【C++】Day16(3)

Class	C++
Date	@December 6, 2021
Material	
Series Number	
Summary	

【Ch6】Function

6.2.4 Array Parameters

Array have two special properties that affect how we define and use functions that operate on arrays: We cannot copy an array, and when we use an array it is usually converted to a pointer.

See the following code for an example:

```
void point(const int*);
void point(const int[]);
void point(const int[10]); //dimension for documentation pruposes
```

Regardless of appearances, these declarations are equivalent: Each declares a function with a single parameter of type const int*. When the compiler checks a call to print, it checks only that the argument has type const int*:

```
int i = 0, j[2] = {0, 1};
print(&i); //ok: &i is an int*
print(j); //ok: j is converted to an int* that points to j[0]
```

Warning: As with any code that uses arrays, functions that take array parameters must ensure that all uses of the array stay within the array bounds.

Because arrays are passed as pointers, functions ordinarily don't know the size of the array they are given. They must rely on additional information provided by the caller. There are three common techniques used to mangae pointer parameters:

1. Using a Marker to Specify the Extent of an Array:

   The first approach to managing array arguments requires the array itself to contain an end marker. C-style character strings are an example of this approach.

   ```
   void ptrin(const char *cp) {
     if(cp) {
       while(*cp)
         cout << *cp++; //print *cp and then advance the pointer by 1
     }
   }
   ```

2. Using the Standard Library Conventions:

   A second technique used to manage array arguments is to pass pointers to the first and one past the last element in the array. This approach is inspired by techniques used in the standard library.

   ```
   void print(const int *beg, const int *end) {
     while(beg != end)
       cout << *beg++ << endl;
   }
   ```

   To call this function:

   ```
   int j[2] = {0, 1};
   print(begin(j), end(j));
   ```

3. Explicitly Passing a Size Parameter:

   A third approach for array arguments, which is common in C programs, is to define a second parameter that indicates the size of the array.

   ```
   void print(const int ia[], size_t size) {
     for(size_t i = 0; i != size; ++i)
       cout << ia[i] << endl;
   }
   ```

To call this function:

```
int j[2] = {0, 1};
print(j, end(j) - begin(j));
```

Note: The parentheses around &arr are necessary:

```
f(int &arr[10]) //errror: declares arr as an array of references
f(int (&arr)[10]) //ok: arr is a reference to an array of ten ints
```

Passing a Multidimensional Array

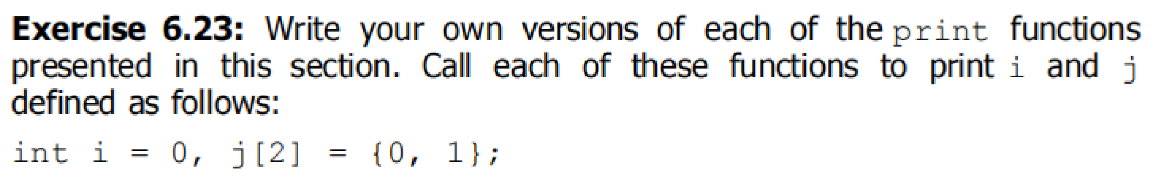
As with any array, a multidimensional array is passed as a pointer to its first element. Because we are dealing with an array of ararys. that element is an array, so the pointer is a pointer to an array. The size of the second dimension is part of the element type and must be specified

```
void print(int (*matrix)[10], int rowSize) {}
```

Note: Again, the parentheses around *matrix are necessary:

```
int *matrix[10];
int (*matrix)[10]; //pionter to an array of ten ints
```

Exercise:



Exercise 6.23: Write your own versions of each of the `print` functions presented in this section. Call each of these functions to print `i` and `j` defined as follows:

```
int i = 0, j[2] = {0, 1};
```

```
void print1(const int i, const int *begin, const int *end) {
  cout << i << endl;
  while(begin && begin != end)
    cout << *begin++ << endl;
}

void print2(const int i, const int arr[], const int size) {
  cout << i << endl;
  for(int i = 0; i != size; i++)
    cout << arr[i] << endl;
}

int main() {
  int i = 0, j[2] = {0, 1};
  print1(i, std::begin(j), std::end(j));
  print2(i, j, std::end(j) - std::begin(j));
  return 0;
}
```

6.2.4 main: Handling Command-Line Options

It turns out that main is a good example of how C++ programs pass arrays to functions.

Sometimes, we need to pass arguments to main. The most common use of arguments to main is to let the user specify a set of options to guide the operation of the program.

For example, assuming our main program is in an executable file named prog, we might pass options to the program as follows:

```
prog -d -o ofile data0
```

Such command-line options are passed to main in two parameters:

```
int main(int argc, char *argv[]) {}
```

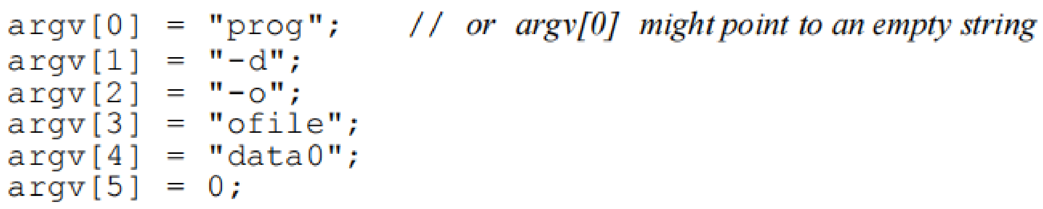
The second parameter, `argv`, is an array of pointers to C-style character strings.

The first parameter, `argc`, passes the number of strings in that array. Because the second parameter is an array, we might alternatively define main as

```
int main(int argc, char **argv) {}
```

indicating that argv points to a char*.

When arguments are passed to main, the first element in argv points either to the name of the program or to the empty string. Subsequent element pass the arguments provided on the command line. The element just past the last pointer is guaranteed to be 0.



```
argv[0] = "prog";    // or argv[0] might point to an empty string
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
```

Warning: When you use the arguments in argv, remember that the optional arguments begin in argv[1]; argv[0] contains the program's name, not user input.