C++ Training Course

Functions, Passing variable to functions

Lesson Objectives

- Understand about function and how to call a function
- Understand about pass by value, reference and address
- Understand about return by value, reference, and address
- Understand about function pointer

Section 1

Function and how to call a function

Agenda

- What is functions
- Structure of a function
- Arguments and parameter
- Function declaration and function prototypes

đối số
tham số
khai báo
nguyên mẫu

What is function

- A function is a self-contained program segment that carries out a specific, well-defined task
  Hàm là một đoạn chương trình độc lập thực hiện một nhiệm vụ cụ thể, được định nghĩa rõ ràng
- Functions are generally used as abbreviations for a series of instructions that are to be executed more than once
  Hàm thường được sử dụng như một cách viết tắt cho một loạt các lệnh sẽ được thực hiện nhiều lần
- Functions are easy to write and understand
  hàm dễ viết và dễ hiểu
- Debugging the program becomes easier as the structure of the program is more apparent, due to its modular form
  quá trình gỡ lỗi trở lên dễ dàng hơn vì cấu trúc của chương tình rõ ràng hơn do cáo dạng module
- Programs containing functions are also easier to maintain, because modifications, if required, are confined to certain functions within the program
  chương trình bao gồm nhiều hàm cũng dễ bảo trì hơn bởi vì các sửa đổi, nếu cần chỉ giới hạn trong các hàm nhất định trong chương trình

The Function Structure

- The general syntax of a function in C is : cú pháp chugn của một hàm trong C là:

```
type_specifier function_name(arguments)
{
        body of the function
}
```

- The type_specifier specifies the data type of the value, which the function will return. type_specifier chỉ định kiểu dữ liệu của giá trị mà hàm sẽ trả về
- A valid function name is to be assigned to identify the function. Một tên hàm hợp lệ phải được gán để xác định hàm.
- Arguments appearing in parentheses are also termed as formal parameters. Các đối số xuất hiện trong dấu ngoặc đơn cũng được gọi là tham số hình thức.

Arguments and parameter

- A parameter is a variable in a method definition.

Tham số (parameter) là một biến trong định nghĩa của hàm.

- A argument is the data you pass into the method's parameters.

Đối số (argument) là dữ liệu bạn truyền vào các tham số của hàm.

```
#include <stdio.h>
main()
{
     int i;
     for (i =1; i <=10; i++)
     printf ("\nSquare of %d is %d ", i,squarer (i));
}

squarer (int x)
/* int x; */
{
     int j;
     j = x * x;
     return (j);
}
```

Function declaration

- Declaring a function becomes compulsory when
  Việc khai báo một hàm trở nên bắt buộc khi hàm đó được sử dụng trước khi định nghĩa nó.
  the function is being used before its definition
- The address() function is called before it is defined
  Hàm address() được gọi trước khi nó được định nghĩa.
- Some C compilers return an error, if the function is
  Một số trình biên dịch C sẽ trả về lỗi nếu hàm không được khai báo trước khi gọi.
  not declared before calling
- This is sometimes referred to as Implicit declaration
  Điều này đôi khi được gọi là khai báo ngầm.

```
#include <stdio.h>
main()
{
      .
      .
      address()
      .
      .
}


address()
{
      .
      .
      .
}
```

Function Prototypes nguyên mẫu hàm

- char abc(int x, int y);

Function Prototypes (Nguyên mẫu hàm) là các khai báo hàm được viết trước khi hàm được sử dụng trong chương trình. Chúng chỉ rõ tên hàm, kiểu trả về và các tham số (nếu có) mà hàm sẽ nhận. Việc sử dụng function prototypes đảm bảo rằng trình biên dịch biết về sự tồn tại của hàm trước khi nó được gọi.

Section 2

Pass by value and pass by reference and pass by address

Agenda

- Pass by value
- Pass by reference
- Pass by address

inline function:
không sử dụng trong hàm đệ quy, hàm có biến static, hàm có ...
nên sử dụng khi function ngắn dưới 10 dòng

khi nào truyền vào tham chiếu khi nào truyền vào con trỏ:
+ trong các hàm thông thường, sử dụng tham chiếu khi muốn thay đổi giá trị của biến gốc được truyền vào hàm
+ truyền con trỏ khi muốn truyền một mảng

truyền tham số vào function đặc biệt (main):
+ hàm main có thể truyền vào 2 đố số: argc (argument count) và argv (argument vector)
+ cấu trúc int main (int argc, char* argv[]){}
+ việc điền đối số khi gọi hàm main giúp tăng tính linh hoạt cho hàm main: với số lượng đối số nhất định thì cách hàm main chạy sẽ khác nhau (cập nhật luật cho chương trình)
+ có thể sử dụng danh sách đối số truyền vào làm các biến local mà không cần khai báo
+ CHÚ Ý:
  .argv luôn phải là các chuỗi, trong đó argv[0] mặc định là tên chương trình
  .argc luôn min = 1 (tên chương trình)
  .để truyền đối số vào hàm ta phải biên dịch thủ công bằng g++

các kiểu khai báo biến:
+ int a = 0;
+ int a(0);
+ int a{0};
+ int a = 0xA // hex
+ int a = 071 // oc
+ int a = 0b1001 // bit
+ int a = 1E2 // dấu phẩy động

Pass by value

- We can pass the literal, value of variable or an operator to a parameter of function

Chúng ta có thể truyền một hằng số, giá trị của biến hoặc một toán tử vào một tham số của hàm.

- Because a copy of the argument is passed to the function, the original argument can not be modified by the function

Bởi vì một bản sao của đối số được truyền vào hàm, nên đối số gốc không thể được sửa đổi bởi hàm.

```
#include <iostream>
 using namespace std;
void vd(int y)
{
    cout << "y = " << y << '\n';
}

int main()
{
    vd(5); // goi lan dau
    int x = 6;
    vd(x); // goi lan hai
    vd(x+1); // goi lan ba

    return 0;
}
```

khi truyền vào giá trị thì thực tế chỉ là sao chép giá trị tại vị trí cũ và lưu vào stack sau đó truyền vào hàm và tính toán; sau khi xong việc vùng nhớ tại stack đó sẽ được giải phóng.

Hàm luôn được lưu ở tex segment của chương trình, chỉ tham số mới được dùng ở stack

Pass by reference

- We pass value of variable to the function

chúng ta có thể truyền giá trị của một biến vào hàm thông qua tham chiếu

- Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument

vì một tham chiếu tới 1 biến được xử lý hoàn toàn giống biến đó, bất kỳ thay đổi nào được thực heinej trên tham chiếu cũng được truyền qua đối số

```
#include <iostream>
using namespace std;
void vd(int &y) // y la mot bien tham chieu
{
    y = y + 1;
} // y bi huy o day

int main()
{
    int x = 5;
    cout << "x = " << x << '\n';
    vd(x);
    cout << "x = " << x << '\n';
    return 0;
}
```

Pass by address

- There is one more way to pass variables to functions, and that is by address. Passing an argument by address involves passing the address of the argument variable rather than the argument variable itself

- It is always a good idea to ensure parameters passed by address are not null pointers before dereferencing them

```
void printArray(int *array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        return;

    for (int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';
}

int main()
{
    int array[6]{ 6, 5, 4, 3, 2, 1 };
    printArray(array, 6);
}
```

+Có một cách khác để truyền biến vào hàm, đó là bằng cách sử dụng địa chỉ. Việc chuyển đổi số bằng địa chỉ bao gồm việc truyền địa chỉ của biến đối số thay vì biến đối số đó chính nó.
+Luôn luôn là ý tưởng tốt khi đảm bảo các tham số được chuyển bằng địa chỉ không phải là con trỏ null trước khi giải tham chiếu chúng.

check null:
```
void processPointer(int* ptr) {
    if (ptr != nullptr) {
    } else {}
}
```

cú pháp inline: // thêm từ khóa inline khi định nghĩa hàm.

Lợi ích:

+Tăng hiệu suất: Việc sử dụng từ khóa inline giúp giảm thiểu overhead của việc gọi hàm, bởi vì mã nguồn của hàm được nhúng trực tiếp vào các điểm gọi của nó, giảm số lần nhảy tới và trở về từ hàm.

+Tiết kiệm bộ nhớ: Không tạo bản sao của hàm khi sử dụng inline giúp tiết kiệm bộ nhớ, đặc biệt là khi hàm được gọi nhiều lần trong chương trình.

+Tăng tốc độ thực thi: Khi hàm ngắn và được gọi nhiều lần, việc sử dụng inline có thể tăng tốc độ thực thi của chương trình.

Rủi ro:

+Tăng kích thước mã: Việc nhúng mã nguồn của hàm vào các điểm gọi của nó có thể làm tăng kích thước mã của chương trình, đặc biệt là khi hàm có thân hàm lớn.

+Tăng thời gian biên dịch: Việc sử dụng từ khóa inline có thể làm tăng thời gian biên dịch của chương trình, vì mã nguồn của hàm được nhúng vào nhiều điểm gọi khác nhau.

+Mất ý kiến: Mặc dù trình biên dịch có thể không thực sự nhúng mã nguồn vào các điểm gọi của hàm, nhưng từ khóa inline vẫn có thể mất đi ý kiến của lập trình viên về việc tối ưu hóa mã.

```
#include <iostream>

// Định nghĩa hàm inline
inline int square(int x) {
    return x * x;
}

int main() {
    int num = 5;

    // Gọi hàm square bằng cách sử dụng từ khóa inline
    int result = square(num);

    std::cout << "Bình phương của " << num << " là: " << result << std::endl;

    return 0;
}
```

Section 3

Return value, reference and address

Agenda

- Return value
- Return reference
- Return address

Return value

- Return by value is the simplest and safest return type to use. When a value is returned by value, a copy of that value is returned to the caller.
- As with pass by value, you can return by value literals (e.g. 5), variables (e.g. x), or expressions (e.g. x+1)

```
int doubleValue(int x)
{
    int value{ x * 2 };
    return value; // A copy of value will be returned here
} // value goes out of scope here
```

Giá trị trả về
+Trả về theo giá trị là loại trả về đơn giản và an toàn nhất để sử dụng. Khi một giá trị được trả về theo giá trị, một bản sao của giá trị đó được trả về cho người gọi.
+Tương tự như truyền theo giá trị, bạn có thể trả về bằng giá trị của hằng số (ví dụ: 5), biến (ví dụ: x), hoặc biểu thức (ví dụ: x+1).

Return address

- Returning by address involves returning the address of a variable to the caller..
- Because return by address just copies an address from the function to the caller, return by address is fast.
- However, return by address has one additional downside that return by value doesn't -- if you try to return the address of a variable local to the function, your program will exhibit undefined behaviour.

```
int* doubleValue(int x)
{
    int value{ x * 2 };
    return &value; // return value by address here
} // value destroyed here
```

không nên return địa chỉ biến local vì biến local chỉ tồn tại trong phạm vi 1 hàm

+Trả về theo địa chỉ liên quan đến việc trả về địa chỉ của một biến cho người gọi.
+Bởi vì trả về theo địa chỉ chỉ sao chép một địa chỉ từ hàm đến người gọi, trả về theo địa chỉ là nhanh chóng.
+Tuy nhiên, trả về theo địa chỉ có một nhược điểm bổ sung mà trả về theo giá trị không có - nếu bạn cố gắng trả về địa chỉ của một biến cục bộ trong hàm, chương trình của bạn sẽ thể hiện hành vi không xác định.

Return reference

- When a variable is returned by reference, a reference to the variable is passed back to the caller.
- The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.
- However, just like return by address, you should not return local variables by reference. Consider the following example.

```
int& doubleValue(int x)
{
    int value{ x * 2 };
    return value; // return a reference to value here
} // value is destroyed here
```

+Khi một biến được trả về bằng tham chiếu, một tham chiếu đến biến được chuyển trở lại cho người gọi.
+ Người gọi sau đó có thể sử dụng tham chiếu này để tiếp tục sửa đổi biến, điều này có thể hữu ích đôi khi. Trả về bằng tham chiếu cũng nhanh chóng, điều này có thể hữu ích khi trả về các cấu trúc và lớp.
+Tuy nhiên, giống như trả về bằng địa chỉ, bạn không nên trả về các biến cục bộ bằng tham chiếu.

trả về địa chỉ hay tham chiếu đều không được trả về biến local

xét trả về một địa chỉ khi bạn muốn trả về một con trỏ đến một biến hoặc cấu trúc được cấp phát động trong hàm và cần tiếp tục sử dụng dữ liệu sau khi hàm kết thúc. - Do cấp phát động vào bộ nhớ heap nên không bị xóa sau khi kết thúc hàm trừ khi dùng delete

xét trả về một tham chiếu khi bạn muốn trả về một biến được lưu trữ ở ngoài hàm và bạn muốn người gọi hàm có thể sửa đổi trực tiếp giá trị của biến đó.

Section 4

Function Pointers

Function Pointer là các biến chứa địa chỉ các hàm (địa chỉ bắt đầu của hàm- được lưu trong text- segment). chúng cho phép truy cập và thực thi các hàm thông qua địa chỉ được lưu trữ trong biến

Function Pointer được xem là 1 cách gọi hàm chứ không phải 1 kiểu trả về

khi nào sử dụng???

con trỏ hàm được sử dụng để truyền hàm A,A1,A2,... vào hàm B:
+ các hàm A,A1,A2,... được khai báo và định nghĩa trước, chúng thường có chung danh sách tham số
+ hàm B là hàm sử dụng hàm A hoặc A1 hoặc A2,... trong kiến trúc của nó

Function Pointers

- Function pointer is similar a pointer, except that instead of pointing to variables, it point to functions
- Definition a function pointer

+Con trỏ hàm tương tự như một con trỏ, ngoại trừ việc thay vì trỏ đến biến, nó trỏ đến hàm.
+Định nghĩa một con trỏ hàm:

```
kieu_du_lieu (*Ten_con_tro)(danh_sach_tham_so);
```

- How to use function pointer

```
int foo();
double goo();
int hoo(int x);

int (*fcnPtr1)() = foo; // dung
int (*fcnPtr2)() = goo; // sai - kieu tra ve khong khop
double (*fcnPtr4)() = goo; // dung
fcnPtr1 = hoo; // sai - fcnPtr1 khong co doi so, hoo co tham so
int (*fcnPtr3)(int) = hoo; // dung
```

```
void myFunction() {
    cout << "This is my function" << endl;
}

int main() {
    // Khai báo một con trỏ hàm trỏ đến hàm myFunction
    void (*functionPtr)() = &myFunction;

    // Gọi hàm thông qua con trỏ hàm
    (*functionPtr)(); // Output: This is my function
    return 0;
}
```

```
int* returnPointer() {
    int* ptr = new int(5); // Cấp phát động một biến nguyên và trả về con trỏ đến nó
    return ptr;
}

int main() {
    int* ptr = returnPointer(); // Gọi hàm và nhận con trỏ trả về
    cout << *ptr << endl; // Output: 5
    delete ptr; // Giải phóng vùng nhớ
    return 0;
}
```

Function Pointers

- Using function pointer like a parameter of other function

```
// tao mot ham voi con tro ham la tham so
kieu_tra_ve Ten_ham_1(kieu_du_lieu(*Ten_con_tro)(danh_sach_tham_so))
// goi ham do voi ham 2 la mot doi so
Ten_ham_1(Ten_ham_2);
```

Chú ý

tạo 1 hàm với con trỏ hàm là tham số

<kiểu trả về> <tên hàm 1> (<kiểu dữ liệu của hàm làm tham số> (*tên con trỏ đại diện cho cái tên hàm) (tham số))

Function Pointers

```
// Note our user-defined comparison is the third parameter
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // bestIndex is the index of the smallest/largest element we've encountered so far.
        int bestIndex{ startIndex };

        // Look for smallest/largest element remaining in the array (starting at startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller/larger than our previously found smallest
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // COMPARISON DONE HERE
            {
                // This is the new smallest/largest number for this iteration
                bestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest/largest element
        std::swap(array[startIndex], array[bestIndex]);
    }
}
```

```
// Here is a comparison function that sorts in ascending order
// (Note: it's exactly the same as the previous ascending() function)
bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}

// Here is a comparison function that sorts in descending order
bool descending(int x, int y)
{
    return x < y; // swap if the second element is greater than the first
}

// This function prints out the values in the array
void printArray(int *array, int size)
{
    for (int index{ 0 }; index < size; ++index)
    {
        std::cout << array[index] << ' ';
    }

    std::cout << '\n';
}
```

```
int main()
{
    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    // Sort the array in descending order using the descending() function
    selectionSort(array, 9, descending);
    printArray(array, 9);

    // Sort the array in ascending order using the ascending() function
    selectionSort(array, 9, ascending);
    printArray(array, 9);

    return 0;
}
```

chỉ cần truyền tên hàm <=> truyền con trỏ hàm

References

- https://www.tutorialspoint.com
- https://www.learncpp.com/

Lesson Summary

- Function
- Pass to function
- Return function
- Function pointer

Thank you