chương trình thường rất phong phú, đa dạng về chủng loại. Trước hết ngôn ngữ C# (hay bất kỳ ngôn ngữ lập trình nào) phải định nghĩa 1 số kiểu được dùng phổ biến nhất trong các ứng dụng, ta gọi các kiểu này là "kiểu định sẵn".

Mỗi dữ liệu thường được để trong 1 biến. Phát biểu định nghĩa biến sẽ đặc tả các thông tin về biến đó :

- tên nhận dạng để truy xuất.
- kiểu dữ liệu để xác định các giá trị nào được lưu trong biến.
- giá trị ban đầu mà biến chứa...

Biến thuộc kiểu định sẵn sẽ chứa trực tiếp giá trị, thí dụ biến nguyên chứa trực tiếp các số nguyên, biến thực chứa trực tiếp các số thực → Ta gọi kiểu định sẵn là kiểu giá trị (value type) để phân biệt với kiểu tham khảo (reference type) trong lập trình hướng đối tượng ở các chương sau.

Kiểu tham khảo (hay kiểu đối tượng) sẽ được trình bày trong chương 2 trở đi. Đây là kiểu quyết định trong lập trình hướng đối tượng. Một biến đối tượng là biến có kiểu là tên interface hay tên class. Biến đối tượng không chứa trực tiếp đối tượng, nó chỉ chứa thông tin để truy xuất được đối tượng → Ta gọi kiểu đối tượng là kiểu tham khảo (reference type).

Sau đây là danh sách các tên kiểu cơ bản định sẵn :

- bool : kiểu luận lý, có 2 giá trị true và false.
- byte : kiểu nguyên dương 1 byte, có tầm trị từ 0 đến 255.
- sbyte : kiểu nguyên có dấu 1 byte, có tầm trị từ -128 đến 127.
- char : kiểu ký tự Unicode 2 byte, có tầm trị từ mã 0000 đến FFFF.
- short : kiểu nguyên có dấu 2 byte, tầm trị từ -32768 đến 32767.
- ushort : kiểu nguyên dương 2 byte, tầm trị từ 0 đến 65535.
- int : kiểu nguyên có dấu 4 byte, tầm trị từ -2,147,483,648 đến 2,147,483,647.

- uint : kiểu nguyên dương 4 byte, tầm trị từ 0 đến 4,294,967,295.
- long : kiểu nguyên có dấu 8 byte, tầm trị từ $-2^{63}$ đến $2^{63}-1$.
- ulong : kiểu nguyên dương 8 byte, tầm trị từ 0 đến $2^{64}-1$.
- float : kiểu thực chính xác đơn, dùng 4 byte để miêu tả 1 giá trị thực, có tầm trị từ $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$. Độ chính xác khoảng 7 ký số thập phân.
- double : kiểu thực chính xác kép, dùng 8 byte để miêu tả 1 giá trị thực, có tầm trị từ $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$. Độ chính xác khoảng 15 ký số thập phân.
- decimal : kiểu thực chính xác cao, dùng 16 byte để miêu tả 1 giá trị thực, có tầm trị từ $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$. Độ chính xác khoảng 28-29 ký số thập phân.
- object (Object) : kiểu đối tượng bất kỳ, đây là 1 class định sẵn đặc biệt.

### 1.3 Kiểu do người lập trình tự định nghĩa - Liệt kê

Ngoài các kiểu cơ bản định sẵn, C# còn hỗ trợ người lập trình tự định nghĩa các kiểu dữ liệu đặc thù trong từng ứng dụng.

Kiểu liệt kê bao gồm 1 tập hữu hạn và nhỏ các giá trị đặc thù cụ thể. Máy sẽ mã hóa các giá trị kiểu liệt kê thành kiểu byte, short...

```
//định nghĩa kiểu chứa các giá trị ngày trong tuần
enum DayInWeek {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
//định nghĩa kiểu chứa các giá trị ngày trong tuần
enum DayInWeek {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
//định nghĩa biến chứa các giá trị ngày trong tuần
DayInWeek day = DayInWeek.Tue;
//định nghĩa kiểu chứa các giá trị nguyên trong tầm trị đặc thù
enum ManAge : byte {Max = 130, Min = 0};
```

## 1.4 Kiểu do người lập trình tự định nghĩa - Record

Kiểu record bao gồm 1 tập hữu hạn các thông tin cần quản lý.

```
//định nghĩa kiểu miêu tả các thông tin của từng sinh viên cần quản lý
    public struct Sinhvien {
        public String hoten;
        public String diachi;
        //các field khác
    }
```

Thật ra kiểu struct là trường hợp đặc biệt của class đối tượng mà ta sẽ trình bày chi tiết từ chương 2.

## 1.5 Kiểu do người lập trình tự định nghĩa - Array

Trong trường hợp ta có nhiều dữ liệu cần xử lý thuộc cùng 1 kiểu (thường xảy ra), nếu ta định nghĩa từng biến đơn để miêu tả từng dữ liệu thì rất nặng nề, thuật giải xử lý chúng cũng gặp nhiều khó khăn. Trong trường hợp này, tốt nhất là dùng kiểu Array để quản lý nhiều dữ liệu cần xử lý. Array có thể là :

- array 1 chiều.
- array nhiều chiều.
- array "jagged".

### Array 1 chiều

```
int[] intList; //1.định nghĩa biến array là danh sách các số nguyên
//2. khi biết được số lượng, thiết lập số phần tử cho biến array
intList = new int[5];
//3. gán giá trị cho từng phần tử khi biết được giá trị của nó
intList[0] = 1; intList[1] = 3; intList[2] = 5;
intList[3] = 7; intList[4] = 9;
```

Nếu có đủ thông tin tại thời điểm lập trình, ta có thể viết lệnh định nghĩa biến array như sau :

```
int[] intList = new int[5] {1, 3, 5, 7, 9};
```
hay đơn giản :
```
int[] intList = new int[] {1, 3, 5, 7, 9};
```
hay đơn giản hơn nữa :
```
int[] intList = {1, 3, 5, 7, 9};
```

## Array nhiều chiều

```
int[,] matran; //1. định nghĩa biến array là ma trận các số nguyên
//2. khi biết được số lượng, thiết lập số phần tử cho biến array
matran = new int[3,2];
//3. gán giá trị cho từng phần tử khi biết được giá trị của nó
matran[0,0] = 1; matran[0,1] = 2; matran[1,0] = 3;
matran[1,1] = 4; matran[2,0] = 5; matran[2,1] = 6;
```

Nếu có đủ thông tin tại thời điểm lập trình, ta có thể viết lệnh định nghĩa biến array như sau :
```
int[,] matran = new int[3,2] {{1, 2}, {3, 4}, {5,6}};
```
hay đơn giản :
```
int[,] matran = new int[,] {{1, 2}, {3, 4}, {5,6}};
```
hay đơn giản hơn nữa :
```
int[,] matran = {{1, 2}, {3, 4}, {5,6}};
```

## Array "jagged"

Array "jagged" là array mà từng phần tử là array khác, các array được chứa trong array "jagged" có thể là array 1 chiều, n chiều hay là array "jagged' khác.

```
int[][] matran; //1. định nghĩa biến array "jagged"
//2. khi biết được số lượng, thiết lập số phần tử cho biến array
matran = new int[3][];
for (int i = 0; i < 3; i++) matran[i] = new int[2];
//3. gán giá trị cho từng phần tử khi biết được giá trị của nó
matran[0][0] = 1; matran[0][1] = 2; matran[1][0] = 3;
matran[1][1] = 4; matran[2][0] = 5; matran[2][1] = 6;
```

Nếu có đủ thông tin tại thời điểm lập trình, ta có thể viết lệnh định nghĩa biến array như sau :

```
int[][] array = new int [3][];
array[0] = new int[] {1, 2};
array[1] = new int[] {3, 4};
array[2] = new int[] {5,6};
```

hay đơn giản :

```
int[][] array = new int [][] {new int[]{1, 2}, new int[]{3, 4}, new int[] {5,6}};
```

hay đơn giản hơn nữa :

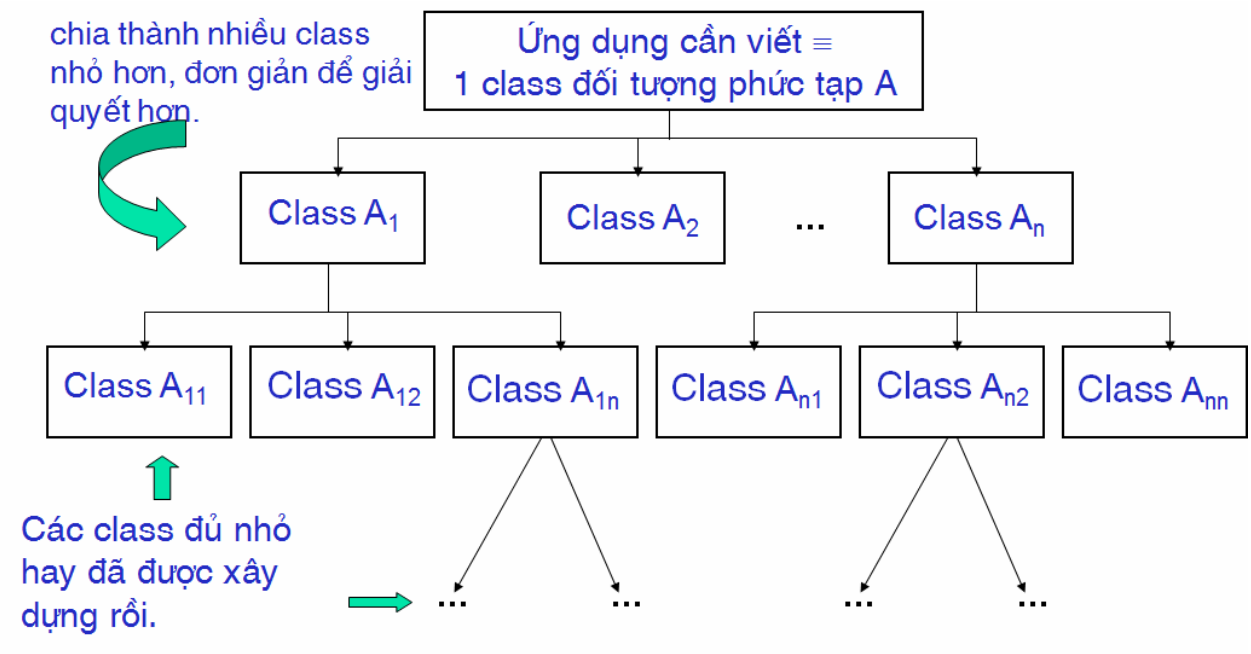
```
int[][] array = {new int[]{1, 2}, new int[]{3, 4}, new int[] {5,6}};
```

## 1.6 Phương pháp phân tích từ-trên-xuống

Như đã thấy ở slide trước, nếu ứng dụng lớn chứa rất nhiều hàm chức năng và phải xử lý rất nhiều dữ liệu thì rất khó quản lý chúng trong 1 class đơn giản → cần 1 cấu trúc phù hợp để quản lý ứng dụng lớn. Phương pháp được dùng phổ biến nhất là phương pháp phân tích top-down.

Nội dung của phương pháp này là phân rã class ứng dụng lớn thành n class nhỏ hơn (với n đủ nhỏ để việc phân rã đơn giản). Mỗi class nhỏ hơn, nếu còn quá phức tạp, lại được phân rã thành m class nhỏ hơn nữa (với m đủ nhỏ), cứ như vậy cho đến khi các class tìm được hoặc là class đã xây dựng rồi hoặc là class khá đơn giản, có thể xây dựng dễ dàng.

Hình vẽ sau đây cho thấy trực quan của việc phân tích top-down theo hướng đối tượng.




## 1.7 Namespace

Trên mỗi máy có 1 hệ thống quản lý các đối tượng được dùng bởi nhiều ứng dụng đang chạy. Mỗi ứng dụng lớn gồm rất nhiều class đối tượng khác nhau. Mỗi phần tử trong hệ thống tổng thể đều phải có tên nhận dạng duy nhất. Để đặt tên các phần tử trong hệ thống lớn sao cho mỗi phần tử có tên hoàn toàn khác nhau (để tránh tranh chấp, nhặp nhằng), C# (và các ngôn ngữ .Net khác) cung cấp phương tiện Namespace (không gian tên).

Namespace là 1 không gian tên theo dạng phân cấp : mỗi namespace sẽ chứa nhiều phần tử như struct, enum, class, interface và namespace con. Để truy xuất 1 phần tử trong namespace, ta phải dùng tên dạng phân cấp, thí dụ System.Windows.Forms.Button là tên của class Button, class miêu tả đối tượng giao diện button trong các form ứng dụng.

Trong file mã nguồn C#, để truy xuất 1 phần tử trong không gian tên khác, ta có thể dùng 1 trong 2 cách :

- dùng tên tuyệt đối dạng cây phân cấp. Thí dụ :

```
//định nghĩa 1 biến Button
System.Windows.Forms.Button objButton;
```

- dùng lệnh using <tên namespace>; Kể từ đây, ta nhận dạng phần tử bất kỳ trong namespace đó thông qua tên cục bộ. Thí dụ :

```
using System.Windows.Forms;
Button objButton; //định nghĩa 1 biến Button
TextBox objText;     //định nghĩa 1 biến TextBox
```

Microsoft đã xây dựng sẵn hàng ngàn class, interface chức năng phổ biến và đặt chúng trong khoảng 500 namespace khác nhau :

- System chứa các class và interface chức năng cơ bản nhất của hệ thống như Console (nhập/xuất văn bản), Math (các hàm toán học),..
- System.Windows.Forms chứa các đối tượng giao diện phổ dụng như Button, TextBox, ListBox, ComboBox,...
- System.Drawing chứa các đối tượng phục vụ xuất dữ liệu ra thiết bị vẽ như class Graphics, Pen, Brush,...
- System.IO chứa các class nhập/xuất dữ liệu ra file.
- System.Data chứa các class truy xuất database theo kỹ thuật ADO .Net.
- ...

## 1.8 Assembly

Ngoài khái niệm namespace là phương tiện đặt tên luận lý các phần tử theo dạng cây phân cấp thì C# còn cung cấp khái niệm assembly.

Assembly là phương tiện đóng gói vật lý nhiều phần tử. Một assembly là 1 file khả thi (EXE, DLL,...) chứa nhiều phần tử bên trong. Khi lập trình bằng môi trường Visual Studio .Net, ta sẽ tạo Project để quản lý việc xây dựng module chức năng nào đó (thư viện hay ứng dụng), mỗi project chứa nhiều file mã nguồn đặc tả

các thành phần trong Project đó. Khi máy dịch Project mã nguồn nó sẽ tạo ra file khả thi, ta gọi file này là 1 assembly.

Mỗi assembly có thể chứa nhiều phần tử nằm trong các namespace luận lý khác nhau. Ngược lại, 1 namespace có thể chứa nhiều phần tử mà về mặt vật lý chúng nằm trong các assembly khác nhau.

## 1.9 Kết chương

Chương này đã giới thiệu cấu trúc của chương trình VC# nhỏ và đơn giản gồm 1 số biến dữ liệu và 1 số hàm xử lý các biến dữ liệu, từ đó tổng kết lại các kiểu dữ liệu khác nhau có thể được dùng trong 1 chương trình, đặc biệt là các kiểu liệt kê, kiểu array, kiểu record.

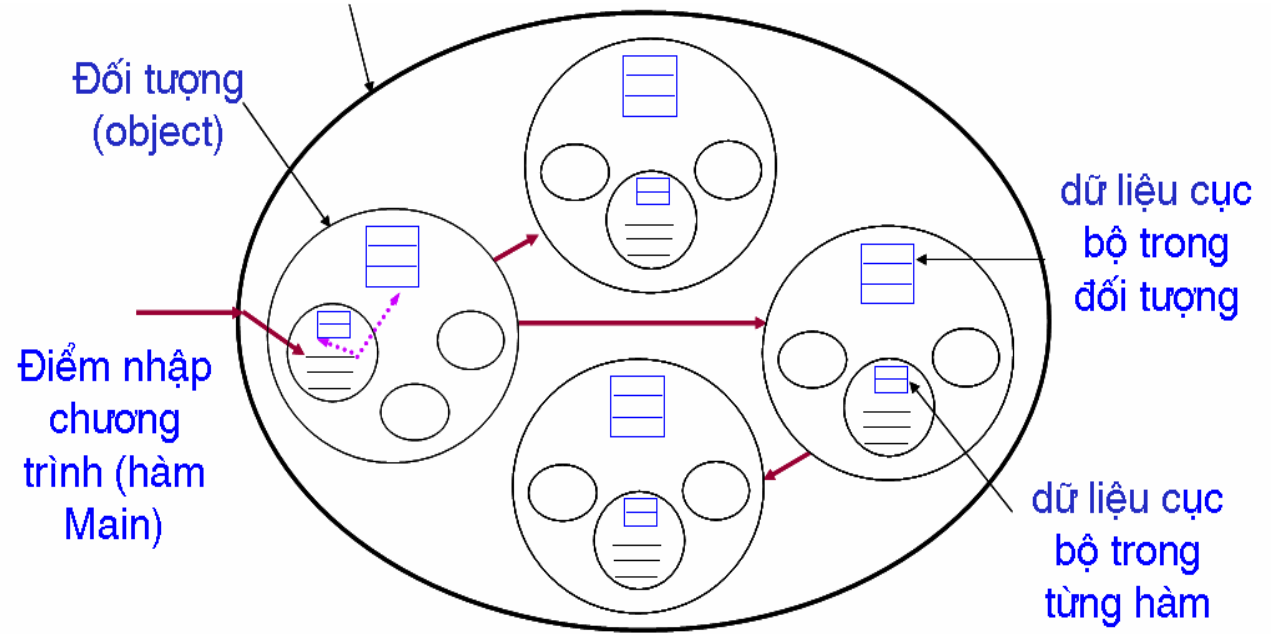
Chương này cũng giới thiệu phương pháp đặt tên cho các phần tử cấu thành ứng dụng lớn 1 cách khoa học thông qua khái niệm namespace dạng cây phân cấp, cách chứa các phần tử cấu thành ứng dụng lớn trong các module vật lý được gọi là assembly.

Chương 2

# Các khái niệm chính của lập trình hướng đối tượng

## 2.1 Cấu trúc chương trình OOP

Chương trình = tập các đối tượng sống độc lập, tương tác nhau khi cần thiết để hoàn thành nhiệm vụ của chương trình (ứng dụng).




Cấu trúc chương trình hướng đối tượng rất thuần nhất, chỉ chứa 1 loại thành phần : đối tượng.

Các đối tượng có tính độc lập rất cao ⇒ quản lý, kiểm soát chương trình rất dễ (cho dù chương trình có thể rất lớn) ⇒ dễ nâng cấp, bảo trì.

Không thể tạo ra dữ liệu toàn cục của chương trình ⇒ điểm yếu nhất của chương trình cấu trúc không tồn tại nữa.

## 2.2 Đối tượng (Object)

Đối tượng là nguyên tử cấu thành ứng dụng.

Đối tượng bao gồm 2 loại thành phần chính yếu :

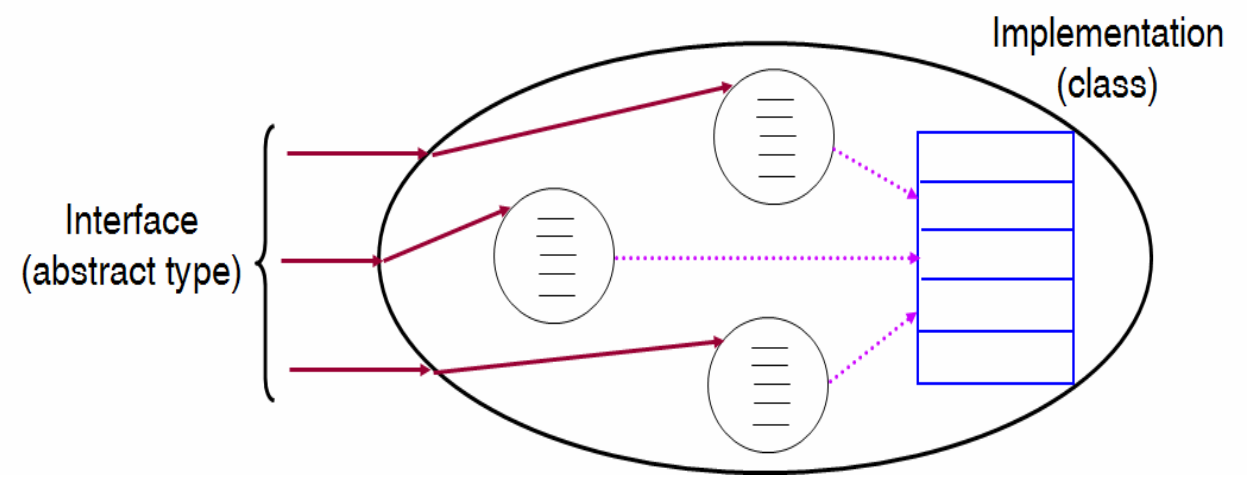
- Tập các tác vụ (operation) : mỗi tác vụ thực hiện 1 chức năng rõ ràng đơn giản nào đó.
- Tập các thuộc tính dữ liệu (attribute) : mỗi thuộc tính có kiểu dữ liệu cụ thể, và chứa 1 giá trị cụ thể thuộc kiểu tương ứng tại từng thời điểm. Các thuộc tính phục vụ cho các tác vụ và là đối tượng xử lý bởi các tác vụ.

Viết phần mềm hướng đối tượng là qui trình đặc tả các loại đối tượng cấu thành ứng dụng.

Đặc tả một loại đối tượng là đặc tả 2 góc nhìn khác nhau về đối tượng :

Góc nhìn sử dụng : dùng phát biểu interface.

Góc nhìn hiện thực cụ thể : dùng phát biểu class.




## 2.3 Kiểu trừu tượng (Abstract type) hay interface

Phát biểu interface định nghĩa thông tin sử dụng đối tượng mà bên ngoài thấy, kết hợp các thông tin này với 1 tên gọi, tên này được gọi là tên kiểu trừu tượng (Abstract type) hay ngắn gọn là type.

Interface là tập hợp các điểm nhập (entry) mà bên ngoài có thể giao tiếp với đối tượng. C# cho phép định nghĩa nhiều loại điểm nhập, nhưng phổ biến nhất là tác vụ chức năng (operation).

Ta dùng chữ ký (signature) để định nghĩa và phân biệt mỗi điểm nhập. Chữ ký của 1 tác vụ gồm :

- tên tác vụ (operation)
- danh sách tham số hình thức, mỗi tham số được đặc tả bởi 3 thuộc tính : tên, type và chiều di chuyển (IN, OUT, INOUT).
- đặc tả chức năng của tác vụ (thường ở dạng chú thích).

Muốn làm việc với 1 đối tượng nào đó, ta thường dùng biến đối tượng. Biến đối tượng nên được đặc tả kiểu bằng tên interface, hạn chế dùng tên class cụ thể.

Biến đối tượng là biến tham khảo, nó không chứa trực tiếp đối tượng, nó chỉ chứa các thông tin để truy xuất được đối tượng, bất chấp đối tượng đang nằm ở đâu.

Biến đối tượng thuộc kiểu interface có thể tham khảo đến nhiều đối tượng thuộc các class cụ thể khác nhau miễn sao các đối tượng này hỗ trợ được interface tương ứng.

Như vậy, nếu ta dùng đối tượng thông qua biến thuộc kiểu interface thì ta không cần biết bất kỳ thông tin hiện thực chi tiết nào về đối tượng mà mình đang dùng, nhờ vậy code ứng dụng sẽ độc lập hoàn toàn với class hiện thực của đối tượng được sử dụng trong ứng dụng.

## Thí dụ interface

Thí dụ sau đây miêu tả 1 interface của đối tượng mà hỗ trợ 2 tác vụ chuẩn hóa chuỗi tiếng Việt về dạng tổ hợp và dựng sẵn. Thông qua interface, người dùng không hề thấy và biết chi tiết về hiện thực của các tác vụ, nhưng điều này không hề ngăn cản họ trong việc dùng đối tượng nào đó có interface IVietLib.

```
interface IVietLib {
    //tác vụ chuẩn hóa chuỗi tiếng Việt về dạng tổ hợp
    int VnPre2Comp(String src, int len, ref String dst);
    //tác vụ chuẩn hóa chuỗi tiếng Việt về dạng dựng sẵn
```

```
    int VnComp2Pre(String src, int len, ref String dst);
}
```

## 2.4 Class (Implementation)

Phát biểu class định nghĩa chi tiết hiện thực đối tượng :

- định nghĩa các thuộc tính, mỗi thuộc tính được đặc tả bởi các thông tin về nó như tên nhận dạng, kiểu dữ liệu, tầm vực truy xuất,... Kiểu của thuộc tính có thể là type cổ điển (kiểu giá trị : số nguyên, thực, ký tự, chuỗi ký tự,...) hay kiểu đối tượng (kiểu tham khảo), trong trường hợp sau thuộc tính sẽ là tham khảo đến đối tượng khác. Trạng thái của đối tượng là tập giá trị của tất cả thuộc tính của đối tượng tại thời điểm tương ứng.
- 'coding' các tác vụ (miêu tả giải thuật chi tiết về hoạt động của tác vụ), các hàm nội bộ trong class và các thành phần khác.

Ngoài các thành phần chức năng, ta còn phải định nghĩa các tác vụ quản lý đối tượng như : khởi tạo trạng thái ban đầu (constructor), dọn dẹp các phần tử liên quan đến đối tượng khi đối tượng bị xóa (destructor).

### Thí dụ về class

Thí dụ sau đây miêu tả 1 class hiện thực interface IVietLib :

```
class MyVietLib : IVietLib {
    //định nghĩa các thuộc tính cần dùng cho 2 tác vụ
        ...
    //định nghĩa 2 tác vụ quản lý đối tượng
    MyVietLib() {…}
    ~MyVietLib() {…}
    //định nghĩa thuật giải tác vụ chuẩn hóa chuỗi tiếng Việt về dạng tổ hợp
    int VnPre2Comp(String src, int len, ref String dst) {….}
    //định nghĩa thuật giải tác vụ chuẩn hóa chuỗi tiếng Việt về dạng dựng
sẵn
```

```
    int VnComp2Pre(String src, int len, ref String dst) {…}
}
```

## 2.5 Tính bao đóng (encapsulation)

Bao đóng : che dấu mọi chi tiết hiện thực của đối tượng, không cho bên ngoài thấy và truy xuất ⇒ tạo độ độc lập cao giữa các đối tượng (tính nối ghép – coupling – hay phụ thuộc giữa các đối tượng rất thấp), nhờ vậy việc quản lý, hiệu chỉnh và nâng cấp từng thành phần phần mềm dễ dàng, không ảnh hưởng đến các thành phần khác.

- che dấu các thuộc tính dữ liệu : nếu cần cho phép bên ngoài truy xuất 1 thuộc tính vật lý, ta tạo 1 thuộc tính luận lý (2 tác vụ get/set tương ứng) để giám sát và kiểm soát việc truy xuất.
- che dấu chi tiết hiện thực các tác vụ.
- che dấu các local function và sự hiện thực của chúng.

C# cung cấp các từ khóa private, protected, internal, public (chương 3) để xác định tầm vực truy xuất từng thành phần của class.

## 2.6 Tính thừa kế (inheritance)

Tính thừa kế cho phép giảm nhẹ công sức định nghĩa interface/class : ta có thể định nghĩa các interface/class không phải từ đầu mà bằng cách kế thừa interface/class có sẵn nhưng gần giống với mình :

- Miêu tả tên cha : mọi thành phần của cha trở thành của mình.
- override 1 số method của class cha, kết quả override chỉ tác dụng trên đối tượng của class con.
- định nghĩa thêm các chi tiết mới (thường khá ít).

Đa thừa kế hay đơn thừa kế. C# cho phép đa thừa kế interface (đa hiện thực), nhưng chỉ hỗ trợ đơn thừa kế class.

Thừa kế tạo ra mối quan hệ cha/con : phần tử đã có là cha, phần tử thừa kế cha được gọi là con. Cha/con có thể là trực tiếp hay gián tiếp.

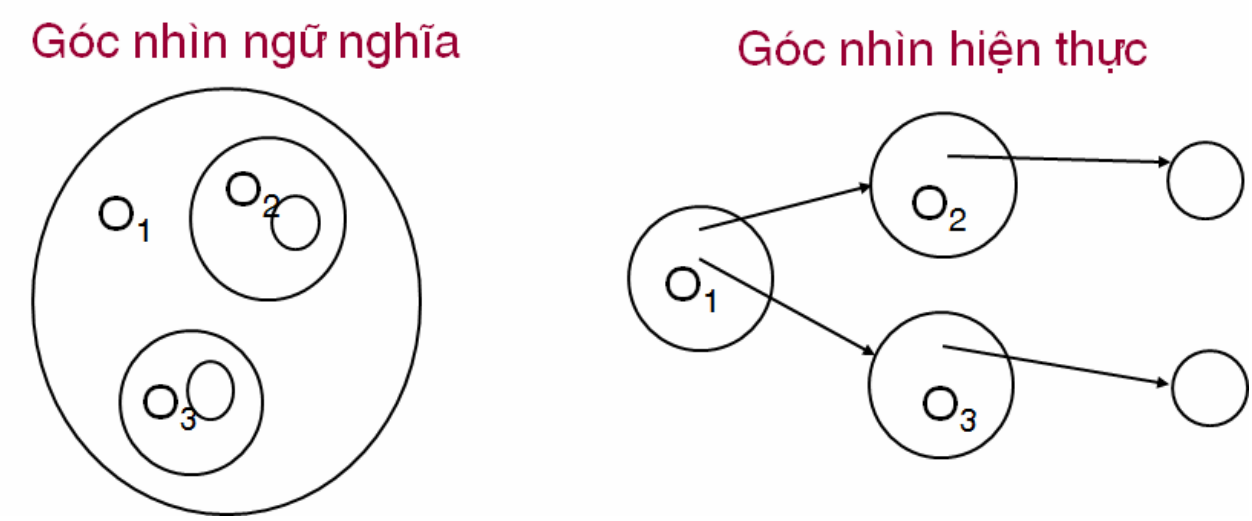
Với các tính chất về thừa kế như slide trước, ta rút ra được 1 số ý tưởng :

- Đối tượng của class con luôn lớn hay hay ít nhất bằng đối tượng class cha (theo góc nhìn người dùng).
- Và như thế, đối tượng class con hoàn toàn có thể đóng vai trò của đối tượng class cha và thay thế đối tượng class cha khi cần thiết, nhưng ngược lại thường không được.

## 2.7 Tính bao gộp (aggregation)

1 đối tượng có thể chứa nhiều đối tượng khác ⇒ tạo nên mối quan hệ bao gộp 1 cách đệ quy giữa các đối tượng. Thí dụ đối tượng quốc gia chứa nhiều đối tượng tỉnh, đối tượng tỉnh chứa nhiều đối tượng quận/huyện,…

Có 2 góc nhìn về tính bao gộp : ngữ nghĩa & hiện thực.




### Ví dụ về bao gộp

```
//định nghĩa class miêu tả đối tượng đồ họa cơ bản
abstract class Geometry {              // abstract base class
```

```
  public abstract void Draw (Graphics g); // abstract operation
  protected int xPos, yPos;
  protected COLORREF color;
};

//định nghĩa class đồ họa phức hợp = tập các đối tượng đồ họa đã có
class GeoGroup : Geometry {
 public override void Draw (Graphics g) {...} ; // override
 private Geometry [] objList; //danh sách các đối tượng thành phần;
 int count; //số lượng các đối tượng thành phần
};
```

## 2.8 Thông điệp (Message), đa xạ (Polymorphism)

Thông điệp là phương tiện giao tiếp (hay tương tác) duy nhất giữa các đối tượng, nó cho phép gọi 1 tác vụ chức năng của 1 đối tượng từ 1 tham khảo đến đối tượng.

Thông điệp bao gồm 3 thành phần :

- tham khảo đến đối tượng cần nhờ.
- tên tác vụ muốn gọi.
- danh sách tham số thực cần truyền cho (hay nhận về từ) tác vụ.

```
public override void Draw (Graphics g) {
   for (int i=0; i < count; i++)
      objList[i].Draw(g);        //gởi thông điệp nhờ đối tượng objList[i]
                                 // tự hiển thị mình lên đối tượng vẽ g
}
```

Xét đoạn lệnh sau :

```
C1 obj = new C1();
obj.func();  //lần 1
obj = new C2();
```

obj.func(); //lần 2

Lệnh gởi thông điệp obj.func() kích hoạt tác vụ func() của class C1 hay tác vụ func() của class C2 ?

1. **Dùng kỹ thuật xác định hàm và liên kết tĩnh** : Dựa vào thông tin tại thời điểm dịch, biến obj thuộc kiểu C1 và máy dịch lời gởi thông điệp obj.func() thành lời gọi hàm C1_func(). Như vậy mỗi khi máy chạy lệnh này, hàm C1_func() sẽ chạy, bất chấp tại thời điểm chạy, obj đang tham khảo đối tượng của class khác (C2). Điều này không đúng với ý muốn người lập trình.
2. **Dùng kỹ thuật xác định hàm và liên kết động** : Lệnh gởi thông điệp obj.func() không được dịch ra 1 lời gọi hàm nào cả mà được dịch thành đoạn lệnh máy với chức năng sau : xác định biến obj đang tham khảo đến đối tượng nào, thuộc class nào, rồi gọi hàm func() của class đó chạy. Như vậy, nếu obj đang tham khảo đối tượng thuộc class C1 thì hàm C1_func() sẽ được gọi, còn nếu obj đang tham khảo đối tượng thuộc class C2 thì hàm C2_func() sẽ được gọi. Ta nói lời gởi thông điệp obj.func() có tính đa xạ. Điều này giải quyết đúng ý muốn người lập trình.

**Tính đa xạ** : cùng 1 lệnh gởi thông điệp đến đối tượng thông qua cùng 1 tham khảo nhưng ở vị trí/thời điểm khác nhau có thể kích hoạt việc thực thi tác vụ khác nhau của các đối tượng khác nhau.

## Kiểm tra kiểu (type check)

Khi lập trình, ta thường phạm nhiều lỗi : lỗi về từ vựng, cú pháp, lỗi về thuật giải... Trong các lỗi thì lỗi về việc gán dữ liệu khác kiểu thường xảy ra nhất.

Để phát hiện triệt để và sớm nhất các lỗi sai về kiểu, máy sẽ dùng cơ chế kiểm tra kiểu chặt và sớm tại thời điểm dịch.

Trong lúc dịch, bất kỳ hoạt động gán dữ liệu nào (lệnh gán, truyền tham số) đều được kiểm tra kỹ lưỡng, nếu dữ liệu và biến lưu trữ không "tương thích" thì báo sai.

Tiêu chí không "tương thích" là gì ?

- dùng kỹ thuật so trùng tên kiểu : tên kiểu không trùng nhau là không tương thích.
- dùng mối quan hệ 'conformity' (tương thích tổng quát). Kiểu A 'conformity' với kiểu B nếu A cung cấp mọi tác vụ mà B có, từng tác vụ của A cung cấp tương thích với tác vụ tương ứng của B. Nói nôm na A lớn hơn hay bằng B thì A 'conformity' với B.

Như vậy, quan hệ so trùng hay quan hệ con/cha (sub/super) là trường hợp đặc biệt của quan hệ tương thích tổng quát.

Nhờ dùng mối quan hệ 'conformity', một biến obj thuộc kiểu C1 có thể chứa tham khảo đến nhiều đối tượng thuộc nhiều class khác nhau, miễn sao các class này tương thích với class được dùng để định nghĩa biến obj.

## 2.9 Tính tổng quát hóa (Generalization)

Viết phần mềm hướng đối tượng là quá trình lặp : viết phát biểu interface/class để đặc tả từng loại đối tượng cấu thành phần mềm.

Nếu số lượng class cấu thành ứng dụng quá lớn thì việc viết phần mềm sẽ khó khăn, tốn nhiều thời gian công sức hơn.

Làm sao giảm nhẹ thời gian, công sức lập trình các ứng dụng lớn ?

- sử dụng cơ chế thừa kế trong định nghĩa interface/class.
- thay vì trực tiếp viết các class cụ thể đặc tả cho các đối tượng trong phần mềm, ta chỉ viết 1 class tổng quát hóa, rồi nhờ class này sinh tự động mã nguồn các class cụ thể.

Thí dụ, thay vì phải viết n class gần giống nhau như danh sách các số nguyên, danh sách các số thực, danh sách các chuỗi, danh sách các record Sinhvien, danh sách các đối tượng đồ họa,... ta chỉ cần viết 1 class tổng quát hóa : danh sách các phần tử có kiểu hình thức T. Khi cần tạo 1 class danh sách các phần tử thuộc kiểu cụ thể nào đó, ta chỉ viết lệnh gọi class tổng quát hóa và truyền tên kiểu cụ thể của phần tử trong danh sách.

Chương 9 sẽ trình bày chi tiết về khả năng, tính chất, mức độ hỗ trợ tổng quát hóa của ngôn ngữ C#.

## 2.10 Kết chương

Chương này đã giới thiệu cấu trúc của chương trình hướng đối tượng, các phương tiện đặc tả đối tượng như interface, class.

Chương này cũng đã giới thiệu các tính chất liên quan đến việc đặc tả và sử dụng đối tượng như thừa kế, bao đóng, bao gộp, tổng quát hóa.

Chương này cũng đã giới thiệu phương tiện giao tiếp duy nhất giữa các đối tượng là thông điệp, nhu cầu cần phải có tính đa xạ trong việc thực hiện lệnh gởi thông điệp.

# Chương 3

# Interface & Class trong C#

## 3.1 Tổng quát về phát biểu class của C#

Ngôn ngữ C# (hay bất kỳ ngôn ngữ lập trình nào khác) cung cấp cho người lập trình nhiều phát biểu (statement) khác nhau, trong đó phát biểu class để đặc tả chi tiết hiện thực từng loại đối tượng cấu thành phần mềm là phát biểu quan trọng nhất. Sau đây là 1 template của 1 class C# :

```
class MyClass : BaseClass, I1, I2, I3 {
  //định nghĩa các thuộc tính vật lý của đối tượng
  //định nghĩa các tác vụ chức năng, các toán tử
  //định nghĩa các thuộc tính giao tiếp (luận lý)
  //định nghĩa các đại diện hàm chức năng (delegate)
  //định nghĩa các sự kiện (event)
  //định nghĩa indexer của class
  //định nghĩa các tác vụ quản lý đời sống đối tượng
}
```

Khi định nghĩa 1 class mới, ta có thể thừa kế tối đa 1 class đã có (đơn thừa kế), tên class này nếu có, phải nằm ở vị trí đầu tiên ngay sau dấu ngăn ":".

Khi định nghĩa 1 class, ta có thể hiện thực nhiều interface khác nhau (đa hiện thực), danh sách này nếu có, phải nằm sau tên class cha. Trong trường hợp nhiều interface có cùng 1 tác vụ (phân biệt bằng chữ ký) và nếu class muốn hiện thực chúng khác nhau thì ta dùng tên dạng phân cấp :

```
class MyClass : BaseClass, I1, I2, I3 {
  //hiện thực các tác vụ cùng chữ ký trong các interface khác nhau
  void I1.func1() {}
  void I2.func1() {}
  void I3.func1() {}
  ...
```

}

## 3.2 Định nghĩa thuộc tính vật lý

Mỗi thuộc tính vật lý của đối tượng là 1 biến dữ liệu cụ thể. Phát biểu định nghĩa 1 thuộc tính vật lý sẽ đặc tả các thông tin sau về thuộc tính tương ứng :

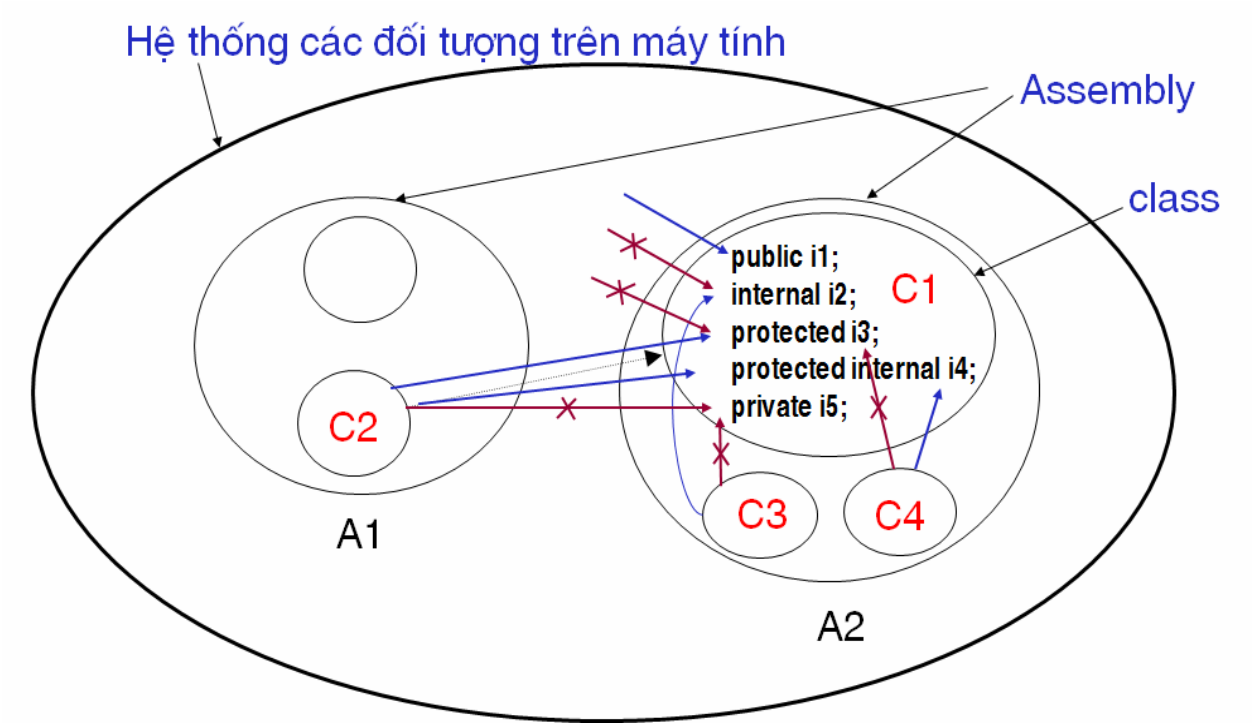
- Tên nhận dạng.
- Kiểu dữ liệu.
- Giá trị ban đầu.
- Tầm vực truy xuất

Cú pháp đơn giản để định nghĩa 1 thuộc tính vật lý như sau :

[scope] *type name* [= *value*];

Thành phần scope miêu tả tầm vực truy xuất của thuộc tính, có thể chọn 1 trong 5 khả năng sau :

- public : thuộc tính có thể được truy xuất bất kỳ đâu.
- internal : thuộc tính có thể được truy xuất bất kỳ đâu trong cùng assembly chứa class.
- protected : thuộc tính có thể được truy xuất bởi class hiện hành và các class con, cháu.
- protected internal : thuộc tính có thể được truy xuất bất kỳ đâu trong cùng assembly chứa class hay các class con, cháu.
- private : thuộc tính chỉ có thể được truy xuất nội bộ trong class hiện hành.
- nếu thành phần scope không được miêu tả tường minh, thuộc tính sẽ có tầm vực internal.




Thành phần *type* thường là tên kiểu dữ liệu của thuộc tính tương ứng, nó có thể là tên kiểu giá trị hay tên kiểu tham khảo.

Thành phần *name* là tên nhận dạng thuộc tính.

Thành phần [= *value*] miêu tả biểu thức xác định trị ban đầu của thuộc tính.

Thành phần nào nằm trong [] là nhiệm ý (optional), có thể có hoặc không. Các thành phần khác bắt buộc phải có.

Thí dụ :

```
private int dorong = 10;
private int docao = 10;
```

### 3.3 Định nghĩa tác vụ chức năng

Mỗi tác vụ (operation) thực hiện 1 chức năng xác định, rõ ràng nào đó mà bên ngoài đối tượng (client) cần dùng. Định nghĩa tác vụ gồm 2 phần : định nghĩa interface sử dụng và định nghĩa thuật giải chi tiết mà tác vụ thực hiện (method).

Lệnh định nghĩa 1 tác vụ thường gồm 5 phần sau :

[scope | attribute] *return_type name* **(**arg_list**)** *body*

- scope miêu tả tầm vực truy xuất của tác vụ : public, protected, internal, protected internal, private.
- attribute miêu tả tính chất hoạt động của tác vụ : static, virtual, sealed, override, abstract, extern.
- return_type là tên kiểu của giá trị mà tác vụ sẽ trả về.
- name là tên tác vụ, arg_list là danh sách từ 0 tới n tham số hình thức cách nhau bởi dấu ',', định nghĩa mỗi tham số hình thức gần giống như định nghĩa thuộc tính vật lý.

### 3.4 Định nghĩa toán tử chức năng

Mỗi toán tử (operator) thực hiện 1 phép toán xác định. Toán tử là trường hợp đặc biệt của tác vụ. Định nghĩa toán tử gồm 2 phần : định nghĩa interface sử dụng và định nghĩa thuật giải chi tiết mà toán tử thực hiện (method).

Lệnh định nghĩa 1 toán tử thường gồm 6 phần sau :

[scope] *return_type* **operator** *name* **(**arg_list**)** *body*

- scope miêu tả tầm vực truy xuất của toán tử : public, static, extern.
- return_type là tên kiểu của giá trị mà toán tử sẽ trả về.
- name là tên toán tử : +,-,*,/,...
- arg_list là danh sách từ 0 (cho toán tử 1 ngôi) tới 2 (cho toán tử 3 ngôi) tham số hình thức cách nhau bởi dấu ',', định nghĩa mỗi tham số hình thức gần giống như định nghĩa thuộc tính vật lý.

### 3.5 Định nghĩa thuộc tính giao tiếp (luận lý)

Mỗi thuộc tính giao tiếp (luận lý) chẳng qua là 1 hay 2 tác vụ get/set (tham khảo/thiết lập) nội dung thuộc tính tương ứng. Định nghĩa thuộc tính giao tiếp là định nghĩa 1 hay 2 tác vụ get/set.

Lệnh định nghĩa 1 thuộc tính thường có dạng sau :

[scope | attribute] *type name* **{[**getdef] [setdef]**};**

- scope, attirbute, type, name có ý nghĩa giống như lệnh định nghĩa tác vụ.
- getdef và setdef là lệnh định nghĩa tác vụ get và set thuộc tính tương ứng.

```
class Rectangle {
    private int m_cao;  //thuộc tính vật lý
    public int Cao {  //thuộc tính luận lý
      get { return m_cao; }
      set { if (value>0 && value <1024) m_cao = value; }
    }
}
```

### 3.6 Định nghĩa đối tượng đại diện hàm (delegate)

Nhiều khi chúng ta cần viết lệnh gọi hàm mà chưa biết tên cụ thể, tên của hàm chỉ có thể xác định tại thời điểm run-time. Delegate của C# cho phép ta giải quyết được yêu cầu này.

Delegate là 1 class đối tượng đặc biệt, đối tượng delegate chỉ chứa 1 field thông tin, field này là địa chỉ của 1 hàm chức năng nào đó.

Delegate đặc biệt hữu dụng khi kết hợp với sự kiện (Event) mà ta sẽ trình bày sau.

Lệnh định nghĩa delegate thường có dạng :

[scope] **delegate** return_*type name* **(***arg_list***);**

- scope, return_type, name, arg_list có ý nghĩa giống như lệnh định nghĩa tác vụ.

### 3.7 Định nghĩa sự kiện (Event)

Tác vụ chỉ có thể được (gọi) kích hoạt bởi người lập trình, trong khi nhiều lúc ta muốn người dùng có thể kích hoạt trực tiếp chức năng nào đó của đối tượng (thí dụ đối tượng giao diện). Event là phương tiện giải quyết yêu cầu này.

Event là 1 đối tượng thuộc class delegate, sau khi được khởi động, nó có thể miêu tả từ 1 tới n tác vụ chức năng mà sẽ được tự kích hoạt mỗi khi event xảy ra.

Lệnh định nghĩa Event giống như lệnh định nghĩa thuộc tính dữ liệu :

[scope] **event** *delegate_type name***;**

- scope, name có ý nghĩa giống như lệnh định nghĩa thuộc tính.
- delegate_type là tên của 1 delegate đã định nghĩa trước.

## 3.8 Định nghĩa phần tử quản lý danh sách (indexer)

Để truy xuất 1 đối tượng thuộc 1 class, ta dùng biến đối tượng. Thông qua biến đối tượng (tham khảo), ta truy xuất từng thành phần được phép (thuộc tính, tác vụ, toán tử,...) thông qua cú pháp gởi thông điệp : objVar.tên thành phần.

Ngoài khả năng thông thường trên, C# còn cho phép kết hợp với đối tượng 1 danh sách các phần tử dữ liệu thuộc 1 kiểu nào đó. Indexer chính là khả năng này.

Nếu thuộc tính giao tiếp cho phép ta miêu tả 1 giá trị luận lý duy nhất thì Indexer cho phép ta miêu tả 1 danh sách nhiều giá trị luận lý. Lệnh định nghĩa Indexer giống như lệnh định nghĩa thuộc tính luận lý :

[scope | attribute] *type* **this [int i] {[**getdef] [setdef]**};**

- scope, attirbute, type có ý nghĩa giống như lệnh định nghĩa thuộc tính.
- getdef và setdef là lệnh định nghĩa tác vụ get và set phần tử thứ i trong danh sách.

```
class Rectangle {
        private int[] arr = new int[100];
        public int this[int index]  { //định nghĩa Indexer
                get {
                        if (index < 0 || index >= 100) {  return 0; }
```

```
                else { return arr[index]; }
            }
            set {
                if (!(index < 0 || index >= 100)) {
                    arr[index] = value;
                }
            }
        }
    }
```

Để truy xuất phần tử thứ i trong danh sách, ta dùng cú pháp giống như truy xuất biến array :

```
Rectangle objRec = new Rectangle();
objRec[0] = 0;
int ret = objRec[10];
```

### 3.9 Thành phần static và thành phần không static

Phát biểu class được dùng để đặc tả các đối tượng cùng loại mà phần mềm dùng. Về nguyên tắc, khi đối tượng được tạo ra (bằng lệnh new), nó sẽ chứa tất cả các thành phần được đặc tả trong class tương ứng. Tuy nhiên, nếu xét chi li thì VC# cho phép đặc tả 2 loại thành phần trong 1 class như sau :

1. Thành phần static : là thành phần có từ khóa static trong lệnh định nghĩa nó. Đây là thành phần kết hợp với class, nó không được nhân bản cho từng đối tượng và như thế đối tượng không thể truy xuất nó. Cách duy nhất để truy xuất thành phần static là thông qua tên class.

```
//Console là tên class chứa các hàm truy xuất
//các thiết bị nhập/xuất chuẩn
Console.Writeln("Nội dung cần hiển thị");
```

2. Thành phần không static : là thành phần không dùng từ khóa static trong lệnh định nghĩa nó. Đây là thành phần kết hợp với từng đối tượng, nó sẽ được nhân bản cho từng đối tượng. Ta truy xuất thành phần không static thông qua tham khảo đối tượng.

```
class Rectangle {
        private int m_cao;  //thuộc tính vật lý
        public int Cao {  //thuộc tính luận lý
            get { return m_cao; }
            set { if (value>0 && value <1024) m_cao =
value; }
        }
    }

    Rectangle r = new Rectangle();
    r.Cao = 10;
```

### 3.10 Lệnh định nghĩa 1 class C# điển hình

```
class MyClass {
    //1. định nghĩa các thuộc tính vật lý
    private int m_x;
    private int[] arr = new int[100];
    //2. định nghĩa các tác vụ & toán tử chức năng
    public void button1_Click(object sender, System.EventArgs
e) {}
    ...
    //3. định nghĩa đối tượng đại diện hàm chức năng
    public delegate void EventHandler (Object sender, EventArgs
e);
    //4. định nghĩa sự kiện Click được xử lý bởi delegate EventHandler.
    public event EventHandler Click;
    //5. định nghĩa thuộc tính luận lý x
    public int x {
        get { return m_x; }
        set { m_x = value; }
    }
    //6. định nghĩa các tác vụ quản lý đối tượng
    public MyClass() { this.Click += new
EventHandler(button1_Click); }
    ~MyClass() {...}       //hàm destructor
```

```
        //còn tiếp ở slide kế sau
        //7. định nghĩa indexer
        public int this[int index]  {
              get {
            //kiểm tra giới hạn để quyết định
            if (index < 0 || index >= 100) {  return 0; }
            else { return arr[index]; }
              }
              set {
            if (!(index < 0 || index >= 100)) { arr[index] = value;  }
              }
        }
};
```

**Lệnh định nghĩa 1 inreface C# điển hình**

```
interface IMyInterface {
        //2. định nghĩa các tác vụ & toán tử chức năng
        void button1_Click(object sender, System.EventArgs e) {}
        //4. định nghĩa sự kiện Click được xử lý bởi delegate EventHandler.
        event EventHandler Click;
        //5. định nghĩa thuộc tính luận lý x
        int x {get; set;}
        //7. định nghĩa indexer
        int this[int index] {get; set;}
}
```

## 3.11 Kết chương

Chương này đã giới thiệu cú pháp của phát biểu class C# được dùng để đặc tả chi tiết hiện thực 1 loại đối tượng được dùng trong chương trình.

Chương này cũng đã giới thiệu cú pháp các phát biểu để định nghĩa các thành phần cấu thành đối tượng như thuộc tính vật lý, thuộc tính giao tiếp, tác vụ chức năng, toán tử, delegate, event, indexer.

Chương này cũng đã phân biệt 2 loại thành phần được đặc tả trong 1 class : thành phần dùng chung (static) và thành phần nhân bản theo từng đối tượng.

# Chương 4

# Vòng đời đối tượng và sự tương tác giữa chúng

## 4.1 Quản lý đời sống đối tượng - Hàm Constructor

Class mô hình các đối tượng cùng loại mà phần mềm dùng. Lúc lập trình, ta chỉ đặc tả class, đối tượng chưa có. Khi ứng dụng chạy, tại thời điểm cần thiết, phần mềm sẽ phải tạo tường minh đối tượng bằng lệnh new :

```
Rectangle objRec = new Rectangle(); //tạo đối tượng
```

Trạng thái của đối tượng là tập giá trị cụ thể của các thuộc tính. Ngay sau đối tượng được tạo ra, nó cần có trạng thái ban đầu xác lập nào đó. Hàm constructor cho phép người lập trình miêu tả hoạt động xác lập trạng thái ban đầu của đối tượng.

Cũng giống như nhiều tác vụ khác, hàm constructor có thể có nhiều "overloaded" khác nhau (với số lượng tham số khác nhau hay tính chất của 1 tham số nào đó khác nhau).

Mỗi lần đối tượng được tạo ra (bởi lệnh new), máy sẽ gọi tự động constructor của class tương ứng. Tùy theo tham số của lệnh new mà constructor nào tương thích sẽ được kích hoạt chạy.

Trong nội bộ 1 class, các tác vụ chỉ có thể truy xuất các thuộc tính của mình và các thuộc tính thừa kế từ cha có tầm vực protected, public, chứ không thể truy xuất trực tiếp các thuộc tính thừa kế từ cha có thuộc tính private. Do đó nếu chỉ chạy constructor của class cần tạo đối tượng thì không thể khởi tạo hết các thuộc tính của đối tượng, cần kích hoạt hết các constructor của các class cha (gián tiếp hay trực tiếp).

Mặc định, khi cần gọi constructor của class cha chạy, máy sẽ gọi constructor không tham số. Nếu người lập trình muốn khác thì phải khai báo lại tường minh "overloaded" nào cần chạy thông qua mệnh đề base() trong lệnh định nghĩa hàm constructor.

```
//class A  có 2 hàm constructor
```

```
class A {
   A() {...}
   A(int i) {...}
   ...
};
//class B thừa kế A, có 2 hàm constructor
class B : A {
   B() : base() {...}
   B(int i) : base (i) {...}
   ...
};
//class C thừa kế B, có 2 hàm constructor
class C : B {
   C() : base () {...}
   C(int i) : base (i) {...}
   ... };
C c1 = new C();  //kích hoạt A() → B() → C()
C c2 = new C(5);   //kích hoạt A(5) → B(5) → C(5)
```

Việc xác định constructor nào được kích hoạt phải theo chiều từ dưới lên bắt đầu từ class được new, nhưng các constructor được chạy thực sự sẽ theo chiều từ trên xuống bắt đầu từ class tổ tiên đời đầu.

## 4.2 Quản lý đời sống đối tượng - Hàm Destructor

Đối tượng là 1 thực thể, nó có đời sống như bao thực thể khác. Như ta đã biết, khi ta gọi lệnh new, 1 đối tượng mới thuộc class tương ứng sẽ được tạo ra (trong không gian hệ thống), trạng thái ban đầu sẽ được xác lập thông qua việc kích hoạt dây chuyền các constructor của các class thừa kế. Chương trình sẽ lưu giữ tham khảo đến đối tượng trong biến tham khảo để khi cần, gởi thông điệp nhờ đối tượng thực thi dùm 1 tác vụ nào đó.

VC# không cung cấp tác vụ delete để xóa đối tượng khi không cần dùng nó nữa. Thật vậy, đánh giá 1 đối tượng nào đó có cần

dùng nữa hay không là việc không dễ dàng, dễ nhầm lẫn nếu để chương trình tự làm.

Tóm lại, trong VC#, chương trình chỉ tạo tường minh đối tượng khi cần dùng nó, chương trình không quan tâm việc xóa đối tượng và cũng không có khả năng xóa đối tượng.

Như vậy, đối tượng sẽ bị xóa lúc nào, bởi ai ? Hệ thống có 1 module đặc biệt tên là "Garbage collection" (trình dọn rác), module này sẽ theo dõi việc dùng các đối tượng, khi thấy đối tượng nào mà không còn ai dùng nữa thì nó sẽ xóa dùm tự động.

Trình dọn rác không biết trạng thái đối tượng tại thời điểm bị xóa nên nó không làm gì ngoài việc thu hồi vùng nhớ mà đối tượng chiếm. Như vậy rất nguy hiểm, thí dụ như đối tượng bị xóa đã mở, khóa file và đang truy xuất file dỡ dang.

Để giải quyết vấn đề xóa đối tượng được triệt để, trình dọn rác sẽ gọi tác vụ destructor của đối tượng sắp bị xóa, nhiệm vụ của người đặc tả class là hiện thực tác vụ này.

Tác vụ destructor không có kiểu trả về, không có tham số hình thức → không có overloaded, chỉ có 1 destructor/class mà thôi.

Mặc dù người đặc tả class sẽ hiện thực tác vụ destructor nếu thấy cần thiết, nhưng code của chương trình không được gọi trực tiếp destructor của đối tượng. Chỉ có trình dọn rác của hệ thống mới gọi destructor của đối tượng ngay trước khi xóa đối tượng đó.

Destructor của 1 class cũng chỉ xử lý trạng thái đối tượng do các thuộc tính của class đó qui định, nó cần gọi destructor của class cha để xử lý tiếp trạng thái đối tượng do các thuộc tính private của class cha qui định, và cứ thế tiếp tục.

Tóm lại trước khi xóa một đối tượng, trình dọn rác sẽ gọi các destructor theo chiều từ dưới lên, bắt đầu từ class hiện hành của đối tượng, sau đó tới class cha, ... và cuối cùng là class tổ tiên đời đầu (root).

## 5.3 Hiệu chỉnh thuộc tính các đối tượng giao diện

Muốn tương tác với đối tượng nào đó, ta phải có tham khảo đến đối tượng đó. Thường ta lưu giữ tham khảo đối tượng cần truy xuất trong biến đối tượng (biến tham khảo). Thông qua tham khảo đến đối tượng, ta có thể thực hiện 1 trong các hành động tương tác sau đây :

- truy xuất 1 thuộc tính vật lý của đối tượng có tầm vực cho phép (public hay internal hay protected).
- truy xuất 1 thuộc tính luận lý của đối tượng.
- gọi 1 tác vụ hay toán tử có tầm vực cho phép.
- truy xuất 1 event của đối tượng.
- truy xuất 1 phần tử trong danh sách indexer của đối tượng.

Gọi 1 tác vụ hay 1 toán tử là giống nhau và cần làm rõ chi tiết trong phần sau.

Xét đoạn lệnh sau :

```
class C1 {
        public void func1() {}         //dịch ra hàm mã máy có tên là C1_func1
        public virtual func2() {} //dịch ra hàm mã máy có tên là
C1_func2
}

class C2 : C1 {
        public override func1() {}         //dịch ra hàm mã máy có tên là C2_func1
        public override func2() {}//dịch ra hàm mã máy có tên là C2_func2
}

C1 obj = new C1();
obj.func1();  //lần 1 → gọi hàm mã máy nào ?
//đoạn code có thể làm obj chỉ về đối tượng của class C2, C3,...
obj.func1(); //lần 2 → gọi hàm mã máy nào ?
```

## 4.4 Liên kết tĩnh trong việc gởi thông điệp

Hai lệnh gởi thông điệp obj.func1() trong slide trước sẽ kích hoạt tác vụ func1() của class C1 hay tác vụ func1() của class C2 ?

### 1. Dùng kỹ thuật xác định hàm và liên kết tĩnh :

Tại thời điểm dịch, chương trình dịch chỉ biết biến obj thuộc kiểu C1 và nó dịch cả 2 lời gởi thông điệp obj.func1() thành lời gọi hàm C1_func1(). Như vậy mỗi khi máy chạy lệnh obj.func1() lần 1, hàm C1_func1() sẽ được gọi, điều này đúng theo yêu cầu của phần mềm. Nhưng khi máy chạy lệnh obj.func1() lần 2, hàm C1_func1() cũng sẽ được gọi, điều này không đúng theo yêu cầu của phần mềm vì lúc này obj đang tham khảo đối tượng của class C2.

Mặc định, VC# dùng kỹ thuật xác định hàm và liên kết tĩnh khi dịch lời gởi thông điệp, do đó tạo ra độ rủi ro cao, chương trình ứng dụng thường chạy không đúng theo yêu cầu mong muốn!!!

## 4.5 Liên kết động để đảm bảo tính đa xạ

Bây giờ nếu ta hiệu chỉnh 2 lệnh gởi thông điệp obj.func1() trong slide trước thành obj.func2() thì máy sẽ kích hoạt tác vụ func2() của class C1 hay tác vụ func2() của class C2 ?

### 2. Dùng kỹ thuật xác định hàm và liên kết động :

Lệnh gởi thông điệp obj.func2() được dịch thành đoạn lệnh máy với chức năng sau : xác định biến obj đang tham khảo đến đối tượng nào, thuộc class nào, rồi gọi hàm func2() của class đó chạy. Như vậy, lần gởi thông điệp 1, biến obj đang tham khảo đối tượng thuộc class C1 nên máy sẽ gọi hàm C1_func2(), điều này đúng theo yêu cầu của phần mềm. Khi máy chạy lệnh obj.func2() lần 2, đoạn code xác định hàm và liên kết động sẽ gọi được hàm C2_func2(), điều này cũng đúng theo yêu cầu của phần mềm. Ta nói lời gởi thông điệp obj.func2() có tính đa xạ.

Trong VC#, nếu dùng từ khóa virtual trong lệnh định nghĩa tác vụ thì tác vụ này sẽ được xử lý theo cơ chế liên kết động và sẽ đảm bảo được tính đa xạ, tức đảm bảo tính đúng đắn trong lời gởi thông điệp. Biết được điều này, từ đây về sau, mỗi lần định nghĩa 1 tác vụ hay 1 toán tử, ta hãy luôn dùng từ khóa virtual kết hợp với nó.

Lưu ý rằng 2 tác vụ constructor và destructor của đối tượng là 2 tác vụ đặc biệt, chúng quản lý đời sống đối tượng và chỉ được gọi bởi hệ thống. Ta không được phép dùng từ khóa virtual khi định nghĩa chúng.

### 4.6 Xử lý sự kiện luôn có tính đa xạ

Chúng ta hãy viết 1 chương trình nhỏ gồm 1 form giao diện, trong form ta tạo 1 Button có thuộc tính Text="Làm gì đây?", thuộc tính (Name) = btnStart, định nghĩa hàm xử lý sự kiện Click cho nó rồi viết code như sau :

```
//hàm xử lý Click chuột trên button do máy tạo ra
private void btnStart_Click(object sender, EventArgs e) {
        //xuất thông báo để kiểm tra
        MessageBox.Show("Hàm btnStart_Click sẽ xử lý đây");
        //thay đổi hàm xử lý Click cho Button
        this.btnStart.Click -= new EventHandler(btnStart_Click);
        this.btnStart.Click += new EventHandler(btnStart_Click1);
}
```

Hãy viết thêm hàm btnStart_Click1() như sau :

```
//hàm xử lý Click chuột trên button tự viết thêm
private void btnStart_Click1(object sender, EventArgs e) {
        //xuất thông báo để kiểm tra
        MessageBox.Show("Hàm btnStart_Click1 sẽ xử lý đây");
        //thay đổi hàm xử lý Click cho Button
        this.btnStart.Click -= new EventHandler(btnStart_Click1);
        this.btnStart.Click += new EventHandler(btnStart_Click);
}
```

Bây giờ nếu chạy chương trình, lần đầu click chuột ta sẽ thấy hàm btnStart_Click() chạy, nhưng nếu click chuột tiếp thì hàm btnStart_Click1() chạy, cứ thế thay phiên nhau chạy (theo ý muốn người lập trình). Ta nói xử lý sự kiện người dùng luôn có tính đa xạ.

## 4.7 Kết chương

Chương này đã giới thiệu vòng đời của từng đối tượng trong chương trình, cách thức quản lý đời sống của đối tượng, các thời điểm quan trọng nhất như lúc tạo mới đối tượng, lúc xóa đối tượng cũng như cách miêu tả các hoạt động xảy ra tại các thời điểm này.

Chương này cũng đã giới thiệu sự tương tác giữa các đối tượng trong lúc chúng đang sống để hoàn thành nhiệm vụ của chương trình. Gởi thông điệp là sự tương tác chính yếu giữa các đối tượng, và cần phải có tính đa xạ.

# Chương 5

# Xây dựng giao diện ứng dụng bằng Visual Studio

## 5.1 Tổng quát về xây dựng ứng dụng bằng VS .Net

Một trong các yêu cầu quan trọng của các ứng dụng hiện nay là phải có tính thân thiện cao, gần gũi với người dùng. Để thỏa mãn yêu cầu này, ứng dụng thường sẽ hoạt động ở chế độ đồ họa trực quan.

Các class cấu thành chương trình dùng giao diện đồ họa được chia làm 2 nhóm chính :

- Các class miêu tả các đối tượng giao diện với người dùng như Form, Button, TextBox, Checkbox,... Nhiệm vụ của các đối tượng này là giúp người dùng có thể tương tác dễ dàng, trực quan với chương trình để nhập/xuất dữ liệu, để điều khiển/giám sát hoạt động của chương trình. Các đối tượng này còn che dấu mọi chi tiết về thuật giải và dữ liệu bên trong chương trình, người dùng không cần quan tâm đến chúng.
- Các class miêu tả các chức năng cần thực hiện của chương trình.

Viết code tường minh để đặc tả các đối tượng giao diện là 1 công việc rất khó khăn và tốn nhiều công sức, thời gian.

Để giảm nhẹ công sức đặc tả các đối tượng giao diện, các môi trường lập trình trực quan (như Visual Studio .Net) đã viết sẵn 1 số đối tượng giao diện thường dùng và cung cấp công cụ để người lập trình thiết kế trực quan giao diện của ứng dụng bằng cách tích hợp các đối tượng giao diện có sẵn này : người lập trình đóng vai trò họa sĩ để vẽ/hiệu chỉnh kích thước, di chuyển vị trí các phần tử giao diện cần cho ứng dụng.

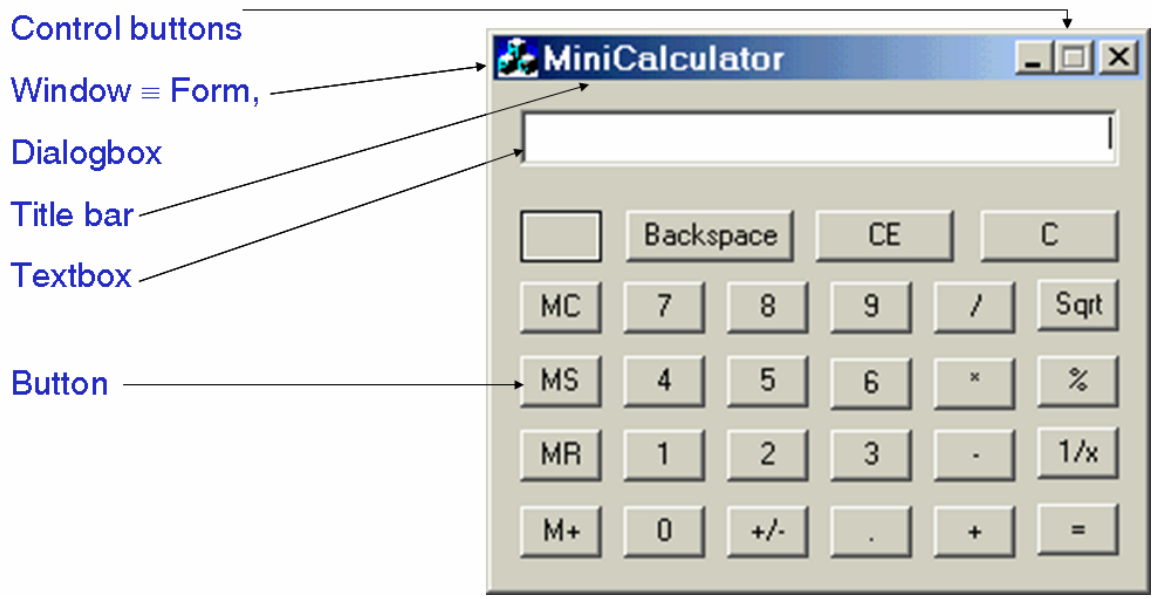
Ngoài ra môi trường trực quan còn cho phép người lập trình tự tạo các đối tượng giao diện mới (User Control) để dùng trong các ứng dụng được viết sau đó (chương 9).

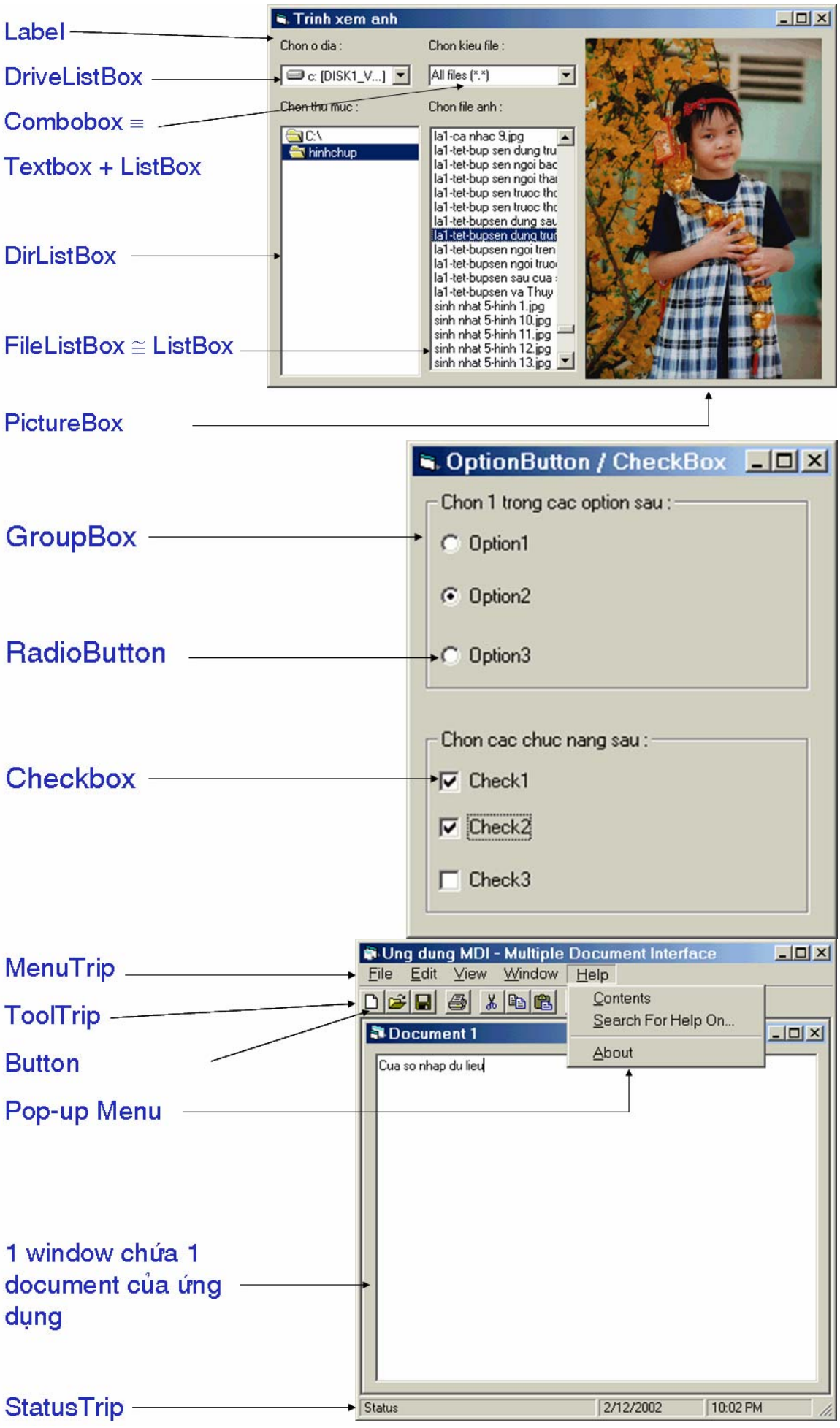
Qui trình viết ứng dụng theo cơ chế này được gọi là viết ứng dụng bằng cách lắp ghép các linh kiện phần mềm, nó giống như việc lắp máy tính từ các linh kiện phần cứng như CPU, RAM, disk, keyboard, monitor,...⇒ rất dễ dàng và nhanh chóng.

Mọi phần tử giao diện, dù nhỏ hay lớn, dù đơn giản hay phức tạp, đều là cửa sổ (window). HĐH Windows sẽ quản lý các cửa sổ làm việc theo thời gian. Một ứng dụng có thể dùng nhiều cửa sổ trong quá trình hoạt động, nhưng từng thời điểm chỉ có 1 số ít cửa sổ được chương trình hiển thị để làm việc với người dùng.

Chúng ta sẽ làm quen 1 số đối tượng giao diện, nắm được tính chất và khả năng của từng đối tượng để khi lập trình ứng dụng nào đó, ta sẽ chủ động chọn lựa và dùng chúng cho phú hợp với từng ngữ cảnh sử dụng.

## 5.2 Một số đối tượng giao diện thường dùng




Label
DriveListBox
Combobox ≡
Textbox + ListBox
DirListBox
FileListBox ≅ ListBox
PictureBox
Trinh xem anh
Chon o dia :
Chon kieu file :
c: [DISK1_V...]
All files (*.*)
Chon thu muc :
Chon file anh :
C:\
hinhchup
la1-ca nhac 9.jpg
la1-tet-bupsen dung sau
la1-tet-bupsen ngoi tren
la1-tet-bupsen sau cua :
la1-tet-bupsen va Thuy
sinh nhat 5-hinh 1.jpg
sinh nhat 5-hinh 10.jpg
sinh nhat 5-hinh 11.jpg
sinh nhat 5-hinh 12.jpg
sinh nhat 5-hinh 13.jpg
GroupBox
RadioButton
Checkbox
OptionButton / CheckBox
Chon 1 trong cac option sau :
Option1
Option2
Option3
Chon cac chuc nang sau :
Check1
Check2
Check3
MenuTrip
ToolTrip
Button
Pop-up Menu
1 window chứa 1 document của ứng dụng
StatusTrip
Ung dung MDI - Multiple Document Interface
File Edit View Window Help
Contents
Search For Help On...
About
Document 1
Cua so nhap du lieu
Status
2/12/2002
10:02 PM

### Các tính chất chung của các đối tượng giao diện

Đối tượng giao diện có những tính chất giống như đối tượng bình thường, nó cũng được cấu thành từ các loại thành phần : thuộc tính, tác vụ, event, delegate...

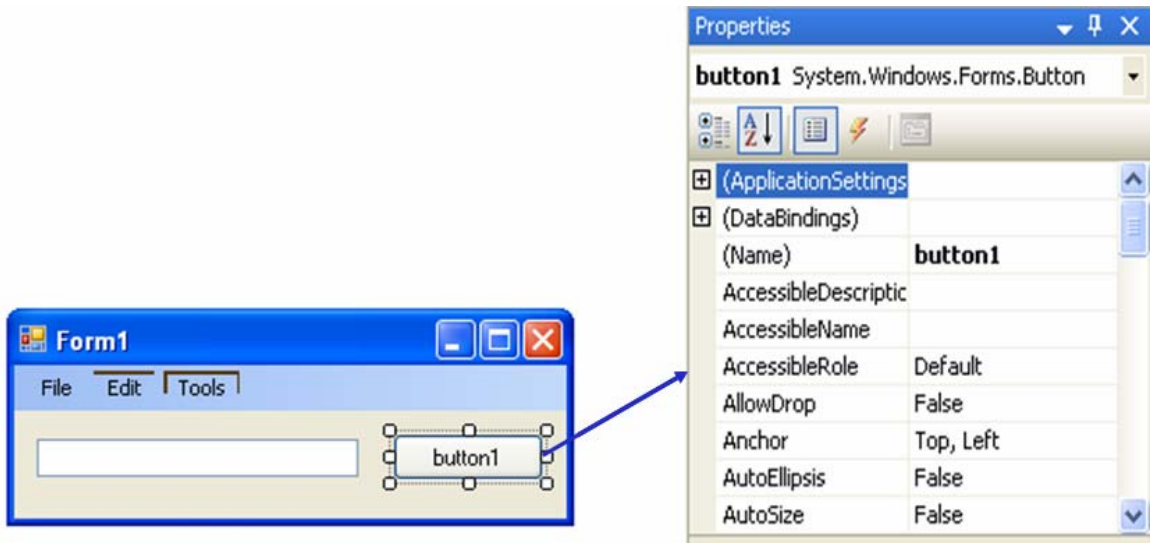
Mỗi đối tượng giao diện chứa khá nhiều thuộc tính liên quan đến nhiều loại trạng thái khác nhau :

- (Name) : đây là thuộc tính đặc biệt, xác định tên nhận dạng của đối tượng, giá trị của thuộc tính này sẽ trở thành biến tham khảo đến đối tượng, code của ứng dụng sẽ dùng biến này để truy xuất đối tượng.
- các thuộc tính xác định vị trí và kích thước (Layout) : Location, Size, Margin...
- các thuộc tính xác định tính chất hiển thị : Text, Font, ForeColor, BackColor,...
- các thuộc tính xác định hành vi (Behavoir) : Enable, Visible...
- các thuộc tính liên kết dữ liệu database : DataBindings,...

### 5.3 Hiệu chỉnh thuộc tính các đối tượng giao diện

Khi tạo trực quan 1 đối tượng giao diện, môi trường đã gán giá trị đầu mặc định cho các thuộc tính, thường ta chỉ cần thay đổi 1 vài thuộc tính là đáp ứng được yêu cầu riêng. Có 2 cách để hiệu chỉnh giá trị 1 thuộc tính :

- trực quan thông qua cửa sổ thuộc tính của đối tượng giao diện.
- lập trình truy xuất thuộc tính của đối tượng giao diện.

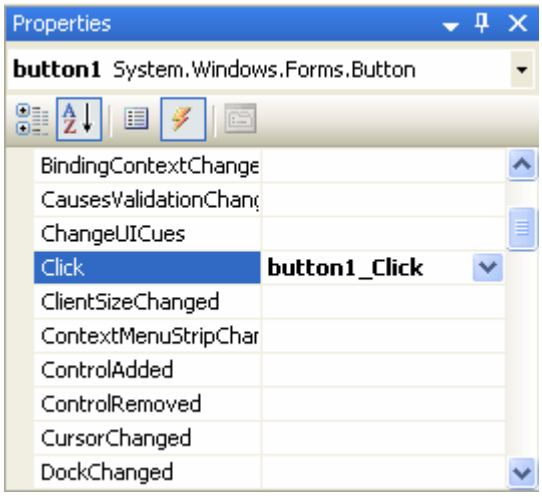



## 5.4 Sự kiện - Hàm xử lý sự kiện

Mỗi đối tượng giao diện có khá nhiều sự kiện để người dùng kích hoạt. Người lập trình có thể định nghĩa hàm xử lý kết hợp với sự kiện cần xử lý. Khi ứng dụng chạy, lúc người dùng kích hoạt sự kiện, hàm xử lý sự kiện tương ứng (nếu có) sẽ chạy.

Thí dụ khi user ấn chuột vào button tên "button1", hệ thống tạo ra sự kiện "Click" để kích khởi hàm button1_Click() chạy.

Muốn tạo hàm xử lý sự kiện trên đối tượng giao diện, ta chọn đối tượng, cửa sổ thuộc tính của nó sẽ hiển thị, click icon  để hiển thị danh sách các sự kiện của đối tượng, duyệt tìm sự kiện cần xử lý, nhập tên hàm xử lý vào combobox bên phải sự kiện (hay ấn kép chuột vào comboBox để máy tạo tự động hàm xử lý).



## 5.5 Qui trình điển hình viết 1 ứng dụng bằng VC#

1. Trước hết phải nắm bắt yêu cầu phần mềm để xác định các chức năng mà ứng dụng phải cung cấp cho người dùng.
2. Phân tích sơ lược từng chức năng và tìm ra các class phân tích cấu thành chức năng tương ứng.
3. Thiết kế chi tiết các class phân tích : xác định các thuộc tính và các tác vụ cũng như phác họa giải thuật của từng tác vụ. Phân loại các class phần mềm thành 2 nhóm : nhóm các đối tượng giao diện (các form giao diện) và nhóm các class miêu tả thuật giải các chức năng bên trong của ứng dụng. Trong các ứng dụng nhỏ dùng thuật giải đơn giản, ta thường đặt các thuật giải chức năng trực tiếp trong các hàm xử lý sự kiện của các đối tượng giao diện.
4. Hiện thực phần mềm bằng VC# gồm 2 công việc chính :
   - thiết kế trực quan các form giao diện người dùng : mỗi form chứa nhiều phần tử giao diện, các phần tử giao diện thường đã có sẵn, nếu không ta phải tạo thêm 1 số đối tượng giao diện mới (User Control). Ứng với mỗi phần tử giao diện vừa tạo ra, nên thiết lập giá trị đầu cho thuộc tính "Name" và 1 vài thuộc tính cần thiết.
   - tạo hàm xử lý sự kiện cho các sự kiện cần thiết trên các phần tử giao diện rồi viết code cho từng hàm xử lý sự kiện vừa tạo ra.

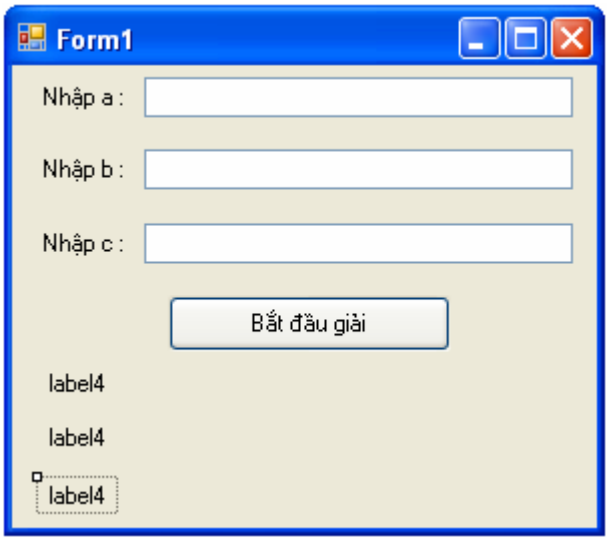
## 5.6 Thí dụ viết ứng dụng giải phương trình bậc 2

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Window, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chức Project trong listbox "Location", nhập tên Project vào

textbox "Name:", click button OK để tạo Project theo các thông số đã khai báo.

3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, việc thiết kế form là quá trình lặp 4 thao tác tạo mới/xóa/hiệu chỉnh thuộc tính/tạo hàm xử lý sự kiện cho từng đối tượng cần dùng trong form.
4. Nếu cửa sổ ToolBox chưa hiển thị chi tiết, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Click chuột vào button  (Auto Hide) nằm ở góc trên phải cửa sổ ToolBox để chuyển nó về chế độ hiển thị thường trực.
5. Duyệt tìm phần tử Label (trong nhóm Common Controls), chọn nó, dời chuột về vị trí thích hợp trong form và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Nhap a :". Nếu cần, hãy thay đổi vị trí và kích thước của Labelvà của Form.
6. Duyệt tìm phần tử TextBox (trong nhóm Common Controls), chọn nó, dời chuột về vị trí bên phải Label vừa vẽ và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính (Name) = txtA. Nếu cần, hãy thay đổi vị trí và kích thước của TextBox.
7. Lặp lại các bước 4, 5 để vẽ 2 Label "Nhập b :", "Nhập c :", 2 TextBox có (Name) = txtB, txtC, 1 button "Bắt đầu giải" có (Name) = btnStart, 3 Label có (Name) lần lượt là lblKetqua, lblX1, lblX2.
    - Đối với các đối tượng giống nhau, ta có thể dùng kỹ thuật Copy-Paste để nhân bản vô tính chúng cho dễ dàng.

Sau khi thiết kế xong, Form có dạng sau :




8. Dời chuột về button "Bắt đầu giải", ấn kép chuột vào nó để tạo hàm xử lý sự kiện Click chuột cho button, cửa sổ mã nguồn sẽ hiển thị để ta bắt đầu viết code cho hàm. Lưu ý rằng để tạo hàm xử lý sự kiện bất kỳ cho đối tượng 1 cách chính quy, ta phải hiển thị cửa sổ thuộc tính của đối tượng, rồi hiển thị danh sách các sự kiện rồi mới định nghĩa hàm xử lý sự kiện mong muốn.

9. Viết code cho hàm btnStart_Click() như sau :

```
private void btnStart_Click(object sender, EventArgs e) {
    //định nghĩa các biến cần dùng
    double a, b, c;
    double delta;
    double x1, x2;
    //mã hóa chuỗi nhập thành giá trị thực a,b,c
    a = Double.Parse(txtA.Text);
    b = Double.Parse(txtB.Text);
    c = Double.Parse(txtC.Text);
    //tính biệt số delta của phương trình
    delta = b * b - 4 * a * c;
    if (delta >= 0)  { //nếu có nghiệm thực
        x1 = (-b + Math.Sqrt(delta)) / 2 / a;
       x2 = (-b - Math.Sqrt(delta)) / 2 / a;
       lblKetqua.Text = "Phương trình có 2 nghiệm thực :";
       lblX1.Text = "X1 = " + x1;
```

```
            lblX2.Text = "X2 = " + x2;
        } else { //nếu vô nghiệm
            lblKetqua.Text = "Phương trình vô nghiệm";
            lblX1.Text = lblX2.Text = "";
        }
    }
```

10. Hiệu chỉnh hàm khởi tạo form như sau :

```
public Form1() {
  InitializeComponent();
    //xóa nội dung ban đầu của các Label kết quả
  lblKetqua.Text = lblX1.Text = lblX2.Text = "";
 }
```

11. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Hãy thử nhập từng bộ ba (a,b,c) của phương trình bậc 2 rồi ấn button "Bắt đầu giải" để giải và xem kết quả.

## 5.7 Kết chương

Chương này đã giới thiệu các đối tượng giao diện phổ dụng, qui trình tạo/xóa/hiệu chỉnh thuộc tính của đối tượng cũng như tạo hàm xử lý sự kiện cho 1 số sự kiện quan tâm trên đối tượng giao diện.

Chương này cũng đã giới thiệu qui trình điển hình để xây dựng chương trình có giao diện đồ họa được thiết kế trực quan (thay vì phải viết code khó khăn).

# Chương 6

# Tương tác với người dùng trong ứng dụng C#

## 6.1 Tổng quát về tương tác người dùng/chương trình

Trong lúc chương trình chạy, nó thường phải tương tác với người dùng. Sự tương tác gồm 2 hoạt động chính :

- chờ nhận dữ liệu do người dùng cung cấp hay chờ nhận lệnh của người dùng để thực thi 1 chức năng nào đó.
- hiển thị thông báo và/hoặc kết quả tính toán ra màn hình/máy in để người dùng biết và sử dụng.

Sự tương tác giữa người dùng và máy tính được thực hiện thông qua các thiết bị nhập/xuất (thiết bị I/O - input/output) như bàn phím/chuột để nhập dữ liệu hay lệnh, màn hình/máy in để xuất kết quả hay thông báo...

Hiện có hàng trăm hãng chế tạo thiết bị I/O, mỗi hãng chế tạo rất nhiều model của cùng 1 thiết bị (td. hãng HP chế rất nhiều model máy in phun mực, máy in laser,...). Mỗi model thiết bị của từng hãng có những tính chất vật lý riêng và khác với các model khác.

Để giúp người lập trình truy xuất các thiết bị I/O dễ dàng, độc lập với tính chất phần cứng của thiết bị, HĐH Windows và VC# đã che dấu mọi tính chất phần cứng của các thiết bị và cung cấp cho người lập trình 1 giao tiếp sử dụng duy nhất, độc lập với thiết bị : người dùng sẽ tương tác với chương trình thông qua các đối tượng giao diện :

- người dùng ra lệnh bằng cách kích hoạt sự kiện xác định của 1 đối tượng giao diện. Thí dụ click chuột vào button "Bắt đầu giải" để ra lệnh chương trình giải dùm phương trình bậc 2 có 3 tham số a, b, c đã nhập.
- nhập giá trị đúng/sai thông qua chọn/cấm chọn RadioButton hay checkbox.

- nhập chọn lựa 1/n thông qua chọn RadioButton tương ứng trong GroupBox, hay chọn mục tương ứng trong Listbox, ComboBox.
- nhập số nguyên, số thực, chuỗi thông qua TextBox...
- xuất kết quả ra màn hình thông qua các đối tượng RadioButton, Checkbox, TextBox, ListBox, ComboxBox, TreeView...

Trong trường hợp cần xuất kết quả phức tạp bất kỳ, ta xem nó như là tập hợp nhiều chuỗi văn bản, nhiều phần tử ảnh bitmap, nhiều phần tử đồ họa toán học như hình chữ nhật, hình tròn,...→ Xuất kết quả phức tạp là quá trình lặp vẽ từng phần tử cấu thành kết quả phức tạp.

## 6.2 Đối tượng vẽ và cơ chế vẽ nội dung

Các đối tượng Form, PictureBox, Printer cho phép vẽ nội dung bất kỳ lên chúng.

Mỗi lần cần vẽ lại nội dung của đối tượng (lúc bắt đầu hiển thị, lúc thay đổi vị trí, kích thước của đối tượng), máy sẽ tạo sự kiện Paint, sự kiện này sẽ kích hoạt hàm xử lý tương ứng của đối tượng. Như vậy, nếu muốn vẽ thông tin chi tiết lên đối tượng, người lập trình phải định nghĩa hàm xử lý sự kiện Paint của đối tượng và hiện thực thuật giải để vẽ chi tiết thông tin lên đối tượng.

Khi cần thiết, người lập trình có thể gọi tác vụ Refresh() của đối tượng để nhờ máy tạo dùm sự kiện Paint hầu vẽ lại đối tượng.

Template của hàm xử lý sự kiện Paint của đối tượng như sau :

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    //xác định đối tượng mục tiêu
    Control control = (Control)sender;
    //thay đổi kích thước, vị trí nếu cần
    //xác định đối tượng graphics (đối tượng vẽ) của đối tượng
    Graphics g = e.Graphics;
    //gọi các tác vụ vẽ của đối tượng vẽ như DrawImage,
```

```
        //DrawString, DrawLine,... để xuất các thông tin bitmap,
        //chuỗi văn bản, hình đồ họa toán học.
    }
```

## 6.3 Xuất chuỗi văn bản

Đối tượng vẽ (graphics) cung cấp khoảng 70 tác vụ vẽ khác nhau, mỗi tác vụ gồm nhiều biến thể (overloaded) để giúp ta điều khiển vẽ nội dung dễ dàng, tiện lợi. Ở đây chúng ta chỉ giới thiệu 1 số tác vụ phổ dụng.

Tác vụ DrawString cho phép xuất chuỗi văn bản theo định dạng xác định. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
    public void DrawString (
            string s,           //chuỗi cần xuất
            Font font,          //các tính chất font chữ cần dùng để vẽ
            Brush brush,        //màu vẽ chuỗi
            float x,            //toạ độ x của điểm canh lề chuỗi
            float y,            //tọa độ y của điểm canh lề chuỗi
            StringFormat format);      //thuộc tính điều khiển vẽ chuỗi
```

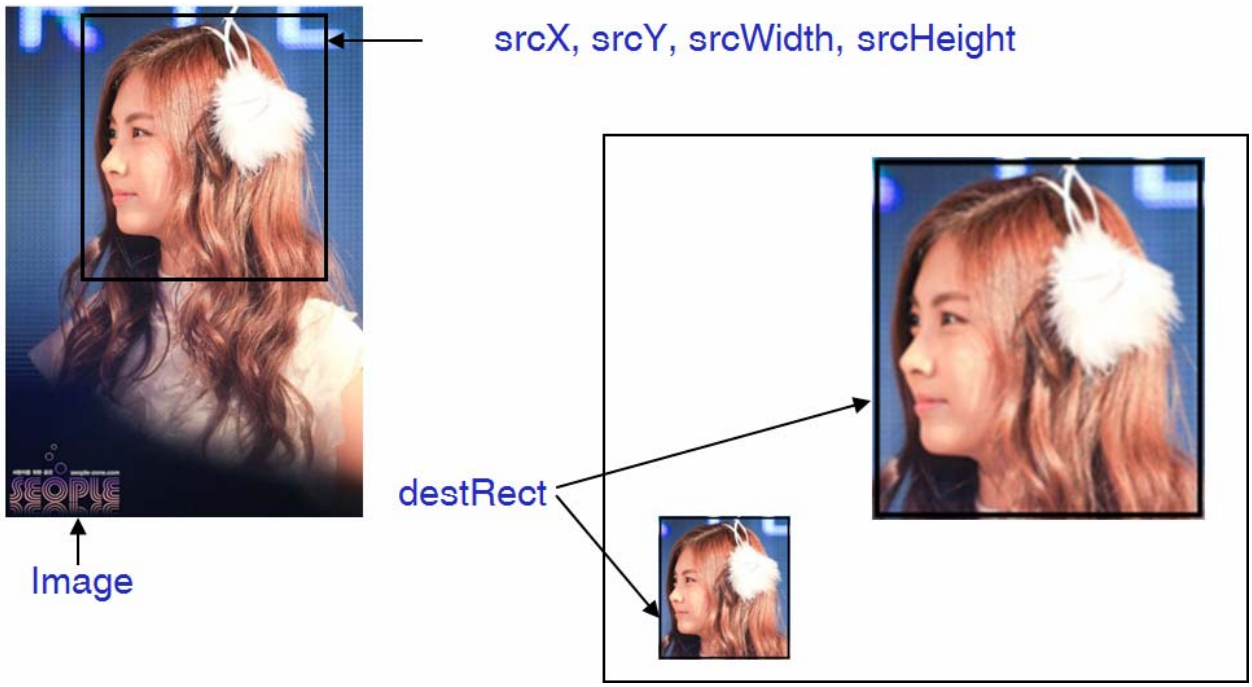
Thí dụ ta có biến now miêu tả thông tin thời điểm hiện hành, ta có thể viết đoạn code sau để rút trích thông tin từ biến now và xuất thông tin giờ/phút/giây ra giữa form ứng dụng :

```
    //tạo chuỗi miêu tả giờ/phút/giây hiện hành
    String buf = "" + now.Hour + ":" + now.Minute + ":" +
now.Second;
    //tạo đối tượng font chữ cần dùng
    Font myFont = new Font("Helvetica", 11);
    //tạo biến miêu tả chế độ canh giữa khi xuất chuỗi
    StringFormat format1 = new StringFormat(StringFormatFlags.NoClip);
    format1.Alignment = StringAlignment.Center;
    //xuất chuỗi miêu tả giờ/phút/giây
    g.DrawString(buf, myFont, System.Drawing.Brushes.Blue,
            xo, rec.Height - 35, format1);
```

## 6.4 Xuất ảnh bitmap

Tác vụ DrawImage cho phép vẽ bitmap từ nguồn có sẵn, thí dụ từ file bitmap. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
public void DrawImage (
        Image image,  //đối tượng chứa ảnh bitmap gốc
        Rectangle destRect,       //vùng chữ nhật chứa kết quả
            //trong đối tượng vẽ
        int srcX,   //tọa độ x của vùng ảnh gốc
        int srcY,   //tọa độ y của vùng ảnh gốc
        int srcWidth,    //độ rộng vùng ảnh gốc cần vẽ
        int srcHeight,   //độ cao vùng ảnh gốc cần vẽ
        GraphicsUnit srcUnit,      //đơn vị đo lường được dùng
        ImageAttributes imageAttr)     //cách thức xử lý
            //từng pixel ảnh gốc khi vẽ
```




## 6.5 Xuất hình đồ họa

### Tác vụ DrawLine

Tác vụ DrawLine cho phép vẽ đoạn thẳng được xác định bởi 2 đỉnh. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
public void DrawLine (
        Pen pen,  //miêu tả nét, màu đường vẽ
        int x1,      //tọa độ x của điểm đầu
        int y1,      //tọa độ y của điểm đầu
        int x2,      //tọa độ x của điểm cuối
        int y2       //tọa độ y của điểm cuối
)
```

Trước khi gọi DrawLine, phải tạo đối tượng Pen miêu tả nét, màu của đường vẽ :

```
//tạo pen với màu Blue, nét vẽ 2 pixel
Pen pen = new Pen(Color.FromArgb(0,0, 255), 2);
```

## Tác vụ DrawRectangle

Tác vụ DrawRectangle cho phép vẽ hình chữ nhật được xác định bởi 2 đỉnh chéo nhau. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
public void DrawRectangle (
        Pen pen,  //miêu tả nét, màu đường vẽ
        int x1,      //tọa độ x của điểm đầu
        int y1,      //tọa độ y của điểm đầu
        int x2,      //tọa độ x của điểm cuối
        int y2)      //tọa độ y của điểm cuối
```

Lưu ý tác vụ DrawRectangle chỉ vẽ đường biên, muốn tô nền hình chữ nhật, ta cần gọi tác vụ FillRectangle (đặc tả giống như tác vụ DrawRectangle), chỉ khác là tham số đầu là đối tượng mẫu tô :

```
//tạo brush với màu đỏ, tô đặc
Brush brush = new SolidBrush(Color.FromArgb(255, 0,
0));
```

## Tác vụ DrawEllipse

Tác vụ DrawEllipse cho phép vẽ hình ellipse được xác định bởi hình chữ nhật bao quanh nó. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
public void DrawEllipse (
```

```
        Pen pen,  //miêu tả nét, màu đường vẽ
        int x1,      //tọa độ x của điểm đầu
        int y1,      //tọa độ y của điểm đầu
        int x2,      //tọa độ x của điểm cuối
        int y2)      //tọa độ y của điểm cuối
```

Lưu ý tác vụ DrawEllipse chỉ vẽ đường biên, muốn tô nền hình ellipse, ta cần gọi tác vụ FillEllipse (đặc tả giống như tác vụ DrawEllipse), chỉ khác là tham số đầu là đối tượng mẫu tô :

```
        //tạo brush với màu đỏ, tô đặc
        Brush brush = new SolidBrush(Color.FromArgb(255, 0,
    0));
```

## Tác vụ DrawPolygon

Tác vụ DrawPolygon cho phép vẽ hình nhiều cạnh khép kín. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
    public void DrawPolygon (
        Pen pen,  //miêu tả nét, màu đường vẽ
        Point[] points)  //danh sách các đỉnh của polygon
```

Lưu ý tác vụ DrawPolygon chỉ vẽ đường biên, muốn tô nền hình polygon, ta cần gọi tác vụ FillPolygon (đặc tả giống như tác vụ DrawPolygon), chỉ khác là tham số đầu là đối tượng mẫu tô :

```
        //tạo brush với màu đỏ, tô đặc
        Brush brush = new SolidBrush(Color.FromArgb(255, 0,
    0));
```

## Tác vụ DrawCurve

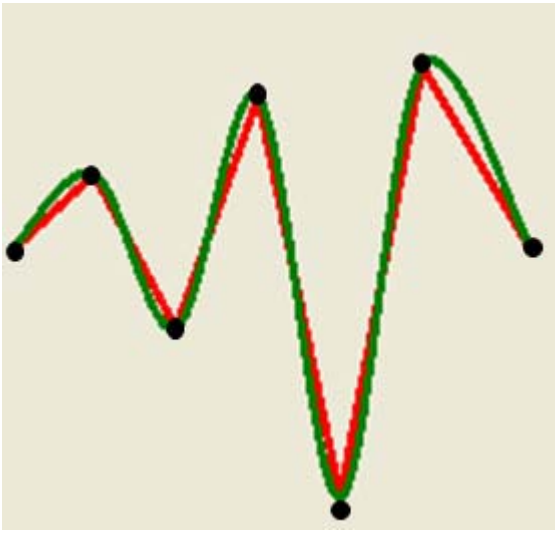
Tác vụ DrawCurve cho phép vẽ cong trơn xuyên qua nhiều điểm theo phép tension xác định. Nó có nhiều biến thể, biến thể khá mạnh và dùng phổ biến có đặc tả như sau :

```
    public void DrawCurve (
        Pen pen,  //miêu tả nét, màu đường vẽ
        Point[] points    //danh sách các đỉnh của polygon
        int offset,  //vị trí điểm bắt đầu vẽ trong danh sách
```

```
        int numberOfSegments, //số đoạn cần vẽ
        float tension     //phép tension được dùng
    )
```

**Thí dụ :**

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    //tạo 2 bút vẽ cho đường thẳng và cong
    Pen redPen = new Pen(Color.Red, 3);
    Pen greenPen = new Pen(Color.Green, 3);
    //tạo các đỉnh
    Point point1 = new Point(10, 100), point2 = new Point(40,
    75);
    Point point3 = new Point(70, 125), point4 = new Point(100,
    50);
    Point point5 = new Point(130, 180), point6 = new Point(160,
    40);
    Point point7 = new Point(200, 100);
    Point[] curvePoints = { point1, point2, point3, point4, point5,
    point6, point7 };
    //vẽ các đoạn thẳng.
    e.Graphics.DrawLines(redPen, curvePoints);
    //thiết lập offset, số đoạn cong, và tension.
    int offset = 0, numSegments = 6;
    float tension = 0.5F;
    //vẽ đường cong trơn qua các đỉnh.
    e.Graphics.DrawCurve(greenPen, curvePoints, offset,
    numSegments, tension);
}
```

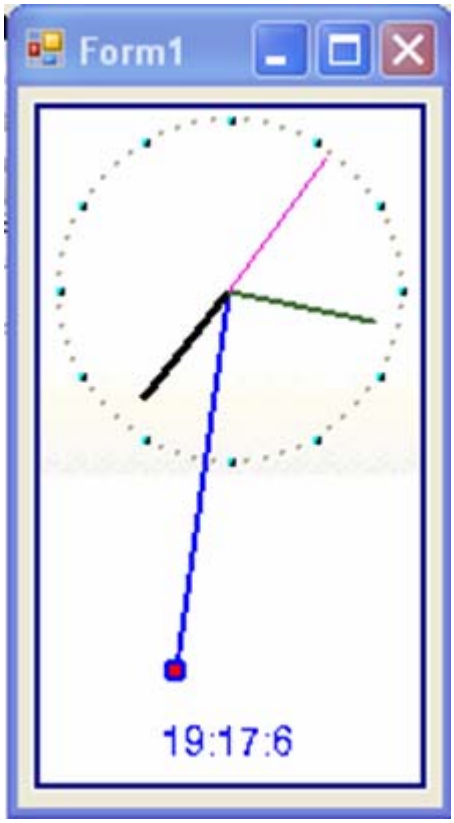


## 6.6 Thí dụ viết ứng dụng vẽ đối tượng phức hợp

Để củng cố kiến thức về các tác vụ xuất nội dung tổng hợp chứa chuỗi văn bản, ảnh bitmap và các hình đồ họa toán học, chúng ta hãy viết ứng dụng giả lập đồng hồ treo tường có 3 kim giờ/phút/giây và có quả lắc theo góc 20 độ.

Phân tích thông tin cần xuất, ta thấy có các thành phần :

- hình bitmap miêu tả khung đồng đồ, bản số đồng hồ.
- 4 đoạn thẳng miêu tả 3 kim giờ/phút/giây và cần lắc. Vòng tròn nhỏ miêu tả quả lắc. Các hình toán học này thay đổi vị trí theo thời gian.
- chuỗi hiển thị giờ/phút/giây.




Dùng đối tượng Timer với thời gian đếm khoảng 40ms, mỗi lần đếm xong nó tạo sự kiện Paint để kích hoạt hàm vẽ lại Form ứng dụng. Như vậy mỗi giây ta vẽ lại khoảng 25 lần, tốc độ như thế này là vừa đủ để người dùng cảm thấy đồng hồ gần như thật.

Qui trình điển hình để xây dựng ứng dụng đồng hồ quả lắc gồm các bước sau đây :

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. VCDongho), click button OK để tạo Project theo các thông số đã khai báo.
3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, lúc này form hoàn toàn trống, chưa chứa đối tượng giao diện nào.

4. Nếu cửa sổ ToolBox chưa hiển thị, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Duyệt tìm phần tử Timer (trong nhóm Comopents hay nhóm All Window Forms), chọn nó, dời chuột vào trong form (ở vị trí nào cũng được vì đối tượng này không được hiển thị) và vẽ nó với kích thước tùy ý. Hiệu chỉnh thuộc tính (Name) = myTimer.
5. Chọn đối tượng myTimer, cửa sổ thuộc tính của nó sẽ hiển thị, click icon để hiển thị danh sách các sự kiện của đối tượng, ấn kép chuột vào comboBox bên phải sự kiện Tick để máy tạo tự động hàm xử lý cho sự kiện này.
6. Viết code cụ thể cho hàm như sau :

```
//hàm phục vụ Timer
private void myTimer_Tick(object sender, EventArgs e) {
    myTimer.Stop();   //dừng đếm timer
    this.Refresh();      //vẽ lại form theo giờ hiện hành
}
```

7. Ấn phải chuột vào mục Form1.cs trong cửa sổ Solution Explorer rồi chọn option View Designer để hiển thị lại cửa sổ thiết kế Form. Chọn Form, cửa sổ thuộc tính của nó sẽ hiển thị, click icon  để hiển thị danh sách các sự kiện của Form, duyệt tìm sự kiện Paint, ấn kép chuột vào comboBox bên phải sự kiện Paint để máy tạo tự động hàm xử lý cho sự kiện này. Viết code cụ thể cho hàm như sau :

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    //tạo đối tượng image gốc
    Image bgimg = Image.FromFile("c:\\bgclock.bmp");
    //xác định đối tượng mục tiêu
    Control control = (Control)sender;
    //thay đổi kích thước form theo ảnh khung đồng hồ
    control.Size = new Size(bgimg.Width + 10 + 8, bgimg.Height
+ 10 + 35);
    //xác định đối tượng graphics (đối tượng vẽ) của đối tượng
```

```
Graphics g = e.Graphics;
//vẽ bitmap miêu tả khung đồng hồ
g.DrawImage(bgimg, 5,5);
//định nghĩa các biến cần dùng
Rectangle rec = control.DisplayRectangle;
Pen hPen;
Brush hBrush;
int xo,yo,rql,rh,rm, rs;
int x, y;
//thiết lập tâm đồng hồ
xo = 76; yo = 74;
//thiết lập bán kính cần lắc, kim giờ/phút/giây
rql = 140; rh = 50; rm = 55; rs = 60;
//tạo pen để vẽ cần lắc
hPen = new Pen (Color.FromArgb(0,0, 255),2);
//tạo brush để tô nền quả lắc
hBrush = new SolidBrush(Color.FromArgb(255, 0, 0));
//xác định giờ/phút/giây hiện hành
DateTime now = DateTime.Now;
//tính góc của cần lắc (góc quay max. là 40 độ)
double goc = 80*now.Millisecond/1000;
if (goc < 40) goc = goc +70;
else goc = 150-goc;
//đổi góc cần lắc từ độ ra radian
goc = goc*3.1416/180;
//xác định tâm quả lắc (điểm còn lại của cần lắc)
x = xo+(int)(rql*Math.Cos(goc));
y = yo+(int)(rql*Math.Sin(goc));
//vẽ cần lắc
g.DrawLine(hPen, xo, yo, x, y);
//vẽ quả lắc
g.FillEllipse(hBrush, x-3, y-3, 5, 5);
g.DrawEllipse(hPen,x-4,y-4,7,7);
//tạo pen để vẽ kim giờ
```

```
hPen = new Pen(Color.FromArgb(0,0,0),3);
//tính góc của kim giờ
goc = 90+360*(now.Hour+(double)now.Minute/60)/12;
//đổi góc từ độ ra radian
goc = goc*3.1416/180;
//xác định tọa độ đỉnh thứ 2 của kim giờ
x = xo - (int)(rh * Math.Cos(goc));
y = yo - (int)(rh * Math.Sin(goc));
//vẽ kim giờ
g.DrawLine(hPen, xo, yo, x, y);

//tạo pen để vẽ kim phút
hPen = new Pen(Color.FromArgb(65,110,55),2);
//tính góc của kim phút
goc = 90+360*now.Minute/60;
//đổi góc từ độ ra radian
goc = goc*3.1416/180;
//xác định tọa độ đỉnh thứ 2 của kim phút
x = xo - (int)(rm * Math.Cos(goc));
y = yo - (int)(rm * Math.Sin(goc));
//vẽ kim phút
g.DrawLine(hPen, xo, yo, x, y);
//tạo pen để vẽ kim giây
hPen = new Pen(Color.FromArgb(237,5,220),1);
//tính góc của kim giây
goc = 90+360*now.Minute/60;
//đổi góc từ độ ra radian
goc = goc*3.1416/180;
//xác định tọa độ đỉnh thứ 2 của kim giây
x = xo - (int)(rs * Math.Cos(goc));
y = yo - (int)(rs * Math.Sin(goc));
//vẽ kim giây
g.DrawLine(hPen, xo, yo, x, y);
//tạo chuỗi miêu tả giờ/phút/giây hiện hành
```

```
        String buf = "" + now.Hour + ":" + now.Minute + ":" +
now.Second;
        //tạo đối tượng font chữ cần dùng
        Font myFont = new Font("Helvetica", 11);
        //tạo biến miêu tả chế độ canh giữa khi xuất chuỗi
        StringFormat format1 = new
StringFormat(StringFormatFlags.NoClip);
        format1.Alignment = StringAlignment.Center;
        //xuất chuỗi miêu tả giờ/phút/giây
        g.DrawString(buf, myFont, System.Drawing.Brushes.Blue,
                xo, rec.Height - 35, format1);
        //cho phép timer chạy tiếp
    myTimer.Start();
}
```

8. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Xem kết quảvà đánh giá kết quả.

## 6.7 Kết chương

Chương này đã giới thiệu cách thức tương tác giữa người dùng và chương trình để nhập/xuất dữ liệu.

Chương này cũng đã giới thiệu các đối tượng giao diện cùng các tác vụ xuất dữ liệu dạng chuỗi, dạng bitmap, dạng hình đồ họa toán học. Kết hợp 3 loại dữ liệu này, ta có thể tạo kết xuất bất kỳ.

# Chương 7

# Ghi/đọc dữ liệu của ứng dụng C# ra file

## 7.1 Tổng quát về đời sống của dữ liệu ⊂ ứng dụng VC#

Khi chương trình bắt đầu chạy, nó sẽ tạo ra dữ liệu, xử lý dữ liệu cho đến khi hoàn thành nhiệm vụ và dừng chương trình.

Khi chương trình kết thúc, thường các dữ liệu của chương trình sẽ bị mất. Do đó, trong lúc chương trình chạy, khi cần thiết, ta sẽ ghi các dữ liệu ra file để lưu giữ lâu dài. Sau này khi cần dùng lại, ta sẽ đọc dữ liệu từ file vào các biến chương trình để xử lý tiếp.

VC# cung cấp 3 class đối tượng FileStream, BinaryWriter, BinaryReader (trong namespace System.IO) để phục vụ việc ghi/đọc biến dữ liệu thuộc các kiểu định sẵn phổ biến ra file ở dạng nhị phân (dạng được mã hóa trong chương trình). Nếu biết được cách thức mã hóa dữ liệu (được trình bày trong môn Nhập môn điện toán), ta sẽ kiểm tra trực tiếp được kết quả được ghi ra file.

## 7.2 Ghi dữ liệu ra file ở dạng nhị phân

Qui trình điển hình để ghi dữ liệu trong chương trình ra file ở dạng nhị phân (không giãi mã dữ liệu) :

```
//1. tạo đối tượng quản lý file
FileStream stream = new FileStream("C:\\data.bin",
    FileMode.Create);
//2. tạo đối tượng phục vụ ghi file
BinaryWriter writer = new BinaryWriter(stream);
//3. xử lý dữ liệu theo yêu cầu chương trình
int i = -15;
double d = -1.5;
String s = "Nguyễn Văn Hiệp";
bool b = true;
//4. ghi dữ liệu ra file
writer.Write(b); writer.Write(i); writer.Write(d); writer.Write(s);
```

```
//5. đóng các đối tượng được dùng lại
writer.Close(); stream.Close();
```

Tác vụ write của class BinaryWriter có 14 biến thể 1 tham số để ghi đýợc 14 kiểu dữ liệu định sẵn phổ biến sau đây :

- Boolean
- Byte, SByte
- Int16, Int32, Int64
- UInt16, UInt32, UInt64
- Single, Double, Decimal
- Byte[] , Char[]
- Char, String

Muốn ghi nội dung của biến thuộc 1 trong 14 kiểu dữ liệu định sẵn trên, ta gọi tác vụ write theo dạng sau :

```
writer.Write(varname); //writer là biến đối tượng
    BinaryWriter
```

Tác vụ write của class BinaryWriter còn có 2 biến thể 3 tham số để ghi được các phần tử chọn lọn trong danh sách :

```
//ghi count byte từ vị trí index trong danh sách buffer
BinaryWriter.write(Byte[] buffer, int index, int count);
//ghi count ký tự từ vị trí index trong danh sách buffer
BinaryWriter.write(Char[] buffer, int index, int count);
```

## 7.3 Đọc dữ liệu từ file ở dạng nhị phân

Qui trình điển hình để đọc dữ liệu từ file nhị phân vào chýõng trình (không mã hóa dữ liệu) :

```
//1. tạo đối tượng quản lý file
FileStream stream = new FileStream("C:\\data.bin",
    FileMode.Open);
//2. tạo đối tượng phục vụ đọc file
BinaryReader reader = new BinaryReader(stream);
//3. định nghĩa các biến dữ liệu theo yêu cầu chương trình
int i; double d; String s; bool b;
//4. đọc dữ liệu từ file vào các biến
```

```
b= reader.ReadBoolean();       //đọc trị luận lý
i = reader.ReadInt32();    //đọc số nguyên 32 bit
d = reader.ReadDouble();       //đọc số thực chính xác kép
s = reader.ReadString();  //đọc chuỗi
//5. đóng các đối tượng được dùng lại
reader.Close(); stream.Close();
```

## 7.4 Ghi dữ liệu ra file ở dạng text

Mặc dù việc ghi/đọc dữ liệu ra file ở dạng nhị phân (y như trong máy) là rất đơn giản, hiệu quả (khỏi phải thực hiện mã hóa/giải mã dữ liệu). Tuy nhiên, file nhị phân cũng có 1 số nhược điểm :

- người dùng khó xem, khó kiểm tra nội dung của file.
- người dùng khó tạo dữ liệu dưới dạng nhị phân để chương trình đọc vào xử lý.

Trong trường hợp cần nhập nhiều thông tin cho chương trình, ta không thể dùng các đối tượng giao diện như textbox, listbox. Trong trường hợp này, ta sẽ dùng trình soạn thảo văn bản để soạn dữ liệu dưới dạng văn bản hầu xem/kiểm tra/sửa chữa dễ dàng. File văn bản chứa dữ liệu là danh sách gồm nhiều chuỗi, mỗi chuỗi miêu tả 1 dữ liệu (luận lý, số nguyên, số thực, chuỗi,...), các chuỗi sẽ được ngăn cách nhau bởi 1 hay nhiều dấu ngăn. Dấu ngăn thường dùng là ký tự gióng cột TAB, ký tự xuống hàng.

VC# cung cấp các class đối tượng có tên là FileStream, StreamWriter, StreamReader (trong namespace System.IO) để phục vụ việc ghi/đọc biến dữ liệu thuộc các kiểu định sẵn phổ biến ra file ở dạng text. Trong trường hợp này, tác vụ ghi dữ liệu sẽ tự động giải mã dạng nhị phân sang dạng chuỗi tương đương trước khi ghi ra file. Khi đọc lại chuỗi miêu tả dữ liệu, ta phải mã hóa dữ liệu từ dạng chuỗi thành dạng nhị phân trước khi chứa vào biến dữ liệu bên trong chương trình.

Qui trình điển hình để ghi dữ liệu trong chương trình ra file ở dạng text (giãi mã dữ liệu nhị phân thành dạng chuỗi) :

```
//1. tạo đối tượng quản lý file
FileStream stream = new FileStream("C:\\data.txt",
    FileMode.Create);
//2. tạo đối tượng phục vụ ghi file
StreamWriter writer = new StreamWriter(stream,
    Encoding.Unicode);
//3. xử lý dữ liệu theo yêu cầu chương trình
int i = -15;
double d = -1.5;
String s = "Nguyễn Văn Hiệp";
bool b = true;
//4. ghi dữ liệu ra file
writer.Write(b); writer.Write("\t");      //ghi 1 dữ liệu và dấu ngăn
writer.Write(i); writer.WriteLine(); //ghi 1 dữ liệu và dấu ngăn
writer.Write(d); writer.Write("\t"); //ghi 1 dữ liệu và dấu ngăn
writer.Write(s); writer.Write("\t"); //ghi 1 dữ liệu và dấu ngăn
//5. đóng các đối tượng được dùng lại
writer.Close();
stream.Close();
```

## 7.5 Đọc dữ liệu từ file text

Qui trình điển hình để đọc dữ liệu từ file text vào chương trình (mã hóa dữ liệu từ chuỗi thành dữ liệu nhị phân) :

```
//1. tạo đối tượng quản lý file
FileStream stream = new FileStream("C:\\data.txt",
    FileMode.Open);
//2. tạo đối tượng phục vụ đọc file
StreamReader reader=new
    StreamReader(stream,Encoding.Unicode);
//3. định nghĩa các biến dữ liệu theo yêu cầu chương trình
int i; double d; String s; bool b; String buf=null;
//4. đọc dữ liệu từ file vào các biến
```

```
ReadItem(reader,ref buf); b = Boolean.Parse(buf);        //đọc trị luận lý
ReadItem(reader,ref buf); i = Int32.Parse(buf);     //đọc số nguyên 32 bit
ReadItem(reader,ref buf); d = Double.Parse(buf); //đọc số thực
ReadItem(reader,ref buf); s = buf;   //đọc chuỗi
//5. đóng các đối tượng được dùng lại
reader.Close(); stream.Close();
//hàm đọc chuỗi miêu tả 1 dữ liệu nào đó
static void ReadItem(StreamReader reader, ref String buf) {
    char ch;
    //thiết lập chuỗi nhập được lúc đầu là rỗng
    buf = "";
    //lặp cho đến khi hết file
    while (reader.EndOfStream != true) {
        ch = (char)reader.Read(); //đọc 1 ký tự
        if (ch != '\t' && ch != '\r' && ch!='\n') //nếu là ký tự bình thường
                buf += ch.ToString();
        else { //nếu là dấu ngăn thì kết thúc việc đọc chuỗi
            if (ch == '\r') reader.Read(); //đọc bỏ luôn ký tự '\n'
            return;       //trả kết quả về nơi gọi
        }
    }
}
```

## 7.6 Thí dụ về đọc/ghi dữ liệu cổ điển

Giả sử ta có 2 file A.txt và B.txt chứa thông tin về 2 ma trận theo qui ước như sau :

- chuỗi đầu tiên miêu tả số hàng
- chuỗi kế tiếp miêu tả số cột
- các chuỗi còn lại miêu tả giá trị từng phần tử, từng hàng từ trên xuống, mỗi hàng từ trái sang phải.
- các chuỗi dữ liệu được ngăn cách nhau bởi dấu ngăn ',', '\r', '\n'

Thí dụ ma trận (5,7) được chứa như sau :

```
5, 7
2, 3, 4, 5, 6, 7, 8
9, 10, 11, 12, 13, 14, 15
16, 17, 18, 19, 20, 21, 22
23, 24, 25, 26, 27, 28, 29
30, 31, 32, 33, 34, 35, 36
```

Ta hãy viết chương trình đọc 2 ma trận A và B vào bộ nhớ, tính ma trận tổng rồi xuất kết quả ra file văn bản S.txt.

Qui trình điển hình để xây dựng chương trình theo yêu cầu trên như sau :

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Console Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. TongMT), click button OK để tạo Project theo các thông số đã khai báo.
3. Ngay sau Project vừa được tạo ra, cửa sổ soạn code cho chương trình được hiển thị. Thêm lệnh using sau đây vào đầu file :

```
using System.IO;
```

4. Viết code cho thân class Program như sau :

```
class Program {
        static double[,] A;     //ma trận A
        static double[,] B;     //ma trận B
        static double[,] S;     //ma trận S
        static int hang, cot;
        //hàm đọc ma trận vào biến bộ nhớ
        static void ReadMT(string path, ref double[,] A, ref int hang, ref int
cot) {
```

```
        //1. tạo đối tượng quản lý file
        FileStream stream = new FileStream(path, FileMode.Open);
        //2. tạo đối tượng phục vụ đọc file
        StreamReader reader = new StreamReader(stream,
Encoding.ASCII);
        //3. định nghĩa các biến dữ liệu theo yêu cầu chýõng
trình
        int i, j; string buf = "";
        //4. đọc dữ liệu từ file vào các biến
        ReadItem(reader, ref buf); hang = Int32.Parse(buf);      //đọc      số
hàng
        ReadItem(reader, ref buf); cot = Int32.Parse(buf);       //đọc số cột
        //phân phối vùng nhớ cho ma trận
        A = new double[hang, cot];
        //đọc từng phần tử ma trận
        for (i = 0; i < hang; i++)
              for (j = 0; j < cot; j++) {
           ReadItem(reader, ref buf);
           A[i, j] = Double.Parse(buf);   //đọc số thực
              }
        //5. đóng các đối týợng đýợc dùng lại
        reader.Close(); stream.Close();
}
//hàm ghi ma trận ra file text
static void WriteMT(string path, double[,] A, int hang, int cot) {
        //1. tạo đối tượng quản lý file
        FileStream stream = new FileStream(path, FileMode.Create);
        //2. tạo đối tượng phục vụ ghi file
        StreamWriter writer = new StreamWriter(stream, Encoding.ASCII);
        //3. định nghĩa các biến dữ liệu theo yêu cầu chýõng
trình
        int i, j;
        //4. ghi dữ liệu từ các biến ra file
```

```
        writer.Write(hang); writer.Write(", "); //ghi số hàng và dấu
ngăn
        writer.Write(cot); writer.WriteLine();  //ghi số cột và dấu
ngăn
        //ghi ma trận
        for (i = 0; i < hang; i++) { //ghi từng hàng ma trận
                for (j = 0; j < cot; j++) {
                    writer.Write(A[i, j]); writer.Write(",");        //ghi phần tử i,j
                }
                writer.WriteLine(); //ghi dấu xuống hàng
        }
        //5. đóng các đối tượng đượC dùng lại
    writer.Close(); stream.Close();
}
//hàm đọc chuỗi miêu tả 1 dữ liệu nào đó
static void ReadItem(StreamReader reader, ref String buf) {
        char ch = '\0';
        //thiết lập chuỗi nhập đượC lúc đầu là rỗng
        buf = "";
        while (reader.EndOfStream != true) {//lặp đọc bỏ các dấu ngăn
                ch = (char)reader.Read(); //đọc 1 ký tự
                if (ch != ',' && ch != '\r' && ch != '\n')
                        break;              //nếu là ký tự bình thýờng thì dừng
        }
        buf += ch.ToString();
        //lặp đọc các ký tự của chuỗi dữ liệu
        while (reader.EndOfStream != true) {
                ch = (char)reader.Read(); //đọc 1 ký tự
                if (ch == ',' || ch == '\r' || ch == '\n')
                        return;      //nếu là dấu ngăn thì dừng
                buf += ch.ToString();       //chứa ký tự vào bộ đệm
        }
}
//điểm nhập chương trình
```

```
        static void Main(string[] args) {
            int i, j, h = 0, c = 0;
            //đọc ma trận A từ file c:\A.txt
            ReadMT("c:\\a.txt", ref A, ref hang, ref cot);
            //đọc ma trận B từ file c:\B.txt
            ReadMT("c:\\b.txt", ref B, ref h, ref c);
            if (h != hang || c != cot)  {
                Console.WriteLine("Hai ma trận A, B không cùng kích thýớc");
                return;
            }
            //phân phối ma trận tổng S
            S = new double[hang, cot];
            //tính ma trận tổng S
            for (i = 0; i < hang; i++)
                for (j = 0; j < cot; j++) S[i, j] = A[i, j] + B[i, j];
            //xuất ma trận kết quả ra file c:\S.txt
            WriteMT("c:\\S.txt", S, hang, cot);
        } //hết hàm Main
} //hết class program
```

5. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Lưu ý là trước khi chạy ứng dụng, hãy dùng chương trình soạn thảo văn bản (NotePad, WordPad,...) soạn thảo 2 file c:\A.txt, c:\B.txt chứa dữ liệu của 2 ma trận A và B theo qui ước ở slide 12.

## 7.7 Ghi/Đọc hệ thống đối tượng ra/vào file

Đọc/ghi dữ liệu trên các biến thuộc kiểu giá trị (int, double, char[],..) rất dễ vì nội dung của các biến này không chứa tham khảo đến các thành phần khác. Ngược lại, việc đọc/ghi nội dung của 1 đối tượng thường rất khó khăn vì đối tượng có thể chứa nhiều tham khảo đến các đối tượng khác và các đối tượng có thể tham khảo vòng lẫn nhau. Để hỗ trợ việc đọc/ghi nội dung của đối tượng, VC# đề nghị kỹ thuật "Serialization".

Một đối tượng chỉ có thể được "serialize/deserialize" (ghi/đọc dùng kỹ thuật Serialization) nếu nó thuộc class "serializable". Để định nghĩa 1 class "serializable" dễ dàng và đơn giản nhất, ta chỉ cần thêm mệnh đề [Serializable] trước phát biểu định nghĩa class đó :

```
[Serializable]
public class A {...}
```

Để ghi 1 đối tượng (và toàn bộ các đối tượng mà nó phụ thuộc) ở dạng nhị phân, ta viết template như sau :

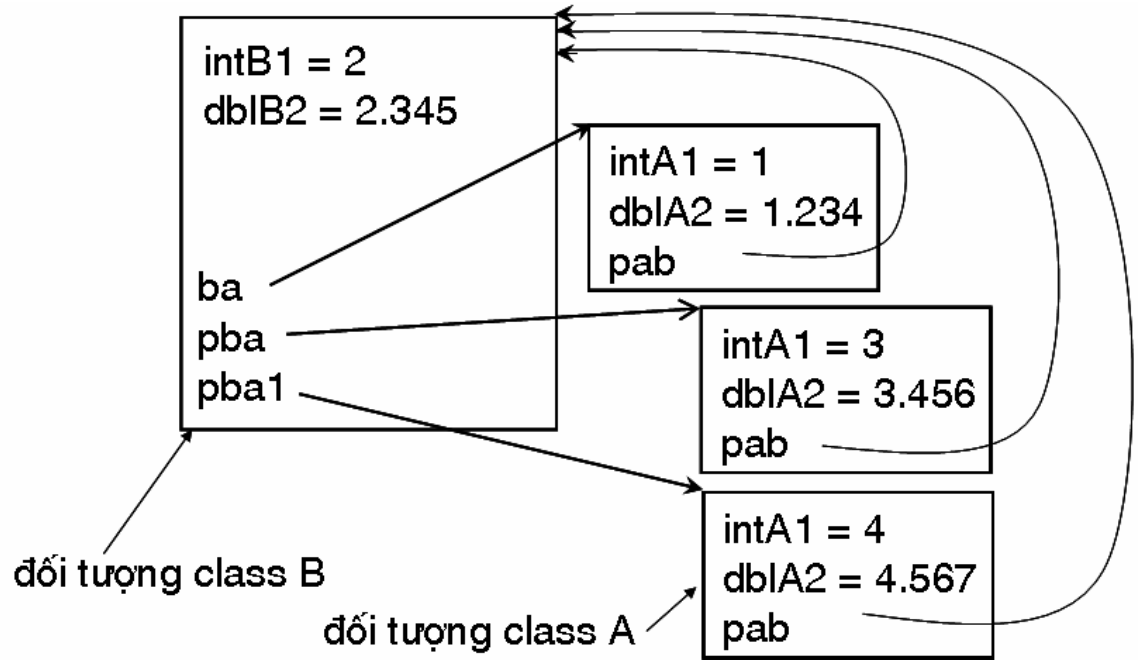
```
//1. xử lý và xây dựng đối tượng
B b;
//2. định nghĩa đối tượng FileStream miêu tả file chứa kết quả
FileStream fs = new FileStream("c:\\data.obj",
    FileMode.Create);
//3. tạo đối tượng BinaryFormatter phục vụ ghi đối tượng
BinaryFormatter formatter = new BinaryFormatter();
//4. gọi tác vụ Serialize  của formatter để ghi đối tượng
formatter.Serialize(fs, b);
//đóng file lại
fs.Flush();
fs.Close();
```

Để đọc lại 1 đối tượng (và toàn bộ các đối tượng mà nó phụ thuộc) ở dạng nhị phân ta, viết template như sau :

```
//1. định nghĩa biến đối tượng để chứa nội dung từ file
B b;
//2. định nghĩa đối tượng FileStream miêu tả file chứa dữ liệu đã có
FileStream fs = new FileStream("c:\\data.obj", FileMode.Open);
//3. tạo đối tượng BinaryFormatter phục vụ đọc đối tượng
BinaryFormatter formatter = new BinaryFormatter();
//4. gọi tác vụ Deserialize để đọc đối tượng từ file vào
b = (B) formatter.Deserialize(fs);
//đóng file lại
fs.Close();
```

## 7.8 Thí dụ về đọc/ghi hệ thống đối tượng

Giả sử ta có hệ thống các đối tượng với trạng thái và mối quan hệ giữa chúng cụ thể như sau. Lưu ý chúng có mối quan hệ bao gộp dạng vòng :




Giả sử biến b đang tham khảo tới đối tượng class B. Hãy viết chương trình ghi hệ thống đối tượng này lên file để khi cần, đọc lại vào bộ nhớ hầu xử lý tiếp.

Qui trình xây dựng ứng dụng giải quyết yêu cầu trên như sau :

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Console Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. WRObject), click button OK để tạo Project theo các thông số đã khai báo.
3. Ngay sau Project vừa được tạo ra, cửa sổ soạn code cho chương trình được hiển thị. Thêm lệnh các using sau đây vào đầu file :

```
using System.IO;
```

```
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
```

4. Viết code cho hàm Main và các hàm dịch vụ khác nhau sau :

```
class Program {
    static String fbuf;
    static void Main(string[] args) { //điểm nhập của chương trình
      //xây dựng hệ thống đối tượng và ghi lên file
      Create_SaveObject();
      //đọc lại hệ thống đối tượng
      //ReadObject();
    }
    //hàm xây dựng hệ thống đối tượng và ghi lên file
    public static void Create_SaveObject() {
      //khở tạo đối tượng b theo hình ở silde 24
      B b = new B();
      b.init(2,2.345);
      b.Setba(1,1.234,b);
      b.Setpba(3,3.1416,b);
      b.Setpba1(4,4.567,b);
      //ghi đối tượng b dùng kỹ thuật Serialization
      try {
            //1. định nghĩa đối tượng FileStream miêu tả file chứa kết
    quả
            FileStream fs = new FileStream("c:\\data.obj", FileMode.Create);
            //2. tạo đối tượng BinaryFormatter phục vụ ghi đối tượng
            BinaryFormatter formatter = new BinaryFormatter();
            //3. gọi tác vụ Serialize  của formatter để ghi đối tượng
            formatter.Serialize(fs,b);
            //4. đóng file lại
            fs.Flush(); fs.Close();
      } catch (Exception e) { Console.WriteLine(e.ToString()); }
  }
    //hàm đọc đối tượng b dùng kỹ thuật Serialization
```

```
    public static void ReadObject() {
     try {
      //1. định nghĩa đối tượng FileStream miêu tả file chứa dữ liệu đã có
      FileStream fs = new FileStream("c:\\data.obj", FileMode.Open);
      //2. tạo đối tượng BinaryFormatter phục vụ đọc đối tượng
      BinaryFormatter formatter = new BinaryFormatter();
      //3. gọi tác vụ Deserialize để đọc đối tượng từ file vào
      B b = (B) formatter.Deserialize(fs);
      //4. đóng file lại
      fs.Close();
      } catch (Exception e) { Console.WriteLine(e.ToString()); }
    } //hết hàm Main
} //hết class program
```

5. Ấn phải chuột vào phần tử gốc của cây Project trong cửa sổ Solution Explorer, chọn option Add.Class, đặt tên là A.cs để tạo ra file đặc tả class A. Khi cửa sổ hiển thị mã nguồn của class A hiển thị, đặc tả class A như đoạn code dưới đây :

```
//thêm lệnh using sau ở đầu file
using System.Runtime.Serialization;
namespace WRObject {
  [Serializable]
  public class A {
        //định nghĩa các thuộc tính dữ liệu
        private int intA1;
        private double dblA2;
        private B pab;
        //định nghĩa các tác vụ
        public A() {}
        public void init(int a1, double a2, B p) {
              this.intA1 = a1;
              this.dblA2 = a2;
              this.pab = p;
        }
```

```
  }
}
```

6. Ấn phải chuột vào phần tử gốc của cây Project trong cửa sổ Solution Explorer, chọn option Add.Class, đặt tên là B.cs để tạo ra file đặc tả class B. Khi cửa sổ hiển thị mã nguồn của class B hiển thị, đặc tả class B như đoạn code dưới đây :

```
//thêm lệnh using sau ở đầu file
using System.Runtime.Serialization;
namespace WRObject  {
  [Serializable]
  public class B {
        //định nghĩa các thuộc tính dữ liệu
        private    int intB1;
        private    double dblB2;
        private    A ba;
        private   A pba;
        private   A pba1;
        //định nghĩa các tác vụ
        public B() { }
public void init(int b1, double b2) {
  this.intB1 = b1; this.dblB2 = b2;
  ba = new A(); pba = new A(); pba1 = new A();
}
public void Setba (int a1, double a2, B b) {
  this.ba.init (a1,a2,b);
}
public void Setpba (int a1, double a2, B b) {
  this.pba.init (a1,a2,b);
}
public void Setpba1 (int a1, double a2, B b) {
  this.pba1.init (a1,a2,b);
}
} //hết class B
```

```
} //hết namespace WRObject
```

7. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Hệ thống đối tượng sẽ được tạo ra và lưu lên file c:\data.obj.
8. Hiển thị cửa sổ soạn mã nguồn file Program.cs, chú thích lệnh gọi Create_SaveObject(); và bỏ chú thích lệnh gọi ReadObject();. Dời chuột về lệnh "B b = (B) formatter.Deserialize(fs);" trong hàm ReadObject(), click chuột vào lệnh trái của lệnh này để thiết lập điểm dừng. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Ứng dụng sẽ dừng ở lệnh dừng. Dời chuột về biến b, ta thấy cửa sổ hiển thị giá trị của biến này lúc này là null.
9. Ấn F10 để thực hiện đúng lệnh này rồi dừng lại, dời chuột về biến b, rồi mở rộng cây phân cấp miêu tả nội dung biến b và thấy nó chứa đúng thông tin như slide 14, nghĩa là chương trình đã đọc được toàn bộ hệ thống đối tượng đã lưu giữ trước đây.

## 7.9 Kết chương

Chương này đã giới thiệu các đối tượng phục vụ ghi/đọc dữ liệu ra/vào file cùng các tác vụ ghi/đọc dữ liệu cổ điển ra/vào file.

Chương này cũng đã giới thiệu các đối tượng phục vụ ghi/đọc hệ thống đối tượng ra/vào file cùng các tác vụ ghi/đọc hệ thống đối tượng có mối quan hệ tham khảo phức tạp ra/vào file.

# Chương 8

# Xây dựng class tổng quát hóa bằng VC#

## 8.0 Dẫn nhập

Chương này giới thiệu một loại class đặc biệt : class tổng quát hóa, nó giúp người lập trình tối thiểu hóa việc viết họ các class có tính chất giống nhau.

Chương này cũng giới thiệu cách miêu tả các thông tin ràng buộc kèm theo từng tên kiểu hình thức được dùng trong class tổng quát hóa, cách dùng class tổng quát hóa để yêu cầu máy sinh mã tự động ra class cụ thể.

## 8.1 Tổng quát về interface và class tổng quát hóa

Trong phương pháp xây dựng chương trình hướng đối tượng, chương trình là tập các đối tượng sống và tương tác lẫn nhau để hoàn thành nhiệm vụ. Số lượng các đối tượng cấu thành phần mềm thường rất lớn, nhưng chúng thường thuộc 1 số loại xác định. Viết phần mềm hướng đối tượng là quá trình lặp đặc tả các loại đối tượng cấu thành chương trình.

Trong các chương trình lớn và phức tạp, số loại đối tượng cần đặc tả có thể lớn nên thời gian, công sức đặc tả chúng cũng sẽ lớn.

Để giảm nhẹ thời gian, công sức đặc tả các đối tượng, mô hình hướng đối tượng đã giới thiệu tính thừa kế : ta không đặc tả đối tượng từ đầu (zero) mà dùng lại đặc tả có sẵn rồi hiệu chỉnh/thêm các thành phần mới. Tuy nhiên, thừa kế cũng chỉ giúp giảm nhẹ công sức đặc tả interface/class, chứ chưa triệt tiêu việc đặc tả.

Trong chương này, chúng ta sẽ thấy được phương pháp khác, nó cũng cho phép ta giảm nhẹ và triệt tiêu việc đặc tả interface/class cho 1 số class cấu thành ứng dụng. Phương pháp này được gọi là tổng quát hóa.

Ta biết, 1 hàm không tham số chỉ có thể thực hiện 1 thuật giải cố định trên các dữ liệu cố định và cho kết quả cố định, cho dù ta gọi nó bao nhiêu lần. Thí dụ hàm Cos() chỉ có thể tính được Cos của góc nào đó (được xác định cứng trong thân hàm).

Nếu thêm tham số cho hàm, nó sẽ thực hiện 1 thuật giải nhưng trên các dữ liệu khác nhau mà những lần gọi khác nhau người ta truyền cho nó, như vậy kết quả cũng sẽ khác nhau. Thí dụ hàm Cos(x) có thể tính Cos của góc x bất kỳ, tùy thuộc mỗi lần gọi nó, người ta truyền góc nào.

Như vậy, ta nói hàm có tham số sẽ có tính năng tổng quát hơn hàm không tham số. Càng có nhiều tham số, hàm càng có tính tổng quát hơn.

Tương tự, nếu ta đặc tả 1 class bình thường như đã thấy trong các chương trước, ta nói class dạng này là class cụ thể. Class cụ thể chỉ có thể chứa và xử lý các dữ liệu xác định trước. Class cụ thể chỉ có thể tạo ra các đối tượng có dữ liệu được class xác định.

Trong lập trình, chúng ta mơ ước có ai đó viết dùm mình các class cụ thể mà chương trình cần. Class tổng quát hóa sẽ giúp ta điều này. Nhiệm vụ của class tổng quát hóa là viết dùm con người các class cụ thể mà chương trình cần dùng.

Sự khác biệt giữa class tổng quát hóa và class cụ thể cũng giống như sự khác biệt giữa hàm có tham số và hàm không có tham số. Cụ thể ta sẽ định nghĩa từ 1 đến n tên kiểu hình thức mà sẽ được dùng trong class tổng quát hóa. Trong thân của class tổng quát hóa, ta sẽ dùng các tên kiểu hình thức để đặc tả cho các dữ liệu. Như vậy class tổng quát hóa không thể tạo đối tượng cụ thể (điều này không có nghĩa), nó chỉ có thể tạo ra class cụ thể khi được truyền các tên kiểu cụ thể.

Để thấy rõ sự khác biệt giữa class tổng quát hóa và class cụ thể, trước tiên ta hãy xây dựng 1 class quản lý Stack các số nguyên có dung lượng tùy ý, nó có 2 tác vụ chức năng là push(int) và pop().

## 8.2 Class cụ thể : Stack các số nguyên

```
//định nghĩa class Stack các số nguyên
public class IntStack {
	//định nghĩa các thuộc tính cần dùng
	private int[] data;	//danh sách đặc các số nguyên trong stack
	private int top;  // chỉ số phần tử đỉnh stack
	private int max; // số lượng max hiện hành của stack
	private int GROWBY = 4; //bước tăng dung lượng stack
	//hàm constructor
	public IntStack() {
		top = 0;
		max =GROWBY;
		//lúc đầu, phân phối GROWBY phần tử
		data = (int[])new int[max];
	}
	//hàm push phần tử vào stack
	public bool push(int newVal) {
	int[] newdata;
	if (top==max) { //kiểm tra xem stack đầy chưa
		//tạo vùng nhớ chứa các phần tử stack
		//hơn GROWBY phần tử
		try {
			newdata = (int[])new int[GROWBY+max];
		} catch (Exception e) {
			return false; //nếu hết bộ nhớ thì báo lỗi
		}
		//copy các phần tử từ stack cũ vào stack mới
		for (int i = 0; i<max; i++) newdata[i] =data[i];
		//ghi nhớ vùng stack mới
		data = newdata;
		max += GROWBY;
	}
	//chứa phần tử mới vào đỉnh stack
```

```
        data[top++] = newVal;
        return true;        //báo thành công
        }
        //hiện thực hàm pop phần tử từ đỉnh stack
        public int pop() {
        if (top == 0) //kiểm tra hết stack chưa
              throw new Exception ("Cạn stack");
        else return data[--top];      //return phần tử ở đỉnh stack
        }
}
```

## 8.3 Class tổng quát hóa : Stack các phần tử kiểu T

Đặc tả class IntStack ở mục 8.2 miêu tả stack các số nguyên. Giả sử trong 1 chương trình nào đó, ta cần thêm stack các số thực, stack các chuỗi, stack các trị luận lý, stack các đối tượng,...

Nếu ta tự viết lấy các class còn lại (thường bằng cách dùng lại đặc tả class IntStack rồi hiệu chỉnh lại các chi tiết cho phù hợp với kiểu phần tử trong stack mới) thì cũng được, nhưng đây là cách làm tốn nhiều thời gian, công sức, nhưng không hiệu quả, không tin cậy, dễ gây lỗi,...

Do đó ta sẽ định nghĩa class tổng quát hóa miêu tả Stack các phần tử thuộc kiểu T nào đó bằng cách :

- dùng class IntStack cụ thể, tìm và thay thế kiểu phần tử cụ thể (int) thành tên kiểu hình thức (T).
- định nghĩa kiểu hình thức T trong danh sách tham số của phát biểu class.

```
//định nghĩa class tổng quát hóa : Stack các phần tử thuộc kiểu T
public class ValueStack <T> {
        //định nghĩa các thuộc tính cần dùng
        private T[] data;//danh sách đặc các phần tử T trong stack
        private int top;   // chỉ số phần tử đỉnh stack
        private int max; // số lượng max hiện hành của stack
        private int GROWBY = 4;  //bước tăng dung lượng stack
```

```
//hàm constrcutor
public ValueStack() {
	top = 0;
	max =GROWBY;
	//lúc đầu, phân phối GROWBY phần tử
	data = (T[])new T[max];
}
//hàm push phần tử vào stack
public bool push(T newVal) {
T[] newdata;
if (top==max) { //kiểm tra xem stack đầy chưa
	//tạo vùng nhớ chứa các phần tử stack
	//hơn GROWBY phần tử
	try {
		newdata = (T[])new T[GROWBY+max];
	} catch (Exception e) {
		return false; //nếu hết bộ nhớ thì báo lỗi
	}
	//copy các phần tử từ stack cũ vào stack mới
	for (int i = 0; i<max; i++) newdata[i] =data[i];
	//ghi nhớ vùng stack mới
	data = newdata;
	max += GROWBY;
}
//chứa phần tử mới vào đỉnh stack
data[top++] = newVal;
return true;	//báo thành công
}
//hiện thực hàm pop phần tử từ đỉnh stack
public T pop() {
if (top == 0) //kiểm tra hết stack chưa
	throw new Exception ("Cạn stack");
else return data[--top];	//return phần tử ở đỉnh stack
}
```

}

## 8.4 Ràng buộc về tham số kiểu hình thức

Trong class tổng quát hóa ValueStack được định nghĩa ở mục 8.3, ta có dùng kiểu hình thức có tên là T. Nếu không có thông tin gì khác ngoài tên hình thức T thì trong thân của class ValueStack, ta chỉ có thể gán các biến dữ liệu thuộc kiểu hình thức T chứ không thể thực hiện được gì khác trên các dữ liệu thuộc kiểu T này.

Trong trường hợp cần thực hiện nhiều hoạt động xử lý khác trên các dữ liệu thuộc kiểu hình thức T, thí dụ gởi thông điệp nhờ thực hiện 1 tác vụ nào đó, ta phải định nghĩa thông tin ràng buộc về kiểu T.

VC# cho phép ta định nghĩa thông tin ràng buộc về kiểu hình thức T theo cú pháp sau :

```
//định nghĩa class có 3 tham số kiểu hình thức
class ValueStack <T, K, L>
where T : <thông tin ràng buộc về kiểu T>
where K : <thông tin ràng buộc về kiểu K>
where L : <thông tin ràng buộc về kiểu L>
```

Ta có thể dùng 1 trong 5 dạng ràng buộc sau đây :

1. where T: struct → kiểu T phải là kiểu giá trị (kiểu cổ điển)
2. where T: class → kiểu T phải là kiểu tham khảo (class, delegate,...)
3. where T: new( ) → kiểu T phải là kiểu đối tượng và phải có hàm contructor không tham số (contructor mặc định)
4. where T: <base class name> → kiểu T phải là kiểu đối tượng và phải tương thích với class <base class name>
5. where T: <interface name> → kiểu T phải là kiểu đối tượng và phải tương thích với interface <interface name>

Mặc định, nếu không khai báo gì thì được hiểu là where T: struct. Như vậy class ValueStack được định nghĩa ở mục 8.3 chỉ quản lý các dữ liệu cổ điển như số nguyên, số thực,...

Nếu muốn viết class tổng quát hóa miêu tả stack các đối tượng bất kỳ, ta chỉ cần hiệu chỉnh lại lệnh đặc tả class từ :

class ValueStack <T>

hay

class ValueStack <T> where T : struct

thành

class RefStack <T> where T : class

## 8.5 Sử dụng class tổng quát hóa

Sau khi có class cụ thể A, chương trình có thể tạo đối tượng cụ thể thuộc class cụ thể bằng cách gọi lệnh new :

```
A obj = new A();
```

Bản thân class tổng quát hóa không thể tạo ra đối tượng được dùng trong chương trình như các class thông thường. Nhiệm vụ của nó là tạo ra đặc tả class cụ thể. Thí dụ sau khi đã đặc tả được 2 class tổng quát hóa ValueStack, RefStack trong mục 8.3 và 8.4, nếu ta cần viết tự động class stack các nguyên, stack các thực,... thì ta chỉ cần viết lệnh như sau :

```
ValueStack <int> si;  //định nghĩa biến stack các số nguyên
ValueStack <double> sd;  //định nghĩa biến stack các số thực
RefStack <MyInt>  sri;       //biến stack các đối tượng nguyên
RefStack <MyDouble>  srd;      //biến stack các đối tượng thực
```

Chương trình sau demo việc dùng ValueStack để quản lý các số nguyên :

```
static void Main(string[] args) {
    int i;
    //định nghĩa class stack các số nguyên và biến thuộc class này
    ValueStack<int> si = new ValueStack<int> ();
    //push lần lượt các trị từ -5 tới 5
```

```
    for (i = -5; i <= 5; i++)  {
        if (!si.push(i)) {
            Console.WriteLine("Không push được nữa!!!");
            return;
        }
    }
    //pop ra từng phần tử cho đến khi hết stack
    try {
        while (true) {
                int ci = si.pop();
                Console.WriteLine("Trị vừa pop ra là : " + ci);
        }
    }
    //xử lý lỗi khi hết stack
    catch (Exception e) {
        Console.Write("Hết stack. Ấn Enter để đóng cửa sổ");
        Console.Read();
    }
}
```

Chương trình sau demo việc dùng RefStack để quản lý các đối tượng, mỗi đối tượng chứa 1 số nguyên :

```
 static void Main(string[] args) {
    int i;
    //định nghĩa class stack các đối tượng nguyên (class MyInt)
    //và biến thuộc class này
    RefStack<MyInt> si = new RefStack<MyInt> ();
    //push lần lượt các trị từ -5 tới 5
    for (i = -5; i <= 5; i++)  {
        if (!si.push(new MyInt(i))) {
            Console.WriteLine("Không push được nữa!!!");
            return;
        }
    }
    //pop ra từng phần tử cho đến khi hết stack
```

```
        try {
            while (true) {
                MyInt ci = si.pop();
                Console.WriteLine("Trị vừa pop ra là : " + ci.Value);
            }
        }
        //xử lý lỗi khi hết stack
        catch (Exception e) {
            Console.Write("Hết stack. Ấn Enter để đóng cửa sổ");
            Console.Read();
        }
}
```

Chương trình ở slide trước có sử dụng class MyInt để quản lý số nguyên được định nghĩa như sau :

```
 class MyInt {
        //định nghĩa thuộc tính vật lý chứa số nguyên
        private int m_value;
        //định nghĩa thuộc tính luận lý để truy xuất số nguyên
        public int Value {
              get { return m_value; }
              set { m_value = value; }
        }
        //định nghĩa hàm contructor
        public MyInt(int val) { m_value = val; }
}
```

## 8.6 Kết chương

Chương này đã giới thiệu một loại class đặc biệt : class tổng quát hóa, nó giúp người lập trình tối thiểu hóa việc viết họ các class có tính chất giống nhau.

Chương này cũng đã giới thiệu cách miêu tả các thông tin ràng buộc kèm theo từng tên kiểu hình thức được dùng trong class tổng quát hóa, cách dùng class tổng quát hóa để yêu cầu máy sinh mã tự động ra class cụ thể.

# Chương 9

# Tạo đối tượng giao diện cá nhân hóa bằng VC#

## 9.0 Dẫn nhập

Chương này giới thiệu cách thức dùng tính thừa kế để tạo mới 3 loại đối tượng giao diện cá nhân hóa phổ biến là User Control, Inherited Control và Owner-drawn Control.

Chương này cũng giới thiệu cách thức viết chương trình sử dụng lại các đối tượng giao diện cá nhân hóa.

## 9.1 Tổng quát về giao diện cá nhân hóa

Mỗi chương trình dùng giao diện đồ họa thường có nhiều cửa sổ giao diện. Mỗi cửa sổ giao diện chứa nhiều đối tượng giao diện. Microsoft đã cung cấp sẵn nhiều đối tượng giao diện (control) phổ dụng để ta thiết kế form giao diện dễ dàng. Tuy nhiên trong từng ứng dụng, có thể ta cần 1 số đối tượng giao diện đặc thù, ta gọi chúng là đối tượng cá nhân hóa (user control).

Thường có 3 dạng đối tượng giao diện cá nhân hóa :

1. User Control : là dạng đơn giản nhất, nó thừa kế class UserControl sẵn có, tích hợp nhiều control có sẵn để tạo đối tượng cá nhân hóa. Thí dụ 1 LoginControl gồm 2 TextBox để nhập username, password và 1 Button đăng nhập.
2. Inherited Control : chức năng và hành vi của nó gần giống control đã có sẵn. Để xây dựng nó, ta thừa kế class có sẵn mà chức năng gần giống nhất, rồi hiệu chỉnh (override) 1 số tác vụ để thể hiện chức năng thay đổi. Ta cũng có thể thêm mới 1 số tác vụ để thể hiện các chức năng tăng cường. Thí dụ MyTextBox có chức năng gần giống như TextBox có sẵn, nhưng nó có nhiều chế độ khác nhau, ở mỗi chế độ nó phản ứng khác nhau. Thí dụ nếu ở chế độ

nhập số nguyên, nó chỉ cho phép nhập ký số, chứ không cho nhập ký tự khác.

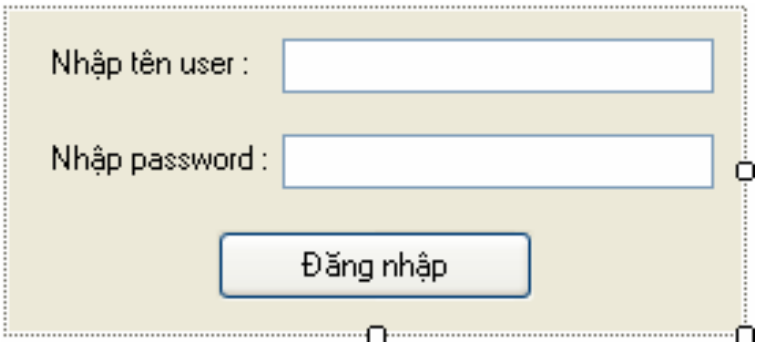
3. Owner-drawn control : chức năng giống y như control có sẵn nhưng bộ mặt giao diện thì khác. Ta sẽ thừa kế class có sẵn mà chức năng giống y rồi override tác vụ OnPaint để vẽ lại bộ mặt mới. Thí dụ HeartControl là một Button nhưng bộ mặt không phải là khung chữ nhật bình thường mà là trái tim màu đỏ tươi.

## 9.2 Xây dựng User control

Qui trình xây dựng 1 hay nhiều User Control gồm các bước chính :

1. chạy Visual Studio .Net, mở/tạo Project loại "Windows Control Library" để quản lý 1 hay nhiều user control.
2. Tạo mới 1 User Control rồi thiết kế giao diện/viết code cho nó.
3. Dịch project ra file *.dll, ta gọi file này là thư viện chứa các user control.

Thí dụ ta hãy xây dựng 1 User Control có tên là LoginControl, nó gồm 2 TextBox và 1 Button để giúp người dùng đăng ký tài khoản để truy xuất hệ thống. Hình ảnh LoginControl như sau :




1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Control Library"

trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. MyUserControls), click button OK để tạo Project theo các thông số đã khai báo.

3. Ngay sau Project vừa được tạo ra, nó có sẵn 1 User Control mới có tên mặc định là UserControl1, nó chỉ là 1 vùng hình chữ nhật trống, chứ chưa có gì. Dời chuột về cửa sổ Solution Explorer (thường ở trên phải màn hình), ấn kép chuột vào mục UserControl1.cs để hiển thị menu lệnh, chọn option Rename, nhập tên mới là LoginControl.cs và chọn button Yes khi được hỏi.

4. Nếu cử sổ ToolBox chưa hiển thị, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Duyệt tìm phần tử Label (trong nhóm Common Controls), chọn nó, dời chuột về vị trí thích hợp trong LoginControl và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Nhập tên user :". Nếu cần, hãy thay đổi vị trí và kích thước của Label và của LoginControl.

5. Dời chuột về cửa sổ ToolBox, duyệt tìm phần tử TextBox (trong nhóm Common Controls), chọn nó, dời chuột về vị trí thích hợp trong LoginControl (bên phải Label vừa vẽ) và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính (Name) = txtUser. Nếu cần, hãy thay đổi vị trí và kích thước của TextBox.

6. Lặp lại các bước 4 và 5 để vẽ Label "Nhập passwword :", TextBox có (Name) = txtPassword, 1 button "Đăng nhập" có (Name) = btnLogin.

7. Dời chuột về và chọn button "Đăng nhập", cửa sổ thuộc tính của nó sẽ hiển thị, click icon  để hiển thị danh sách các sự kiện Button, duyệt tìm sự kiện Click, ấn kép chuột vào comboBox bên phải của Click để máy tạo tự động hàm xử lý rồi viết code cho hàm này như sau :

```
 private void btnLogin_Click(object sender, EventArgs e)  {
    //kiểm tra đã nhập user name chưa
    if (txtUser.Text.Length == 0) {
        MessageBox.Show("Hãy nhập tên user."); return;
    }
    //kiểm tra đã nhập password chưa
    if (txtPassword.Text.Length == 0) {
        MessageBox.Show("Hãy nhập password."); return;
    }
    //tạo sự kiện Click để gọi hàm xử lý sự kiện Click
    //do người lập trình ứng dụng viết
    OnSubmitClicked(sender,e);
}
```

8. Viết thêm đoạn code định nghĩa delegate, event và 2 thuộc tính UserName, Password như sau (nằm trước hay sau hàm sự lý Click chuột cho button) :

```
//định nghĩa delegate phục vụ cho event
public delegate void SubmitClickedHandler(object sender, EventArgs e);
//định nghĩa event SubmitClicked
public event SubmitClickedHandler SubmitClicked;
//định nghĩa hàm xử lý sự kiện SubmitClicked
protected virtual void OnSubmitClicked(object sender, EventArgs e) {
    // kiểm tra xem có hàm xử lý sự kiện SubmitClicked ?
    //nếu có thì gọi nó
    if (SubmitClicked != null)  {
        SubmitClicked(sender, e);  // Notify Subscribers
    }
}
//định nghĩa thuộc tính giao tiếp có tên là UserName
public string UserName {
    get { return txtUser.Text; }
    set { txtUser.Text = value; }
```

```
}
//định nghĩa thuộc tính giao tiếp có tên là Password
public string Password {
    get { return txtPassword.Text; }
    set { txtPassword.Text = value; }
}
```

9. Chọn menu Build.Build Solution để dịch và tạo file thư viện chứa các user control. Nếu có lỗi thì sữa và dịch lại.
10. Nếu dịch thành công, file thư viện có tên là MyUserControls.dll sẽ được tạo ra trong thư mục con Debug (hay Release tùy chế độ dịch) trong thư mục chứa Project. Ta nên copy file này vào thư mục chung chứa các file thư viện để sau này dùng tiện lợi hơn.

## Xây dựng ứng dụng dùng User Control

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. UseLoginControl), click button OK để tạo Project theo các thông số đã khai báo.
3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, việc thiết kế form là quá trình lặp 4 thao tác tạo mới/xóa/hiệu chỉnh thuộc tính/tạo hàm xử lý sự kiện cho từng đối tượng cần dùng trong form.
4. Nếu cửa sổ ToolBox chưa hiển thị, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Dời chuột vào trong cửa sổ Toolbox, ấn phải chuột để hiển thị menu lệnh, chọn option "Choose Items". Khi cửa sổ "Choose Toolbox Items" hiển thị, click chuột vào

button Browse để hiển thị cửa sổ duyệt tìm file, hãy duyệt tìm đến thư mục chứa file MyUserControls.dll vừa xây dựng được trong các slide trước, chọn file dll rồi click button OK để "add" các usercontrol trong thư viện này vào cửa sổ Toolbox của Project ứng dụng. Bây giờ việc dùng LoginControl giống y như các điều khiển có sẵn khác.

5. Duyệt tìm phần tử LoginControl (trong nhóm General ở cuối cửa sổ Toolbox), chọn nó, dời chuột về vị trí thích hợp trong Form và vẽ nó với kích thước mong muốn.

6. Chọn đối tượng LoginControl để hiển thị cửa sổ thuộc tính của nó, click chuột vào button Events để hiển thị các event của nó. duyệt tìm event SubmitClicked vào tạo hàm xử lý cho event này. Viết code cho hàm xử lý như sau :

```
private void loginControl1_SubmitClicked(object sender,
EventArgs e) {
    //viết code xử lý việc đăng nhập tài khoản
    //ở đây chỉ hiển thị thông báo để kiểm tra
    MessageBox.Show("Đã đăng ký tài khoản : "
            + loginControl1.UserName);
}
```

7. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Hãy thử sử dụng đối tượng LoginControl và đánh giá kết quả.

## 9.3 Xây dựng Inherited control

Qui trình xây dựng 1 hay nhiều Inherited Control gồm các bước chính :

1. chạy Visual Studio .Net, mở/tạo Project loại "Windows Control Library" để quản lý 1 hay nhiều user control.

2. Ấn phải chuột vào gốc của cây Project trong cửa sổ "Solution Explorer", chọn option Add.User Control để tạo mới 1 User Control.

3. Hiển thị cửa sổ soạn mã nguồn của user Control, hiệu chỉnh lại tên class base cần thừa kế rồi override/tăng cường các tác vụ chức năng mong muốn.
4. Dịch project ra file *.dll, ta gọi file này là thư viện chứa các user control.

Thí dụ ta hãy xây dựng 1 Inherited Control có tên là MyTextBox, nó là TextBox nhưng có thể hoạt động ở 1 trong nhiều chế độ khác nhau :

- Common (giống như textbox của .Net),
- Text (chỉ cho nhập các ký tự alphabet),
- NumInt (chỉ cho phép nhập các ký số),
- NumReal (chỉ cho phép nhập các ký số và dấu chấm thập phân).

Qui trình xây dựng MyTextBox và chứa nó trong thư viện có sẵn (thư viện chứa đối tượng LoginControl) như sau :

1. Chạy VS .Net, chọn menu File.Open.Project để hiển thị cửa sổ duyệt file. Duyệt và tìm file *.sln quản lý Project "Windows Control Library" có sẵn để mở lại Project này.
2. Quan sát cây Project, chúng ta đã thấy có mục LoginControl.cs quản lý user control đã xây dựng trong mục 9.2. Ấn phải chuột vào gốc của cây Project trong cửa sổ "Solution Explorer", chọn option Add.User Control để tạo mới 1 User Control, nhập tên là MyTextBox.cs rồi click button Add để tạo mới nó.
3. Lúc này control mới chỉ là 1 vùng hình chữ nhật trống. Dời chuột về mục MyTextBox.cs trong cửa sổ Project, ấn phải chuột trên nó rồi chọn option "View Code" để hiển thị cửa sổ soạn mã nguồn cho MyTextBox control.
4. Thêm lệnh định nghĩa kiểu liệt kê các chế độ làm việc của MyTextBox :

```
public enum ValidationType {
    Common = 0,     //giống như TextBox bình thường
    NumInt,         //chỉ nhận các ký số
    NumReal,        //chỉ nhận các ký số và dấu chấm thập phân
    Text }          //chỉ nhận các ký tự chữ
```

5. Hiệu chỉnh lại lệnh định nghĩa class MyTextBox để thừa kế class TextBox (thay vì UserControl như mặc định). Nội dung chi tiết của class MyTextBox được liệt kê ở các slide sau.

6. Chọn menu Build.Build Solution để dịch và tạo file thư viện chứa các user control. Nếu có lỗi thì sữa và dịch lại. Lưu ý khi máy báo lỗi ở hàng lệnh this.AutoScaleMode = ... thì hãy chú thích hàng lệnh này hay xóa nó luôn cũng được.

```
public partial class MyTextBox : TextBox {
    bool fPoint;
    //hàm contructor
    public MyTextBox() : base() {
        InitializeComponent();
        //đăng ký hàm xử lý sự kiện KeyPress
        this.KeyPress += new KeyPressEventHandler(OnKeyPress);
    }
    //định nghĩa thuộc tính ValidateFor miêu tả chế độ làm việc
    private int intValidType = (int)ValidationType.Text;
    public ValidationType ValidateFor {
            get { return (ValidationType)intValidType; }
            set { intValidType = (int)value; }
    }
    //hàmxử lý sự kiện gỏ phím KeyPress
    protected void OnKeyPress(object sender,
        KeyPressEventArgs e) {
            //xác định mã ký tự được nhập
           char ch = e.KeyChar;
         //kiểm tra chế độ hoạt động để phản ứng
            switch (intValidType)  {
```

```
        case (int)ValidationType.Common:
            //nếu là kiểu tổng quát, thì không xử lý thêm gì cả
            return;
        case (int)ValidationType.NumInt:
            //nếu là kiểu số nguyên thì chỉ nhận ký số
            if (!Char.IsDigit(ch)) e.KeyChar = (char)0;
            return;
       case (int)ValidationType.NumReal:
            //nếu là kiểu số thực thì chỉ nhận ký số + dấu .
            if (Char.IsDigit(ch)) return;
            if (ch == '.' && fPoint==false) {
                fPoint = true; return;
            }
            e.KeyChar = (char)0;
            return;
        case (int)ValidationType.Text:
            //nếu là kiểu chuỗi văn bản thì chỉ nhận ký tự chữ
            ch = Char.ToLower(ch);
            if (ch < 'a' || 'z' < ch) e.KeyChar = (char)0;
            return;
    } } //kết thúc lệnh switch, hàm OnKeyPress, ...
```

## Xây dựng ứng dụng dùng Inherited control

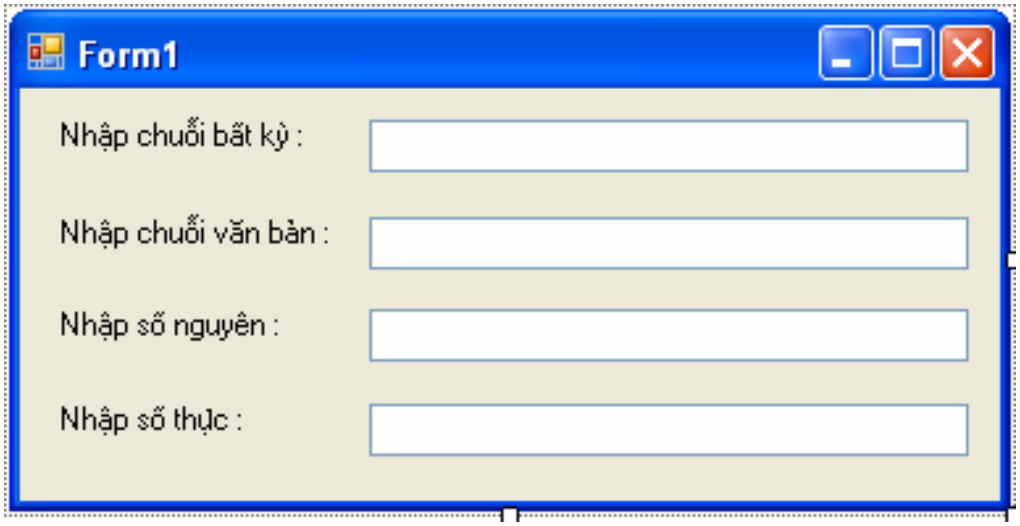
1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. UseMyTextBox), click button OK để tạo Project theo các thông số đã khai báo.
3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, việc thiết kế form là quá trình lặp 4 thao tác tạo

mới/xóa/hiệu chỉnh thuộc tính/tạo hàm xử lý sự kiện cho từng đối tượng cần dùng trong form.

4. Nếu cửa sổ ToolBox chưa hiển thị, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Dời chuột vào trong cửa sổ Toolbox, ấn phải chuột để hiển thị menu lệnh, chọn option "Choose Items". Khi cửa sổ "Choose Toolbox Items" hiển thị, click chuột vào button Browse để hiển thị cửa sổ duyệt tìm file, hãy duyệt tìm đến thư mục chứa file MyUserControls.dll vừa xây dựng được trong các slide trước, chọn file dll rồi click button OK để "add" các usercontrol trong thư viện này vào cửa sổ Toolbox của Project ứng dụng.
5. Duyệt tìm phần tử Label (trong nhóm Common Controls), chọn nó, dời chuột về vị trí thích hợp trong form và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Nhập chuỗi bất kỳ :". Nếu cần, hãy thay đổi vị trí và kích thước của Label và Form.
6. Duyệt tìm phần tử MyTextBox (trong nhóm General ở cuối cửa sổ Toolbox), chọn nó, dời chuột về vị trí thích hợp trong Form (bên phải Label vừa vẽ) và vẽ nó với kích thước mong muốn. Vào cửa sổ thuộc tính của đối tượng MyTextBox vừa vẽ, đặt thuộc tính (Name) = txtCommon, thuộc tính ValidateFor = Common để nó hoạt động ở chế độ nhập ký tự bình thường.
7. Lặp 2 bước 5 và 6 ba lần để tạo thêm 3 cặp (Label, MyTextBox) khác, các MyTextBox tạo mới lần lượt có thuộc tính ValidateFor = Text, NumInt, NumReal để hoạt động trên hoặc chuỗi văn bản, hoặc số nguyên, hoặc số thực.

   Đối với các đối tượng giống nhau, ta có thể dùng kỹ thuật Copy-Paste để nhân bản vô tính chúng cho dễ dàng.

Sau khi thiết kế xong, Form có dạng sau :




8. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Hãy thử nhập các loại ký tự vào các đối tượng MyTextBox và đánh giá kết quả.

## 9.4 Xây dựng Owner-drawn control

Qui trình xây dựng 1 hay nhiều Owner-drawn Control cũng giống như Inherited control, gồm các bước chính :

1. chạy Visual Studio .Net, mở/tạo Project loại "Windows Control Library" để quản lý 1 hay nhiều user control.
2. Ấn phải chuột vào gốc của cây Project trong cửa sổ "Solution Explorer", chọn option Add.User Control để tạo mới 1 User Control.
3. Hiển thị cửa sổ soạn mã nguồn của User Control, hiệu chỉnh lại tên class base cần thừa kế rồi override/tăng cường các tác vụ chức năng mong muốn, trong đó thiết yếu nhất là hàm OnPaint để vẽ bộ mặt của đối tượng giao diện.
4. Dịch project ra file *.dll, ta gọi file này là thư viện chứa các user control.

Thí dụ ta hãy xây dựng 1 Owner-drawn Control có tên là HeartButton, nó là Button nhưng bộ mặt không phải là hình chữ nhật có đường viền thông thường mà là một trái tim màu đỏ tươi.

Qui trình xây dựng HeartButton và chứa nó trong thư viện có sẵn (thư viện đã tạo ra trong mục 9.2 và 9.3) như sau :

1. Chạy VS .Net, chọn menu File.Open.Project để hiển thị cửa sổ duyệt file. Duyệt và tìm file *.sln quản lý Project "Windows Control Library" có sẵn để mở lại Project này.
2. Quan sát cây Project, ta thấy có mục LoginControl.cs quản lý usercontrol đã xây dựng trong mục 9.2, mục MyTextBox.cs quản lý inherited control đã xây dựng trong mục 9.3. Ấn phải chuột vào gốc cây Project trong cửa sổ "Solution Explorer", chọn option Add.User Control để tạo mới 1 User Control, nhập tên là HeartButton.cs rồi click button Add để tạo mới nó.
3. Lúc này control mới chỉ là 1 vùng hình chữ nhật trống. Dời chuột về mục HeartButton.cs trong cửa sổ Project, ấn phải chuột trên nó rồi chọn option "View Code" để hiển thị cửa sổ soạn mã nguồn cho HeartButton control.
4. Hiệu chỉnh lại lệnh định nghĩa class HeartButton để thừa kế class Button (thay vì UserControl như mặc định). Nội dung chi tiết của class HeartButton được liệt kê ở các slide sau.
5. Chọn menu Build.Build Solution để dịch và tạo file thư viện chứa các user control. Nếu có lỗi thì sữa và dịch lại. Lưu ý khi máy báo lỗi ở hàng lệnh this.AutoScaleMode = ... thì hãy chú thích hàng lệnh này hay xóa nó luôn cũng được.

```
public partial class HeartButton : Button {
    //hàm contructor của class
    public HeartButton() {
        InitializeComponent();
    }
   //hàm vẽ bộ mặt của button
    protected override void OnPaint(PaintEventArgs e) {
      //xác định đối tượng vẽ của Button
```

```
        Graphics g = e.Graphics;
         //tạo đối tượng image gốc chứa ảnh trái tim màu đỏ
         Image bgimg = Image.FromFile("c:\\bgbutton.jpg");
         //vẽ inage gốc theo chế độ zoom vào button
         g.DrawImage(bgimg, 0, 0,this.Width,this.Height);
        //định nghĩa đối tượng miêu tả cách thức hiển thị chuỗi
         StringFormat format1 =
           new StringFormat (StringFormatFlags.NoClip);
         format1.Alignment = StringAlignment.Center;
        //vẽ chuỗi caption của button
         g.DrawString(this.Text, this.Font, Brushes.White,
                    this.Width / 2, this.Height / 3, format1);
      } //hết hàm OnPaint
    } //hết class HeartButton
```

## Xây dựng ứng dụng dùng Owner-drawn control

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. UseHeartButton), click button OK để tạo Project theo các thông số đã khai báo.
3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, việc thiết kế form là quá trình lặp 4 thao tác tạo mới/xóa/hiệu chỉnh thuộc tính/tạo hàm xử lý sự kiện cho từng đối tượng cần dùng trong form.
4. Nếu cửa sổ ToolBox chưa hiển thị, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Dời chuột vào trong cửa sổ Toolbox, ấn phải chuột để hiển thị menu lệnh, chọn option "Choose Items". Khi cửa sổ "Choose Toolbox Items" hiển thị, click chuột vào

button Browse để hiển thị cửa sổ duyệt tìm file, hãy duyệt tìm đến thư mục chứa file MyUserControls.dll vừa xây dựng được trong các slide trước, chọn file dll rồi click button OK để "add" các usercontrol trong thư viện này vào cửa sổ Toolbox của Project ứng dụng.

5. Duyệt tìm phần tử HeartButton (trong nhóm General ở chuối cửa sổ), chọn nó, dời chuột về vị trí thích hợp trong form và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Bắt đầu thực hiện". Nếu cần, hãy thay đổi vị trí và kích thước của Button và Form.

6. Ấn kép chuột vào button vừa tạo để tạo hàm xử lý sự kiện Click của Button rồi viết code như sau :

```
//hàm xử lý Click chuột trên button
private void btnStart_Click (object sender, EventArgs e) {
   MessageBox.Show("Bạn vừa ấn chuột trên Button");
   //thử thay đổi nội dung Caption
   btnStart.Text = "Kết thúc";
}
```

7. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Hãy thử click chuột trên đối tượng HeartButton và đánh giá kết quả.

## 9.5 Kết chương

Chương này đã giới thiệu cách thức dùng tính thừa kế để tạo mới 3 loại đối tượng giao diện cá nhân hóa phổ biến là User Control, Inherited Control và Owner-drawn Control.

Chương này cũng đã giới thiệu cách thức viết chương trình sử dụng lại các đối tượng giao diện cá nhân hóa.

# Chương 10

# Truy xuất database trong chương trình VC#

## 10.0 Dẫn nhập

Chương này giới thiệu cách thức dùng các đối tượng trong thư viện ADO .Net để truy xuất database dễ dàng, tin cậy.

Chương này cũng giới thiệu cách thức dùng khả năng databinding của các đối tượng giao diện trong môi trường VS .Net để xây dựng chương trình truy xuất database được dễ dàng, nhanh chóng, tin cậy, và nhiều trường hợp không cần viết code cho chương trình.

## 10.1 Tổng quát về truy xuất database

Mục tiêu của chương trình là xử lý các dữ liệu của nó. Dữ liệu của chương trình có thể rất nhiều và đa dạng phong phú về tính chất. Trong chương 7, chúng ta đã giới thiệu cách lập trình để ghi/đọc dữ liệu cổ điển hay đối tượng ra/vào file.

Hầu hết các ứng dụng hiện nay (nhất là các ứng dụng nghiệp vụ) đều phải truy xuất dữ liệu rất lớn. Thí dụ chương trình quản lý công dân Việt Nam phải xử lý hàng trăm triệu hồ sơ chứa thông tin về các công dân.

Việc xử lý dữ liệu bao gồm nhiều tác vụ như tạo file mới với cấu trúc record cụ thể, thêm/bớt/hiệu chỉnh/duyệt các record, tìm kiếm các record thỏa mãn 1 tiêu chuẩn nào đó,... Để thực hiện các tác vụ trên (nhất là tìm kiếm record thỏa mãn 1 số tiêu chuẩn nào đó) hiệu quả, tin cậy, ta cần nhiều kiến thức khác nhau và phải tốn nhiều công sức.

Hiện nay các record dữ liệu có cùng cấu trúc (thí dụ như các record sinh viên) cần xử lý của chương trình thường được lưu giữ trong 1 bảng dữ liệu (table). Nhiều bảng dữ liệu có mối quan hệ lẫn nhau được chứa trong 1 database quan hệ. Có nhiều định dạng

database quan hệ khác nhau đang được dùng như FoxPro, Access, SQL Server, MySQL, Oracle...

Để giải phóng ứng dụng khỏi các chi tiết quản lý database, người ta đã xây dựng ứng dụng đặc biệt : DBMS (Database Management System).

Mỗi DBMS cung cấp ít nhất 1 Provider. Provider là module phần mềm cung cấp các hàm chức năng để chương trình ứng dụng gọi khi cần thiết hầu truy xuất dữ liệu trong database mà không cần biết chi tiết về cấu trúc vật lý của các record dữ liệu trong database.

Mỗi lần cần truy xuất dữ liệu trong database, ứng dụng sẽ nhờ DBMS thực hiện dùm thông qua việc dùng 1 trong các cấp dịch vụ sau đây (từ cao xuống thấp) :

- Các lệnh truy vấn của ngôn ngữ SQL
- Các đối tượng trong thư viện ADO .Net (ActiveX Data Objects)
- Các đối tượng trong thư viện ADO (ActiveX Data Objects)
- Các đối tượng trong thư viện DAO (Data Access Objects)
- Các hàm trong thư viện ODBC (Open Database Connectivity)

Ngôn ngữ truy vấn SQL là ngôn ngữ phi thủ tục, nó cung cấp 1 tập các lệnh SQL rất mạnh và dễ dàng dùng để xử lý database. Thí dụ để tìm tất cả sinh viên nam quê ở Bến tre đang theo học tại trường Bách Khoa Tp.HCM, ta chỉ cần dùng 1 lệnh SQL như sau :

```
Select * from Sinhvien where Phai = 1 and Quequan = 71
```

Thư viện ADO .Net cung cấp 1 số đối tượng để giúp người lập trình truy xuất database rất dễ dàng thông qua mô hình hướng đối tượng.

Ngôn ngữ VC# cho phép ta kết hợp 2 cấp truy xuất database dễ dàng, đơn giản nhất : dùng các đối tượng ADO .Net để thực hiện các lệnh truy vấn SQL.

## 10.2 Truy xuất database thông qua ADO .Net

Các đối tượng ADO .Net được tổ chức theo từng namespace, mỗi namespace chứa đối tượng dùng cùng Provider truy xuất database :

- System.Data.OleDb chứa các đối tượng ADO .Net để truy xuất database do bộ Microsoft Office quản lý như Visual FoxPro, Access, Excel,...
- System.Data.Sql và System.Data.SqlClient chứa các đối tượng ADO .Net để truy xuất database do serer "SQLServer" quản lý.
- System.Data.Odbc chứa các đối tượng ADO .Net để truy xuất database thông qua chuẩn giao tiếp ODBC. Hầu hết các hệ quản trị database (DBMS) hiện nay đều hỗ trợ chuẩn giao tiếp này.
- ...

Trong từng namespace, để lập trình truy xuất database, ta thường dùng các đối tượng ADO .Net chính yếu sau đây :

1. preConnection, trong đó pre là phần tiếp đầu ngữ miêu tả tên namespace như OleDb, Obdc, Sql, SqlClient,... Đối tượng này có chức năng quản lý cầu nối đến nguồn database mà chúng ta cần truy xuất.
2. preCommand có chức năng quản lý lệnh truy vấn SQL mà ta cần thực hiện.
3. preDataReader cho phép duyệt đọc/xử lý các record từ 1 bảng dữ liệu.

4. preDataAdapter quản lý 1 tập các lệnh truy vấn và 1 connection tới nguồn database để cho phép việc đọc/ghi dữ liệu.
5. ...

## 10.3 Thí dụ lập trình dùng ADO .Net

Thí dụ trong một tổ chức quản lý việc nhập/xuất/tồn các sản phẩm, ta dùng 1 database quản lý dữ liệu. Database chứa 3 bảng dữ liệu sau đây :

- SPNhap chứa số lượng các sản phẩm nhập, mỗi record có các field như MaSP, Soluong,...
- SPXuat chứa số lượng các sản phẩm xuất, mỗi record có các field như MaSP, Soluong,...
- SPTon chứa số lượng các sản phẩm tồn kho, mỗi record có các field như MaSP, Soluong,...

Thường thì người làm công tác nghiệp vụ sẽ thực hiện việc cập nhật bảng sản phẩm nhập, bảng sản phẩm xuất theo thời gian. Chúng ta hãy viết chương trình tạo bảng sản phẩm tồn theo nội dung hiện hành của 2 bảng sản phẩm nhập/xuất.

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Console Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. TaoSPTon), click button OK để tạo Project theo các thông số đã khai báo.
3. Ngay sau khi Project vừa được tạo ra, cửa sổ soạn code cho chương trình được hiển thị. Thêm lệnh using sau đây vào đầu file :

   using System.data.OleDb;

4. Viết code cho thân hàm Main như sau :

```
static void Main(string[] args) {
    //định nghĩa các biến đối tượng cần dùng
    String ConnectionString;
    OleDbConnection cn;
    OleDbCommand cmd = new OleDbCommand();
    //xây dựng chuỗi đặc tả database cần truy xuất
    ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=d:\\MyDatabase.mdb;";
    //tạo đối tượng Connection đến database & mở Connection
    cn = new OleDbConnection(ConnectionString);
    cn.Open();
    //cấu hình cho đối tượng Command
    cmd.Connection = cn;
    //thực hiện lệnh SQL xóa bảng SPTon nếu đã có
    cmd.CommandText = "Drop Table SPTon";
    try { cmd.ExecuteNonQuery(); }
    catch { }
    //thực hiện lệnh SQL tạo bảng sản phẩm tồn
    cmd.CommandText = "Create Table SPTon(MaSP Text, Soluong int)";
    cmd.ExecuteNonQuery();
    //thực hiện lệnh SQL tạo các sản phẩm tồn có MaSP tồn tại trong bảng
SPNhap
    cmd.CommandText = "Insert into SPTon select SPNhap.MaSP,
IIf(IsNull(SPNhap.Soluong), 0, SPNhap.Soluong)-
IIf(IsNull(SPXuat.Soluong), 0, SPXuat.Soluong) as Soluong from
SPXuat right join SPNhap on SPXuat.MaSP = SPNhap.MaSP";
    cmd.ExecuteNonQuery();
    //thực hiện lệnh SQL tạo bảng sản phẩm Tam
    cmd.CommandText = "Create Table Tam(MaSP Text, Soluong int)";
    cmd.ExecuteNonQuery();
    //thực hiện lệnh SQL tạo các sản phẩm tồn có MaSP tồn tại trong bảng
SPXuat
```

```
    cmd.CommandText = "Insert into Tam select SPXuat.MaSP,
IIf(IsNull(SPNhap.Soluong), 0, SPNhap.Soluong) -
IIf(IsNull(SPXuat.Soluong), 0, SPXuat.Soluong) as Soluong from
SPNhap right join SPXuat on SPXuat.MaSP = SPNhap.MaSP";
    cmd.ExecuteNonQuery();
    //Trộn 2 bảng kết quả lại thành bảng SPTon
    cmd.CommandText = "Insert into SPTon select MaSP, Soluong
from Tam where Soluong < 0";
    cmd.ExecuteNonQuery();
    //Xóa bảng Tam
    cmd.CommandText = "Drop Table Tam";
    cmd.ExecuteNonQuery();
}
```

5. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng.
6. Sau khi ứng dụng chạy xong, chạy ứng dụng Access, mở file database, kiểm tra nội dung bảng SPTon để đánh giá kết quả có đúng yêu cầu không.

## 10.4 Databinding (Kết nối động đến dữ liệu)

Khi viết chương trình truy xuất database có giao diện đồ họa trực quan, chúng ta thường phải thực hiện các chức năng chính :

- thiết kế các đối tượng giao diện để giúp người dùng tương tác với database.
- viết code thiết lập các thông tin về database, về các dữ liệu cần truy xuất.
- mỗi lần record trên database bị thay đổi nội dung (do bị xử lý bên trong, do bị thay đổi vị trí truy xuất), ta phải viết code đọc thông tin của record hiện hành và hiển thị lên các đối tượng giao diện tương ứng.
- mỗi lần người dùng cập nhật nội dung trong các đối tượng giao diện, ta phải viết code đọc thông tin của các đối tượng

giao diện và ghi lên record tương ứng trên database để đảm bảo tính nhất quán dữ liệu.

Trong 4 công việc cần thực hiện trong slide trước thì công việc 3 và 4 khá nặng nề. VC# cung cấp khả năng Databinding để giúp người lập trình không cần thực hiện 2 công việc nặng nề này.

Thật vậy, hầu hết các đối tượng giao diện có sẵn trong framework .Net như TextBox, ListBox, ComboBox, TreeView, DataGridView,... đều có thuộc tính DataBindings. Nó giúp ta chỉ cần thiết lập sự kết hợp giữa đối tượng giao diện với dữ liệu nào đó trong database để khi chương trình hoạt động, máy sẽ tự hiển thị thông tin từ database lên đối tượng giao diện và mỗi khi nội dung đối tượng giao diện bị thay đổi bởi người dùng, máy cũng sẽ tự động ghi lên database để đảm bảo tính nhất quán dữ liệu theo suốt thời gian làm việc.

Việc thiết lập sự kết hợp giữa đối tượng giao diện với dữ liệu trong database có thể thực hiện bằng thiết kế trực quan hay viết code tường minh.

Nếu ta thiết lập sự kết hợp giữa đối tượng giao diện với dữ liệu trong database bằng thiết kế trực quan thì sẽ dẫn đến kết quả hết sức hấp dẫn : xây dựng chương trình truy xuất database mà không cần viết code nào cả. Thí dụ trong mục 10.5 sẽ cho thấy kết luận này.

Tuy nhiên để chủ động hơn trong việc xác lập mối quan hệ ràng buộc giữa các nội dung hiển thị trong các đối tượng giao diện, ta thường sẽ viết đoạn code để thiết lập sự kết hợp giữa đối tượng giao diện với dữ liệu trong database cũng như các ràng buộc giữa chúng. Thí dụ trong mục 10.6 sẽ cho thấy kết luận này.

## 10.5 Thí dụ về databinding mà không viết code

Trong bộ Microsoft Office, Microsoft có cung cấp 1 database Access có tên là NorthWind.mdb, database này chứa thông tin

quản lý của 1 siêu thị điện tử ảo, gồm nhiều bảng dữ liệu, trong đó có bảng Customers miêu tả các khách hàng của siêu thị.

Chúng ta hãy thử viết 1 chương trình đơn giản có chức năng hiển thị thông tin về các khách hàng trong bảng Customers cho người dùng xem.

Sau đây là qui trình điển hình để xây dựng chương trình theo yêu cầu trên mà không cần viết code cho chương trình.

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. DBAccess), click button OK để tạo Project theo các thông số đã khai báo.
3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, việc thiết kế form là quá trình lặp 4 thao tác tạo mới/xóa/hiệu chỉnh thuộc tính/tạo hàm xử lý sự kiện cho từng đối tượng cần dùng trong form.
4. Duyệt tìm phần tử DataGridView (trong nhóm Data), chọn nó, dời chuột vào trong Form và vẽ nó với kích thước mong muốn (chiếm hết form). Hiệu chỉnh thuộc tính (Name) = grdCustomers.
5. Ngay sau khi vẽ xong DataGridView, máy sẽ hiển thị cửa sổ "DataGridView Tasks". Nếu sơ xuất làm mất nó thì bạn hãy click chuột vào button nhỏ ở phía trên phải DataGridView để hiển thị lại. Click chuột vào icon chỉ xuống trong listbox "Choose data source" để hiển thị cửa sổ hỗ trợ. Click chuột vào mục "Add Project Data Source", để hiển thị cửa sổ "Choose a Data Source type". chọn icon Database rồi click button Next để hiển thị cửa sổ "Choose Your Database Connection".

6. Click button New Connection để hiển thị cửa sổ "Add Connection", xác định Provider truy xuất database, thí dụ như Provider "Microsoft Access Database File (OLE DB)" để truy xuất file Access, provider "Microsoft SQL Server" để truy xuất database do SQL Server quản lý...
7. Xác định database cần truy xuất trong "Database file name" rồi click button OK để quay về cửa sổ trước. Click buttonNext để hiển thị cửa sổ "Choose Your Database Object".
8. Mở rộng mục Tables để hiển thị đầy đủ các tên bảng dữ liệu có trong database, duyệt tìm và đánh dấu chọn vào bảng Customers rồi click button Finish để hoàn tất việc khai báo trực quan.
9. Bây giờ chương trình đã được viết xong. hãy chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Cửa sổ ứng dụng sẽ hiển thị đối tượng DataGridView, đối tượng này hiển thị đầy đủ danh sách thông tin các khách hàng trong bảng Customers, người dùng có thể "scroll" lên/xuống hay trái/phải để xem thông tin thích hợp.

## 10.6 Thí dụ về databinding có viết code khởi tạo

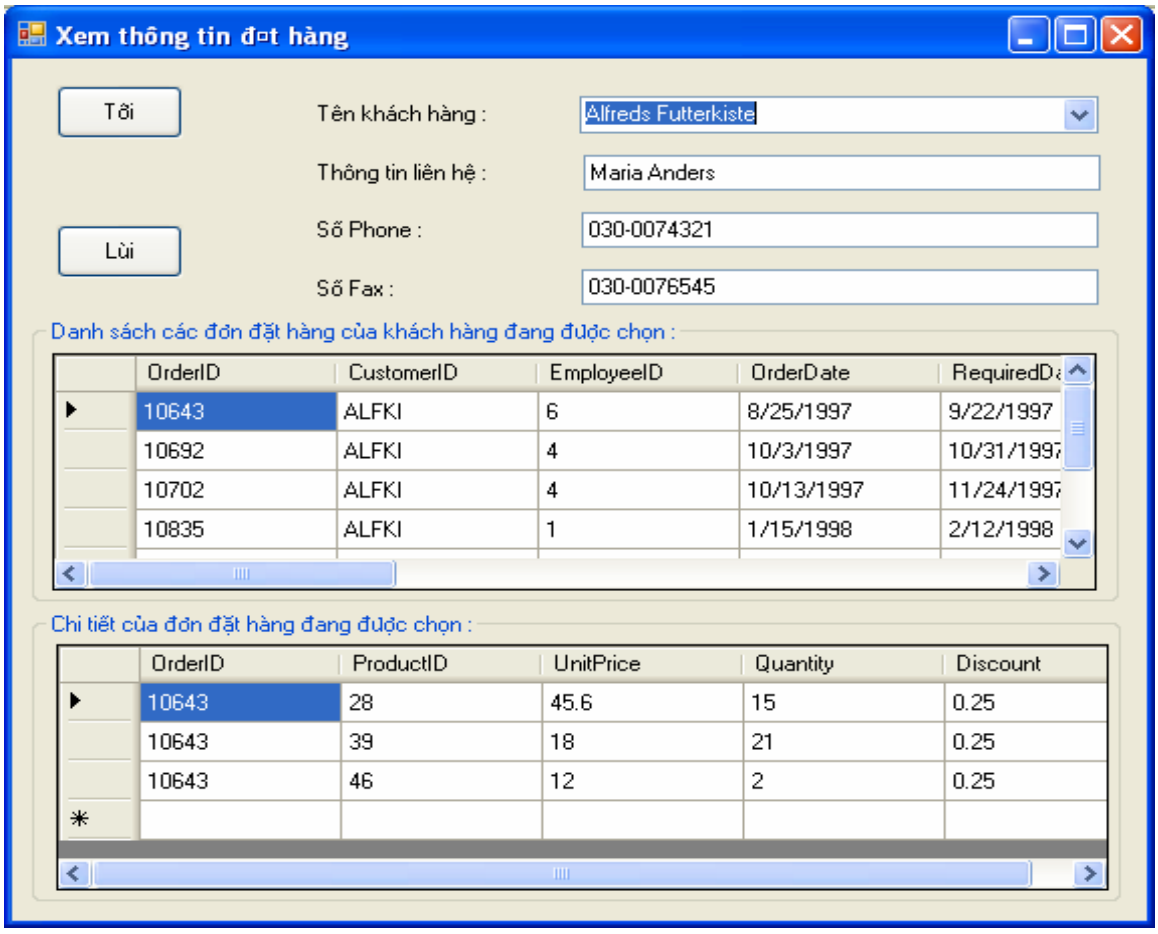
Trong bộ Microsoft Office, Microsoft có cung cấp 1 database Access có tên là NorthWind.mdb, database này chứa thông tin quản lý của 1 siêu thị điện tử ảo, gồm nhiều bảng dữ liệu và mối quan hệ giữa chúng :

- Customers : là bảng dữ liệu miêu tả các khách hàng của siêu thị, mỗi khác hàng có các field thông tin như CustomerID, CustomerName,...
- Orders : là bảng dữ liệu miêu tả các đơn đặt hàng của các khách hàng, mỗi đơn đặt hàng có các field thông tin như CustomerID, OrderID,...

- OrderDetails : là bảng dữ liệu miêu tả nội dung chi tiết của từng đơn đặt hàng, có các field thông tin như OrderID, ProductID,...

Chúng ta hãy viết chương trình cho phép người dùng xem thông tin mua hàng của các khách hàng, mỗi thời điểm xem chi tiết 1 khách hàng cần khảo sát, người dùng có thể dời tới/dời lui để xem thông tin khách hàng kế cận, người dùng có thể chọn ngẫu nhiên 1 khách hàng để xem thông tin. Thông tin chi tiết về khách hàng gồm tên, địa chỉ liên hệ, số phone, số fax, danh sách các đơn đặt hàng đã đặt, nội dung chi tiết của từng đơn đặt hàng.

Sau khi phân tích kỹ yêu cầu của chương trình trên, ta thiết kế được nội dung chi tiết của form giao diện của chương trình như sau :




Vai trò cụ thể của các đối tượng giao diện như sau :

- 2 Button : cho phép người dùng tiến tới/lùi 1 khách hàng.

- ComboBox : hiển thị tên khách hàng đang chọn khảo sát, nó còn cho phép người dùng chọn ngẫu nhiên 1 khác hàng khác nếu muốn.
- 3 textbox : hiển thị các thông tin về khác hàng như địa chỉ liên hệ, số phone, số fax.
- 1 DataGridView : hiển thị danh sách chi tiết về các đơn hàng của khách hàng đang quan tâm.
- 1 DataGridView : hiển thị danh sách chi tiết về các mặt hàng của đơn hàng đang quan tâm.

1. Chạy VS .Net, chọn menu File.New.Project để hiển thị cửa sổ New Project.
2. Mở rộng mục Visual C# trong TreeView "Project Types", chọn mục Windows, chọn icon "Windows Application" trong listbox "Templates" bên phải, thiết lập thư mục chứa Project trong listbox "Location", nhập tên Project vào textbox "Name:" (td. DBAccess), click button OK để tạo Project theo các thông số đã khai báo.
3. Form đầu tiên của ứng dụng đã hiển thị trong cửa sổ thiết kế, việc thiết kế form là quá trình lặp 4 thao tác tạo mới/xóa/hiệu chỉnh thuộc tính/tạo hàm xử lý sự kiện cho từng đối tượng cần dùng trong form.
4. Nếu cửa sổ ToolBox chưa hiển thị, chọn menu View.Toolbox để hiển thị nó (thường nằm ở bên trái màn hình). Thay đổi kích thước của form lớn ra theo yêu cầu.
5. Duyệt tìm phần tử Button (trong nhóm Common Controls), chọn nó, dời chuột về vị trí thích hợp trong form và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Tới" và thuộc tính (Name) = btnToi.
6. Lặp lại bước 5 để vẽ Button thứ 2 với thuộc tính Text = "Lùi" và thuộc tính (Name) = btnLui.

7. Duyệt tìm phần tử Label (trong nhóm Common Controls), chọn nó, dời chuột về vị trí thích hợp trong form và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Tên khách hàng :".
8. Lặp lại bước 7 để vẽ 3 Label còn lại với thuộc tính Text tuần tự là "Địa chỉ liên hệ :", "Số Phone : ", "Số Fax :"
9. Duyệt tìm phần tử ComboBox (trong nhóm Common Controls), chọn nó, dời chuột về bên phải Label "Tên khách hàng :" và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính (Name) = cbCust.
10. Duyệt tìm phần tử TextBox (trong nhóm Common Controls), chọn nó, dời chuột về vị trí bên phải Label "Địa chỉ liên hệ :" và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính (Name)= txtContact.
11. Lặp lại bước 10 để vẽ 2 TextBox còn lại với thuộc tính (Name) tuần tự là txtPhoneNo, txtFaxNo.
12. Duyệt tìm phần tử GroupBox (trong nhóm Common Controls), chọn nó, dời chuột về vị trí ngay dưới Label "Số Fax :" và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính Text = "Danh sách các đơn đặt hàng của khách hàng đang được chọn :".
13. Duyệt tìm phần tử DataGridView (trong nhóm Data), chọn nó, dời chuột vào trong GroupBox vừa vẽ và vẽ nó với kích thước mong muốn. Hiệu chỉnh thuộc tính (Name)= grdOrders.
14. Lặp lại bước 12 và 13 để vẽ GroupBox có thuộc tính Text = "Chi tiết của đơn đặt hàng đang được chọn :" và DataGridView bên trong có thuộc tính (Name) = grdOrderDetails.
15. Tạo hàm xử lý sự kiện cho 2 button btnToi và btnLui rồi viết code cho chúng như sau :

```
//hàm xử lý Click chuột trên button "Tới"
private void btnToi_Click(object sender, EventArgs e) {
   CurrencyManager cm = (CurrencyManager)this.BindingContext[
dsView, "Customers"];
   //nếu không phải khách hàng cuối thì tiến tới 1 khách hàng
   if (cm.Position < cm.Count - 1) cm.Position++;
}
//hàm xử lý Click chuột trên button"Lùi"
private void btnLui_Click(object sender, EventArgs e) {
   //nếu không phải khách hàng đầu tiên thì lùi 1 khách hàng
   if (this.BindingContext[dsView, "Customers"].Position > 0)
      this.BindingContext[dsView, "Customers"].Position--;
}
```

16. Thêm lệnh using sau vào đầu file đặc tả class Form :

```
using System.Data.OleDb;
```

17. Thêm các lệnh định nghĩa các thuộc tính dữ liệu cần dùng sau đây vào ở vị trí đầu class đặc tả Form :

```
   //định nghĩa các thuộc tính dữ liệu cần dùng
   private String ConnectionString;
   private DataViewManager dsView;
   private DataSet ds;
   private OleDbConnection cn;
```

18. Hiệu chỉnh lại hàm constructor của Form để có nội dung như sau:

```
 public Form1() {
   InitializeComponent();
   //xây dựng chuỗi đặc tả database cần truy xuất
   ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=c:\\NorthWind.mdb;";
   //tạo đối tượng Connection đến database
   cn = new OleDbConnection(ConnectionString);
   //tạo đối tượng DataSet
   ds = new DataSet("CustOrders");
```

```
    //tạo đối tượng DataApdater quản lý danh sách các khách hàng
    OleDbDataAdapter da1 = new OleDbDataAdapter
        ("SELECT * FROM Customers", cn);
    //ánh xạ Tablename "Table" tới bảng dữ liệu "Customers"
    da1.TableMappings.Add("Table","Customers");
    //chứa bảng Customers vào Dataset
    da1.Fill(ds);
    //tạo đối tượng DataApdater quản lý danh sách các đơn đặt
hàng
    OleDbDataAdapter da2 = new OleDbDataAdapter
        ("SELECT * FROM Orders", cn);
    //ánh xạ Tablename "Table" tới bảng dữ liệu "Orders"
    da2.TableMappings.Add("Table","Orders");
    // chứa bảng Orders vào Dataset
    da2.Fill(ds);
    //tạo đối tượng DataApdater quản lý danh sách các mặt hàng
    OleDbDataAdapter da3 = new OleDbDataAdapter
        ("SELECT * FROM [Order Details]", cn);
    //ánh xạ Tablename "Table" tới bảng dữ liệu "Orders"
    da3.TableMappings.Add("Table","OrderDetails");
    // chứa bảng [Orders Details] vào Dataset
    da3.Fill(ds);
    //thiết lập quan hệ "RelCustOrd" giữa bảng Customers và
Orders
    System.Data.DataRelation relCustOrd;
    System.Data.DataColumn  colMaster1;
    System.Data.DataColumn  colDetail1;
    colMaster1 = ds.Tables["Customers"].Columns["CustomerID"];
    colDetail1 = ds.Tables["Orders"].Columns["CustomerID"];
    relCustOrd = new System.Data.DataRelation
    ("RelCustOrd",colMaster1,colDetail1);
    //"add" quan hệ vừa tạo vào dataSet
    ds.Relations.Add(relCustOrd);
```

```
    //thiết lập quan hệ "relOrdDet" giữa bảng Orders & [Order Details]
    System.Data.DataRelation relOrdDet;
    System.Data.DataColumn  colMaster2;
    System.Data.DataColumn  colDetail2;
    colMaster2 = ds.Tables["Orders"].Columns["OrderID"];
    colDetail2 = ds.Tables["OrderDetails"].Columns["OrderID"];
    relOrdDet = new DataRelation("RelOrdDet",colMaster2,colDetail2);
    //"add" quan hệ vừa tạo vào dataSet
    ds.Relations.Add(relOrdDet);
    //Xác định DataViewManager của DataSet.
    dsView = ds.DefaultViewManager;
    //thiết lập Databinding giữa database với 2 DataGridView
    grdOrders.DataSource = dsView;
    grdOrders.DataMember = "Customers.RelCustOrd";
    grdOrderDetails.DataSource = dsView;
    grdOrderDetails.DataMember = "Customers.RelCustOrd.RelOrdDet";
    //thiết lập Databinding giữa database với ComboBox
    cbCust.DataSource = dsView;
    cbCust.DisplayMember = "Customers.CompanyName";
    cbCust.ValueMember = "Customers.CustomerID";
    //thiết lập Databinding giữa database với 3 Textbox
    txtContact.DataBindings.Add("Text",dsView,"Customers.ContactName");
    txtPhoneNo.DataBindings.Add("Text",dsView,"Customers.Phone");
    txtFaxNo.DataBindings.Add("Text",dsView,"Customers.Fax");
}
```

19. Chọn menu Debug.Start Debugging để dịch và chạy ứng dụng. Lúc đầu, form sẽ hiển thị thông tin về khách hàng đầu tiên trong bảng, khi bạn click vào button "Tới" hay "Lùi", thông tin khách hàng tương ứng sẽ tự được hiển thị. Bạn cũng có thể chọn 1 khách hàng tùy ý trong ComboBox "Tên khách hàng :" để chương trình tự hiển thị thông tin chi tiết về khách hàng đó.

20. Tóm lại Databinding trong VC# giúp ta giảm nhẹ rất nhiều công sức viết chương trình truy xuất database : chúng ta chỉ viết đoạn code thiết lập databinding giữa các đối tượng giao diện với dữ liệu tương ứng trong database chứ chúng ta không cần viết đoạn code cập nhật nội dung của các phần tử giao diện theo sự biến động của database, chúng ta cũng không cần viết code cập nhật database theo nội dung mà người dùng thay đổi trên các đối tượng giao diện.

## 10.7 Kết chương

Chương này đã giới thiệu cách thức dùng các đối tượng trong thư viện ADO .Net để truy xuất database dễ dàng, tin cậy.

Chương này cũng đã giới thiệu cách thức dùng khả năng databinding của các đối tượng giao diện trong môi trường VS .Net để xây dựng chương trình truy xuất database được dễ dàng, nhanh chóng, tin cậy, và nhiều trường hợp không cần viết code cho chương trình.

# Chương 11

# Lập trình song song bằng VC#

## 11.0 Dẫn nhập

Chương này giới thiệu kiến thức tổng quát về lập trình song song, class Process của môi trường .Net phục vụ lập trình multi-process, class Thread phục vụ lập trình multi-thread.

Chương này cũng giới thiệu các chương trình demo cho tính hiệu quả của lập trình multi-thread so với lập trình tuần tự, vấn đề tranh chấp giữa các thread về việc truy xuất tài nguyên dùng chung đồng thời, cách thức giải quyết tranh chấp và hệ lụy của việc giải quyết tranh chấp.

## 11.1 Tổng quát về lập trình song song

Thường để giải quyết bài toán nào đó, ta thường dùng giải thuật tuần tự nhờ tính dễ hiểu, dễ kiểm soát của nó. Chương trình dùng thuật giải tuần tự khi chạy trở thành process mono-thread hay process tuần tự.

Process tuần tự hoạt động không hiệu quả vì không lợi dụng triệt để được các CPU xử lý trên máy tính vật lý. Lưu ý rằng hiện nay các máy PC, smartphone hay tablet đều dùng CPU đa nhân. Thí dụ galaxy S4 ở thị trường Việt Nam có 8 nhân.

Để máy giải quyết bài toán hiệu quả hơn, ta nên dùng thuật toán song song bằng cách nhận dạng các hoạt động có thể thực hiện đồng thời rồi nhờ nhiều CPU thực hiện chúng đồng thời.

Một trong các phương pháp hiện thực thuật toán song song là lập trình multi-process và multi-thread.

## 11.2 Lập trình multi-process bằng class Process

Môi trường .Net cung cấp class tên là Process để giúp ta lập trình multi-process dễ dàng.

Class Process thuộc namespace System.Diagnostics, nó chứa các thuộc tính và tác vụ giúp ta quản lý process dễ dàng, thuận lợi.

Thí dụ thuộc tính StartInfo là 1 đối tượng gồm nhiều thuộc tính xác định thông tin để kích hoạt ứng dụng xác định :

```
//tạo đối tượng quản lý process mới
Process myProcess = new Process();
//thiết lập file khả thi cần chạy
myProcess.StartInfo.FileName = txtPath.Text;
//không dùng Shell để chạy process mới
myProcess.StartInfo.UseShellExecute = false;
//tạo cửa sổ giao diện riêng cho process mới
myProcess.StartInfo.CreateNoWindow = true;
....
```

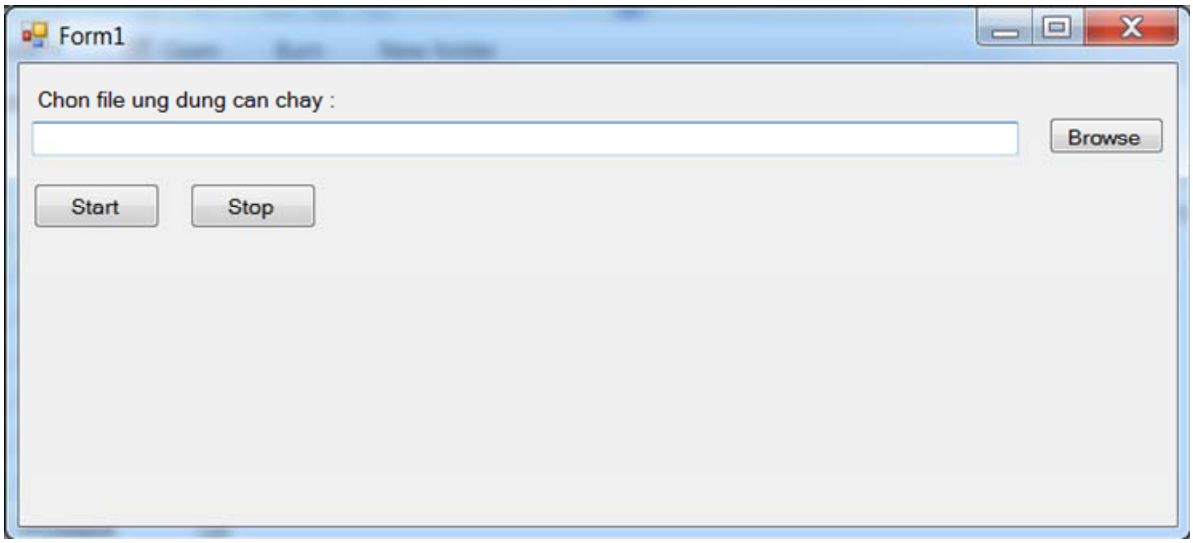
Sau khi thiết lập đầy đủ các thông tin để khởi tạo process, ta có thể gọi tác vụ Start để kích hoạt nó chạy :

```
myProcess.Start();
```

Sau khi được kích hoạt, process sẽ chạy song hành và độc lập với process kích hoạt nó cho đến khi kết thúc theo thuật giải của nó. Tuy nhiên, từ bên ngoài ta có thể giết process nhờ tác vụ Kill :

```
myProcess.Kill();
```

Ta hãy thử viết 1 ứng dụng quản lý process đơn giản có form giao diện như sau :




## 11.3 Lập trình multi-threads bằng class Thread

Môi trường .Net cung cấp class tên là Thread để giúp ta lập trình multi-thread dễ dàng.

Class Thread thuộc namespace System.Threading, nó chứa các thuộc tính và tác vụ giúp ta quản lý thread dễ dàng, thuận lợi.

Thường mỗi thread sẽ chạy đoạn code được miêu tả trong 1 hàm chức năng xác định. Thí dụ khi process được kích hoạt, HĐH sẽ tạo tường minh thread ban đấu cho process đó, thread chính này sẽ chạy đoạn code của hàm Main của class ứng dụng.

Để tạo thread mới, ta có thể dùng lệnh :

```
Thread t = new Thread
        (new ParameterizedThreadStart(tenhamcanchay));
```

Để kích hoạt chạy thread, ta có thể gọi tác vụ Start :

```
t.Start (new Params(danhsachthamso));
```

với Params là class đối tượng chứa các thông số mà ta muốn truyền/nhận cho thread mới.

Lưu ý tác vụ mà thread sẽ chạy phải được đặc tả với tham số hình thức là kiểu object :

```
void TinhTich (object obj) {    //tác vụ mà thread sẽ chạy
    Params p = (Params)obj; //ép kiểu tham số về kiểu mong muốn
```

```
    ...
}
```

Để tạm dừng thread, ta có thể gọi tác vụ Suspend :

```
t.Suspend();
```

Để chạy tiếp thread, ta có thể gọi tác vụ Resume :

```
t.Resume();
```

Để dừng và xóa thread, ta có thể gọi tác vụ Abort :

```
t.Abort();
```

Để thay đổi quyền ưu tiên thread, ta thực hiện lệnh gán :

```
t.Priority = ThreadPriority.Normal;
```

Trên Windows, mỗi process có thể ở 1 trong 6 cấp quyền ưu tiên sau đây :

IDLE_PRIORITY_CLASS

BELOW_NORMAL_PRIORITY_CLASS

NORMAL_PRIORITY_CLASS

ABOVE_NORMAL_PRIORITY_CLASS

HIGH_PRIORITY_CLASS

REALTIME_PRIORITY_CLASS

Cấp quyền ưu tiên của process sẽ quyết định các thread trong process đó chạy theo quyền ưu tiên như thế nào.

Trên Windows, mỗi thread trong process có thể ở 1 trong 7 cấp quyền ưu tiên sau đây :

THREAD_PRIORITY_IDLE

THREAD_PRIORITY_LOWEST

THREAD_PRIORITY_BELOW_NORMAL

THREAD_PRIORITY_NORMAL

THREAD_PRIORITY_ABOVE_NORMAL

THREAD_PRIORITY_HIGHEST

THREAD_PRIORITY_TIME_CRITICAL

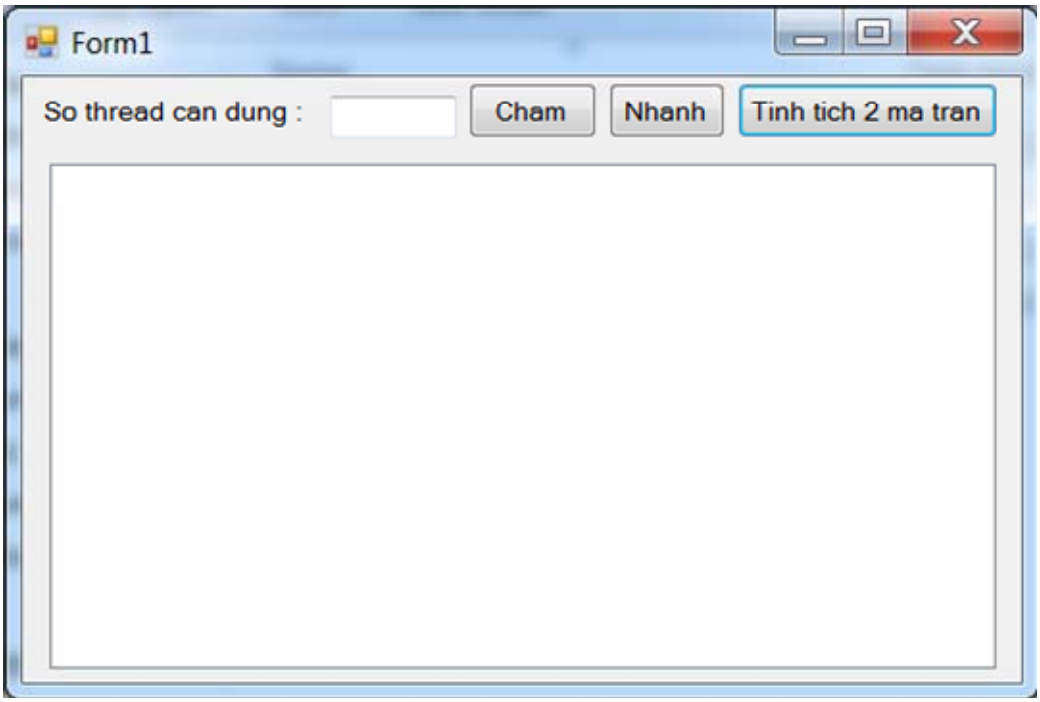
Cấp quyền ưu tiên của process sẽ quyết định các thread với từng cấp quyền ưu tiên trên đây sẽ được xử lý như thế nào.

| Process priority class | Thread priority level | Base priority |
|---|---|---|
| IDLE_PRIORITY_CLASS | THREAD_PRIORITY_IDLE | 1 |
| | THREAD_PRIORITY_LOWEST | 2 |
| | THREAD_PRIORITY_BELOW_NORMAL | 3 |
| | THREAD_PRIORITY_NORMAL | 4 |
| | THREAD_PRIORITY_ABOVE_NORMAL | 5 |
| | THREAD_PRIORITY_HIGHEST | 6 |
| | THREAD_PRIORITY_TIME_CRITICAL | 15 |
| BELOW_NORMAL_PRIORITY_CLASS | THREAD_PRIORITY_IDLE | 1 |
| | THREAD_PRIORITY_LOWEST | 4 |
| | THREAD_PRIORITY_BELOW_NORMAL | 5 |
| | THREAD_PRIORITY_NORMAL | 6 |
| | THREAD_PRIORITY_ABOVE_NORMAL | 7 |
| | THREAD_PRIORITY_HIGHEST | 8 |
| | THREAD_PRIORITY_TIME_CRITICAL | 15 |
| NORMAL_PRIORITY_CLASS | THREAD_PRIORITY_IDLE | 1 |
| | THREAD_PRIORITY_LOWEST | 6 |
| | THREAD_PRIORITY_BELOW_NORMAL | 7 |
| | THREAD_PRIORITY_NORMAL | 8 |
| | THREAD_PRIORITY_ABOVE_NORMAL | 9 |
| | THREAD_PRIORITY_HIGHEST | 10 |
| | THREAD_PRIORITY_TIME_CRITICAL | 15 |
| ABOVE_NORMAL_PRIORITY_CLASS | THREAD_PRIORITY_IDLE | 1 |
| | THREAD_PRIORITY_LOWEST | 8 |
| | THREAD_PRIORITY_BELOW_NORMAL | 9 |

| | | |
|---|---|---|
| | THREAD_PRIORITY_NORMAL | 10 |
| | THREAD_PRIORITY_ABOVE_NORMAL | 11 |
| | THREAD_PRIORITY_HIGHEST | 12 |
| | THREAD_PRIORITY_TIME_CRITICAL | 15 |
| HIGH_PRIORITY_CLASS | THREAD_PRIORITY_IDLE | 1 |
| | THREAD_PRIORITY_LOWEST | 11 |
| | THREAD_PRIORITY_BELOW_NORMAL | 12 |
| | THREAD_PRIORITY_NORMAL | 13 |
| | THREAD_PRIORITY_ABOVE_NORMAL | 14 |
| | THREAD_PRIORITY_HIGHEST | 15 |
| | THREAD_PRIORITY_TIME_CRITICAL | 15 |
| REALTIME_PRIORITY_CLASS | THREAD_PRIORITY_IDLE | 16 |
| | THREAD_PRIORITY_LOWEST | 22 |
| | THREAD_PRIORITY_BELOW_NORMAL | 23 |
| | THREAD_PRIORITY_NORMAL | 24 |
| | THREAD_PRIORITY_ABOVE_NORMAL | 25 |
| | THREAD_PRIORITY_HIGHEST | 26 |
| | THREAD_PRIORITY_TIME_CRITICAL | 31 |

## 11.4 Demo tính hiệu quả của multi-thread

Để demo tính hiệu quả của lập trình multi-thread trên các máy tính đa nhân, ta hãy thử viết ứng dụng tính tích của 2 ma trận có kích thước lớn (thí dụ 2000*2000). Ta thiết kế form ứng dụng như sau :

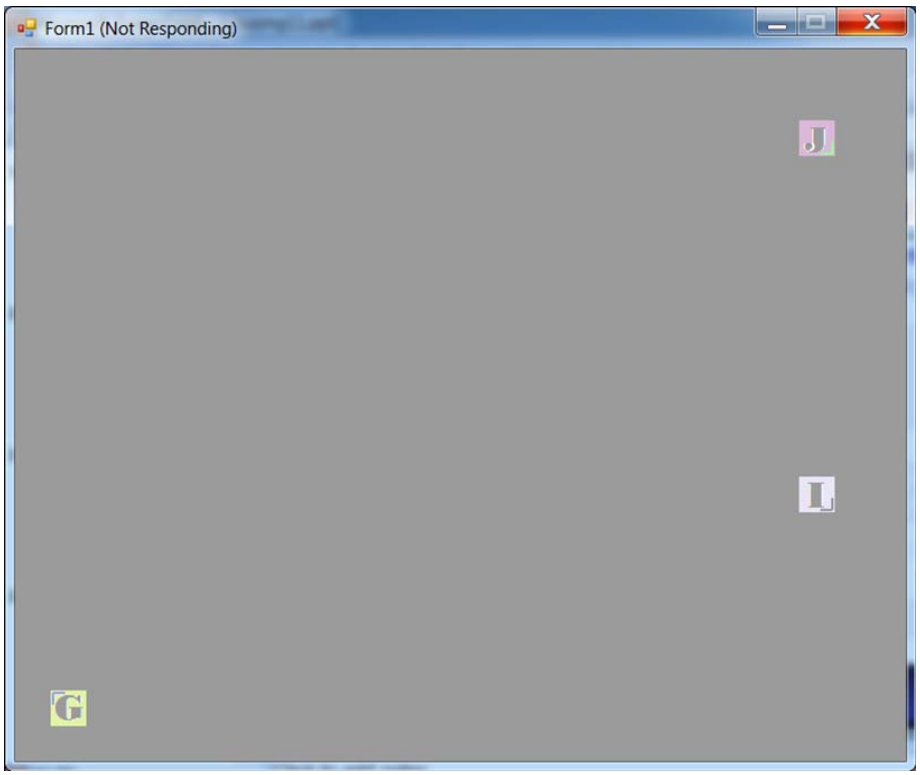



Ứng dụng cho phép người dùng nhập số thread cần dùng (n), rồi chia ma trận tích thành N/n nhóm hàng rồi tạo thread con để tính dùm từng nhóm hàng. Sau khi tính ma trận tích xong, ứng dụng sẽ hiển thị cho người dùng thấy thời gian tính toán để người dùng đánh giá độ hiệu quả.

```
void TinhTich (object obj) {
    DateTime t1 = DateTime.Now; //ghi nhận thời điểm bắt đầu chạy
    Params p = (Params)obj; //ép kiểu tham số về đối tượng cần dùng
    int h, c, k;
    for (h = p.sr; h < p.er; h++) //lặp theo hàng
        for (c = 0; c < N; c++) {  //lặp theo cột
            double s = 0;
            for (k = 0; k < N; k++)
                s = s + A[h, k] * B[k, c];
            C[h, c] = s;
        }
    stateLst[p.id] = 1;        //ghi nhận trạng thái hoàn thành
    dateLst[p.id] = DateTime.Now.Subtract(t1);//tính thời gian chạy
}
```

## 11.5 Demo vấn đề tương tranh giữa các thread

Để demo vấn đề tương tranh giữa các thread, ta hãy thử viết ứng dụng quản lý các thread với giao diện như sau :




Form là ma trận gồm nhiều cell, mỗi cell chứa được icon ảnh cho 1 thread đang chạy. Lúc đầu, chưa có thread nào chạy hết. Người dùng có thể ấn phím để quản lý các thread như sau :

- Ấn phím từ A-Z để kích hoạt chạy thread có tên tương ứng.
- Ấn phím Ctrl-Alt-X để tạm dừng chạy thread X.
- Ấn phím Alt-X để chạy tiếp thread X.
- Ấn phím Shift-X để tăng độ ưu tiên chạy cho thread X.
- Ấn phím Ctrl-X để giảm độ ưu tiên chạy cho thread X.
- Ấn phím Ctrl-Shift-X để dừng và thoát thread X.

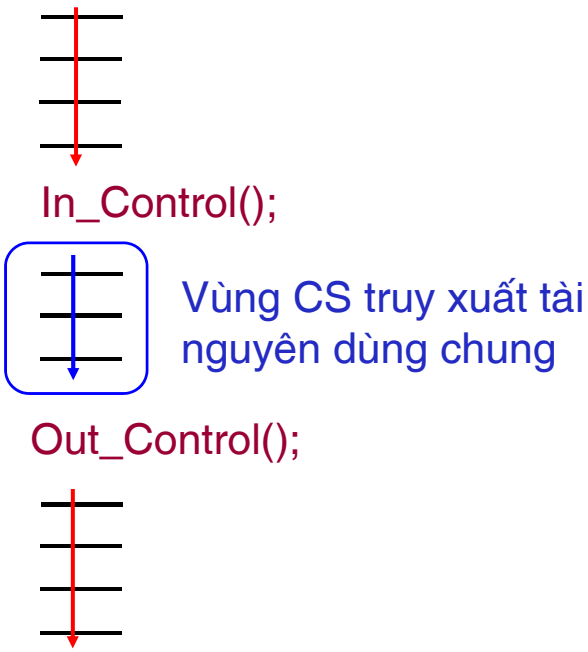
Để cho thấy hành vi hoạt động của thread, hoạt động của thread là hiển thị icon miêu tả mình lên form, icon này sẽ chạy theo 1 phương xác định, khi đụng thành form thì dội lại theo nguyên lý vật lý.

Quan sát quỹ đạo chạy icon miêu tả từng thread ta thấy thỉnh thoảng có hiện tượng icon thread này đè mất icon thread khác. Đây là hiện tượng lỗi không mong muốn do các thread được quyền tự do chiếm dụng từng cell hiển thị của form. Ta dùng thuật ngữ "tương tranh" giữa các thread trên các tài nguyên dùng chung (các cell của form).

Cần có biện pháp quản lý tương tranh sao cho các thread không được quyền truy xuất tài nguyên dùng chung đồng thời. Hiện nay ta dùng phương pháp loại trừ tương hỗ để giải quyết vấn đề này.

## 11.6 Demo việc giải quyết tương tranh giữa các thread

Mỗi lần muốn vào vùng CS, ta phải gọi hàm In_Control() để kiểm soát việc thi hành vùng CS, khi hoàn thành vùng CS, ta phải gọi hàm Out_Control() để thông báo cho các thread khác đang chờ để chúng kiểm tra lại việc đi vào.




Phương pháp loại trừ tương hỗ phổ dụng hiện nay là dùng semaphore nhị phân (Mutex). Semaphore là 1 đối tượng đơn giản chứa :

- 1 thuộc tính kiểu nguyên dương (s), ta còn gọi nó là biến semaphore.
- Tác vụ down(), có nhiệm vụ giảm s 1 đơn vị và luôn phải hoàn thành. Do đó trong trường hợp s = 0, tác vụ down sẽ phải ngủ chờ đến khi s <> 0 thì cố gắng thực hiện lại… ➔ Thời gian thi hành tác vụ down là không xác định.
- Tác vụ up(), có nhiệm vụ tăng s 1 đơn vị và luôn phải hoàn thành. Trong trường hợp s = 0, tác vụ up sẽ phải đánh thức các thread đang ngủ chờ down s dậy.

Môi trường .Net cung cấp class Mutex để quản lý semaphore nhị phân. Ta kết hợp mỗi tài nguyên dùng chung 1 mutex m với giá trị đầu = 1.

Hàm In_Control() sẽ là lệnh m.WaitOne(); Thread nào thực hiện lệnh này đầu tiên sẽ thành công ngay và sẽ chạy được đoạn lệnh CS truy xuất tài nguyên tương ứng. Các thread khác thực hiện lệnh trên để truy xuất tài nguyên dùng chung sẽ thất bại và bị ngủ trong khi thread đầu chưa hoàn thành truy xuất.

Khi hoàn thành việc truy xuất tài nguyên, thread gọi hàm Out_Control(). Trong trường hợp dùng Mutex, hàm Out_Control() sẽ là lệnh m.ReleaseMutex(); Nó sẽ đánh thức các thread đang ngủ chờ nếu có.

```
//xin khóa truy xuất cell bắt đầu (x1,y1)
mutList[y1, x1].WaitOne();
while (p.start) { //lặp trong khi chưa có yêu cầu kết thúc
    //hiển thị icon miêu tả mình lên cell
    //thực hiện công việc của thread
    //xác định vị trí mới của thread (x2,y2)
    //xin khóa truy xuất cell (x2,y2)
    while (true) {
        kq = mutList[y2, x2].WaitOne(new TimeSpan(0,0,2));
        if (kq==true || p.start==false) break;
    }
```

```
    // Xóa vị trí cũ
    //giải phóng cell (x1,y1) cho các thread khác truy xuất
    mutList[y1, x1].ReleaseMutex();
}
```

## 10.7 Kết chương

Chương này đã giới thiệu kiến thức tổng quát về lập trình song song, class Process của môi trường .Net phục vụ lập trình multi-process, class Thread phục vụ lập trình multi-thread.

Chương này cũng đã giới thiệu các chương trình demo cho tính hiệu quả của lập trình multi-thread so với lập trình tuần tự, vấn đề tranh chấp giữa các thread về việc truy xuất tài nguyên dùng chung đồng thời, cách thức giải quyết tranh chấp và hệ lụy của việc giải quyết tranh chấp.