Contecnt

Version

Version	Author	Date	Changes
0.1	Jin Feng	2020.01.31	Initial Draft
0.8	Jin Feng	2020.01.31	Add array type and comparasion of funcName and &funcName
0.9	Jin Feng	2020.02.16	Post on personal blog

Overview

It is useful to understand the *Types, Identifiers, Declarations, and Definitions* firstly. Please see article ***Types, Identifiers, Declarations, and Defintions in C++*** . In this article, The array type and poiter will be introduced for comparing with function type.

Array

Array Type

Think of *array type* as a ordinary elementary type.

```
// array type
int [3];

// array variable declaration
// usual form
int a[3];
// canonical form
int [3] a;

// array pointer type
// usual form
int (*) [3]
// canonical form
int [3] *

// array pointer varaible
// usual form
int (*p) [3];
// canonical form
int [3] * p
```

Array Pointer

It is worthwhile to distinguish ***arrayName*** and ***&arrayName***. Let us assume *arr* with the definition of "int arr[3];", thus

- The type of arr is "int [3]", it is array type.
- The type of &arr is "int (*)[3]", it is array pointer type.
- The "int [3]" can be converted to "int*" while in function's parameter, assignning to a pointer of int, arr+1*
- "arr+1" is different from "&arr+1", the former is the address of next array element, the later is just the addres after the last array element.

Concret coding bellow can demonstrate our conclusions above

```
#include <stdio.h>
#include <iostream>
#include <boost/type_index.hpp>

using boost::typeindex::type_id_with_cvr;

void func(int arr[3])
{
        std::cout<<"func(arr) is "<<type_id_with_cvr<decltype(arr)>().pretty_name()<<" (
                typeid(arr).name()<<std::endl;
}

int main()
{
        int arr[3] = {1,2,3};
        int (*arrPtr)[3];
        arrPtr = & arr;
        std::cout<<"arr is "<<type_id_with_cvr<decltype(arr)>().pretty_name()<<" or "<<
                typeid(arr).name()<<std::endl;
        std::cout<<"*arr is "<<type_id_with_cvr<decltype(*arr)>().pretty_name()<<" or "<
                typeid(*arr).name()<<std::endl;
        // ***arr does not exist
        //std::cout<<"**arr is "<<type_id_with_cvr<decltype(**arr)>().pretty_name()<<" (
                //typeid(**arr).name()<<std::endl;
        std::cout<<"&arr is "<<type_id_with_cvr<decltype(&arr)>().pretty_name()<<" or "<
                typeid(&arr).name()<<std::endl;
        std::cout<<"*&arr is "<<type_id_with_cvr<decltype(*&arr)>().pretty_name()<<" or
                typeid(*&arr).name()<<std::endl;
        std::cout<<"**&arr is "<<type_id_with_cvr<decltype(**&arr)>().pretty_name()<<" (
                typeid(**&arr).name()<<std::endl;
        // ***&arr does not exist
        //std::cout<<"***&arr is "<<type_id_with_cvr<decltype(***&arr)>().pretty_name()<
                //typeid(***&arr).name()<<std::endl;

        printf("value/addr(arr):  %d/%p, value/addr(arr+1): %d/%p\n", *arr, arr, *(arr+1
        printf("value/addr(&arr):  %d/%p, value/addr(&arr+1): %d/%p\n", *&arr, &arr, *(&
        printf("value/addr(arrPtr):  %d/%p, value/addr(arrPtr+1): %d/%p\n", *arrPtr, &ar

   func(arr);
   //funcArray(arrPtr);
        int *intPtr = arr;
        //intPtr = arrPtr;

        return 0;
}
```

The output the codes is

```
arr is int [3] or A3_i
*arr is int& or i
&arr is int (*) [3] or PA3_i
*&arr is int (&) [3] or A3_i
**&arr is int& or i
value/addr(arr):  1/0x7ffee8c26908, value/addr(arr+1): 2/0x7ffee8c2690c
value/addr(&arr):  -389912312/0x7ffee8c26908, value/addr(&arr+1): -389912300/0x7ffee8c26
value/addr(arrPtr):  -389912312/0x7ffee8c26880, value/addr(arrPtr+1): -389912300/0x7ffe
func(arr) is int* or Pi
```

Function

Function Type

Think of *function type* as a ordinary elementary type.

```
// function type
void(int, double);

// function variable declaration
// usual form
void someFunc(int, double);
// canonical form
void(int, double) someFunc;

// function pointer type
// usual form
void (*)(int, double);
// canonical form
void(int, double)*

// function pointer varaible declaration
// usual form
void (*sfp)(int, double);
void (*)(int, double) sfp;
// canonical form
void (int, double)* sfp;

// function reference type
// usual form
void (&)(int, double);
// canonical form
void (int, double)&;

// function reference variable
// usual form
void (&sfr)(int, double);
void (&)(int, double) sfr;
// canonical form
void(int,double)& sfr;
```

Function Pointer

C does not have function objects or lambda expressions, pointers to functions are widely used as function arguments in C-style code. Dereferecing a pointer to function using * is optional, similarly, using & to get the address of a function is optional.

It is worthwhile to distinguish the ***FunctionName*** and ***&FunctionName***. Let's asuume *func* with definition of "*void func() {std::cout<<"hello word\n";}*", thus

- The type of func is "void ( )", it is function type.
- The type of &func is "void (*)( )", it is pointer type pointed to function type

- The type of *func, **func, ... is equal to func
- The type of *&func, **&func, ... is equal to func
- The effects of these callings are equal:

```
func() == (&func)() // functionName() == fuctionPointer()
== (*func)() == (**func)() == (*&func)() == (**&func)()
```

Concret coding bellow can demonstrate our conclusions above

```
#include <stdio.h>
#include <iostream>
#include <boost/core/demangle.hpp>
#include <boost/type_index.hpp>

using boost::typeindex::type_id_with_cvr;

void func() {std::cout<<"hello word\n";}

void my_int_func(int x)
{
        printf( "%d\n", x );
}

int main()
{
        // functionName v.s. &functionName
        std::cout<<"\nfunctionName v.s. &functionName"<<std::endl;
        std::cout<<"======================================"<<std::endl;

        std::cout<<"func is "<<type_id_with_cvr<decltype(func)>().pretty_name()<<" or "<
                typeid(func).name()<<std::endl;
        std::cout<<"*func is "<<type_id_with_cvr<decltype(*func)>().pretty_name()<<" or
                typeid(*func).name()<<std::endl;
        std::cout<<"**func is "<<type_id_with_cvr<decltype(**func)>().pretty_name()<<" o
                typeid(**func).name()<<std::endl;
        std::cout<<"&func is "<<type_id_with_cvr<decltype(&func)>().pretty_name()<<" or
                typeid(&func).name()<<std::endl;
        std::cout<<"*&func is "<<type_id_with_cvr<decltype(*&func)>().pretty_name()<<" o
                typeid(*&func).name()<<std::endl;
        std::cout<<"**&func is "<<type_id_with_cvr<decltype(*&func)>().pretty_name()<<"
                typeid(**&func).name()<<std::endl;

        printf("The address of func is %p\n", func);
        printf("The address of &func %p\n", &func);
        std::cout<<"Call func(): "; func();
        std::cout<<"Call (*func)(): "; (*func)();
        std::cout<<"Call (**func)(): "; (**func)();
        std::cout<<"Call (&func)(): "; (&func)();
        std::cout<<"Call (*(&func))(): "; (*(&func))();
        std::cout<<"Call (**(&func))(): "; (**(&func))();

        std::cout<<"\n Miscs "<<std::endl;
        std::cout<<"======================================"<<std::endl;
        void (*foo)(int);
        void (*foo1)(int);
        void foo2(int);

        foo = &my_int_func;
        foo1 = my_int_func;
        //foo2 = &my_int_func;
```

```
        /* call my_int_func (note that you do not need to write (*foo)(2) ) */
        std::cout<<"Call foo(2): ";
        foo(2);
        /* but if you want to, you may */
        std::cout<<"Call (*foo)(2): ";
        (*foo)( 2 );

        std::cout<<"Call foo1(3): ";
        foo1(3);
        std::cout<<"Call (*foo1)(3): ";
        (*foo1)(3);

        return 0;
}
```

The output the codes is

```
functionName v.s. &functionName
====================================
func is void () or FvvE
*func is void (&)() or FvvE // something wrong with type_id_with_cvr ?
**func is void (&)() or FvvE // something wrong with type_id_with_cvr ?
&func is void (*)() or PFvvE
*&func is void (&)() or FvvE  // something wrong with type_id_with_cvr ?
**&func is void (&)() or FvvE // something wrong with type_id_with_cvr ?
The address of func is 0x101e59730
The address of &func 0x101e59730
Call func(): hello word
Call (*func)(): hello word
Call (**func)(): hello word
Call (&func)(): hello word
Call (*(&func))(): hello word
Call (**(&func))(): hello word

 Miscs
====================================
Call foo(2): 2
Call (*foo)(2): 2
Call foo1(3): 3
Call (*foo1)(3): 3
```

Conclusion

Things to remember

Array Type

- Take array type as ordinary elementary type
- Array type is different from array pointer type
- The type of array name is array type
- The type of addresing array name is array pointer type
- Array type can be converted to associated element's type pointer
- Increasing/decreasing variable with array type is totaly different from the varaible with array pointer type

Function Type

- Take function type as ordinary elementary type
- Function type is different from function pointer type
- The type of function name is ordinay function type
- The type of addressing function name is function pointer type
- The style of functionPointer assignment
  - For logical clarity, &functionName is encouraged to assign to functionPointer
    - functionPointer = functionName // optional
    - functionPointer = &functionName // encouraged
  - For simplifying code, functionName is encouraged to assign to functionPointer
    - functionPointer = functionName // encouraged
    - functionPointer = &functionName // optional
- The style of calling functionPointer
  - For logical clarity, (*functionPointer)() is encouraged to use
    - functionPointer() // optional
    - (*functionPointer)() // encouraged
  - For simplifying code, functionPointer() is encouraged to use
    - functionPointer() // encouraged
    - (*functionPointer)() // optional

Appendix

- This article's source code and c++ source code: https://gitee.com/emmix/languages/tree/master/cpp/arrayTypeAndFunctionType

References

1. http://www.newty.de/fpt/zip/e_fpt.pdf

2. <The C++ Programming Language> Forth Edition, Chapter 12.5
3. typesIdentifiersDeclarationsAndDefintitionsInCpp.pdf
4. http://blog.csdn.net/qq575787460/article/details/8531397
5. http://www.umich.edu/~eecs381/handouts/bind.pdf
6. https://thenewcpp.wordpress.com/2012/04/25/deprecated-binders-and-adaptors/
7. http://rastergrid.com/blog/2010/02/one-more-degree-of-freedom-for-c/
8. http://tipsandtricks.runicsoft.com/Cpp/MemberFunctionPointers.html
9. http://www.newty.de/fpt/zip/e_fpt.pdf