

Chapter 3: Strings, Vectors and Arrays

2.5.3 decltype Type Specifier

```
int i;
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x has type const int
decltype (i) a; // a is an uninitialized int
decltype(cj) y = x; // y has type const int& and is bound to x
```

```
double f() {return 3.01;}
decltype(f()) sum = x;
// sum has whatever type f returns, double in this case
```

Quick Check: Assignment is an example of an expression that yields a reference type. The type is a reference to the type of the left-hand operand. That is, if `i` is an `int`, then the type of the expression `i = x` is `int&`. Using this knowledge, determine the type of `d` deduced from `decltype` statement

```
int a = 3, b = 4;
decltype(a = b) d = a;
```

A:

DeclEx.cpp

Q: What are the outputs?

```
#include <iostream>

using namespace std;

int main()
{
    int a = 3, b = 4;
    decltype(a) c = a;
    decltype(a = b) d = a; c++;
    d += 2;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    return 0;
}
```

A:

Remark: the `decltype` is very useful for template programming as we will realize later in the class.

(see ppt)

3.1 Namespace using Declaration

nameSpaceExample1.cpp

```
#include <iostream>

// using declarations for names from the standard library
using std::cin;
using std::cout;
using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;

    int v1, v2;
    cin >> v1 >> v2;

    cout << "The sum of " << v1
        << " and " << v2
        << " is " << v1 + v2 << endl;

    return 0;
}
```

Or using entire std namespace

nameSpaceExample2.cpp

```
#include <iostream>

// using declarations for the entire standard library
using namespace std;

int main()
{
    cout << "Enter two numbers:" << endl;

    int v1, v2;
    cin >> v1 >> v2;

    cout << "The sum of " << v1
        << " and " << v2
        << " is " << v1 + v2 << endl;

    return 0;
}
```

(see ppt)

3.2 Library string Type

string I/O

StringIO.cpp

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s;
    cin >> s;
    cout << s << endl;
    return 0;
}
```

Q: what are the outputs if we enter Hello World! from inputs?

A:

Now another example:

StringGetLine.cpp

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;
    getline(cin, line);
    cout << line << endl;
    return 0;
}
```

Q: what are the outputs if we enter Hello World! from inputs?

A:

string IO is often combined with the while loop to read strings:

StringIOEx1.cpp

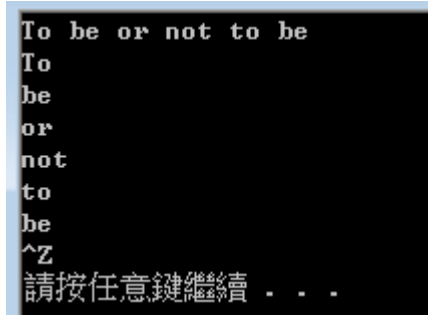
```
#include <iostream>
#include <string>
using namespace std;

int main()
```

```
{  
    string s;
```

```
while (cin >> s)
    cout << s << endl;
return 0;
}
```

The program read string or strings. Print each string in its own line but discards any whitespace (e.g., spaces, newlines, tabs) before or after the string.



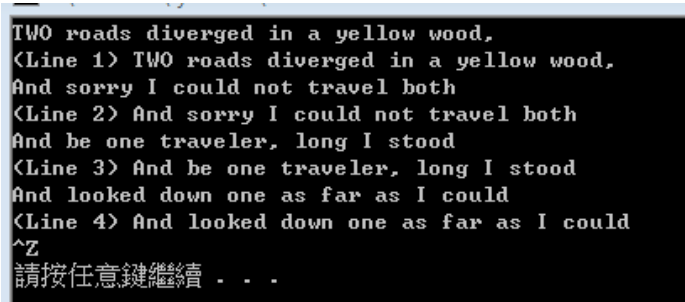
```
To be or not to be
To
be
or
not
to
be
^Z
請按任意鍵繼續 . . .
```

In Windows command, we press **CTRL-Z** to stop the program. In Unix (mac terminal for example), we press **CTRL-D** to stop the program.

Exercise 3.1 In-class Coding Exercise

Ex31.cpp

Write a program to read line from input and output the line with a proper line number. Below is a sample run:



```
TWO roads diverged in a yellow wood,
<Line 1> TWO roads diverged in a yellow wood,
And sorry I could not travel both
<Line 2> And sorry I could not travel both
And he one traveler, long I stood
<Line 3> And he one traveler, long I stood
And looked down one as far as I could
<Line 4> And looked down one as far as I could
^Z
請按任意鍵繼續 . . .
```

Answer

Table 3.2 `string` Operation

<code>os << s</code>	Writes <code>s</code> onto output stream <code>os</code> . Returns <code>os</code> .
<code>is >> s</code>	Reads whitespace-separated string from <code>is</code> into <code>s</code> . Returns <code>is</code> .
<code>getline(is, s)</code>	Reads a line of input from <code>is</code> into <code>s</code> . Returns <code>is</code> .
<code>s.empty()</code>	Returns <code>true</code> if <code>s</code> is empty; otherwise returns <code>false</code> .
<code>s.size()</code>	Returns the number of characters in <code>s</code> .
<code>s[n]</code>	Returns a reference to the char at position <code>n</code> in <code>s</code> ; positions start at 0.
<code>s1 + s2</code>	Returns a string that is the concatenation of <code>s1</code> and <code>s2</code> .
<code>s1 = s2</code>	Replaces characters in <code>s1</code> with a copy of <code>s2</code> .
<code>s1 == s2</code>	The strings <code>s1</code> and <code>s2</code> are equal if they contain the same characters.
<code>s1 != s2</code>	Equality is case-sensitive.
<code><, <=, >, >=</code>	Comparisons are case-sensitive and use dictionary ordering.

string size Operation:

It might be logical to expect that `s.size()` returns an `int` or an `unsigned`. Instead, `s.size()` returns a `string::size_type` value. The reason is that the `string` class—and most other library types—defines several **companion types**. These companion types make it possible to use the library types in a machine independent manner. The type `size_type` is one of these companion types.

To use the `size_type` defined by `string`, we use the scope operator to say that the name `size_type` is defined in the `string` class. Although we don't know the precise type of `string::size_type`, **we do know that it is an unsigned type big enough to hold the size of any string.**

It can be tedious to type `string::size_type`. Under the new standard, we can ask the compiler to provide the appropriate type by using `auto`

```
string s = "I am a C++ string";
string::size_type len1 = s.size(); //C++98
auto len2 = s.size(); // len has type string::size_type, C++11
```

3.2.3 Dealing with Characters in a `string`

We often need to manipulate each character in a string. The `cctype.h` from C (`cctype` in

C++) provides useful utility functions to accomplish the task.

Table 3.3 ctype Functions

<code>isalnum(c)</code>	true if <code>c</code> is a letter or a digit.
<code>isalpha(c)</code>	true if <code>c</code> is a letter.
<code>iscntrl(c)</code>	true if <code>c</code> is a control character.
<code>isdigit(c)</code>	true if <code>c</code> is a digit.
<code>isgraph(c)</code>	true if <code>c</code> is not a space but is printable.
<code>islower(c)</code>	true if <code>c</code> is a lowercase letter.
<code>isprint(c)</code>	true if <code>c</code> is a printable character (i.e., a space or a character that has a visible representation).
<code>ispunct(c)</code>	true if <code>c</code> is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
<code>isspace(c)</code>	true if <code>c</code> is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
<code>isupper(c)</code>	true if <code>c</code> is an uppercase letter.
<code>isxdigit(c)</code>	true if <code>c</code> is a hexadecimal digit.
<code>tolower(c)</code>	If <code>c</code> is an uppercase letter, returns its lowercase equivalent; otherwise returns <code>c</code> unchanged.
<code>toupper(c)</code>	If <code>c</code> is a lowercase letter, returns its uppercase equivalent; otherwise returns <code>c</code> unchanged.



If we want to do something to every character in a string, by far the best approach is to use a statement introduced by the new standard: the **range for** statement. This statement iterates through the elements in a given sequence and performs some operation on each value in that sequence. The syntactic form is

```
for (declaration : expression)
    statement
```

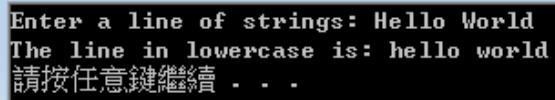
For example, the following code using the range for statement to count the number of punctuations in a string

```
unsigned punct_cnt = 0;
for (auto e: s)
    if (ispunct(e)) ++punct_cnt;
```

Exercise 3.2 In-class Coding Exercise

Ex32.cpp

Write a program to read a line of strings from the standard input and change all the characters to lowercase. A sample run looks like:

A terminal window with a black background and white text. The first line is the prompt "Enter a line of strings:" followed by the user input "Hello World". The second line is the output "The line in lowercase is:" followed by "hello world". The third line is the prompt "請按任意鍵繼續" followed by three dots " . . .".

```
Enter a line of strings: Hello World
The line in lowercase is: hello world
請按任意鍵繼續 . . .
```

Answer:

(see ppt)

3.3.1 Defining and Initializing vectors

Table 3.4 The ways to Initialize a vector

<code>vector<T> v1</code>	vector that holds objects of type T. Default initialization; v1 is empty.
<code>vector<T> v2 (v1)</code>	v2 has a copy of each element in v1.
<code>vector<T> v2 = v1</code>	Equivalent to <code>v2 (v1)</code> , v2 is a copy of the elements in v1.
<code>vector<T> v3 (n, val)</code>	v3 has n elements with value val.
<code>vector<T> v4 (n)</code>	v4 has n copies of a value-initialized object.
<code>vector<T> v5 {a,b,c ... }</code>	v5 has as many elements as there are initializers; elements are initialized by corresponding initializers.
<code>vector<T> v5 = {a,b,c ... }</code>	Equivalent to <code>v5 {a,b,c ... }</code> .

List each element in the following `vector` initialization

```
vector<int> ivec(10, -1);
vector<string> svec(10, "hi");
```

```
vector<int> ivec(10);
vector<int> ivec(10, 1);
vector<int> ivec{10, 1};
```

```
vector<string> svec(10);
vector<Sales_item> salesVec(10);
```

Table 3.5 vector Operation

<code>v.empty()</code>	Returns true if v is empty; otherwise returns false.
<code>v.size()</code>	Returns the number of elements in v.
<code>v.push_back(t)</code>	Adds an element with value t to end of v.
<code>v[n]</code>	Returns a reference to the element at position n in v.
<code>v1 = v2</code>	Replaces the elements in v1 with a copy of the elements in v2.
<code>v1 = {a,b,c ... }</code>	Replaces the elements in v1 with a copy of the elements in the comma-separated list.
<code>v1 == v2</code>	v1 and v2 are equal if they have the same number of elements and each element in v1 is equal to the corresponding element in v2.
<code>v1 != v2</code>	
<code><, <=, >, >=</code>	Have their normal meanings using dictionary ordering.

Using `push_back` member function

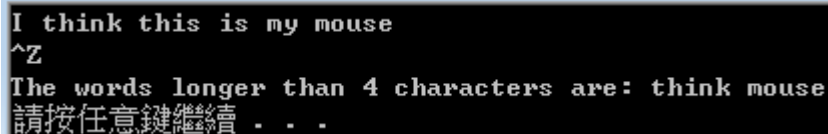
To store a value in a `vector` that does not have a starting size or that is already full, you should use the `push_back` member function. This function accepts a value as an argument and store it in a new element placed at the end of vector. It “pushes” the value at the “back” of the `vector`. For example,

```
vector<int> x;
x.push_back(12);
```

Q: what happens?

A:

With introduction of `string` and `vector`, we can easily store word from standard input into the `vector` container and process these words upon request. For example, we can ask users to input a few words, store them in a `vector` and parse and print those words that are longer than 4 characters.



```
I think this is my mouse
^Z
The words longer than 4 characters are: think mouse
請按任意鍵繼續 . . .
```

VectorStringEx.cpp

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    string word;
    vector<string> text;
    while (cin >> word)
        text.push_back(word);
    for (auto s : text)
        if (s.size() > 4) cout << s << endl;
    return 0;
}
```

3.3.3 Other vector Operations

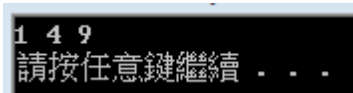
We can access the elements of a `vector` the same way as we access the characters in a

string:

vecEx1.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v{ 1, 2, 3 };
    for (auto &i : v) // for each element in v (note: i is a reference)
        i *= i; // square the element value
    for (auto i : v) // for each element in v
        cout << i << " "; // print the element
    cout << endl;
    return 0;
}
```



Quick Check on Concept: What is the difference between `for (auto &i : v)` and `for (auto i : v)`?

```
| for (auto &i : v)
```

We define our control variable, `i`, as a reference so that we can use `i` to assign new values to the elements in `v`.

```
| for (auto i : v)
```

Control variable, `i`, is a copy of an element in `v`. Any change in `i` will not affect the elements in `v`. Thus, we use this kind of range `for` for **read only access** in a container.

Subscript Operator []

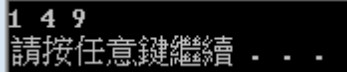
We can obtain a given element in `vector` using the subscript operator `[]` as with `strings`. Subscripts for `vector` start at 0 (a typical C/C++ convention). For example, the previous code can now be modified as:

vecEx2.cpp

```
| #include <iostream>
| #include <vector>
```

```
using namespace std;

int main(){
    vector<int> v{ 1, 2, 3 };
    for (decltype(v.size()) idx = 0; idx != v.size();
        ++idx){ v[idx] = v[idx] * v[idx];
        cout << v[idx] << " ";
    }
    cout << endl;
    return 0;
}
```

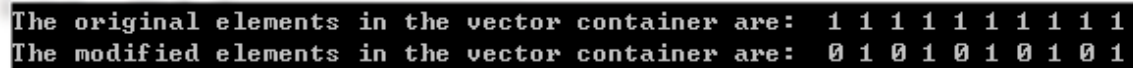


1 4 9
請按任意鍵繼續 . . .

Exercise 3.3 In-class Coding Exercise

Ex33.cpp

Define and initialize a vector with 10 elements of 1 and print the contents. Modify all the even elements in the vector to 0 and print the modified contents. A sample output looks like:



The original elements in the vector container are: 1 1 1 1 1 1 1 1 1 1
The modified elements in the vector container are: 0 1 0 1 0 1 0 1 0 1

(Answer)

3.4 Introducing Iterators

(see ppt)

Looping through containers

Subscript Operator []

```
// reset all the elements in ivec to 0
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

C++11 way

```
// reset all the elements in ivec to 0
for (decltype(ivec.size()) ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

Iterator

```
// using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin(); iter != ivec.end();
    ++iter)
    *iter = 0; // set element to which iter refers to 0
```

C++11 way

```
// using iterators to reset all the elements in ivec to 0
for (auto iter = ivec.begin(); iter != ivec.end(); ++iter)
    *iter = 0; // set element to which iter refers to 0
```

const_iterator for reading but not writing to the elements in the container.

```
string word;
vector<string> text;
while (cin >> word) {
    text.push_back(word);
}
for (vector<string>::const_iterator iter = text.begin();
    iter != text.end(); ++iter)
    cout << *iter << endl;
```

C++11 way

```
string word;
vector<string> text;
while (cin >> word) {
    text.push_back(word);
}
for (auto iter = text.cbegin(); iter != text.cend(); ++iter)
    cout << *iter << endl;
```

Exercise 3.4 In-class Coding ExerciseEx34.cpp

Rewrite Exercise 3.3 using iterator. Fill up the blank below.

```
The original elements in the vector container are: 1 1 1 1 1 1 1 1 1 1
The modified elements in the vector container are: 0 1 0 1 0 1 0 1 0 1
請按任意鍵繼續 . . .
```

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> ivec(10, 1);
    cout << "The original elements in the vector container are: "
          << " ";

    //(your codes)

    cout << endl;
    cout << "The modified elements in the vector container are: "
          << " ";

    //(your codes)

    cout << endl;
    return 0;
}
```

Ans:

Key Concept: Generic Programming

Programmers coming to C++ from C or Java might be surprised that we used `!=` rather than `<` in our for loops. C++ programmers use `!=` as a matter of habit. They do so for the same reason that **they use iterators rather than subscripts**: This coding style applies equally well to various kinds of containers provided by the standard library.

As we will learn later, only a few standard library types, `vector` and `string` being among them, have the subscript operator. Similarly, **all of the library containers** have iterators that define the `==` and `!=` operators. Most of those iterators do not have the `<` operator. By routinely using iterators and `!=`, we don't have to worry about the precise type of container we're processing.

Iterator Operations

Standard iterators support only a few operations, which are listed below (Table 3.6).

<code>*iter</code>	Returns a reference to the element denoted by the iterator <code>iter</code> .
<code>iter->mem</code>	Dereferences <code>iter</code> and fetches the member named <code>mem</code> from the underlying element. Equivalent to <code>(*iter).mem</code> .
<code>++iter</code>	Increments <code>iter</code> to refer to the next element in the container.
<code>--iter</code>	Decrements <code>iter</code> to refer to the previous element in the container.
<code>iter1 == iter2</code>	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.
<code>iter1 != iter2</code>	

Remark: We can compare two valid iterators using `==` or `!=`. Iterators are equal (1) if they denote the same element or (2) if they are both off-the-end iterators for the same container. Otherwise, they are unequal. For example, we can simply write a code fragment that will capitalize the first character of a string.

```
string s("some string");
if (s.begin() != s.end()) { // make sure s is not empty
    auto it = s.begin(); // it denotes the first character in s
    *it = toupper(*it); // make that character uppercase
}
```

Exercise 3.5 In-class Coding Exercise**Ex35.cpp**

A textfile `input.txt` contains sentences of text. A line with an empty string indicates a paragraph break. Write a program to store all the lines in a vector container and print the lines in the first paragraph. For example, if our `input.txt` has the following contents:

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

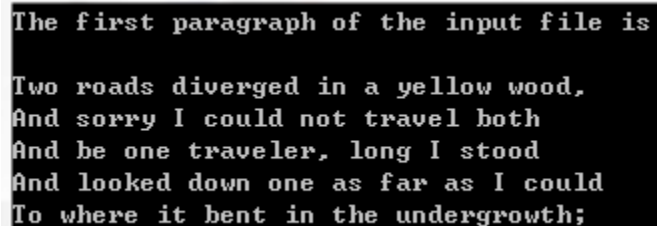
Then took the other, as just as fair,
And having perhaps the better claim
Because it was grassy and wanted wear,
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I,
I took the one less traveled by,
And that has made all the difference.

The Road Not Taken by Robert Frost

A sample output looks like:



```
The first paragraph of the input file is  
  
Two roads diverged in a yellow wood,  
And sorry I could not travel both  
And be one traveler, long I stood  
And looked down one as far as I could  
To where it bent in the undergrowth;
```

Answer:

3.4.2. Iterator Arithmetic

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time. They also support all the relational operators. These operations, which are often referred to as **iterator arithmetic**, are described below (Table 3.7).

<code>iter + n</code>	Adding (subtracting) an integral value <code>n</code> to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
<code>iter - n</code>	
<code>iter1 += n</code>	Compound-assignment for iterator addition and subtraction. Assigns to <code>iter1</code> the value of adding <code>n</code> to, or subtracting <code>n</code> from, <code>iter1</code> .
<code>iter1 -= n</code>	
<code>iter1 - iter2</code>	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
<code>>, >=, <, <=</code>	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

3.5 Array

(see ppt)

The **general form** for declaring an array is:

```
| typeName arrayName[arraySize];
```

The expression `arraySize`, which is the number of elements, must be a constant expression (must be known at compile time).

```
| int arr[10]; // array of ten ints
| //
| unsigned cnt = 42; // not a constant expression
| string bad[cnt]; // error: cnt is not a constant expression
| //
| const unsigned s = 42; // constant expression
| int *parr[s]; // array of 42 pointers to int
```

Initialization (see ppt)

3.5.2. Accessing the Elements of an Array

As with the library `vector` and `string` types, we can use a `range for` or the subscript

operator `[]` to access elements of an array.

When we use a variable to subscript an array, we normally should define that variable to have type `size_t`. `size_t` is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The `size_t` type is defined in the `cstdint` header, which is the C++ version of the `stdint.h` header from the C library. (you often do not need to explicitly include `cstdint` since many header files are likely included it already)

(Example) suppose we have the following array defined,

```
const size_t array_size = 10;
int ia[array_size];
```

(a) How to assign value of each element equal to its index?

```
for (size_t i=0; i != array_size ; ++i)
    ia[i] = i;
```

(b) How to copy one array `ia` into the other `ia2`?

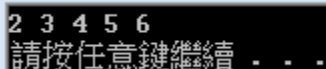
```
int ia2[array_size];
for (size_t i=0; i != array_size ; ++i)
    ia2[i] = ia[i];
```

We can use range for loop to access the element in an array as we did for string and vector.

arrAuto.cpp

```
#include <iostream>
using namespace std;

int main(){
    const size_t array_size = 5;
    int ia[array_size] = {0, 1, 2, 3, 4};
    for (auto& i : ia)
        i += 2;
    for (auto i : ia)
        cout << i << " ";
    cout << endl;
    return 0;
}
```



```
2 3 4 5 6
請按任意鍵繼續 . . .
```

3.5.3. Pointers and Array

(see ppt)

Pointers Are Iterators

Pointers to array elements support the same operations as iterators on vectors or strings. For example, we can use the increment operator to move from one element in an array to the next:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p; // p points to arr[1]
```

(C++11) Library begin and end functions

To make it easier and safer to use pointers, the new library includes two functions, named `begin` and `end`. These functions are defined in `<iterator>` and act like the similarly named container members.

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *beg = begin(ia); // pointer to the first element in ia
int *last = end(ia); // pointer one past the last element in ia
```

Quick Check: how to do the same thing in vector?

A:

Using `begin` and `end`, it is rather easy to write a loop to process the elements in an array.

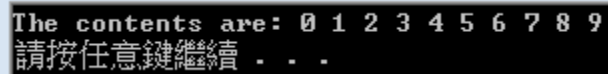
```
int arr[20];
int *pbeg = begin(arr), *pend = end(arr);
while (pbeg != pend){
    //do something
    ++pbeg;
}
```

Quick Check: The following code assign value of each element equal to its index

arrayEx.cpp

```
#include <iostream>
using namespace std;
```

```
int main(){
    const size_t array_size = 10;
    int ia[array_size];
    cout << "The contents are: ";
    for (size_t ix = 0; ix < array_size; ++ix)
        { ia[ix] = ix;
          cout << ia[ix] << ' ';
        }
    cout << endl;
    return 0;
}
```



```
The contents are: 0 1 2 3 4 5 6 7 8 9
請按任意鍵繼續 . . .
```

Rewrite the codes using `begin` and `end`.

A:

arrayExIter.cpp