**CHƯƠNG IV: DỮ LIỆU CÓ CẤU TRÚC**

**§1: MẢNG**

Khi cần lưu trữ một số lượng lớn các phần tử có cùng kiểu dữ liệu, chẳng hạn như cần lưu trữ 30 số nguyên được nhập từ bàn phím, ta có thể dùng 30 biến số nguyên riêng biệt để lưu trữ. Tuy nhiên, cách trên có một số nhược điểm như sau:

* Khi cần lưu trữ một số lượng lớn các phần tử có cùng kiểu dữ liệu, chẳng hạn như cần lưu trữ 30 số nguyên được nhập từ bàn phím, ta có thể dùng 30 biến số nguyên riêng biệt để lưu trữ. Tuy nhiên, cách trên có một số nhược điểm như sau:
* Phải thực hiện thủ công các thao tác với biến đó. Do các biến là độc lập nên ta không thể sử dụng các cấu trúc lặp để thao tác với nó.
* Chương trình sẽ dài và việc nâng cấp cũng trở nên khó khăn.

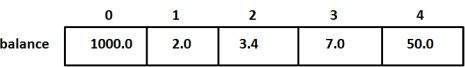
Do đó, việc nhóm các phần tử đó lại với một cái tên duy nhất sẽ khắc phục được những nhược điểm trên. Các phần tử vẫn hoạt động độc lập, nhưng dưới một tên duy nhất, cùng với chỉ số (index) của nó để phân biệt với các phần tử khác.

**1. Khai báo và sử dụng mảng**

Cách khai báo mảng như sau:

**<Kiểu dữ liệu> <Tên mảng>[số lượng phần tử];**

Kiểu dữ liệu của mảng sẽ quy định kiểu dữ liệu của các phần tử có trong mảng. Tên mảng được đặt tên theo quy tắc đặt tên biến trong C/C++. Khi đó, hệ thống sẽ tìm một vùng nhớ liên tiếp phù hợp để cấp phát cho biến mảng. Lưu ý, số lượng phần tử phải là một hằng số. Lưu ý, phần tử đầu tiên của mảng là 0, do đó nếu khai báo mảng n phần tử thì phần tử cuối cùng trong mảng là n – 1. Ví dụ, ta có mảng balance = {1000.0, 2.0, 3.4, 7.0, 50.0}.



Để truy cập tới phần tử của mảng ta sẽ dùng cú pháp:

**<tên mảng>[<index cần truy cập>]**

Ví dụ, balance[2] = 3.4.

Ngoài ra, ta còn có thể khai báo mảng kèm theo giá trị khởi tạo của mảng. Ví dụ:

|  |
| --- |
| int main()  {  int a[10]; //Explicity Declare  for(int i = 0; i < 10; i++)  {  scanf("%d", &a[i]);  }  for(int i = 0; i < 10; i++)  {  printf("%d ", a[i]);  }  printf("\n");    int b[10] = {7, 4, 1, 1}; //Declare and set value  for(int i = 0; i < 10; i++)  {  printf("%d ", \*(b + i)); //Addition with pointer  }  printf("\n");    char str [] = "Welcome to Stdio"; //Implicity Declare  printf("%d", strlen(str));    return 0;  } |

**2. Mảng nhiều chiều**

Khi mỗi phần tử của mảng có kiểu dữ liệu là một mảng khác, ta được mảng 2 chiều. Khi đó, ngoài index của các phần tử trong mảng, mỗi phần tử trong các phần tử đó cũng có index của nó. Do đó để truy xuất đến từng phần tử nhỏ, ta sử dụng cả 2 chỉ số index để chỉ ra vị trí của nó trong mảng 2 chiều.

Chúng ta xét ví dụ sau để nắm được cách khai báo và sử dụng mảng 2 chiều.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main()  {  int a[3][3];  for(int i = 0; i < 3; i++)  {  for(int j = 0; j < 3; j++)  {  scanf("%d", &a[i][j]);  }  }  for(int i = 0; i < 3; i++)  {  for(int j = 0; j < 3; j++)  {  printf("%d ", a[i][j]);  }  printf("\n");  }    int b[3][3] = {{1, 3}, {2, 4, 6}};  for(int i = 0; i < 3; i++)  {  for(int j = 0; j < 3; j++)  {  printf("%d ", b[i][j]);  }  printf("\n");  }    return 0;  } |

Tương tự như vậy ta có thể sử dụng mảng n chiều trong C++ với cùng cách thức làm việc như mảng 1 chiều và 2 chiều.

**3. Sử dụng mảng bằng cách cấp pháp động**

Khi lập trình, có những trường hợp chúng ta không xác định được số lượng ô nhớ cần thiết khi code, chỉ có thể xác định được khi chương trình đã được thực thi. Do đó sẽ là một sự lãng phí khi ta không dùng hết bộ nhớ được cấp phát, hoặc cấp phát không đủ bộ nhớ để sử dụng. Việc cấp phát động giúp chúng ta chủ động trong việc cấp phát bộ nhớ, tránh các trường hợp thừa/thiếu bộ nhớ.

Một ưu điểm khác của việc cấp phát động so với cấp phát tĩnh là ta có thể cấp phát vùng nhớ lớn, miễn sao chương trình tìm được một vùng nhớ liên tục trên RAM có kích thước phù hợp. Vùng nhớ Stack Segment được quản lý chặt chẽ bởi hệ thống, nhưng dung lượng khá nhỏ nên sẽ không đáp ứng hết được nhu cầu của lập trình viên.

Ngoài ra còn một số lợi ích khác của cấp phát động như sau:

* Có thể cấp phát vùng nhớ có kích thước bất kỳ bằng cách truyền tham số vào trong cặp dấu [ ] khi cấp phát vùng nhớ. Việc cấp phát tĩnh không cho phép truyền vào tham số để cấp phát vùng nhớ.
* Có thể tái sử dụng vùng nhớ sau khi được huỷ bằng toán tử delete.

Để cấp phát động bộ nhớ cho đối tượng, ta sử dụng toán tử new. Khi cần cấp phát nhiều ô nhớ cho biến, ta thay toán tử new bằng toán tử new[]. Cú pháp như sau:

**int \*a = new int;**

**int \*b = new int[10];**

Khi đó, chương trình sẽ cấp phát cho các biến một vùng nhớ liên tục có kích thước là chỉ số \* sizeof(kiểu dữ liệu). Các biến quản lý vùng nhớ được cấp phát là một biến con trỏ, lưu trữ địa chỉ ô nhớ đầu tiên của vùng nhớ đó. Các biến con trỏ này được lưu trên Stack Segment, trong khi vùng nhớ mà nó trỏ đến sẽ được lưu trên Data Segment (Vấn đề này chúng ta sẽ tìm hiểu kĩ hơn ở phần sau).

Sau khi thao tác xong, nếu không sử dụng vùng nhớ này nữa, ta sẽ huỷ nó bằng toán tử delete hoặc delete[] tương ứng. Nếu không huỷ vùng nhớ này trước khi chương trình kết thúc sẽ dẫn đến lỗi Memory Leak (rò rỉ vùng nhớ), vùng nhớ đó sẽ không thể được tái sử dụng cho đến khi khởi động lại máy tính (ngắt nguồn điện ở RAM). Do đó, chúng ta cần đặc biệt chú ý tới điều này, có new là phải có delete.

Ví dụ:

|  |
| --- |
| #include <stdio.h>    int main()  {  int \*a = new int[10];    for (int i = 0; i < 10; i++)  {  scanf("%d", &a[i]);  }  for (int i = 0; i < 10; i++)  {  printf("%-4d", \*(a + i));  }    delete[] a;  return 0;  } |

**§2: CHUỖI (XÂU KÍ TỰ)**

Khái niệm chuỗi ký tự do con người đặt ra để thuận tiện trong việc sử dụng. Có thể hiểu đơn giản, chuỗi là tập hợp các ký tự được lưu trữ liên tiếp trong vùng nhớ máy tính. Mỗi ký tự có kích thước 1 byte, do đó kích thước của chuỗi ký tự sẽ bằng tổng số các ký tự có mặt trong chuỗi. Bài viết này sẽ giúp các bạn có được những kiến thức cơ bản trong thao tác xử lý chuỗi trong ngôn ngữ C/C++.

**1. Chuỗi là gì ?**

Khái niệm chuỗi ký tự do con người đặt ra để thuận tiện trong việc sử dụng. Có thể hiểu đơn giản, chuỗi là tập hợp các ký tự được lưu trữ liên tiếp trong vùng nhớ máy tính. Mỗi ký tự có kích thước 1 byte, do đó kích thước của chuỗi ký tự sẽ bằng tổng số các ký tự có mặt trong chuỗi.

Thực chất, chuỗi ký tự là một mảng dữ liệu (array), mỗi phần tử trong mảng có kích thước 1 byte. Do đó, chuỗi ký tự có đầy đủ các tính chất của một mảng dữ liệu. Ngoài ra, nó còn có nhiều đặc tính và phương thức riêng để thao tác được dễ dàng hơn.

**2. Chuỗi trong C**

Dạng chuỗi này bắt nguồn từ ngôn ngữ C và tiếp tục được hỗ trợ trong C/C++. Chuỗi trong ngôn ngữ lập trình C thực chất là mảng một chiều của các ký tự mà kết thúc bởi một ký tự null '\0'.

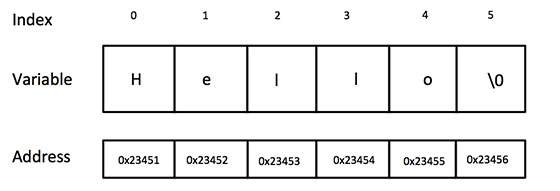
Phần khai báo và khởi tạo dưới đây tạo ra một chuỗi bao gồm một từ "Hello". Để giữ các giá trị null tại cuối của mảng, cỡ của mảng các ký tự bao gồm một chuỗi phải nhiều hơn số lượng các ký tự trong từ khóa "Hello".

**char loiChao[6] = {'H', 'e', 'l', 'l', 'o', '\0'};**

Ngoài ra ta còn cách khác để khởi tạo chuỗi:

**char loiChao[] = "Hello";**

Dưới đây là phần biểu diễn ô nhớ cho đoạn chuỗi trên trong ngôn ngữ C/C++:



Thực tế, ta không đặt ký tự null tại vị trí cuối cùng của biến hằng số. Bộ biên dịch C tự động thêm '\0' tại ví trí cuối cùng của chuỗi khi nó khởi tạo chuỗi. Dưới đây là bảng các hàm hay dùng với chuỗi theo phong cách C.

|  |  |
| --- | --- |
| **STT** | **Hàm & Mục đích** |
| 1 | **strcpy(s1, s2);**  Sao chép chuỗi s2 cho chuỗi s1. |
| 2 | **strcat(s1, s2);**  Nối chuỗi s2 vào cuối chuỗi s1. |
| 3 | **strlen(s1);**  Trả về độ dài của chuỗi s1. |
| 4 | **strcmp(s1, s2);**  Trả về 0 nếu s1 và s2 là như nhau; nhỏ hơn 0 nếu s1<s2; lớn hơn 0 nếu s1>s2. |
| 5 | **strchr(s1, ch);**  Trả về con trỏ tới vị trí đầu tiên của ch trong s1. |
| 6 | **strstr(s1, s2);**  Trả về con trỏ tới vị trí đầu tiên của chuỗi s2 trong chuỗi s1. |

Có nhiều hạn chế của việc dùng chuỗi theo phong cách C có thể kể đến như là:

* Người lập trình phải chủ động kiểm soát bộ nhớ cấp phát cho chuỗi ký tự. Nói chung là phải am hiểu và rất thông thạo về kỹ thuật dùng bộ nhớ và con trỏ thì chương trình mới tránh được các lỗi về kỹ thuật.
* Không thể gán giá trị hay sử dụng phép toán + (ghép chuỗi) và các phép toán so sánh như: > (lớn hơn), < (nhỏ hơn),... mà phải gọi các hàm thư viện trong <string.h>;
* Nếu dùng kỹ thuật cấp phát động thì phải quản lý việc cấp thêm bộ nhớ khi chuỗi dãn ra (chẳng hạn do ghép chuỗi) và phải hủy bộ nhớ (khi không dùng nữa) để tránh việc cạn kiệt bộ nhớ của máy tính trong trường hợp có nhiều chương trình hoạt động đồng thời.

**3. Chuỗi trong STL (Standard Template Library) C++**

**a. String trong C++**

Thư viện chuẩn STL (Standard Template Library) cung cấp kiểu string (xâu ký tự), giúp chúng ta tránh khỏi hoàn toàn các phiền phức nêu trên. Tuy vậy, nếu ta muốn sử dụng các hàm của C-string, ta cũng có thể chuyển đổi giữa hai loại chuỗi này:

* Chuyển từ string sang C-string: **string s; s.c\_str();**
* Chuyển từ C-string sang string:

**char\* s\_old = “ABC”; string s(s\_old);**

**b. Các phương thức phép toán với string (method & operation)**

Các toán tử +, += dùng đẻ ghép hai chuỗi và cũng để ghép một kí tự vào một chuỗi.

Các phép so sánh theo thứ tự từ điển: == (bằng nhau), != (khác nhau), > (lớn hơn), >= (lớn hơn hay bằng), < (nhỏ hơn), <= (nhỏ hơn hay bằng).

Hàm length() trả về kết quả là độ dài của chuỗi.

Do có tính chất mảng nên ta có thể xử lí chuỗi như một mảng các kí tự.

Các phương thức thông dụng:

* Phương thức substr(int pos, int nchar) trích ra chuỗi con của một chuỗi cho trước, ví dụ str.substr(2,4) trả về chuỗi con gồm 4 ký tự của chuỗi str kể từ ký tự ở vị trí thứ 2 (ký tự đầu tiên của chuỗi ở vị trí 0). Ví dụ:

|  |
| --- |
| //get substring  #include <iostream>  #include <string>  #include <conio.h>  using namespace std;  int main ()  {  string s="ConCho chay qua rao";  cout << s.substr(2,4) << endl;  // cout << new string(str.begin()+2, str.begin()+2+4);  getchar();  return 0;  } |

* Phương thức insert() chèn thêm ký tự hay chuỗi vào một vị trí nào đó của chuỗi str cho trước. Có nhiều cách dùng phương thức này:
  + str.insert(int pos, char\* s): chèn s (kiểu c-string) vào vị trí pos của str.
  + str.insert(int pos, string s): chèn s (kiểu string) vào vị trí pos của str.
  + str.insert(int pos, int n, int ch): chèn n lần kí tự ch vào vị trí pos của chuỗi str.
  + Ví dụ:

|  |
| --- |
| // inserting into a string  #include <iostream>  #include <string>  #include <conio.h>  using namespace std;  int main ()  {  string str="day la .. xau thu";  string istr = "them";  str.insert(8, istr);  cout << str << endl;  getchar();  return 0;  } |

* Phương thức str.erase(int pos, int n) xóa n kí tự của chuỗi str kể từ vị trí pos. Nếu không quy định giá trị n thì tất cả các giá trị của str từ pos trở đi sẽ bị xóa. Ví dụ:

|  |
| --- |
| // erase from a string  #include <iostream>  #include <string>  #include <conio.h>  using namespace std;  int main () {  string str="day cung la xau thu";  str.erase(0, 3); // " cung la xau thu"  cout << str << endl;  str.erase(6, 2);  cout << str << endl; // " cung xau thu"  getchar();  return 0;  } |

* So sánh:
  + Bạn có thể đơn giản là sử dụng những toán tử quan hệ (==, !=, <, <=, >=) được định nghĩa sẵn. Tuy nhiên, nếu muốn so sánh một phần của một chuỗi thì sẽ cần sử dụng phương thức compare():

**int compare ( const string& str ) const;**

**int compare ( const char\* s ) const;**

**int compare ( size\_t pos1, size\_t n1, const string& str ) const;**

**int compare ( size\_t pos1, size\_t n1, const char\* s) const;**

**int compare ( size\_t pos1, size\_t n1, const string& str, size\_t pos2, size\_t n2 ) const;**

**int compare ( size\_t pos1, size\_t n1, const char\* s, size\_t n2) const;**

* + Hàm trả về 0 khi hai chuỗi bằng nhau và lớn hơn hoặc nhỏ hơn 0 cho trường hợp khác Ví dụ:

|  |
| --- |
| // comparing apples with apples  #include <iostream>  #include <string>  using namespace std;  int main ()  {  string str1 ("green apple");  string str2 ("red apple");  if (str1.compare(str2) != 0)  cout << str1 << " is not " << str2 << "\n";  if (str1.compare(6,5,"apple") == 0)  cout << "still, " << str1 << " is an apple\n";  if (str2.compare(str2.size()-5,5,"apple") == 0)  cout << "and " << str2 << " is also an apple\n";  if (str1.compare(6,5,str2,4,5) == 0)  cout << "therefore, both are apples\n";  return 0;  } |

* Tìm kiếm và thay thế:
  + Phương thức find() tìm kiếm xem một ký tự hay một chuỗi nào đó có xuất hiện trong một chuỗi str cho trước hay không. Có nhiều cách dùng phương thức này:

str.find(int ch, int pos = 0); tìm ký tự ch kể từ vị trí pos đến cuối chuỗi str

str.find(char \*s, int pos = 0); tìm s (mảng ký tự kết thúc ‘\0’) kể từ vị trí pos đến cuối

str.find(string& s, int pos = 0); tìm chuỗi s kể từ vị trí pos đến cuối chuỗi.

* + Nếu không quy định giá trị pos thì hiểu mặc nhiên là 0; nếu tìm có thì phương thức trả về vị trí xuất hiện đầu tiên, ngược lại trả về giá trị -1.

|  |
| --- |
| //find substring  #include <iostream>  #include <string>  #include <conio.h>  using namespace std;  int main ()  {  string str="ConCho chay qua rao";  cout << str.find("chay") << endl; // 7  cout << (int)str.find("Chay") << endl; // -1  getchar();  return 0;  } |

* + Hàm tìm kiếm ngược (rfind)

|  |
| --- |
| //find from back  #include <iostream>  #include <string>  #include <conio.h>  using namespace std;  int main ()  {  string str="ConCho chay qua chay qua rao";  cout << str.find("chay") << endl; // 7  cout << (int)str.rfind("chay") << endl; // 16  getchar();  return 0;  } |

* + Phương thức replace() thay thế một đoạn con trong chuỗi str cho trước (đoạn con kể từ một vị trí pos và đếm tới nchar ký tự ký tự về phía cuối chuỗi) bởi một chuỗi s nào đó, hoặc bởi n ký tự ch nào đó. Có nhiều cách dùng, thứ tự tham số như sau:

str.replace(int pos, int nchar, char \*s);

str.replace(int pos, int nchar, string s);

str.replace(int pos, int nchar, int n, int ch);

string str="con cho la con cho con. Con meo ko phai la con cho";

str.replace(4, 3, "CHO"); // "con CHO la con cho con. Con meo ko phai la con cho";

cout << str << endl;

getchar();

* Tách chuỗi:
  + Trong việc xử lý xâu ký tự, không thể thiếu được các thao tác tách xâu ký tự thành nhiều xâu ký tự con thông qua các ký tự ngăn cách. Các hàm này có sẵn trong các ngôn ngữ khác như Visual Basic, Java, hay thậm chí là trong <string.h>. Với STL, chúng ta có thể dễ dàng làm điều này với stringstream:

string S = "Xin chao tat ca cac ban"; // Khởi tạo giá trị của xâu

stringstream ss(S); // Khởi tạo stringstream từ xâu S

while (ss >> token) { // Đọc lần lượt các phần của xâu. Các phần tách nhau bởi dấu cách hoặc xuống dòng.

cout << token << endl;

}

**§2: KIỂU DỮ LIỆU TỰ ĐỊNH NGHĨA**

Bên cạnh những kiểu dữ liệu có sẵn như char, int, float,… C++ cũng cho phép chúng ta tự định nghĩa kiểu dữ liệu để phù hợp với mục đích cũng như tạo mã nguồn tối ưu nhất.

**1. Struct**

Struct trong C/C++ Các mảng trong C/C++ cho phép ta định nghĩa một vài loại biến có thể giữ giá trị của một vài thành viên cùng kiểu dữ liêu. Nhưng structure - cấu trúc là một loại dữ liệu khác trong ngôn ngữ lập trình C/C++, cho phép ta kết hợp các dữ liệu khác kiểu nhau. Các biến trong cùng 1 struct có thể cùng kiểu hoặc khác kiểu với nhau.

**a. Định nghĩa một struct**

Cú pháp định nghĩa một struct:

**struct <Tên struct>**

**{**

**<Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;**

**<Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;**

**…**

**};**

Ví dụ:

**struct Subject**

**{**

**string m\_name\_subject;**

**string m\_pre\_subject;**

**int m\_hour\_per\_week;**

**string m\_trainer;**

**};**

**b. Sử dụng struct**

Để truy cập vào các thành viên của struct ta sử dụng toán tử dấu chấm (.)đặt giữa tên biến kiểu struct và tên thành viên của struct cần truy cập.

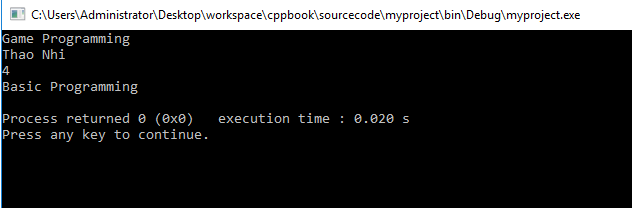
Một biến thuộc kiểu struct cũng có thể trở thành tham số của một hàm.

Có thể truy cập thành phần của struct thông qua biến con trỏ.

Dưới đây là ví dụ cho cách sử dụng cấu trúc trong C++:

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;    /\*Declare a struct called Subject\*/  struct Subject  {  string m\_name\_subject; //delare member types  string m\_pre\_subject;  int m\_hour\_per\_week;  string m\_trainer;  };    void PrintSubjectInformation(Subject object)  {  cout << object.m\_name\_subject << endl;  cout << object.m\_trainer << endl;  cout << object.m\_hour\_per\_week << endl;  cout << object.m\_pre\_subject << endl;  }    int main()  {  Subject subject;    // Access the member of struct Subject through subject  subject.m\_name\_subject = "Game Programming";  subject.m\_pre\_subject = "Basic Programming";  subject.m\_hour\_per\_week = 4;  subject.m\_trainer = "Thao Nhi";    PrintSubjectInformation(subject);    return 0;  } |

Chạy chương trình trên ta sẽ được kết quả như sau:



**2. Union**

Cũng giống như struct, union là một tập hợp các phần tử dữ liệu cùng kiểu hoặc khác kiểu được gộp chung thành một nhóm. Các phần tử này được gọi là thành viên của union

**a. Định nghĩa một union**

Cú pháp:

**union <Tên union>**

**{**

**<Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;**

**<Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;**

**…**

**};**

Ví dụ:

**union <Tên struct>**

**{**

**<Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;**

**<Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;**

**…**

**};**

**b. Sử dụng union**

Để truy cập vào các thành viên của union ta sử dụng toán tử dấu chấm (.)đặt giữa tên biến kiểu union và tên thành viên của union cần truy cập.

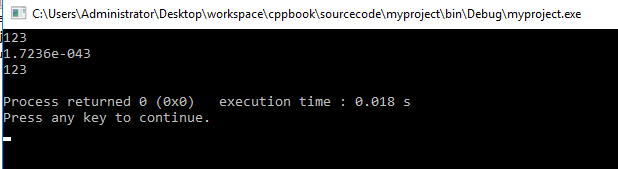
Một biến thuộc kiểu union cũng có thể trở thành tham số của một hàm.

Có thể truy cập thành viên của union thông qua biến con trỏ.

Ta sẽ xét một ví dụ:

|  |
| --- |
| #include <iostream>    using namespace std;    /\* Declare \*/  union Post  {  int m\_iValue;  float m\_fValue; //delare member types  long m\_lValue;  };    void PrintSubjectInformation(Post post)  {  cout << post.m\_iValue << endl;  cout << post.m\_fValue << endl;  cout << post.m\_lValue << endl;  }    int main()  {  Post post;  post.m\_iValue = 169; // Access the member of union  post.m\_fValue = 39.01;  post.m\_lValue = 123;    PrintSubjectInformation(post);    return 0;  } |

Chạy chương trình trên ta sẽ được:



Khi thay đổi giá trị post.m\_lValue = 123;. Ta đã làm ảnh hướng đến giá trị của các thành viên còn lại. Điều này sẽ được giải thích ở phần sau.

**3. So sánh struct và union**

Về mặt ý nghĩa, struct và union cơ bản giống nhau. Tuy nhiên, về mặt lưu trữ trong bộ nhớ, chúng có sự khác biệt rõ rệt như sau:

* struct: Dữ liệu của các thành viên của struct được lưu trữ ở những vùng nhớ khác nhau. Do đó kích thước của một struct chắc chắn lớn hơn kích thước của các thành viên cộng lại. Vì sao lại như vậy ? Câu trả lời là vì kích thước của khối nhớ là do hệ thống quy định. Ví dụ: Hệ thống nhớ quy định khối nhớ là 1 byte, thì kích thước của kiễu dữ liệu sSinhVien là 13 byte.

struct sSinhVien

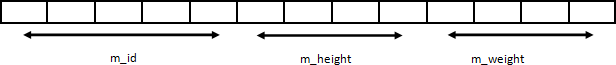
{

char m\_id[5];

float m\_height;

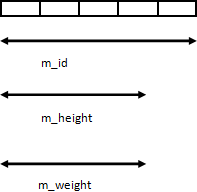
float m\_weight;

};



Thực chất struct trên nếu lấy kích thước thông thường sẽ có hiệu ứng Alignment nên kích thước sẽ không phải là 13 byte, nhưng đó là một vấn đề khác không bàn ở đây.

* union: với union, các thành viên sẽ dùng chung một vùng nhớ. Kích thước của union được tính là kích thước lớn nhất của kiểu dữ liệu trong union. Việc thay đổi nội dung của một thành viên sẽ dẫn đến thay đổi nội dung của các thành viên khác.



**CHƯƠNG V: LẬP TRÌNH CÓ CẤU TRÚC**

**§1: HÀM**

**1. Khái niệm hàm**

Nói một cách dễ hiểu ta có thể tưởng tượng hàm là một chương trình con trong chương trình lớn. Hàm nhận (hoặc không) các đối số và trả lại (hoặc không) một giá trị cho chương trình gọi nó.

Ta có 2 số nguyên 3 và 4. Khi truyền vào hàm nó sẽ giúp bạn trả về một giá trị là 7. Hàm này là hàm thực hiện công việc tính tổng của 2 số nhận vào và trả về giá trị là tổng của hai số đó.

Một chương trình dạng C++ console là tập hợp các hàm, trong đó có một hàm chính với tên gọi main(). Khi chạy chương trình, hàm main() sẽ được chay đầu đầu tiên và gọi đến hàm khác. Kết thúc hàm main() cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những thành phần riêng lẻ, hoạt động độc lập với chương trình lớn, có nghĩa một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác, dễ cho việc kiểm tra và bảo trì chương trình.

Ví dụ: Trường hợp không dùng hàm tính trung bình cộng 2 số

#include <iostream>

using namespace std;

int main()

{

// Trung bình cộng của 3 và 7

cout << (3 + 7)/2.0;

// Trung bình cộng của 4 và 6

cout << (4 + 6)/2.0

return 0;

}

Trường hợp dùng hàm tính trung bình cộng 2 số

#include <iostream>

using namespace std;

// Định nghĩa hàm tính trung bình cộng 2 số

double avg(int a, int b)

{

return (a + b)/2.0;

}

int main()

{

// Trung bình cộng của 3 và 7

cout << avg(3, 7);

// Trung bình cộng của 4 và 6

cout << avg(4, 6);

return 0;

}

Ngoài ra, chúng ta có thể kể chi tiết những lợi ích khi sử dụng hàm như sau:

* Tránh việc lặp đi lặp lại một khối lệnh cùng chức năng.
* Hỗ trợ việc thực hiện các chương trình lớn.
* Phục vụ các quá trình trừ tượng hóa.
* Mở rộng khả năng ngôn ngữ.
* Thuận tiện phát triển, nâng cấp chương trình.

**2. Khai báo hàm và định nghĩa hàm**

**a. Khai báo hàm**

Thông thường hàm thực hiện việc tính toán trên các tham số và cho lại giá trị kết quả, hoặc chỉ thực hiện một chức năng nào đó mà không cần trả lại kết quả. Kiểu giá trị trả về này được gọi là kiểu của hàm. Hàm thường được khai báo ở đầu chương trình. Cách để khai báo một hàm như sau:

**<kiểu của hàm> <tên hàm>(danh sách tham số) ;**

Trong đó:

* Kiểu của hàm: Là kiểu dữ liệu mà hàm trả về. Nếu không có giá trị trả về thì kiểu của hàm là void.
* Tên hàm: Là tên của hàm, do người lập trình đặt. Tên hàm không được chứa ký tự đặt biệt, không được bắt đầu bằng số.
* Danh sách tham số: là danh sách các tham số dùng như các biến cục bộ. Nếu có nhiều tham số, thì chúng sẽ được phân tách theo các dấu phẩy.
* Ví dụ:

int test(int); // Hàm test có tham số kiểu int và kiểu hàm là int int rand(); // Không tham số, kiểu hàm (giá trị trả lại) là int void write(float) ; //Hàm không trả lại giá trị (kiểu void).

**b. Định nghĩa hàm**

Cấu trúc một hàm bất kỳ được bố trí cũng giống như hàm main(). Cụ thể:

<kiểu hàm> <tên hàm>(danh sách tham số)

{

dãy lệnh của hàm ;

return (biểu thức trả về); // có thể nằm đâu đó trong dãy lệnh.

}

Trong đó return là để truyền dữ liệu cho nơi gọi hàm. Câu lệnh return bắt buộc là phải có đối với hàm có giá trị trả về. Và có thể có hoặc không đối với hàm không có giá trị trả về (có kiểu trả về là void). Đối với hàm có giá trị trả về thì kèm theo sau return là một biểu thức. Kiểu giá trị của biểu thức này chính là kiểu của hàm. Tuỳ theo mục đích của hàm mà chúng ta có thể đặt return bất cứ đâu trong dấu {} ở trên. Khi gặp return chương trình tức khắc thoát khỏi hàm và trả lại giá trị biểu thức sau return. Có thể có nhiều câu lệnh return trong hàm.

Ví dụ: Tính tổng 2 số nguyên (kiểu int). Hàm này có 2 tham số truyền vào (số nguyên a và số nguyên b) và giá trị trả về là số nguyên a+b.

int sum(int a, int b)

{

return (a+b);

}

Một ví dụ về hàm không có giá trị trả về:

void printNumber(int a)

{

std::cout << a;

}

Hàm trên thực hiện công việc in một số nguyên (được truyền vào) ra màn hình console. Không có kiểu trả về nên không cần return.

Danh sách tham số trong khai báo hàm có thể có hoặc không tên của biến, thong thường ta chỉ cần khai báo kiểu dữ liệu của tham số chứ không cần khai báo tên của biến. Trong khi trong định nghĩa hàm thì phải có đầy đủ. Hàm có thể có hoặc không có tham số, tuy nhiên cặp dấu () sau tên hàm vẫn phải được viết.

Ví dụ:

//Khai báo hàm

int sum(int, int); //không cần tên biến

int sum(int a, int b); //có tên biến

//Định nghĩa hàm

int sum(int a, int b) // có tên biến

{

return (a + b);

}

void print()

{

cout << “Hello”;

}

Tuy nhiên, ta có thể vừa khai báo vừa định nghĩa hàm cùng lúc, cho nên để cho ngắn gọn người ta thường viết đoạn chương trình trên như sau:

//Định nghĩa hàm

int sum(int a, int b) // có tên biến

{

return (a + b);

}

void print()

{

cout << “Hello”;

}

**3. Tham số của hàm** Thông thường ta có 2 cách để truyền tham số cho hàm đó là truyền tham trị và tham chiếu. Ta sẽ lần lượt xem xét 2 cách truyền tham sô này.

**a. Truyền tham trị**

Hiểu một cách đơn giản thì truyền theo tham trị là truyền giá trị của tham số. Khi hàm được gọi. Chương trình sẽ khởi tạo các ô nhớ tương ứng với danh sách tham số truyền vào. Sau đó sẽ sao chép các giá trị của tham số vào các ô nhớ mới này. Vì vậy những thay đổi trong ô nhớ mới này sẽ không ảnh hưởng đến các tham số truyền vào.

Cú pháp truyền tham trị: **<kiểu tham trị> <tên biến>;**

Để hiểu hơn về truyền theo kiểu tham trị cũng như cách thực hiện, ta xét các ví dụ sau:

Chương trình tính tổng 2 số:

|  |
| --- |
| #include <iostream>  using namespace std;    int sum(int a, int b)  {  return (a+b);  }    int main()  {  int a = 2, b = 3;  int s;  s = sum(a, b); // Gán giá trị là tổng a và b cho biến s;  cout << "Tong của a va b: " << s << endl;  return 0;  } |

Xét một ví dụ khác là chương trình hoán đổi 2 số:

|  |
| --- |
| #include <iostream>  using namespace std;  void swap(int a, int b)  {  int tamp = a;  a = b;  b = tamp;  }    int main()  {  int a = 2, b = 3;  cout << "Truoc khi hoan doi: " << a << ' ' << b << endl;;  swap(a, b); // hoán đổi  cout << "Sau khi hoan doi: " << a << ' ' << b << endl;  return 0;  } |

Sau khi chạy chương trình trên sẽ thấy kết quả trước và sau vẫn không đổi. Lý do là bởi vì lúc truyền tham số, chương trình sẽ khởi tạo 2 ô nhớ khác và sao chép giá trị truyền vào. Nên mọi thay đổi trong hàm sẽ không ảnh hưởng đến chương trình chính. Do đó không thể dùng cách truyền tham trị cho chương trình trên.

**b. Truyền tham chiếu**

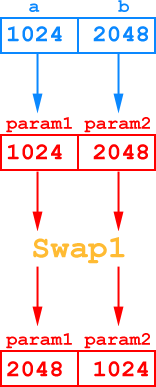
Tham chiếu nói một cách đơn giản là một bí danh của một biến. Nói cách khác, tham chiếu là một tên gọi khác của một biến đã có sẵn. Điều đó có nghĩa là, có thể thao tác trên biến đó thông qua tên biến hoặc tham chiếu.

Cú pháp truyền tham chiếu: **<kiểu tham chiếu> &<tên biến>;**

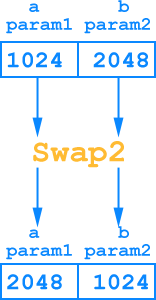
Để hiểu rõ lí do nên dùng tham chiếu ta sẽ xét lại ví dụ hoán đổi hai số.

|  |
| --- |
| #include <iostream>  using namespace std;  void Swap1(int param1, int param2) {  int temp = param1;  param1 = param2;  param2 = temp;  }  void Swap2(int& param1, int& param2) {  int temp = param1;  param1 = param2;  param2 = temp;  }  int main() {  int a = 1024;  int b = 2048;  cout << "a = " << a << ", b = " << b << endl;  cout << "Swap1: ";  Swap1(a, b);  cout << "a = " << a << ", b = " << b << endl;  cout << "Swap2: ";  Swap2(a, b);  cout << "a = " << a << ", b = " << b << endl;  return 0;  }  // Difference between value parameter and reference parameter  // a = 1024, b = 2048  // Swap1: a = 1024, b = 2048  // Swap2: a = 2048, b = 1024 |

Hàm swap1 sử dụng cách truyền tham trị, trong khi đó hàm swap2 sử dụng cách truyền tham chiếu. Kết quả hoàn toàn khác nhau là vì: Khi truyền tham trị, chương trình sẽ tạo ra hai biến mới và copy giá trị của hai tham trị vào, cho nên khi đó hàm chỉ hoán đổi hai biến là bản sao của hai tham trị với nhau mà thôi, tham trị hoàn toàn không tham gia vào quá trình này; Trong khi đó, khi truyền tham chiếu, chương trình sẽ không tạo ra bản sao nào cả mà chỉ đơn giản là dùng chính hai tham chiếu cho những lệnh trong hàm, cho nên giá trị của các tham chiếu đã bị thay đổi.



Khi truyền tham trị giá trị của a, b sẽ được sao chép vào 2 biến param1, param2 hay nói cách khác là đồ “fake”



Còn với tham chiếu thì chương trình sử dụng chính bản gốc “official”

**c. Hàm với tham số có giá trị ngầm định**

Xét một ví dụ sau:

|  |
| --- |
| #include <iostream>    int sum(int a, int b, int c, int d)  {  return (a + b + c + d);  }    int main()  {  int s = sum(1, 3, 4, 5);  std::cout << "Tong la: " << s;  return 0;  } |

Hàm sum trong chương trình trên thực hiện việc tính tổng 4 số nguyên. Tuy nhiên, chúng ta chỉ cần tính tổng của 2 hay 3 số nguyên mà không phải là 4 số nguyên. Vấn đề được giải quyết khi chúng ta viết thêm một hàm tính 2 số nguyên và 3 số nguyên khác. Hoặc cũng có thể gọi hàm với giá trị có tham số bằng 0 như sau:

int s = sum(1, 2, 0, 0);

Lời gọi hàm như vậy sẽ giúp chúng ta tính được tổng 2 số nguyên một và 2 từ hàm tính tổng 4 số nguyên ở trên. Một vấn đề khác được đặt ra khi trong chương trình có quá nhiều lần hàm được gọi để tính tổng 2 số nguyên. Như vậy, cứ mỗi lần gọi hàm, lúc nào cũng phải viết thêm 2 tham số ngầm định giá trị 0 ở sau.

Đó là trường hợp với ví dụ trên. Trong thực tế, hãy thử tưởng tượng nếu một hàm được gọi nhiều lần với nhiều tham số có giá trị ngầm định (không thay đổi giá trị qua các lần gọi). Mỗi lần như vậy lúc nào cũng phải viết thêm một danh sách dài các tham số có giá trị giống nhau. Đó thật sự là một công việc không mấy thú vị. Từ đó hàm có tham số với giá trị ngầm định được C++ mở rông ra (so với C) để giải quyết vấn đề này.

Cú pháp:

**<Kiểu hàm> <Tên hàm> (t/s1, t/s2, ..,t/sn, t/smd1 = gt1, t/smd2 = gt2,…, t/smdn = gtn);**

Trong đó:

* t/s1, t/s2,…,t/sn là các tham số được khai báo bình thường. Tức là gồm có kiểu dữ liệu và tên tham số.
* Các tham số ngầm định t/smd1 ,t/smd2, ..,t/smdn có gán thêm các giá trị ngầm định gt1, gt2,…, gtn.
* Một lời gọi bất kỳ đến hàm này đều phải có đầy đủ các tham số ứng với t/s1,…, t/sn (không phải tham số ngầm định). Nhưng có thể có hoặc không các tham số ngầm định ứng với t/smd1,…,t/smdn. Nếu tham số nào không có thì nó sẽ tự động gán với giá trị ngầm định đã khai báo.

Ví dụ: Hàm tính giá trị lũy thừa.

**int exponential(int x, int n = 2);**

Hàm này có một tham số có giá trị ngầm định là số mũ n. Nếu lời gọi hàm bỏ qua số mũ này thì chương trình sẽ hiểu là n = 2 và tính bình phương của x.

Khi gọi int e = exponential(4, 3); chương trình sẽ tính e = 43.

Khi gọi int e = exponential(4); chương trình sẽ tính e = 42.

Xét hàm tính tổng 4 số nguyên.

**int sum(int a, int b, int c = 0, int d = 0);**

Chúng ta có thể:

* Tính tổng của 4 số nguyên 1, 2, 3, 4 bằng cách gọi sum(1, 2, 3, 4).
* Tính tổng của 3 số nguyên 1, 2, 3 bằng cách gọi sum(1, 2, 3).
* Tính tổng của 2 số nguyên 1, 2 bằng cách gọi sum(1, 2).

**Chú ý:**

* Các tham số với giá trị ngầm định phải được khai báo liên tục và xuất hiện cuối cùng trong danh sách tham số.

**int sum(int a, int b = 0, int c, int d = 0);**

Trường hợp này sai vì các tham số ngầm định không liên tục.

**int sum(int a, int b = 0, int c = 0, int d);**

Trường hợp này sai vì tham số có giá trị ngầm định không ở cuối.

**4. Nạp chồng hàm (Function overloading)**

Xét một ví dụ:

|  |
| --- |
| #include <iostream>    int max(int a, int b)  {  if(a > b) return a;  else return b;  }    int main()  {  int c = max(4, 5);  std::cout << "c = " << c;  return 0;  } |

Khi chạy chương trình chúng ta nhận được kết quả c = 5 đúng như mong muống.

Vậy nếu thay dòng lệnh c = max(4, 5) thành max(4.3, 5.5) thì ta lại được kết quả là 5, không phải kết quả ta mong muốn (5.5). Trường hợp này ta phải viết chương trình tính max của hai số thực:

float fmax(float a, float b)

{

if(a > b) return a;

else return b;

}

Tương tự ta sẽ có:

double dmax(double a, double b)

{

if(a > b) return a;

else return b;

}

hay:

char cmax(char a, char b)

{

if(a > b) return a;

else return b;

}

Vậy chúng ta sẽ gặp phải 2 bất lợi lớn đó là:

* Có quá nhiều tên hàm khác nhau.
* Viết quá nhiều lần một hàm giống nhau.

Ta có cách để giải quyết cả hai vấn đề này. Tuy nhiên, nó sẽ ở phần sau. Bây giờ ta chỉ xét cách để giải quyết vấn đề thứ nhất. Đó là nạp chồng hàm (function overloading). Nạp chồng hàm cho phép ta khai báo và định nghĩa các hàm trên cùng với một tên gọi. Khi đó ta sẽ được:

|  |
| --- |
| int max(int a, int b)  {  if(a > b) return a;  else return b;  }    float max(float a, float b)  {  if(a > b) return a;  else return b;  }    double max(double a, double b)  {  if(a > b) return a;  else return b;  }    char max(char a, char b)  {  if(a > b) return a;  else return b;  } |

Và khi gọi hàm ở bất kì dạng nào như:

max(4, 5)

max(4.3, 5.5)

max(‘A’, ‘B’)

Ta đều có được kết quả mong muốn.

Lưu ý: Ta không thể nạp chồng hàm nếu như chỉ khác biệt về kiểu trả về.

**CHƯƠNG VI: TỆP**

**§1: KIỂU DỮ LIỆU TỆP**

**1. Vai trò của tệp**

Tất cả các dữ liệu ở thuộc các kiểu dữ liệu đã xét đều lưu ở bộ nhớ trong (RAM) và do đó sẽ bị mất khi tắt máy. Với một số bài toán khối lượng dữ liệu lớn, có yêu cầu lưu trữ xử lí nhiều lần, cần có kiểu dữ liệu tệp (file).

Kiểu dữ liệu tệp có những đặc điểm sau:

* Dữ liệu tệp được lưu trữ lâu dài ở bộ nhớ ngoài, không bị mất đi khi tắt nguồn điện.
* Lượng dữ liệu của tệp có thể rất lớn, phụ thuộc vào dung lượng đĩa.

**2. Phân loại tệp**

Xếp theo cách tổ chức dữ liệu, tệp có thể phân thành hai loại:

* Tệp văn bản: là tệp mà dữ liệu được ghi dưới dạng các kí tự theo mãi ASCII. Trong tệp văn bản, dãy kí tự kết thúc bởi kí tự xuống dòng hay kí tự kết thúc tệp tạo thành một dòng.
* Tệp có cấu trúc là tệp mà thành phần của nó được tổ chức theo một cấu trúc nhất định. Tệp nhị phân là một trường hợp riêng của tệp có cấu trúc. Dữ liệu ảnh, âm thanh, .. thường được lưu trữ dưới dạng tệp có cấu trúc.

Xếp theo cách truy cập tệp, ta có thể phân thành hai loại sau:

* Tệp truy cập tuần tự: cho phép truy cập đến một dữ liệu nào đó trong tệp chỉ bằng cách bắt đầu từ đầu tệp và đi qua lần lượt các dữ liệu trước nó.
* Tệp truy cập trực tiếp là tệp cho phép tham chiếu đến dữ liệu cần truy cập bằng cách xác địng trực tiếp vị trị (thường là số hiệu) của dữ liệu đó.

**§2: THAO TÁC VỚI TỆP TRONG C++**

**1. Đọc dữ liệu từ fstream**

Tới bây giờ, chúng ta đã sử dụng thư viện chuẩn iostream, cung cấp các phương thức cin và cout để đọc từ Standard Input và ghi tới Standard Output tương ứng. Bây giờ ta cần phải đọc và ghi dữ liệu với file. Điều này cần một Thư viện chuẩn C++ khác là fstream, mà định nghĩa 3 kiểu dữ liệu mới:

|  |  |
| --- | --- |
| **Kiểu dữ liệu** | **Miêu tả** |
| ofstream | Kiểu dữ liệu này biểu diễn Output File Stream và được sử dụng để tạo các file và để ghi thông tin tới các file đó |
| ifstream | Kiểu dữ liệu này biểu diễn Input File Stream và được sử dụng để đọc thông tin từ các file |
| fstream | Kiểu dữ liệu này nói chung biểu diễn File Stream, và có các khả năng của cả ofstream và ifstream, nghĩa là nó có thể tạo file, ghi thông tin tới file và đọc thông tin từ file |

Để thực hiện tiến trình xử lý file trong C++, bạn bao các header file là <iostream> và <fstream> trong source file của chương trình C++ của chúng ta.

**a. Mở một file trong C++ với fstream**

Một file phải được mở trước khi bạn có thể đọc thông tin từ nó hoặc ghi thông tin tới nó. Hoặc đối tượng ofstream hoặc đối tượng fstream có thể được sử dụng để mở một file với mục đích viết hoặc đối tượng ifstream được sử dụng để mở file chỉ với mục đích đọc.

Dưới đây là prototype của hàm open(), là một thành viên của các đối tượng fstream, ifstream và ofstream trong C++:

**void open(const char \*ten\_file, ios::mode);**

Tại đây, tham số đầu tiên xác định tên và vị trí của file để được mở và tham số thứ hai của hàm thành viên open() định nghĩa chế độ mà file nên được mở.

|  |  |
| --- | --- |
| **Chế độ** | **Miêu tả** |
| ios::app | Chế độ Append. Tất cả output tới file đó được phụ thêm vào cuối file đó |
| ios::ate | Mở một file cho outpur và di chuyển điều khiển read/write tới cuối của file |
| ios::in | Mở một file để đọc |
| ios::out | Mở một file để ghi |
| ios::trunc | Nếu file này đã tồn tại, nội dung của nó sẽ được cắt (truncate) trước khi mở file |

Bạn có thể kết hợp hai hoặc nhiều giá trị này bằng việc hoặc chúng cùng với nhau (sử dụng (|). Ví dụ, nếu bạn muốn mở một file trong chế độ ghi và muốn cắt (truncate) nó trong trường hợp nó đã tồn tại, bạn theo cú pháp sau:

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );

Tuy nhiên, đó là cách tổng quát. Ta có cách ngắn gọn hơn để làm điều này:

**ofstream fo(“file.dat”);**

**ifstream fi(“file.dat”);**

Tương đương với:

ofstream fo;

fo.open(“file.dat”, ios::out);

ifstream fi;

fi.open(“file.dat”, ios::in);

**b. Đóng một file trong C++ với fstream**

Để đóng một file ta thực hiện cú pháp:

**<tên biến file>.close();**

Ví dụ: **outfile.close();**

Ví dụ làm việc với file

|  |
| --- |
| #include <fstream>  #include <iostream>  using namespace std;    int main ()  {    char data[100];  // mo mot file trong che do write.  ofstream outfile;  outfile.open("hdi.dat");  cout << "Ghi du lieu toi file!" << endl;  cout << "Nhap ten cua ban: ";  cin.getline(data, 100);  // ghi du lieu da nhap vao trong file.  outfile << data << endl;  cout << "Nhap tuoi cua ban: ";  cin >> data;  cin.ignore();    // ghi du lieu da nhap vao trong file.  outfile << data << endl;  // dong file da mo.  outfile.close();  // mo mot file trong che do read.  ifstream infile;  infile.open("hdi.dat");    cout << "\n===========================\n" ;  cout << "Doc du lieu co trong file!" << endl;  infile >> data;  // ghi du lieu tren man hinh.  cout << data << endl;    // tiep tuc doc va hien thi du lieu.  infile >> data;  cout << data << endl;  // dong file da mo.  infile.close();  return 0;  } |

**2. Hàm freopen()**

Thông thường trong các cuộc thi thông thường, vấn đề làm việc với file thường chỉ là đọc và ghi file text. Cho nên có một cách thao tác với file khá đơn giản là sử dụng hàm mở file freopen() trong thư viện stdio.h của C. Prototype của hàm này như sau:

**FILE \* freopen ( const char \* filename, const char \* mode, FILE \* stream );**

Trong đó:

* filename: Là chuỗi chứa tên file được mở (có thể là đường dẫn đến file).
* mode: Chế độ truy cập file. Mode có thể là một trong những giá trị dưới đây:

|  |  |
| --- | --- |
| **mode** | **Miêu tả** |
| "r" | Mở một file để đọc. File phải tồn tại |
| "w" | Tạo một file trống để ghi. Nếu một file với cùng tên đã tồn tại, nội dung của nó bị tẩy đi và file được xem như một file trống mới |
| "a" | Phụ thêm (append) tới một file. Với các hoạt động ghi, phụ thêm dữ liệu tại cuối file. File được tạo nếu nó chưa tồn tại |
| "r+" | Mở một file để ghi và đọc. File phải tồn tại |
| "w+" | Tạo một file trống để ghi và đọc |
| "a+" | Mở một file để đọc và phụ thêm |

* stream: Là con trỏ tới đối tượng file mà nhận diện stream đang được mở.

Nếu file được mở lại thành công, hàm trả về một con trỏ tới một đối tượng nhận diện Stream đó, nếu không thì con trỏ null được trả về. Tức là sau khi mở file bằng freopen() ta vẫn sẽ sử dụng luồng dữ liệu giống như standard i/o stream (scanf, printf, cin, cout, …) hoạt động bình thường như khi làm việc với màn hình. Ví dụ:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int main() {  freopen("IN.TXT", "r", stdin);  freopen("OUT.TXT", "w", stdout);  int a, b, ans = 0;  cin >> a >> b;  for (int i = a; i <= b; i++)  ans += b;  cout << ans;  return 0;  } |

**CHƯƠNG VII: CON TRỎ**

**§1: CON TRỎ**

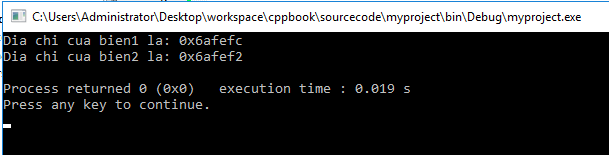
Con trỏ - Pointer trong ngôn ngữ C /C++ rất dễ học và thú vị. Một vài tác vụ trong ngôn ngữ C /C++ được thực hiện dễ dàng hơn nhờ con trỏ, và những tác vụ khác trở nên linh hoạt hơn, như trong việc cấp phát bộ nhớ, không thể thực hiện mà không dùng con trỏ

**1. Khái niệm con trỏ**

Như bạn biết, mỗi biến trong một vùng nhớ nhất định và mỗi vùng nhớ này có địa chỉ có nó được định nghĩa để dễ dàng trong việc truy cập sử dụng toán tử (&) tương ứng với địa chỉ của nó trong bộ nhớ. Xem xét ví dụ dưới đây, sẽ in ra địa chỉ của biến được định nghĩa:

|  |
| --- |
| #include <iostream>  using namespace std;  int main ()  {  int bien1;  char bien2[10];  cout << "Dia chi cua bien1 la: ";  cout << &bien1 << endl;  cout << "Dia chi cua bien2 la: ";  cout << &bien2 << endl;  return 0;  } |

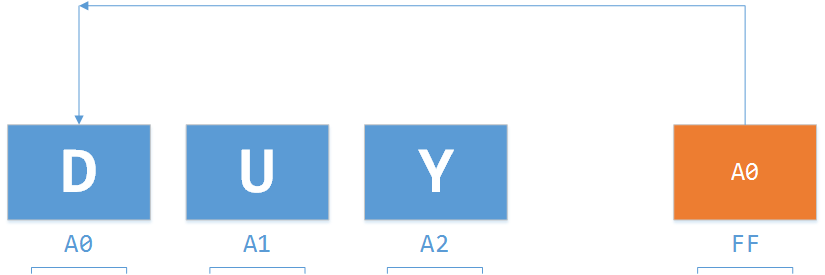
Chạy chương trình trên sẽ được kết quả như sau:



Vậy con trỏ là gì ?

Một con trỏ là một biến mà trong đó giá trị của nó là địa chỉ của biến khác. Ví dụ như địa chỉ của vùng nhớ. Giống như các biến và hằng số, ta phải khai báo con trỏ trước khi có thể sử dụng nó để lưu trữ bất kì địa chỉ của biến nào.

Nếu chưa hình dung được ta sẽ xem xét ví dụ sau:



Trong hình minh họa trên, biến con trỏ (có địa chỉ là FF) sẽ quản lí vùng nhớ chuỗi “DUY” (bắt đầu từ địa chỉ A0). Dĩ nhiên đây chỉ là hình minh họa cho định nghĩa, bản chất nó còn nhiều điểm cần làm rõ bên dưới.

Dạng tổng quát của việc khai báo con trỏ như sau:

**<kiểu dữ liệu> \*<tên biến>;**

Ở đây, **<kiểu dữ liệu>** là kiểu dữ liệu cơ bản con trỏ, nó là kiểu hợp lệ trong ngôn ngữ C++ và **<tên biến>** là tên giá trị của con trỏ. Phần ký tự \* sử dụng trong khai báo con trỏ giống như việc bạn sử dụng cho phép nhân. Mặc dù vậy, trong khai báo này, ký tự \* được thiết kế để sử dụng các biến của con trỏ. Dưới đây là một số cách khai báo hợp lệ của con trỏ:

**int \*sv; // tro toi mot gia tri nguyen**

**double \*nv; // tro toi mot gia tri double**

**float \*luong; // tro toi mot gia tri float**

**char \*ten // tro toi mot ky tu**

Kiểu dữ liệu thực sự của giá trị của tất cả các con trỏ, có thể là integer, float, character, hoặc kiểu khác, là giống như, một số long hexa biểu diễn một địa chỉ bộ nhớ. Điểm khác nhau duy nhất của các con trỏ của các kiểu dữ liệu khác nhau là kiểu dữ liệu của biến hoặc hằng số mà con trỏ chỉ tới.

**2. Sử dụng con trỏ trong C++**

Bởi vì con trỏ chỉ lưu giữ địa chỉ, nên khi bạn gán giá trị cho một con trỏ, giá trị đó phải là địa chỉ. Để lấy địa chỉ của một biến, chúng ta sử dụng address-of operator (&).

int main()

{

int a = 5;

int \*ip = &a;

return 0;

}

Khi đó biến ip sẽ lưu trữ địa chỉ của byte đầu tiên của biến a. Lưu ý toán tử & ở đây là unary operator – toán tử một ngôi, hoàn toàn khác với toán tử & hai ngôi (bitwise).

Một toán tử khác thường sử dụng với con trỏ là dereference operator (\*). Dereference operator dùng để lấy nội dung của địa chỉ mà biến đó trỏ vào.

|  |
| --- |
| #include <iostream>    using namespace std;    int main()  {  int a = 5;    int \*ip = &a;  cout << \*ip << endl;    void \*vp = &a;  cout << \*(int \*)vp << endl;    return 0;  } |

Trong chương trình trên, để lấy giá trị được lưu trữ trong biến a có hai cách: gọi trực tiếp biến a hoặc gọi gián tiếp qua con trỏ ip. Khi đó, ta có thể thay đổi giá trị của biến a thông qua con trỏ ip như sau: \*ip = 11;

Một lưu ý là con trỏ kiểu void có thể lưu địa chỉ của biến bất kì, nhưng không thể dereference trực tiếp. Nghĩa là nó không thể lấy nội dung của memory address chứa trong nó, vì con trỏ void không biết kiểu dữ liệu của đối tượng mà nó trỏ đến. Do đó, lập trình viên cần định kiểu hoặc gán nó cho một con trỏ khác có kiểu dữ liệu cụ thể trước khi sử dụng.

Ngoài ra ta có thể gán trực tiếp một con trỏ cho một con trỏ khác. Tuy nhiên, khi đó hai con trỏ sẽ cùng lưu trữ memory address giống nhau, dễ xảy ra tình trạng "dangling pointer" và gây ra lỗi memory leak nếu người lập trình không quản lý chương trình tốt.

**3. Kích thước của con trỏ**

Như ta biết, những biến có kiểu dữ liệu khác nhau sẽ được cấp phát vùng nhớ có kích thước khác nhau. Còn với biến con trỏ thì như thế nào ? Để trả lời cho câu hỏi trên, trước tiên chúng ta cần trả lời câu hỏi: Biến con trỏ dùng để làm gì ? Như đề cập ở trên, biến con trỏ dùng để lưu trữ memory address của một biến khác. Dù cho biến con trỏ được khai báo như thế nào đi nữa thì giá trị được lưu trữ trong nó cũng là một memory address. Đến đây người không có kiến thúc lập trình cũng thấy được rằng đối với biến con trỏ, kích thước của nó là một hằng số, bất kể kiểu dữ liệu mà nó được khai báo. Hằng số này phụ thuộc vào OS, IDE, Compiler, … Trên hệ điều hành 32-bit, hằng số này là 4 byte.

Vậy nếu kích thước của con trỏ là một hằng số, tại sao lại có nhiều kiểu biến con trỏ như vậy ? Chúng ta cùng xét ví dụ sau:

|  |
| --- |
| #include <iostream>    using namespace std;    int main ()  {  int a = 257;  char \*cp = (char \*)&a;    cout << \*cp;    return 0;  } |

Kết quả in ra màn hình là ký tự ☺. Để lý giải cho trường hợp này, minh họa bằng cách cụ thể hóa vùng nhớ của biến a như sau: 0001 0001 0000 0000

Do kiểu char có kích thước 1 byte, khi ép kiểu (char \*), biến cp chỉ nhận được dữ liệu trong byte đầu tiên (byte trái nhất) của biến a, dẫn đến việc đọc thiếu một phần dữ liệu có trong biến a. Kiểu dữ liệu sẽ quy định cho biến con trỏ đọc số byte tương ứng kể từ ô nhớ đầu tiên đó.

**4. Các phép toán trên con trỏ**

C++ cho phép thực hiện các phép toán cộng, trừ trên con trỏ thông qua các toán tử +, -, ++, --. Khi đó, giá trị địa chỉ lưu trữ trong con trỏ sẽ được cộng/trừ số byte tương ứng với kiểu dữ liệu của con trỏ. Thao tác tương tự như với dữ liệu kiểu số:

|  |
| --- |
| int main()  {  int a = 5;  int \*ip = &a;    cout << ip + 1;  cout << ip++; //cout << ip; ip = ip + 1;  cout << ++ip; //ip = ip + 1; cout << ip;    return 0;  } |

Ở dòng thứ 6, giá trị in ra màn hình là địa chỉ của biến a + 1, con trỏ ip vẫn mang giá trị là địa chỉ của biến a. Để dễ hiểu hơn, giả sử biến a có địa chỉ là D0, giá trị in ra màn hình ở dòng 6 là D4 (không phải là D1!). Dòng 7 và 8 thao tác tương tự như với dữ liệu kiểu số, sau khi thực hiện, giá trị chứa trong con trỏ ip đã bị thay đổi. Ngoài các phép toán giữa con trỏ với số, ta còn có thể thực hiện phép trừ hai con trỏ cùng kiểu. Khi đó, giá trị trả về là một số nguyên độ chênh lệch địa chỉ giữa hai con trỏ đó với đơn vị là kiểu dữ liệu của con trỏ.

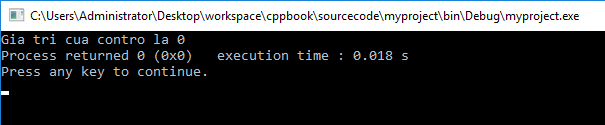
**5. Một vài vấn đề khác về con trỏ**

a. Con trỏ NULL

Con trỏ NULL là một hằng với một giá trị là 0 được định nghĩa trong một vài thư viện chuẩn, gồm iostream. Xét ví dụ sau:

|  |
| --- |
| #include <iostream>  using namespace std;  int main ()  {  int \*ptr = NULL;  cout << "Gia tri cua contro la " << ptr ;    return 0;  } |

Kết quả nếu chạy chương trình là:



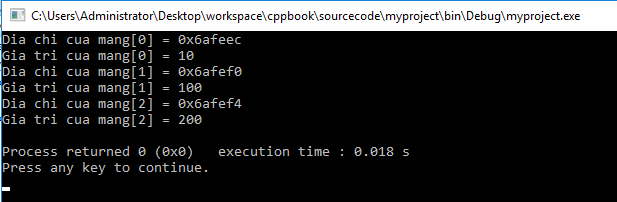
Trên hầu hết Hệ điều hành, các chương trình không được cho phép để truy cập bộ nhớ tại địa chỉ 0, bởi vì, bộ nhớ đó được dự trữ bởi Hệ điều hành. Tuy nhiên, địa chỉ bộ nhớ 0 có ý nghĩa đặc biệt; nó báo hiệu rằng con trỏ không được trỏ tới một vị trí ô nhớ có thể truy cập. Nhưng theo qui ước, nếu một con trỏ chứa giá trị 0, nó được xem như là không trỏ tới bất cứ thứ gì.

**b. Con trỏ và mảng**

Con trỏ và Mảng có mối liên hệ chặt chẽ. Thực tế, con trỏ và mảng là có thể thay thế cho nhau trong một số trường hợp. Ví dụ, một con trỏ mà trỏ tới phần đầu mảng có thể truy cập mảng đó bởi sử dụng: hoặc con trỏ số học hoặc chỉ mục mảng. Xét ví dụ sau:

|  |
| --- |
| #include <iostream>    using namespace std;  const int MAX = 3;    int main ()  {  int mang[MAX] = {10, 100, 200};  int \*contro;    // chung ta co mot mang dia chi trong con tro.  contro = mang;  for (int i = 0; i < MAX; i++)  {  cout << "Dia chi cua mang[" << i << "] = ";  cout << contro << endl;    cout << "Gia tri cua mang[" << i << "] = ";  cout << \*contro << endl;    // tro toi vi tri tiep theo  contro++;  }  return 0;  } |

Ouput:



Tuy nhiên, con trỏ và mảng không hoàn toàn thay thế được cho nhau. Ví dụ:

#include <iostream>

using namespace std;

const int MAX = 3;

int main ()

{

int mang[MAX] = {10, 100, 200};

for (int i = 0; i < MAX; i++)

{

\*mang = i; // Day la cu phap chinh xac

mang++; // cu phap nay la sai, nen chu y.

}

return 0;

}

Việc áp dụng toán tử con trỏ \* tới biến mang là hoàn hảo, nhưng nó không hợp lệ khi sửa đổi giá trị biến mang. Lý do là biến mang là một constant mà trỏ tới phần đầu mảng và không thể được sử dụng như là l-value. Bởi vì, một tên mảng tạo một hằng con trỏ, nó có thể vẫn được sử dụng trong các biểu thức con trỏ, miễn là nó không bị sửa đổi. Ví dụ sau là một lệnh hợp lệ mà gán mang[2] giá trị 500.

**\*(mang + 2) = 500;**

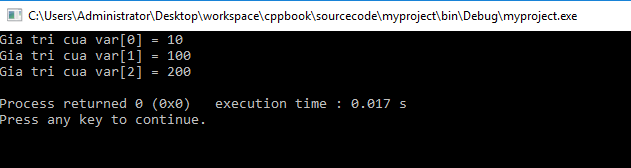
Lệnh trên là hợp lệ và sẽ biên dịch thành công bởi vì mang không bị thay đổi.

**c. Mảng con trỏ**

Trước khi chúng ta hiểu về khái niệm mảng các con trỏ, chúng ta xem xét ví dụ sau, mà sử dụng một mảng gồm 3 số integer:

|  |
| --- |
| #include <iostream>    using namespace std;  const int MAX = 3;    int main ()  {  int var[MAX] = {10, 100, 200};    for (int i = 0; i < MAX; i++)  {  cout << "Gia tri cua var[" << i << "] = ";  cout << var[i] << endl;  }  return 0;  } |

Khi code trên được biên dịch và thực thi, nó cho kết quả sau:

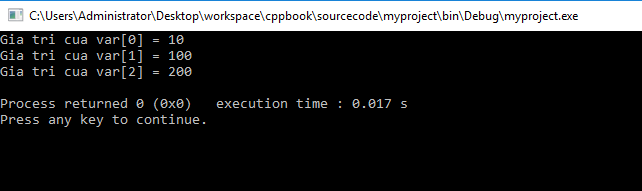


Có một tình huống khi chúng ta muốn duy trì một mảng, mà có thể lưu giữ các con trỏ tới một kiểu dữ liệu int hoặc char hoặc bất kỳ kiểu nào khác. Sau đây là khai báo một mảng của các con trỏ tới một integer: **int \*contro[MAX];**

Nó khai báo contro như là một mảng các con trỏ MAX kiểu integer. Vì thế, mỗi phần tử trong contro, bây giờ giữ một con trỏ tới một giá trị int. Ví dụ sau sử dụng 3 số integer, mà sẽ được lưu giữ trong một mảng các con trỏ như sau:

|  |
| --- |
| #include <iostream>  using namespace std;  const int MAX = 3;  int main ()  {  int var[MAX] = {10, 100, 200};  int \*contro[MAX];  for (int i = 0; i < MAX; i++)  {  contro[i] = &var[i]; // gan dia chi cua so nguyen.  }  for (int i = 0; i < MAX; i++)  {  cout << "Gia tri cua var[" << i << "] = ";  cout << \*contro[i] << endl;  }  return 0;  } |

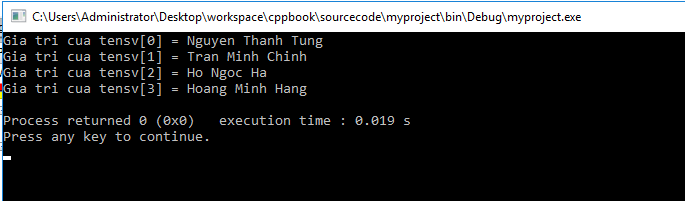
Output:



Ta có thể sử dụng một mảng các con trỏ tới ký tự để lưu giữ một danh sách các chuỗi như sau:

|  |
| --- |
| #include <iostream>    using namespace std;  const int MAX = 4;    int main ()  {  char \*tensv[MAX] = {  "Nguyen Thanh Tung",  "Tran Minh Chinh",  "Ho Ngoc Ha",  "Hoang Minh Hang",  };  for (int i = 0; i < MAX; i++)  {  cout << "Gia tri cua tensv[" << i << "] = ";  cout << tensv[i] << endl;  }  return 0;  } |

Output:



**d. Con trỏ tới con trỏ**

Một con trỏ tới một con trỏ là một form không định hướng hoặc một chuỗi con trỏ. Thông thường, một con trỏ chứa địa chỉ của một biến. Khi chúng ta định nghĩa một con trỏ tới một con trỏ, con trỏ đầu tiên chứa địa chỉ của con trỏ thứ hai, mà trỏ tới vị trí mà chứa giá trị thực sự như hiển thị trong sơ đồ dưới đây:

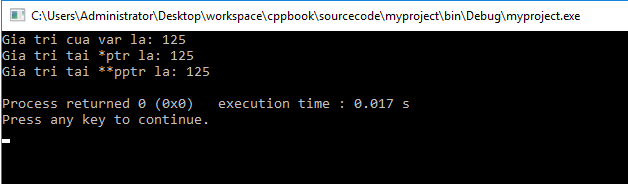


Một biến, mà là một con trỏ tới một con trỏ, phải được khai báo. Điều này được thực hiện bởi việc đặt một dấu sao (\*) ở trước tên của nó. Ví dụ, sau đây là khai báo một con trỏ tới một con trỏ trong kiểu int: **int \*\*var;**

Khi một giá trị mục tiêu được trỏ không định hướng bởi một con trỏ tới một con trỏ, truy cập giá trị đó yêu cầu rằng toán tử dấu sao được áp dụng hai lần, như dưới ví dụ:

|  |
| --- |
| #include <iostream>    using namespace std;    int main ()  {  int var;  int \*ptr;  int \*\*pptr;  var = 125;  // lay dia chi cua var  ptr = &var;  // lay dia chi cua ptr boi su dung dia chi cua toan tu &  pptr = &ptr;  // Lay gia tri boi su dung pptr  cout << "Gia tri cua var la: " << var << endl;  cout << "Gia tri tai \*ptr la: " << \*ptr << endl;  cout << "Gia tri tai \*\*pptr la: " << \*\*pptr << endl;  return 0;  } |

Output:

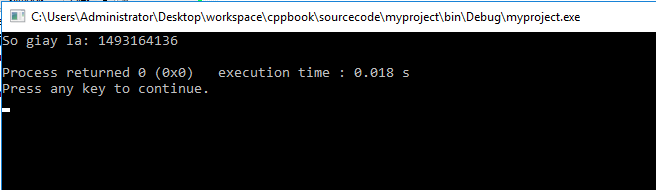


**e. Truyền con trỏ tới hàm**

C++ cho phép truyền một con trỏ tới một hàm. Để làm điều này, đơn giản chỉ cần khai báo tham số hàm như ở dạng một kiểu con trỏ. Ở ví dụ đơn giản dưới đây, chúng ta truyền một con trỏ unsigned long tới một hàm và thay đổi giá trị bên trong hàm, mà phản chiếu trở lại trong khi gọi hàm:

|  |
| --- |
| #include <iostream>  #include <ctime>    using namespace std;  void laySoGiay(unsigned long \*par);  int main ()  {  unsigned long sogiay;  laySoGiay( &sogiay );  // in gia tri  cout << "So giay la: " << sogiay << endl;  return 0;  }  void laySoGiay(unsigned long \*par)  {  // Lay so giay hien tai  \*par = time( NULL );  return;  } |

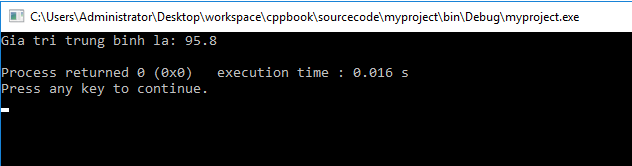
Output:



Hàm, mà có thể chấp nhận một con trỏ, cũng có thể chấp nhận một mảng như ví dụ sau:

|  |
| --- |
| #include <iostream>  using namespace std;    // khai bao ham:  double giaTriTB(int \*arr, int size);    int main ()  {  // mot mang so nguyen co 5 phan tu.  int doanhSoBH[5] = {122, 35, 27, 235, 60};  double trungbinh;    // truyen con tro toi mang duoi dang mot tham so.  trungbinh = giaTriTB( doanhSoBH, 5 ) ;    // hien thi ket qua  cout << "Gia tri trung binh la: " << trungbinh << endl;    return 0;  }  double giaTriTB(int \*arr, int size)  {  int i, sum = 0;  double trungbinh;    for (i = 0; i < size; ++i)  {  sum += arr[i];  }    trungbinh = double(sum) / size;    return trungbinh;  } |

Output:



**f. Trả về con trỏ từ hàm**

Như chúng ta đã thấy cách C++ cho phép trả về một mảng từ một hàm, tương tự như vậy, C++ cho phép trả về một con trỏ từ một hàm. Để làm điều này, phải khai báo một hàm trả về một con trỏ như sau:

**int \* tenHam()**

**{**

**.**

**.**

**.**

**}**

Điều thứ hai cần ghi nhớ là, nó không là ý kiến tốt để trả về địa chỉ của một biến cục bộ tới ngoại vi của một hàm, vì thế bạn sẽ phải định nghĩa biến cục bộ như là biến static.

Bây giờ, giả sử hàm sau sẽ tạo 10 số ngẫu nhiên và trả về chúng bởi sử dụng một tên mảng mà biểu diễn một con trỏ, ví dụ, địa chỉ đầu tiên của phần tử mảng đầu tiên.

|  |
| --- |
| #include <iostream>  #include <ctime>  #include <stdlib.h>  using namespace std;    // phan dinh nghia ham de tao va tra ve cac so ngau nhien.  int \* soNgauNhien( )  {  static int r[10];      srand( (unsigned)time( NULL ) );  for (int i = 0; i < 10; ++i)  {  r[i] = rand();  cout << r[i] << endl;  }    return r;  }    // ham main de goi phan dinh nghia ham tren.  int main ()  {  // mot con tro tro toi mot so nguyen.  int \*p;    p = soNgauNhien();  for ( int i = 0; i < 10; i++ )  {  cout << "Gia tri cua \*(p + " << i << ") : ";  cout << \*(p + i) << endl;  }    return 0;  } |

Output:

