5 Pointers, Iterators, and Memory Management

Pointers were used in C as a machine-oriented feature for systems programming. C++ provides pointers because it is compatible with C. But C++ also provides a more abstract concept: the iterator. In this lecture, we discuss pointers and iterators, the relationship between them, and the importance of (manual) memory management in C++ programs.

5.1 Pointers

An important difference between C++ and Java is that, in C++, dynamic memory allocation and deallocation is ***explicit*** (done by code written by programmers) whereas in Java it is ***implicit*** (done by built-in system code). It is not quite true to say "Java does not have pointers", but it is true to say "Java does not allow direct access to pointers". (Weiss 2004, Chapter 3) includes a detailed comparison of the Java vs. C++ types and memory models.

> ***A pointer is a variable that contains the memory address of another variable.***

Like all variables in C++, a pointer has a type. The type depends on the object addressed by the pointer. For example, a pointer that points to an `int` has type "pointer to `int`", written `int*`.

> ***For every type T, there is a type $T*$ or pointer to T.***

To assign a value to a pointer, we need a way of obtaining addresses. There are two common ways, and we define the unimportant one first. The operator `&`, applied to a variable, yields the address of that variable. The address can be stored as a pointer. The following code defines two pointers, `pk1` and `pk2`, both pointing to the integer `k`, as shown in Figure 49 on the next page.

address-of operator &

130

131

```
int k = 42;
int *pk1 = &k;
int *pk2 = &k;
```

If we have a pointer to an object, we can obtain the object itself by applying the ***prefix*** operator `*`. In the example above, `*pk1` is an integer variable (it is actually, of course, the integer `k`).

operator *

The declaration `int *pk1` is a kind of pun. We can read it as `(int *)pk1` ("`pk1` has type `int *`") or as `int (*pk1)` ("`*pk1` has type `int`"). The forms with parentheses are illegal, but the C++ compiler is not fussy about spaces: we can write any of

132

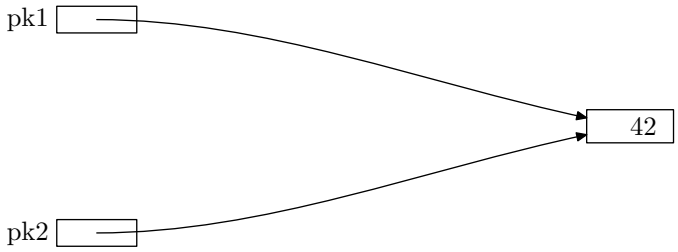

Figure 49: Two pointers, one value

```
int*pk0 = &k;
int *pk1 = &k;
int* pk2 = &k;
int * pk3 = &k;
```

Some programmers prefer the second form and some the third; few use the first or the fourth, although they are legal. There is one syntactic trap that you should be aware of. The statement

```
int* p, q;
```

declares `p` to be a pointer to `int`, but `q` is just a plain `int`.

Using pointers can have strange results – although they are not all that strange if you think
133 about them carefully. For example, the code

```
int k = 42;
int *pk1 = &k;
int *pk2 = &k;
++(*pk1);
cout << k << ' ' << *pk2 << endl;
```

displays `43 43`. Since both pointers point to the same integer, any change to its value will be seen by ***all*** pointers.

5.1.1 Pointer Arithmetic

pointer arithmetic

C++ provides arithmetic operations (+ and −) on pointers. However, the arithmetic is rather special. If `p1` and `p2` are pointers and `i` is an integer, then:
134

`++p1`	is a pointer
`--p1`	is a pointer
`p1 + i`	is a pointer
`p1 - i`	is a pointer
`p1 - p2`	is a signed integer (`p1` and `p2` must have the same type)

These operations would be utterly meaningless if it were not for the fact that C++ uses the size of the object pointed to when doing arithmetic. For example, suppose that an instance of type `T` occupies 20 bytes and that we write: 135

```
T t;
T *pt = &t;
```

Then `pt+1` is an address 20 bytes greater than `pt`.

This kind of arithmetic is exactly what we need for arrays. In fact, subscript operations in C++ are ***defined*** in terms of pointer arithmetic. The declaration 136

```
double a[6];
```

introduces `a` as an array of 6 `doubles`. C++ treats `a` as a `const` pointer. We have: arrays and pointers

$$\begin{array}{lcll} \texttt{a[0]} & \equiv & *\texttt{a} & \\ \texttt{a[1]} & \equiv & *(\texttt{a}+1) & \\ \texttt{a[2]} & \equiv & *(\texttt{a}+2) & \\ & \cdots & & \\ \texttt{a}[i] & \equiv & *(\texttt{a}+i) & \text{for any integer } i \end{array}$$

5.1.2 Pointer types vs. reference types

The use of `&` as the "address-of" operator is different from its use as the "reference-to" operator, as you have seen it in Chapter 3.2 for pass by reference parameter definitions `(int & k)`. However, the uses are closely related, in the sense that a reference is an address. Compare this code, working with pointers: 137

```
int i;
int *pi = &i;
++*pi;
cout << i << ' ' << *pi << endl;
```

with this code, working with references:

```
int i;
int &ri = i;
++ri;
cout << i << ' ' << ri << endl;
```

In C programming, there are no reference data types, so pointer types were used to achieve the same effect as pass by reference. C++ introduced reference types for this purpose and you should not use pointers anymore for modifying a function's arguments.

5.1.3 A note on null pointers

Every pointer type has a special value called ***null***. The definition of the null pointer is that it does not point to an object. The representation is the value zero; this works because, on almost all machines, the address zero is reserved for the operating system and cannot be used by programs to store data.

In old-style C programming, the null pointer was defined as a constant `NULL`, typically by a
138 directive such as

```
#define NULL 0
```

This definition is compatible with the spirit of C's rather loose approach to types, but is not consistent with C++'s safer typing. Improvements such as

```
#define NULL (int) 0
#define NULL (void*) 0
```

do not help, because the first version does not work for pointers and the second version requires an ugly-looking cast whenever we use `NULL` for a pointer type other than `void*`. For example, to obtain the null pointer for integers, we would have to write

```
static_cast<int*>(NULL)
```

In C++, `#define`s are deprecated, and we are supposed to use constant declarations instead. Unfortunately, any reasonable declaration of `NULL` has the same problems as the attempts to `#define NULL`:

```
const int NULL = 0;
const void * NULL = 0;
```

Even if we could find a satisfactory definition for `NULL`, there would still be problems. Suppose
139 that we declare these functions:

```
void f(int n);     // first overload
void f(char *p);   // second overload
```

and call `f(NULL)` thinking that the compiler would say to itself: "The programmer has used `NULL`, which suggests a pointer, and therefore I will pass `static_cast<char*>(NULL)` to the second overload".

However, the compiler does not reason like this. Instead, it says: "`NULL` is 0 and 0 is an `int`, and therefore I will pass 0 to the first overload".

As Meyers (Meyers 1998, Item 25) explains, this is an unusual case because people tend to think that there is an ambiguity but the compiler does not. (Usually, people think their meaning is perfectly obvious and are annoyed when the compiler calls it ambiguous.) Meyers recommends:

Prefer not to overload a function with integer types and pointer types.

The simplest solution for writing null pointers is just:

Forget about `NULL` ***– just use*** `0`***.***

5.1.4 Null pointers in C++11

C++11 includes a new 'null pointer constant', `nullptr`. This makes the confusing double-use of '0' as both the integer zero and the `NULL` pointer obsolete. The `nullptr` is of type `nullptr_t` and can be used like this:

`nullptr`

140

```
char *p = nullptr;
```

Use `nullptr` ***instead of*** `0` ***or*** `NULL` ***for*** **C++11** ***and above.***

5.2 Dynamic Memory Allocation

So far, we did not worry about managing memory for variables: even for dynamically growing data structures like vectors, we relied on automatic memory management (in case of vectores, this is handled by the STL). However, for most C++ programs you will need to take care of memory management yourself, as C++ does not offer built-in ***garbage collection*** like Java. Hence, it is important to understand when and how to manually allocate and de-allocate memory.

5.2.1 Stack Allocation

Most data in C++, and all the data we have seen so far in this course, is allocated on the ***run-time stack***.

The run-time stack, "stack" for short, is initially empty. When execution starts, the runtime parameter (program arguments) are pushed onto the stack. On entry to a function, the local data associated with the function (including its arguments) are pushed onto the stack. When the function returns, the local data is popped off the stack. Thus the stack varies in size as the program runs.

When a function has returned, its local data no longer exists.[20] Normally, this is not a problem, because the function's local variables are no longer accessible. Playing tricks with pointers, however, can cause problems. Consider the code shown in Figure 50 on the following page.

141

This program compiles because there are no syntactic or semantic errors. The final assignment, `p = f()`, makes `p` a pointer to `k`. But `k` no longer exists, having been popped off the stack when `f` returns. Any attempt to use `p` will have unpredictable results.

```
int * f()
{
    int k = 42;
    int *pk = &k;
    return pk;
}

int main()
{
    int *p;
    p = f();
    return 0;
}
```

Figure 50: Dangerous tricks

```
char *read()
{
    char buffer[20];
    cin >> buffer;
    return buffer;
}

int main()
{
    char *pc = read();
    cout << pc << endl;
    return 0;
}
```

Figure 51: Subtler dangerous tricks

142

Figure 51 shows a more subtle version of the same problem. The function `read` has result type `char*` but actually returns an array of characters: this works because the compiler treats these types as the same. Similarly, the main function treats function `read` as having type `char*`.

The reason that this is a bad program is that the array `buffer` is allocated on the stack. It is destroyed when the function returns. The pointer `pc` is undefined and cannot be used safely.

Some compilers, including GCC, actually recognize the mistake and issue a warning message:

```
In function char* read():
warning: address of local variable buffer returned
```

[20] Actually, it's still there, on the stack, but will be overwritten when the next function is called. Consequently, we must ***assume*** that it no longer exists.

This provides another reason for paying attention to warnings from the compiler!

Problems like this tend to occur when we use "old-style" C++ code. No problems arise if we use the STL class `string` instead of an array of characters.

> ***Use* `string`*, in preference to* `char *`*, wherever possible.***

5.2.2 Heap Allocation

Stack allocation works because function calls and returns match the "last-in-first-out" (LIFO) discipline of a stack. Sometimes this is not good enough. For example, we might want to allocate data within a function, use that data for a while after the function has returned, and then deallocate the data. We can do this by allocating the data on the ***heap***, an area of memory that does not obey any particular discipline such as LIFO or FIFO. The heap is used like this: 143

```
T *p = new T();
....
delete p;
```

The operator `new` requires a call to a constructor on its right. The value returned by `new` is a pointer to the constructed object. We can use this pointer to perform operations on the object. When we have finished with the object, we apply `delete` to the pointer to destroy the object and deallocate the heap memory it was using.

operator new for heap allocation

Suppose class `T` provides a public function `f`. To call `f` using the pointer `p`, we must first use '`*`' to dereference `p` and then use '`.`' to call `f`. Thus we write `(*p).f()`. (The parentheses around `*p` are necessary, because the compiler reads `*p.f()` as `*(p.f())`, which is wrong.) Since this construction occurs often, there is an abbreviation for it:

-> operator

$$\texttt{p->f()} \quad \equiv \quad \texttt{(*p).f()}$$

144

Figure 52 on the following page provides a simple example of heap allocation and deallocation. The function `maketest` constructs a new instance of class `Test` on the heap, calls its function `f`, and returns a pointer to it. The function `killTest` deletes the object. The program displays the following output: 145 146

```
makeTest
Constructor
Function
killTest
Destructor
```

In C++, it is important to always be aware of the memory location where a particular object is stored at run-time (stack or heap) – in Java, this question does not arise, because objects are only created on the heap, never on the stack.

```
class Test
{
public:
    Test() { cout << "Constructor\n"; }
    ~Test() { cout << "Destructor\n"; }
    void f() { cout << "Function\n"; }
};

Test *makeTest()
{
    cout << "makeTest\n";
    Test *pt = new Test();
    pt->f();
    return pt;
}

void killTest(Test *p)
{
    cout << "killTest\n";
    delete p;
}

int main()
{
    Test *p = makeTest();
    killTest(p);
    return 0;
}
```

Figure 52: Heap allocation and deallocation

5.3 Iterators

iterators, containers, and algorithms

Iterators are one of the keys to the flexibility of the STL. We have seen that the STL provides containers and algorithms. Iterators provide the glue that allows us to attach one to the other:

- Each container specifies the iterators that it provides
- Each algorithm specifies the iterators that it needs

For example, `vector<T>` provides the kind of iterators that `sort` requires; it follows that we can sort vectors.

A pair of iterators specifies a range of container elements. The range defines a semi-closed interval: the `first` iterator of a range accesses the first element of the range, and the `last` iterator accesses the first element ***not*** in the range. In this typical loop:

147

```
for (⟨iterator⟩ it = first; it != last; ++it)
    .... *it ....
```

⟨*iterator*⟩ stands for some iterator type, and we see that the iterator must provide the operations:

- `first != last` (and therefore `first == last`): equality comparison
- `++it`: increment, or step to next element
- `*it`: dereference to provide access to the container element

C++ programmers will recognize that all of these operations are provided by pointers. In fact, we can use raw pointers as iterators:

pointers as iterators

148

```
const int MAX = 20;
double values[MAX] = .... ;
sort(&values[0], &values[MAX]);
```

5.3.1 Kinds of iterator

149

There are several ways of classifying iterators. Below, we define individual properties; most iterators possess several of these properties. In each of the following cases, we use `it` to stand for an iterator with the given property. Figure 53 summarizes the properties that various kinds of iterators provide.

150

- All iterators implement the operator `=` (assignment).
- With the exception of the output iterator, all iterators implement the operators `==` and `!=` (comparison).
- An ***input iterator*** can be used to read elements from a container but does not provide write access. That is, `*it` is an Rvalue.
- An ***output iterator*** can be used to update elements in a container but may not provide read access. That is, `*it` is a Lvalue. Note that output iterators ***cannot*** be compared with `==` or `!=`.

Property	Input	Output	Forward	Bidirectional	Random Access	Insert
Assign (`=`)	√	√	√	√	√	√
Compare (`==`, `!=`)	√		√	√	√	√
Read (`*it`)	√		√	√	√	
Write (`*it`)		√	√	√	√	√
Order (`<`, `<=`, `>`, `>=`)					√	
Increment (`++`)			√	√	√	
Decrement (`--`)				√	√	
Arithmetic ($\pm i$, `+=`, `-=`, `-`)					√	

Figure 53: Properties of iterators

An iterator may be both an input and an output iterator. That is, it may allow both read and write access to elements of the container. In fact, this is probably the most common kind of iterator.

- A ***forward iterator*** is an input and output iterator that can traverse the container in one direction. A forward iterator must implement `++`.
- A ***bidirectional iterator*** is an input and output iterator that can traverse the container forwards and backwards. A bidirectional iterator must implement `++` and `--`.

  There are no "backwards only" iterators; an iterator is either forward or bidirectional.
- A ***random access iterator*** must allow "jumps" in access as well as traversal. The principal operation that a random access iterator must provide is indexing: `it[n]`. Random access iterators also provide:
  - Addition and subtraction of integers: `it + n` and `it - n`
  - Assignment operators: `it += n` and `it -= n`
  - Subtraction: `it1 - it2` yields a signed integer
  - Comparisons: the operators `<`, `<=`, `>=`, and `>`
- An ***insert iterator*** is an output iterator that puts new values into a container rather than just updating the values that are already there.

All of the STL containers provide bidirectional iterators. It follows that they all provide input, output, and forward iterators. These categories are useful because we can construct special iterators that may not have all of the properties.

The only container classes that provide random access iterators are `deque`, `string`, and `vector`. This is because these containers are required to store elements in consecutive locations, which
151 means that random access is a simple address calculation.[21] Figure 54 shows the iterators associated with a ***selection*** of containers and algorithms. The table is far from complete and, for details, you should consult an STL reference, such as (Josuttis 2012).

Container	Iterator provided	Algorithm	Iterator required
`vector`	random access	`find`	input
`list`	bidirectional	`search`	forward
`slist`	forward	`count`	input
`deque`	random access	`copy`	input/output
`set`	constant bidirectional	`reverse`	bidirectional
`multiset`	bidirectional	`sort`	random access
`map`	bidirectional	`stable_sort`	random access
`multimap`	bidirectional	`binary_search`	forward

Figure 54: Iterators, Containers, and Algorithms

[21] The address of `c[i]` is $\&c + s \times i$, where $\&c$ is the address of the container and s is the size of an element.

We have often mentioned that a range is specified by an iterator accessing the first element of the range and another iterator accessing the element following the last element of the range – which, in most cases, does not even exist. Here are some reasons for this choice:

1. Two equal iterators specify an empty range.
2. We can use `==` and `!=` to test for an empty range and for end of range – we do not need `<` and friends.
3. We have an easy way to indicate "out of range", namely, the `last` iterator of the range.

   A function can return an iterator for a valid element to indicate success or an iterator for an invalid element to indicate failure. This avoids the need for special values, flags, etc.

An iterator with type `iterator` is allowed to change the contents of its associated container. An iterator with type `const_iterator` (called a ***constant iterator***) is ***not*** allowed to change the contents of its associated container. It is best to use constant iterators whenever possible. const_iterator

5.3.2 Using iterators

Suppose that we want to divide the students of the grading program into two groups, according as to whether they passed or failed the course. The first thing we will need is a criterion for deciding whether a student has passed. We add the following member function to class `Student`: 152

```
bool passed() { return total > 50; }
```

Note:

- The definition of this function appears within the class declaration. This is allowed, and a consequence is that the compiler may ***inline*** calls to the function. We will discuss the implications of this later.
- Beginners might write `passed` in this way: 153

  ```
  bool passed()
  {
     if (total > 50)
        return true;
     else
        return false;
  }
  ```

  This is ***stupid***! The statements `return true` and `return false` are occasionally required but, in many cases, a predicate should return a boolean expression.

Let us first consider a straightforward approach, in which we create two empty vectors for passed and failed students, and iterate through the class assigning each student to one or the other vector: see Figure 55 on the next page. In this code, remember that `it->passed()` is an abbreviation for `(*it).passed()`. 154

Unfortunately, this code does not compile: see Figure 56 on the following page. The reason is rather subtle: 155

```
vector<Student> passes;
vector<Student> failures;
for (  vector<Student>::const_iterator it = data.begin();
       it != data.end();
       ++it )
    if (it->passed())
        passes.push_back(*it);
    else
        failures.push_back(*it);
```

Figure 55: Separating passes and failures: first version

```
f:\....\grader.cpp(111):
   error C2662: 'Student::passed' :
   cannot convert 'this' pointer from
       'const std::allocator<_Ty>::value_type'
   to  'Student &'
   with
        [
            _Ty=Student
        ]
```

Figure 56: Compiling Figure 55 fails

- By using `const_iterator`, we are asserting that the iterator cannot be used to change the value of a `Student`.
- The loop contains the call `it->passed()`. ***We*** know that this call will not change the value of the `Student` object `*it`, but the ***compiler*** doesn't.
- Consequently, the compiler rejects the program.

156

There is a "quick and dirty" fix for this error: we could just use an `iterator` instead of a `const_iterator`. This is a ***bad*** solution because it does not address the real cause of the problem. The correct solution is to convince the compiler that `passed` does not change the value of the object it acts on. We can do this by adding `const` to the function declaration:

const member functions

157

```
bool passed() const { return total > 50; }
```

After executing the code in Figure 55, we now have ***three*** vectors and two copies of each student record, one in the original vector and the other in either `passes` or `failures`. It would be more efficient in terms of space to move the failed students into a new vector and remove them from the original vector. Vectors provide the function `erase` to remove elements from a container. The code in Figure 57 on the facing page does this.

The first point to note is that we have to use a `while` loop rather than a `for` loop, because the loop step is not necessarily `++it`. However, for students that pass the course, `++it` is all we have to do.

```
vector<Student> failures;
vector<Student>::iterator it = data.begin();
while (it != data.end())
{
    if (it->passed())
        ++it;
    else
    {
        failures.push_back(*it);
        it = data.erase(it);
    }
}
```

Figure 57: Separating passes and failures: second version

When a student fails, the corresponding record is stored in `failures`. In order to remove the record from the class data vector, we write

```
it = data.erase(it);
```

The effect of `data.erase(it)` is to remove the element indicated by `it` from the vector `data`. After this has been done, the iterator is ***invalid*** because it accesses an element that no longer exists. The function `erase` returns an iterator that accesses the next element of the vector.

iterator invalidation

It is important to use the iterator returned by `erase` and not to assume that it is just `++it`. An iterator operation that invalidates an iterator may do other things as well, even including moving the underlying data. When an iterator operation invalidates an iterator, it potentially invalidates ***all*** iterators.

For example, calling `erase` in Figure 57 invalidates the iterator for the end of the vector, because removing one component forces the remaining components to move. It would therefore be a serious mistake to attempt to "optimize" Figure 57 like this:

158

```
vector<Student> failures;
vector<Student>::iterator it = data.begin();
vector<Student>::iterator last = data.end(); // Save final iterator
while (it != last)
    ....
```

> ***Check whether an operation invalidates iterators before using it.***

The solution we have developed works correctly, but is inefficient. The inefficiency is negligible for a class of sixty students but could be a problem if we used the same strategy for very large vectors. To understand the reason for the inefficiency, suppose that an entire class of 50 students fails. The program would execute as follows:

Remove first record, move remaining 49 records
Remove second record, move remaining 48 records
Remove third record, move remaining 47 records
....

The total number of operations is $\underbrace{49 + 48 + 47 + \cdots + 1}_{49 \text{ terms}}$ and is clearly $\mathcal{O}(N^2)$ for N students.

To improve the performance, we must change the data structure. A list can erase in constant time (i.e., $\mathcal{O}(1)$) and can perform the other operations that we require. It is a straightforward exercise to replace each `vector` declaration by a corresponding `list` declaration.

It is obviously important to know which operations invalidate iterators. Fortunately, good STL reference documents usually provide this information. If you are not sure, you can make a good guess by thinking about how the operation must work on a given data structure – but it's much safer to look up the correct answer.

5.3.3 Range Functions

Most containers have ***range functions*** – that is, functions with a pair of parameters representing a range of elements in the container. If a suitable range function exists, it is better to use it than to use a loop.

Suppose that you want to create a vector `v1` consisting of the back half of the vector `v2` (Meyers 159 2001, Item 5). You could do it with a loop:

```
v1.clear();
for (   vector<Widget>::const_iterator ci = v2.begin() + v2.size()/2;
        ci != v2.end();
        ++ci )
   v1.push_back(*ci);
```

but it is quicker to write any one of the following statements:

```
v1.assign(v2.begin() + v2.size()/2, v2.end());

v1.clear();
copy(v2.begin() + v2.size()/2, v2.end(), back_inserter(v1));

v1.insert(v1.end(), v2.begin() + v2.size()/2, v2.end());
```

In this case, `insert` is probably the best choice.

References

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Meyers, S. (1998). *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* (2nd ed.). Addison-Wesley.

Meyers, S. (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.