【C++】Day17(2)

Class	C++
Date	@December 7, 2021
Material	
Series Number	
Summary	

【Ch6】Return Statement

6.3 Return Types and the return Statement

6.3.2 Functions that Return a Value

Never Return a Reference or Pointer to a Local Object

When a function completes, its storage is freed. After a function terminates, references to local objects refer to memory that is no longer valid:

```
//disaster: this function returns a reference to a local object
const string &manip() {
  string ret;
  if(!ret.empty())
    return ret; //Wrong: returning a reference to a local object
  else
    return "Empty"; //Wrong: "Empty" is a local temporary object.
}
```

Tip: One good way to ensure that the return is safe is to ask: To what preexisting object is the reference referring?

Reference Returns are Lvalues

Whether a function call is an lvalue depends on the return type of the functions. Calls to functions that return references are lvalues; other return types yield rvalues.

```
char &get_val(string str, string::size_type ix) {
  return str[ix];
}

int main() {
  string s("a value");
  cout << s << endl; //Print "a value"
  get_val(s, 0) = 'A';
  cout << s << endl; //Print "A value"
  return 0;
}
```

If the return type is a reference to const, then (as usual) we may not assign to the result of the call.

List Initializing the Return Value

Functions can return a braced list of values. As in any other return, the list is used to initialize the temporary that represents the function's return.

If the list is empty, that temporary is value initialized. Otherwise, the value of the return depends on the function's return type.

Return from main

There is one exception to the rule that a function with a return type other than void must return a value: The main function is allowed to terminate without a return.

If control reaches the end of main and there is no return, then the compiler implicitly inserts a return of 0.

A zero indicates success; most other values indicate failure.

Exercise

Exercise 6.35: In the call to `fact`, why did we pass `val - 1` rather than `val--`?

The order of evaluation is undefined for the operator *, the results can be:

```
//Version 1
auto temp = fact(val) * val;
--val;
return temp;

//Version 2
auto temp = fact(val);
--val;
return temp * val;
```

6.3.3 Returning a Pointer to an Array

Because we cannot copy an array, a function cannot return an array. However, a function can return a pointer or a reference to an array.

Unfortunately, the syntax used to define functions that return pointers or reference to arrays can be intimidating. Fortunately, there are ways to simplify such declarations. The most straightforward way is to use a type alias:

```
typedef int arrT[10]; //arrT is a synonym for the type array of ten ints
using arrT = int [10]; //equivalent declaration of arrT
arrT* func(int i); //func returns a pointer to an array of five ints
```

Declaring a Function That Returns a Pointer to an Array

To declare func without using a type alias, we must remember that the dimension of an array follows the name being defined:

```
int arr[10]; //arr is an array of ten ints
int *p1[10]; //p1 is an array of ten points;
int (*p2)[10] = &arr; //p2 points to an array of ten ints.
```

As with these declarations, if we want to define a function that returns a pointer to an array, the dimension must follow the function's name.

The form of a function that returns a pointer to an array is:

```
Type (*function (parameter_list))[dimension]
```

As an example, the following declares func without using a type alias:

```
int (*func(int i))[10];
```

Using a Trailing Return Type

Another way to simplify the declaration of func is by using a trailing return type.

Trailing returns can be defined for any function, but are most useful for functions with complicated return types, such as pointers(or references) to arrays. A trailing return type follows the parameter list and is preceded by ->.

To signal that the return follows the parameter list, we use auto where the return type ordinarily appears:

```
//func takes an int argument and returns a pointer to an array of ten ints
auto func(int i) -> int (*)[10];
```

Using decltype

As another alternative, if we know the arrays to which our function can return a pointer, we can use `decltype` to declare the return type.

For example, the following function returns a pointer to one of two arrays, depending on the value of its parameter:

```
int odd[] = {1, 3, 5, 7, 9};
int even[] = {2, 4, 6, 8, 10};
//returns a pointer to an array of five int elements
decltype(odd) *arrPtr(int i) {
  return (i % 2)  ? &odd : &even;
}
```

The tricky part is that we must remember that decltype does not automatically convert an array to its corresponding pointer type. The type returned by decltype is an array type, to which we must add a * to indicate that arrPtr returns a pointer.

Exercise

Exercises Section 6.3.3

Exercise 6.36: Write the declaration for a function that returns a reference to an array of ten `string`s, without using either a trailing return, `decltype`, or a type alias.

Exercise 6.37: Write three additional declarations for the function in the previous exercise. One should use a type alias, one should use a trailing return, and the third should use `decltype`. Which form do you prefer and why?

Exercise 6.38: Revise the `arrPtr` function on to return a reference to the array.

```
#include <string>
using std::string;

string (&foo1())[10];

using arr_str_type = string[10];
//typedef string arr_str_type[10];
arr_str_type &foo2();

auto foo3() -> string (&)[10];

string str[10] = {};
decltype(str) &foo4();

// I prefer the trailing return form, because it's much easier to understand.
// I also prefer the type alias form, becasue it's easy to use, especially when
// the type being used many times.

string (&foo1())[10] {
  return str;
}

arr_str_type &foo2() {
  return str;
}

auto foo3() -> string (&)[10] {
```

```
  return str;
}

decltype(str) &foo4() {
  return str;
}

int main() {
  foo1();
  foo2();
  foo3();
  foo4();

  return 0;
}
```