```
--------------------------------------------------------------------------------
Magdalen College
Nuffield College
Kellogg College
```

Listing 3-5: A program illustrating a common idiom for passing arrays to functions

The `print_names` function accepts an array in two arguments: a pointer to the first `College` element ❶ and the number of elements `n_colleges` ❷. Within `print_names`, you iterate with a `for` loop and an index `i`. The value of `i` iterates from `0` to `n_colleges-1` ❸.

You extract the corresponding college name by accessing the `i`th element ❹ and then get the `name` member ❺.

This pointer-plus-size approach to passing arrays is ubiquitous in C-style APIs, for example, in Windows or Linux system programming.

Pointer Arithmetic

To obtain the address of the `n`th element of an array, you have two options. First, you can take the direct approach of obtaining the `n`th element with square brackets (`[]`) and then use the address-of (`&`) operator:

```
College* third_college_ptr = &oxford[2];
```

Pointer arithmetic, the set of rules for addition and subtraction on pointers, provides an alternate approach. When you add or subtract integers to pointers, the compiler figures out the correct byte offset using the size of the pointed-to type. For example, adding 4 to a `uint64_t` pointer adds 32 bytes: a `uint64_t` takes up 8 bytes, so 4 of them take up 32 bytes. The following is therefore equivalent to the previous option of obtaining the address of the `n`th element of an array:

```
College* third_college_ptr = oxford + 2;
```

Pointers Are Dangerous

It's not possible to convert a pointer to an array, which is a good thing. You shouldn't need to, and besides it wouldn't be possible in general for a compiler to recover the size of the array from a pointer. But the compiler can't save you from all the dangerous things you might try to do.

Buffer Overflows

For arrays and pointers, you can access arbitrary array elements with the bracket operator (`[]`) or with pointer arithmetic. These are very powerful tools for low-level programming because you can interact with memory more or less without abstraction. This gives you exquisite control over the system, which you need in some environments (for example, in system programming contexts like implementing network protocols or with embedded controllers). With great power comes great responsibility, however, and you must be very careful. Simple mistakes with pointers can have catastrophic and mysterious consequences.

Listing 3-6 performs low-level manipulation on two strings.

```
#include <cstdio>
int main() {
  char lower[] = "abc?e";
  char upper[] = "ABC?E";
  char* upper_ptr = upper;     ❶ // Equivalent: &upper[0]

  lower[3] = 'd';              ❷ // lower now contains a b c d e \0
  upper_ptr[3] = 'D';            // upper now contains A B C D E \0

  char letter_d = lower[3];    ❸ // letter_d equals 'd'
  char letter_D = upper_ptr[3];  // letter_D equals 'D'

  printf("lower: %s\nupper: %s", lower, upper); ❹

  lower[7] = 'g';              ❺ // Super bad. You must never do this.
}
-------------------------------------------------------------------------------
lower: abcde ❹
upper: ABCDE
The time is 2:14 a.m. Eastern time, August 29th. Skynet is now online. ❺
```

Listing 3-6: A program containing a buffer overflow

After initializing the strings `lower` and `upper`, you initialize `upper_ptr` pointing to the first element ❶ in `upper`. You then reassign the fourth elements of both `lower` and `upper` (the question marks) to `d` and `D` ❷❸. Notice that `lower` is an array and `upper_ptr` is a pointer, but the mechanism is the same. So far, so good.

Finally, you make a major boo-boo by writing out-of-bounds memory ❺. By accessing the element at index `7` ❹, you've gone past the storage allotted to `lower`. No bounds checking occurs; this code compiles without warning.

At runtime, you get *undefined behavior*. Undefined behavior means the C++ language specification doesn't prescribe what happens, so your program might crash, open a security vulnerability, or spawn an artificial general intelligence ❺.

The Connection Between Brackets and Pointer Arithmetic

To understand the ramifications of out-of-bounds access, you must understand the connection between bracket operators and pointer arithmetic. Consider that you could have written Listing 3-6 with pointer arithmetic and dereference operators rather than bracket operators, as demonstrated in Listing 3-7.

```
#include <cstdio>
int main() {
  char lower[] = "abc?e";
  char upper[] = "ABC?E";
  char* upper_ptr = &upper[0];

  *(lower + 3) = 'd';
  *(upper_ptr + 3) = 'D';

  char letter_d = *(lower + 4); // lower decays into a pointer when we add
  char letter_D = *(upper_ptr + 4);

  printf("lower: %s\nupper: %s", lower, upper);

  *(lower + 7) = 'g'; ❶
}
```

Listing 3-7: An equivalent program to Listing 3-6 that uses pointer arithmetic

The `lower` array has length 6 (the letters *a*–*e* plus a null terminator). It should now be clear why assigning `lower[7]` ❶ is perilous. In this case, you're writing to some memory that doesn't belong to `lower`. This can result in access violations, program crashes, security vulnerabilities, and corrupted data. These kinds of errors can be very insidious, because the point at which the bad write occurs might be far removed from the point at which the bug manifests.

void Pointers and std::byte Pointers

Sometimes the pointed-to type is irrelevant. In such situations, you use the *void pointer* `void*`. The `void` pointers have important restrictions, the principal of which is that you cannot dereference a `void*`. Because the pointed-to type has been erased, dereferencing makes no sense (recall that the set of values for `void` objects is empty). For similar reasons, C++ forbids `void` pointer arithmetic.

Other times, you want to interact with raw memory at the byte level. Examples include low-level operations like copying raw data between files and memory, encryption, and compression. You cannot use a `void` pointer for such purposes because bit-wise and arithmetic operations are disabled. In such situations, you can use a `std::byte` pointer.

nullptr and Boolean Expressions

Pointers can have a special literal value, `nullptr`. Generally, a pointer that equals `nullptr` doesn't point to anything. You could use `nullptr` to indicate, for example, that there's no more memory left to allocate or that some error occurred.

Pointers have an implicit conversion to `bool`. Any value that is not `nullptr` converts implicitly to `true`, whereas `nullptr` converts implicitly to `false`. This is useful when a function returning a pointer ran successfully. A common idiom is that such a function returns `nullptr` in the case of failure. The canonical example is memory allocation.

References

References are safer, more convenient versions of pointers. You declare references with the & declarator appended to the type name. References cannot be assigned to null (easily), and they cannot be *reseated* (or reassigned). These characteristics eliminate some bugs endemic to pointers.

The syntax for dealing in references is much cleaner than for pointers. Rather than using the member-of-pointer and dereference operators, you use references exactly as if they're of the pointed-to type.

Listing 3-8 features a reference argument.

```
#include <cstdio>

struct ClockOfTheLongNow {
  --snip--
};

void add_year(ClockOfTheLongNow&❶ clock) {
  clock.set_year(clock.get_year() + 1); ❷ // No deref operator needed
}

int main() {
  ClockOfTheLongNow clock;
  printf("The year is %d.\n", clock.get_year()); ❸
  add_year(clock); ❹ // Clock is implicitly passed by reference!
  printf("The year is %d.\n", clock.get_year()); ❺
}
```

```
The year is 2019. ❸
The year is 2020. ❺
```

Listing 3-8: A program using references

You declare the `clock` argument as a `ClockOfTheLongNow` reference using the ampersand rather than the asterisk ❶. Within `add_year`, you use clock as if it were of type `ClockOfTheLongNow` ❷: there's no need to use clumsy dereference and pointer-to-reference operators. First, you print the value of `year` ❸. Next, at the call site, you pass a `ClockOfTheLongNow` object directly into `add_year` ❹: there's no need to take its address. Finally, you print the value of `year` again to illustrate that it has incremented ❺.

Usage of Pointers and References

Pointers and references are largely interchangeable, but both have tradeoffs. If you must sometimes change your reference type's value—that is, if you must change what your reference type refers to—you must use a pointer. Many data structures (including forward-linked lists, which are covered in the next section) require that you be able to change a pointer's value. Because references cannot be reseated and they shouldn't generally be assigned to `nullptr`, they're sometimes not suitable.

Forward-Linked Lists: The Canonical Pointer-Based Data Structure

A *forward-linked list* is a simple data structure made up of a series of elements. Each element holds a pointer to the next element. The last element in the linked list holds a `nullptr`. Inserting elements into a linked list is very efficient, and elements can be discontinuous in memory. Figure 3-1 illustrates their layout.

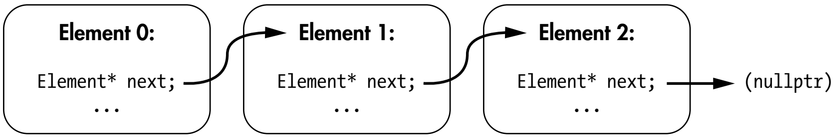


Figure 3-1: A linked list

Listing 3-9 demonstrates a possible implementation of a singly linked list element.

```
struct Element {
  Element* next{}; ❶
  void insert_after(Element* new_element) { ❷
    new_element->next = next; ❸
    next = new_element; ❹
  }
  char prefix[2]; ❺
  short operating_number; ❻
};
```

Listing 3-9: An implementation of a linked list `Element` with an operating number

Each `element` has a pointer to the `next` element in the linked list ❶, which initializes to `nullptr`. You insert a new element using the `insert_after` method ❷. It sets the `next` member of `new_element` to the `next` of `this` ❸ and then sets `next` of `this` to `new_element` ❹. Figure 3-2 illustrates this insertion. You haven't changed the memory location of any `Element` objects in this listing; you're only modifying pointer values.

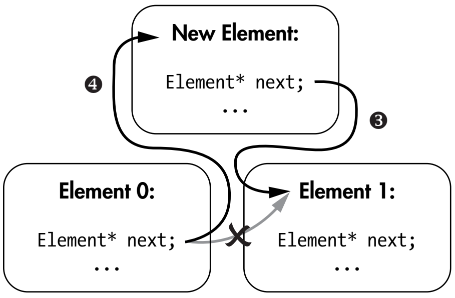


Figure 3-2: Inserting an element into a linked list

Each `Element` also contains a `prefix` array ❺ and an `operating_number` pointer ❻.

Listing 3-10 traverses a linked list of stormtroopers of type `Element`, printing their operating numbers along the way.

```
#include <cstdio>

struct Element {
  --snip--
};

int main() {
  Element trooper1, trooper2, trooper3; ❶
  trooper1.prefix[0] = 'T';
  trooper1.prefix[1] = 'K';
  trooper1.operating_number = 421;
  trooper1.insert_after(&trooper2); ❷
  trooper2.prefix[0] = 'F';
  trooper2.prefix[1] = 'N';
  trooper2.operating_number = 2187;
  trooper2.insert_after(&trooper3); ❸
  trooper3.prefix[0] = 'L';
  trooper3.prefix[1] = 'S';
  trooper3.operating_number = 005; ❹

  for (Element *cursor = &trooper1❺; cursor❻; cursor = cursor->next❼) {
    printf("stormtrooper %c%c-%d\n",
           cursor->prefix[0],
           cursor->prefix[1],
           cursor->operating_number); ❽
  }
}
```

```
stormtrooper TK-421 ❽
stormtrooper FN-2187 ❽
stormtrooper LS-5 ❽
```

Listing 3-10: A program illustrating a forward-linked list

Listing 3-10 initializes three stormtroopers ❶. The element `trooper1` is assigned the operating number TK-421, and then you insert it as the next element in the list ❷. The elements `trooper2` and `trooper3` have operating numbers FN-2187 and LS-005 and are also inserted into the list ❸❹.

The `for` loop iterates through the linked list. First, you assign the cursor pointer to the address of `trooper1` ❺. This is the beginning of the list. Before each iteration, you make sure that `cursor` is not `nullptr` ❻. After each iteration, you set `cursor` to the `next` element ❼. Within the loop, you print each stormtrooper's operating number ❽.

Employing References

Pointers provide a lot of flexibility, but this flexibility comes at a safety cost. If you don't need the flexibility of reseatability and `nullptr`, references are the go-to reference type.

Let's drive home the point that references cannot be reseated. Listing 3-11 initializes an `int` reference and then attempts to reseat it with a `new_value`.

```
#include <cstdio>

int main() {
  int original = 100;
  int& original_ref = original;
  printf("Original:  %d\n", original); ❶
  printf("Reference: %d\n", original_ref); ❷

  int new_value = 200;
  original_ref = new_value; ❸
  printf("Original:  %d\n", original); ❹
  printf("New Value: %d\n", new_value); ❺
  printf("Reference: %d\n", original_ref); ❻
}
--------------------------------------------------------------------------------
Original:  100 ❶
Reference: 100 ❷
Original:  200 ❹
New Value: 200 ❺
Reference: 200 ❻
```

Listing 3-11: A program illustrating that you cannot reseat references

This program initializes an `int` called `original` to 100. Then it declares a reference to `original` called `original_ref`. From this point on, `original_ref` will *always* refer to `original`. This is illustrated by printing the value of `original` ❶ and the value referred to by `original_ref` ❷. They're the same.

Next, you initialize another `int` called `new_value` to 200 and assign `original` to it ❸. Read that carefully: this assignment ❸ doesn't reseat `original_ref` so that it points to `new_value`. Rather, it assigns the value of `new_value` to the object it points to (`original`).

The upshot is that all of these variables—`original`, `original_ref`, and `new_value`—evaluate to 200 ❹❺❻.

this Pointers

Remember that methods are associated with classes and that instances of classes are objects. When you program a method, sometimes you need to access the *current object*, which is the object that is executing the method.

Within method definitions, you can access the current object using the `this` pointer. Usually, `this` isn't needed, because `this` is implicit when accessing members. But sometimes you might need to disambiguate—for example, if you declare a method parameter whose name collides with a member variable. For example, you can rewrite Listing 3-9 to make explicit which `Element` you're referring to, as demonstrated in Listing 3-12.

```
struct Element {
  Element* next{};
  void insert_after(Element* new_element) {
    new_element->next = this->next; ❶
    this->next ❷ = new_element;
  }
  char prefix[2];
  short operating_number;
};
```

Listing 3-12: A rewriting of Listing 3-9 using the `this` *pointer*

Here, `next` is replaced with `this->next` ❶❷. The listings are functionally identical.

Sometimes, you need `this` to resolve ambiguity between members and arguments, as demonstrated in Listing 3-13.

```
struct ClockOfTheLongNow {
  bool set_year(int year❶) {
    if (year < 2019) return false;
    this->year = year; ❷
    return true;
  }
--snip--
private:
  int year; ❸
};
```

Listing 3-13: A verbose `ClockOfTheLongNow` *definition using* `this`

The `year` argument ❶ has the same name as the `year` member ❸. Method arguments will always mask members, meaning when you type `year` within this method, it refers to the `year` argument ❶, not the `year` member ❸. That's no problem: you disambiguate with `this` ❷.

const Correctness

The keyword `const` (short for "constant") roughly means "I promise not to modify." It's a safety mechanism that prevents unintended (and potentially catastrophic) modifications of member variables. You'll use `const` in function and class definitions to specify that a variable (usually a reference or a pointer) won't be modified by that function or class. If code attempts to modify a `const` variable, the compiler will emit an error. When used correctly, `const` is one of the most powerful language features in all modern programming languages because it helps you to eliminate many kinds of common programming mistakes at compile time.

Let's look at a few common usages of `const`.

const Arguments

Marking an argument `const` precludes its modification within a function's scope. A `const` pointer or reference provides you with an efficient mechanism to pass an object into a function for read-only use. The function in Listing 3-14 takes a `const` pointer.

```
void petruchio(const char* shrew❶) {
  printf("Fear not, sweet wench, they shall not touch thee, %s.", shrew❷);
  shrew[0] = "K"; ❸ // Compiler error! The shrew cannot be tamed.
}
```

Listing 3-14: A function taking a const pointer (This code doesn't compile.)

The `petruchio` function takes a `shrew` string by `const` reference ❶. You can read from `shrew` ❷, but attempting to write to it results in a compiler error ❸.

const Methods

Marking a method `const` communicates that you promise not to modify the current object's state within the `const` method. Put another way, these are read-only methods.

To mark a method `const`, place the `const` keyword after the argument list but before the method body. For example, you could update the `ClockOfTheLongNow` object's `get_year` with `const`, as demonstrated in Listing 3-15.

```
struct ClockOfTheLongNow {
  --snip--
  int get_year() const ❶{
      return year;
  }
private:
  int year;
};
```

Listing 3-15: Updating `ClockOfTheLongNow` with `const`

All you need to do is place `const` between the argument list and the method body ❶. Had you attempted to modify `year` within `get_year`, the compiler would have generated an error.

Holders of `const` references and pointers cannot invoke methods that are not `const`, because methods that are not `const` might modify an object's state.

The `is_leap_year` function in Listing 3-16 takes a `const ClockOfTheLongNow` reference and determines whether it's a leap year.

```
bool is_leap_year(const ClockOfTheLongNow& clock) {
  if (clock.get_year() % 4 > 0) return false;
  if (clock.get_year() % 100 > 0) return true;
  if (clock.get_year() % 400 > 0) return false;
  return true;
}
```

Listing 3-16: A function for determining leap years

Had `get_year` not been marked a `const` method, Listing 3-16 would not compile because `clock` is a `const` reference and cannot be modified within `is_leap_year`.

const Member Variables

You can mark member variables const by adding the keyword to the member's type. The const member variables cannot be modified after their initialization.

In Listing 3-17, the Avout class contains two member variables, one const and one not const.

```
struct Avout {
  const❶ char* name = "Erasmas";
  ClockOfTheLongNow apert; ❷
};
```

Listing 3-17: An Avout class with a const member

The name member is const, meaning the pointed-to value cannot be modified ❶. On the other hand, apert is not const ❷.

Of course, a const Avout reference cannot be modified, so the usual rules would still apply to apert:

```
void does_not_compile(const Avout& avout) {
  avout.apert.add_year(); // Compiler error: avout is const
}
```

Sometimes you want the safety of marking a member variable const but would also like to initialize the member with arguments passed into a constructor. For this, you employ member initializer lists.

Member Initializer Lists

Member initializer lists are the primary mechanism for initializing class members. To declare a member initializer list, place a colon after the argument list in a constructor. Then insert one or more comma-separated *member initializers*. A member initializer is the name of the member followed by a braced initialization { }. Member initializers allow you to set the value of const fields at runtime.

The example in Listing 3-18 improves Listing 3-17 by introducing a member initialization list.

```
#include <cstdio>

struct ClockOfTheLongNow {
  --snip--
};

struct Avout {
  Avout(const char* name, long year_of_apert) ❶
    :❷ name❸{ name }❹, apert❺{ year_of_apert }❻ {
  }
  void announce() const { ❼
    printf("My name is %s and my next apert is %d.\n", name, apert.get_year());
  }
  const char* name;
  ClockOfTheLongNow apert;
};

int main() {
  Avout raz{ "Erasmas", 3010 };
  Avout jad{ "Jad", 4000 };
  raz.announce();
  jad.announce();
}
-----------------------------------------------------------------------
My name is Erasmas and my next apert is 3010.
My name is Jad and my next apert is 4000.
```

Listing 3-18: A program declaring and announcing two `Avout` *objects*

The `Avout` constructor takes two arguments, a `name` and the `year_of_apert` ❶. A member initializer list is added by inserting a colon ❷ followed by the names of each member you're initializing ❸❺ and braced initializations ❹❻. A `const` method `announce` is also added to print the `Avout` constructor's status ❼.

All member initializations execute before the constructor's body. This has two advantages:

- It ensures validity of all members before the constructor executes, so you can focus on initialization logic rather than member error checking.
- The members initialize once. If you reassign members in the constructor, you potentially do extra work.

NOTE *You should order the member initializers in the same order they appear in the class definition, because their constructors will be called in this order.*

Speaking of eliminating extra work, it's time to meet `auto`.

auto Type Deduction

As a strongly typed language, C++ affords its compiler a lot of information. When you initialize elements or return from functions, the compiler can divine type information from context. The `auto` keyword tells the compiler to perform such a divination for you, relieving you from inputting redundant type information.

Initialization with auto

In almost all situations, the compiler can determine the correct type of an object using the initialization value. This assignment contains redundant information:

```
int answer = 42;
```

The compiler knows answer is an int because 42 is an int.

You can use auto instead:

```
auto the_answer { 42 };           // int
auto foot { 12L };                // long
auto rootbeer { 5.0F };           // float
auto cheeseburger { 10.0 };       // double
auto politifact_claims { false }; // bool
auto cheese { "string" };         // char[7]
```

This also works when you're initializing with parentheses () and the lone =:

```
auto the_answer = 42;
auto foot(12L);
--snip--
```

Because you've committed to universal initialization with {} as much as possible, this section will say no more of these alternatives.

Alone, all of this simple initialization help doesn't buy you much; however, when types become more complicated—for example, dealing with iterators from stdlib containers—it really saves quite a bit of typing. It also makes your code more resilient to refactoring.

auto and Reference Types

It's common to add modifiers like &, *, and const to auto. Such modifications add the intended meanings (reference, pointer, and const, respectively):

```
auto year { 2019 };              // int
auto& year_ref = year;           // int&
const auto& year_cref = year;    // const int&
auto* year_ptr = &year;          // int*
const auto* year_cptr = &year;   // const int*
```

Adding modifiers to the auto declaration behaves just as you'd expect: if you add a modifier, the resulting type is guaranteed to have that modifier.

auto and Code Refactorings

The auto keyword assists in making code simpler and more resilient to refactoring. Consider the example in Listing 3-19 with a range-based for loop.

```
struct Dwarf {
  --snip--
};

Dwarf dwarves[13];

struct Contract {
  void add(const Dwarf&);
};
```

```
void form_company(Contract &contract) {
  for (const auto& dwarf : dwarves) { ❶
    contract.add(dwarf);
  }
}
```

Listing 3-19: An example using `auto` in a range-based `for` loop

If ever the type of `dwarves` changes, the assignment in the range-based `for` loop ❶ doesn't need to change. The `dwarf` type will adapt to its surroundings, in much the same way that the dwarves of Middle Earth don't.

As a general rule, use `auto` always.

NOTE *There are some corner cases to using braced initialization where you might get surprising results, but these are few, especially after C++17 fixed some pedantic nonsense behavior. Prior to C++17, using `auto` with braces `{}` specified a special object called a `std::initializer_list`, which you'll meet in Chapter 13.*

Summary

This chapter covered the two reference types: references and pointers. Along the way, you learned about the member-of-pointer operator, how pointers and arrays interplay, and `void`/`byte` pointers. You also learned about the meaning of `const` and its basic usage, the `this` pointer, and member initializer lists. Additionally, the chapter introduced `auto` type deduction.

EXERCISES

3-1. Read about CVE-2001-0500, a buffer overflow in Microsoft's Internet Information Services. (This vulnerability is commonly referred to as the Code Red worm vulnerability.)

3-2. Add a `read_from` and a `write_to` function to Listing 3-6. These functions should read or write to `upper` or `lower` as appropriate. Perform bounds checking to prevent buffer overflows.

3-3. Add an `Element* previous` to Listing 3-9 to make a *doubly linked list*. Add an `insert_before` method to `Element`. Traverse the list from front to back, then from back to front, using two separate `for` loops. Print the `operating_number` inside each loop.

3-4. Reimplement Listing 3-11 using no explicit types. (Hint: use `auto`.)

3-5. Scan the listings in Chapter 2. Which methods could be marked `const`? Where could you use `auto`?

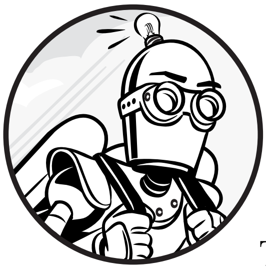
FURTHER READING

- *The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup (Pearson Education, 2013)
- "C++ Core Guidelines" by Bjarne Stroustrup and Herb Sutter (*https://github.com/isocpp/CppCoreGuidelines/*)
- "East End Functions" by Phil Nash (2018; *https://levelofindirection.com/blog/east-end-functions.html*)
- "References FAQ" by the Standard C++ Foundation (*https://isocpp.org/wiki/faq/references/*)

4

THE OBJECT LIFE CYCLE

Things you used to own, now they own you.
—Chuck Palahniuk, Fight Club



The object life cycle is the series of stages a C++ object goes through during its lifetime. This chapter begins with a discussion of an object's storage duration, the time during which storage is allocated for an object. You'll learn about how the object life cycle dovetails with exceptions to handle error conditions and cleanup in a robust, safe, and elegant way. The chapter closes with a discussion of move and copy semantics that provides you with granular control over an object's life cycle.

An Object's Storage Duration

An *object* is a region of storage that has a type and a value. When you declare a variable, you create an object. A variable is simply an object that has a name.

Allocation, Deallocation, and Lifetime

Every object requires storage. You reserve storage for objects in a process called *allocation*. When you're done with an object, you release the object's storage in a process called *deallocation*.

An object's *storage duration* begins when the object is allocated and ends when the object is deallocated. The *lifetime* of an object is a runtime property that is bound by the object's storage duration. An object's lifetime begins once its constructor completes, and it ends just before a destructor is invoked. In summary, each object passes through the following stages:

1. The object's storage duration begins, and storage is allocated.
2. The object's constructor is called.
3. The object's lifetime begins.
4. You can use the object in your program.
5. The object's lifetime ends.
6. The object's destructor is called.
7. The object's storage duration ends, and storage is deallocated.

Memory Management

If you've been programming in an application language, chances are you've used an *automatic memory manager*, or a *garbage collector*. At runtime, programs create objects. Periodically, the garbage collector determines which objects are no longer required by the program and safely deallocates them. This approach frees the programmer from worrying about managing an object's life cycle, but it incurs several costs, including runtime performance, and requires some powerful programming techniques like deterministic resource management.

C++ takes a more efficient approach. The trade-off is that C++ programmers must have intimate knowledge of storage durations. It's *our* job, not the garbage collector's, to craft object lifetimes.

Automatic Storage Duration

An *automatic object* is allocated at the beginning of an enclosing code block, and it's deallocated at the end. The enclosing block is the automatic object's *scope*. Automatic objects are said to have *automatic storage duration*. Note that function parameters are automatic, even though notationally they appear outside the function body.

In Listing 4-1, the function `power_up_rat_thing` is the scope for the automatic variables `nuclear_isotopes` and `waste_heat`.

```
void power_up_rat_thing(int nuclear_isotopes) {
  int waste_heat = 0;
  --snip--
}
```

Listing 4-1: A function with two automatic variables, `nuclear_isotopes` and `waste_heat`