10.5 —按值，引用和地址返回值

作者：亞歷克斯（ALEX），2008年2月25日| 由NASCARDRIVER於2021年2月5日最後修改

在前面的三課中，您學習瞭如何通過值，引用和地址將參數傳遞給函數。在本節中，我們將考慮通過所有三種方法將值返回給調用方的問題。

事實證明，按值，地址或引用將函數的值返回給調用方的方法幾乎與將參數傳遞給函數的方法完全相同。每種方法都具有相同的優點和缺點。兩者之間的主要區別只是數據流的方向相反。但是，還增加了一點複雜性-因為函數中的局部變量超出範圍並在函數返回時被破壞，所以我們需要考慮此操作對每種返回類型的影響。

按價值回報

按值返回是最簡單，最安全的返回類型。當按值返回值時，該值的副本將返回給調用方。與按值傳遞一樣，您可以按值返回字面量（例如5），變量（例如x）或表達式（例如x + 1），這使得按值返回非常靈活。

按值返回的另一個優點是，您可以返回涉及在函數中聲明的局部變量的變量（或表達式），而不必擔心範圍問題。因為變量是在函數返回之前求值的，並且值的副本已返回給調用者，所以當函數的變量在函數末尾超出範圍時，就沒有問題。

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

當返回在函數內部聲明的變量或返回通過值傳遞的函數參數時，按值返回最合適。但是，像按值傳遞一樣，對於結構和大型類，按值返回很慢。

何時使用按值返回：

- 返回在函數內部聲明的變量時
- 返回按值傳遞的函數參數時

什麼時候不使用按值返回：

- 返回內置數組或指針時（使用按地址返回）
- 返回大型結構或類時（使用按引用返回）

按地址返回

按地址返回涉及將變量的地址返回給調用方。與按地址傳遞類似，按地址返回只能返回變量的地址，而不能返回文字或表達式（沒有地址）。由於按地址返回只是將地址從函數複製到調用者，因此按地址返回速度很快。

但是，按地址返回還有一個缺點，即按值返回沒有-如果您嘗試返回該函數局部變量的地址，則程序將表現出未定義的行為。考慮以下示例：

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

如您在此處看到的，值在返回給調用者之後就被銷毀了。最終結果是，調用方以未分配的內存（懸空的指針）結尾，如果使用該地址將導致問題。這是新程序員經常犯的錯誤。如果程序員嘗試按地址返回局部變量，許多較新的編譯器會發出警告（不是錯誤）-但是，有很多方法可以欺騙編譯器，使您在不生成警告的情況下進行非法操作，因此確保程序返回的指針將在函數返回後指向有效變量是程序員的重擔。

按地址返回通常用於將動態分配的內存返回給調用方：

```
int* allocateArray(int size)
{
    return new int[size];
}

```

```
int main()
{
    int *array{ allocateArray(25) };

    // do stuff with array

    delete[] array;
    return 0;
}
```

之所以可行，是因為動態分配的內存在聲明它的塊的末尾不會超出範圍，因此當地址返回給調用者時，內存仍將存在。跟踪手動分配可能很困難。將分配和刪除分為不同的功能，這使得了解誰負責刪除資源或根本是否需要刪除資源變得更加困難。應該使用智能指針（稍後介紹）和在其自身之後清理的類型，而不是手動分配。

何時使用按地址返回：

- 返回動態分配的內存時，您不能使用為您處理分配的類型
- 返回通過地址傳遞的函數參數時

什麼時候不使用按地址返回：

- 當返回在函數內部聲明的變量或按值傳遞的參數時（使用按值返回）
- 返回通過引用傳遞的大型結構或類時（使用按引用返回）

參考退貨

與按地址返回類似，按引用返回的值必須是變量（您不應返回對文字或解析為臨時值的表達式的引用，因為它們將超出函數結尾的範圍，您將最終返回一個懸空的參考）。通過引用返回變量時，對該變量的引用將傳遞回調用方。然後，調用者可以使用該引用來繼續修改變量，這有時會很有用。通過引用返回也很快，這在返回結構和類時很有用。

但是，就像按地址返回一樣，您不應該通過引用返回局部變量。考慮以下示例：

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

在上面的程序中，程序將返回對值的引用，該值將在函數返回時銷毀。這將意味著調用方收到對垃圾的引用。幸運的是，如果您嘗試這樣做，編譯器可能會給您警告或錯誤。

按引用返回通常用於將通過引用傳遞給函數的參數返回給調用者。在下面的示例中，我們返回（通過引用）通過引用傳遞給函數的數組元素：

```
#include <array>
#include <iostream>

// Returns a reference to the index element of array
int& getElement(std::array<int, 25>& array, int index)
{
    // we know that array[index] will not be destroyed when we return to the caller (since t
he caller passed in the array in the first place!)
    // so it's okay to return it by reference
    return array[index];
}

int main()
{
    std::array<int, 25> array;

    // Set the element of array with index 10 to the value 5
    getElement(array, 10) = 5;

    std::cout << array[10] << '\n';

    return 0;
}
```

打印：

```
5
```

當我們調用時`getElement(array, 10)`，getElement（）返回對索引為10的數組元素的引用。main（）然後使用該引用為該元素分配值5。

儘管這是一個人為的示例（因為您可以直接訪問array [10]），但是一旦了解了類，您將發現更多用於通過引用返回值的用法。

何時使用參考歸還：

- 返回參考參數時
- 返回通過引用或地址傳遞到函數中的對象的成員時
- 當返回一個大型結構或類時，該結構或類在函數結束時不會被銷毀（例如，通過引用傳入的結構或類）

什麼時候不使用引用返回：

- 當返回在函數內部聲明的變量或按值傳遞的參數時（使用按值返回）
- 返回內置數組或指針值時（使用按地址返回）

混合返回引用和值

儘管函數可以返回值或引用，但調用方可以也可以不將結果分配給變量或引用。讓我們看看當我們混合使用值和引用類型時會發生什麼。

```
int returnByValue()
{
    return 5;
}

int& returnByReference()
{
     static int x{ 5 }; // static ensures x isn't destroyed when the function ends
     return x;
}

int main()
{
    int giana{ returnByReference() }; // case A -- ok, treated as return by value
    int& ref{ returnByValue() }; // case B -- compile error since the value is an r-value, and an r-value can't bind to a non-const reference
    const int& cref{ returnByValue() }; // case C -- ok, the lifetime of the return value is extended to the lifetime of cref

    return 0;
}
```

在情況A中，我們正在將參考返回值分配給非參考變量。由於giana不是引用，因此將返回值複製到giana中，就像returnByReference（）已按值返回一樣。

在情況B中，我們嘗試使用returnByValue（）返回的返回值的副本來初始化引用ref。但是，由於返回的值沒有地址（它是r值），因此將導致編譯錯誤。

在情況C中，我們嘗試使用returnByValue（）返回的返回值的副本來初始化const引用cref。因為const引用可以綁定到r值，所以這裡沒有問題。通常，r值在創建它們的表達式的末尾過期-但是，當綁定到const引用時，r值的生存期（在這種情況下，該函數的返回值）將延長為匹配引用的生存期（在這種情況下為cref）

終身擴展不會保存懸空的引用

考慮以下程序：

```
const int& returnByReference()
{
    return 5;
}

int main()
```

```
{
    const int& ref { returnByReference() }; // runtime error
}
```

在上面的程序中，returnByReference（）返回的是對值的const引用，該值將在函數結束時超出範圍。通常這是一個禁忌，因為它將導致懸掛的參考。但是，我們也知道將值分配給const引用可以延長該值的壽命。那麼，這裡優先考慮什麼呢？5是否先超出範圍，還是ref延長5的壽命？

答案是5首先超出範圍，然後將對5的引用複制回調用方，然後ref延長當前懸而未決的引用的生存期。

但是，以下確實可以正常工作：

```
const int returnByValue()
{
     return 5;
}

int main()
{
    const int& ref { returnByValue() }; // ok, we're extending the lifetime of the copy passed back to main
}
```

在這種情況下，首先將字面值5複製回調用方（主）的作用域，然後ref延長該副本的生命週期。

返回多個值

C ++不包含將多個值傳遞回調用者的直接方法。儘管有時可以以可以分別傳遞每個數據項的方式來重組代碼（例如，不是讓一個函數返回兩個值，而是讓兩個函數每個返回一個值），但這可能既麻煩又不直觀。

幸運的是，可以使用幾種間接方法。

如課程**10.3-通過引用傳遞參數中所述**，out參數提供了一種將多個數據位傳遞回調用方的方法。我們不建議使用此方法。

第二種方法涉及使用僅數據結構：

```
#include <iostream>

struct S
{
    int m_x;
    double m_y;
};

S returnStruct()
{
    S s;
    s.m_x = 5;
    s.m_y = 6.7;
    return s;
}

int main()
{
    S s{ returnStruct() };
    std::cout << s.m_x << ' ' << s.m_y << '\n';

    return 0;
}
```

第三種方法（在C ++ 11中引入）是使用std :: tuple。元組是一系列可能是不同類型的元素，其中每個元素的類型必須明確指定。

這是一個返回元組並使用std :: get的示例 得到元組的第n個元素：

```
#include <tuple>
#include <iostream>

std::tuple<int, double> returnTuple() // return a tuple that contains an int and a double
{
```

```
    return { 5, 6.7 };
}

int main()
{
    std::tuple s{ returnTuple() }; // get our tuple
    std::cout << std::get<0>(s) << ' ' << std::get<1>(s) << '\n'; // use std::get<n> to get the nth element of the tuple

    return 0;
}
```

這與前面的示例相同。

您還可以使用std :: tie將元組解壓縮為預定義的變量，如下所示：

```
#include <tuple>
#include <iostream>

std::tuple<int, double> returnTuple() // return a tuple that contains an int and a double
{
    return { 5, 6.7 };
}

int main()
{
    int a;
    double b;
    std::tie(a, b) = returnTuple(); // put elements of tuple in variables a and b
    std::cout << a << ' ' << b << '\n';

    return 0;
}
```

從C ++ 17開始，結構化綁定聲明可用於簡化將多個返回值拆分為單獨的變量的操作：

```
#include <tuple>
#include <iostream>

std::tuple<int, double> returnTuple() // return a tuple that contains an int and a double
{
    return { 5, 6.7 };
}

int main()
{
    auto [a, b]{ returnTuple() }; // used structured binding declaration to put results of tuple in variables a and b
    std::cout << a << ' ' << b << '\n';

    return 0;
}
```

如果要在多個位置使用結構，則使用結構比使用元組更好。但是，對於僅打包這些值以返回並且不會從定義新結構重複使用的情況，元組會更乾淨一些，因為它不會引入新的用戶定義數據類型。

結論

在大多數情況下，按價值回報將足以滿足您的需求。這也是將信息返回給呼叫者的最靈活，最安全的方法。但是，按引用或地址返回也很有用，特別是在使用動態分配的類或結構時。使用按引用返回或地址返回時，請確保您未返回對變量的引用或地址，該變量將在函數返回時超出範圍！

小測驗時間

為以下每個函數編寫函數原型。使用最合適的參數和返回類型（按值，按地址或按引用），包括在適當的地方使用const。

1）一個名為sumTo（）的函數，該函數帶有一個整數參數，並返回介於1和輸入數字之間的所有數字的和。

顯示解決方案

2）一個名為printEmployeeName（）的函數，它使用Employee結構作為輸入。

顯示解決方案

3）名為minmax（）的函數，該函數接受兩個整數作為輸入，並將a中較小和較大的數字返回給調用方`std::pair`。一個`std::pair`與a相同的作品，`std::tuple`但恰好存儲了兩個元素。

顯示解決方案

4）名為getIndexOfLargestValue（）的函數採用整數數組（作為`std::vector`），並返回數組中最大元素的索引。

顯示解決方案

5）名為getElement（）的函數，該函數接受`std::string`（作為`std::vector`）的數組和一個索引，並在該索引處返回數組元素（而不是副本）。假設索引是有效的，並且返回值是const。

顯示解決方案

10.6-內聯函數

指數

10.4-通過地址傳遞參數

C ++教程 | 打印這篇文章

301註釋到10.5 —按值，引用和地址返回值

«較舊的評論 1 2 3 4

貝里
2021年2月4日，上午11:21 ·回复

反饋的次要點：

對於問題2，函數參數為const Employee＆employee。
＆在變量旁邊。對於其他問題，＆旁邊的參數類型。

那是不一致的。保持＆旁邊的類型。根據9.16的最佳做法。

J34NP3T3R
2021年1月23日，下午5:10 ·回复

```
1  int& doubleValue(int x)
2  {
3      int value{ x * 2 };
4      return value; // return a reference to value here  ( it seems we are returning a norma
   l variable ? )
5  } // value is destroyed here
```