Managing Memory & Low-Level Data Structures

Laurent Mathy

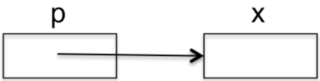
Object-Oriented Programming Projects

April 20, 2020

Pointers

A **pointer** is a value that represents the **address** of an object in memory.

If you can access an object, you can access its address, and vice-versa.



Address operator: `&x` is the address of `x`.
Do not confuse with `&` to define reference to types.

Dereference operator: `*p` is the value pointed to by `p`.
You can think of pointer as an iterator.

Pointers (2)

Pointers are built-in types:

- Default-initialisation: garbage.
- Value-initialisation: `0` (a.k.a. *null* pointer).
  - `0` is only integer that can be converted to a pointer.
  - Only pointer value guaranteed to be distinct from a pointer to any object.

Type of address of object of type `T` is `T*` (pointer to T). *E.g.*

- `int *p; // *p has type int`
  `*p` is a **declarator**: part of the definition of a single variable.
- `int* p; // p has type int*`
  Identical to previous declaration.
- `int* p, q; // What does this declare?`

Pointers (2)

Pointers are built-in types:

- Default-initialisation: garbage.
- Value-initialisation: `0` (a.k.a. *null* pointer).
  - `0` is only integer that can be converted to a pointer.
  - Only pointer value guaranteed to be distinct from a pointer to any object.

Type of address of object of type `T` is `T*` (pointer to T). *E.g.*

- `int *p; // *p has type int`
  `*p` is a **declarator**: part of the definition of a single variable.
- `int* p; // p has type int*`
  Identical to previous declaration.
- `int* p, q; // What does this declare?`
  `p` is `int *`, but `q` is `int` $\implies$ avoid multiple declarations.

C++11 **nullptr**

In C++11, `nullptr` replaces `0` (and `NULL`) for null pointers.

Avoids confusion with `int`. *E.g.*

```
void f(int i) { cout << "i = " << i << endl };

void f(char *s) { cout << "s = " << s << endl };

void difficult_choice() {
    // Should I call f(int), or f(char *)?
    f(0);    // Compiler error, ambiguous
    f(NULL); // Idem
}

void trivial_choice() {
    // Calling f(char *) confidently
    f(nullptr);
}
```

Pointers: simple example

```
int main() {
    int x = 5;

    // `p` is a pointer to `x`, holds the address of `x`
    int *p = &x;
    cout << "x = " << x << endl;

    // Change the value of `x` through `p`
    *p = 6;
    cout << "x = " << x << endl;

    return 0;
}
```

Output will be:

```
x = 5
x = 6
```

Pointers to functions

A program can only do two things with a function:

- call it;
- take its address.

But we passed functions as argument to other functions...

Pointers to functions

A program can only do two things with a function:

- call it;
- take its address.

But we passed functions as argument to other functions...
In this case, the compiler quietly passed a **function pointer** instead of the function itself.

Once you have dereferenced a pointer to a function, all you can do with the resulting function is:

- call it;
- take its address, again.

Pointers to functions (2)

```
int (*fp)(int);
```

When `fp` is dereferenced, you get a function that takes an **`int`** as argument and returns an **`int`**.

Because *all* you can do with a function is either call it or take its address:

- any use that is not a call, is assumed to be taking the address, *even without an explicit* `&`;
- you can call a function via a pointer *without* dereferencing the pointer.

Pointers to functions: example

```c
static int next(int n) {
    return n + 1;
}

int main() {
    int i = 0;
    int (*fp)(int);

    // These two statements are equivalent
    fp = &next;
    fp = next;

    // And these two are equivalent too
    i = (*fp)(i);
    i = fp(i);

    assert(i == 2);

    return 0;
}
```

Simplified notation for functions as parameters

```
// C notation
vector<int> map_c(int (*f)(int),
                  const vector<int>& xs);

// C++ notation
vector<int> map_cpp(int f(int),
                    const vector<int>& xs);
```

In C++, the same notation used for declaring a function can be used in a parameter type.

Built-in arrays

- Built-in arrays are a kind of container, but part of the core language (not to be confused with similar `std::array` introduced by C++11).
- Contains objects all of the same type.
- Array size must be known at compile time (no growing or shrinking).
- They have no member.
- `size_t` (in `<cstddef>`) is type to represent size of array.
- The name of an array represents a **pointer to first element** of the array.

Example definition:

```
const size_t N_DIMS = 10;
double coords[N_DIMS];
```

Constant expressions for array size

Built-in array size must be known at **compile time**.

C++98: const + macros

```
#define IMAGE_SIZE(width, height, n_channels, depth) \
    (width * height * n_channels \
     * ((depth % 8 == 0) ? (depth / 8) : (depth / 8 + 1)))

static uint8_t a98[IMAGE_SIZE(1980, 720, 3, 8)];
```

C++11: constexpr

```
constexpr size_t image_size(size_t width, size_t height,
                            size_t n_channels, size_t depth) {
    size_t bytes_per_pixel =
        (depth % 8 == 0) ? (depth / 8) : (depth / 8 + 1);
    return width * height * n_channels * bytes_per_pixel;
}

static uint8_t a11[image_size(1980, 720, 3, 8)];
```

Pointer arithmetic and arrays

- A pointer is a random-access iterator.
- If `p` points to the m^{th} element of an array then
  - `p + n` points to the $(m+n)^{\text{th}}$ element of the array.
  - `p - n` points to the $(m-n)^{\text{th}}$ element of the array.
- Pointers inside or one-past the end of an array are *valid*, but **one-past the end** pointer can be used **only** for comparison.
- All bets are off if you dereference a pointer not pointing into the array.
- If `p` and `q` are pointers to elements of same array, `p - q` is integer distance between these elements:
  - `(p - q) + q == p`.
  - `p - q` is signed integer of type `ptrdiff_t` (defined in `<cstddef>`).

Indexing, initialization and string literals

`p[n]` is equivalent to `*(p + n)`.

```
const int month_lengths[] = {
    31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31
};
```

String literals are *anonymous*, null-terminated (`'\0'`), arrays of `const char`s.

See `<cstring>` for functions to manipulate null-terminated `char` arrays.

Example: find_if implementation

```
template<class In, class Pred>
In find_if(In begin, In end, Pred f) { // Note `f` type
    while (begin != end && !f(*begin)) // Note `f` call
        ++begin;
    return begin;
}

static bool is_two(int i) { return i == 2; }

int main() {
    int a[] = { 1, 2, 3 };
    // `a` is pointer to first element, i.e. &a[0]
    assert(find_if(a,        &a[3],  is_two) == &a[1]);
    assert(find_if(begin(a), end(a), is_two) == &a[1]);
    return 0;
}
```

begin and end are defined in <iterator>. They return iterators to the beginning and end of a container.

Arguments to `main`

Same as in C:

```
int main(int argc, char* argv[]) {
    for (int i = 1; i < argc; ++i) {
        cout << "Arg " << i << ": "
             << argv[i] // `argv[i]` is a `char *`
             << endl;
    }
    return 0;
}
```

- `argv[0]` is program name.

Files

Standard error:

- `cerr` unbuffered error stream.
- `clog` buffered error stream.

`<fstream>`

- `ifstream` class to represent input files.
- `ofstream` class to represent output file.
- Can use `ifstream` where `istream` would be used.
- Can use `ofstream` where `ostream` would be used.
- `fstream` constructors take pointer to null-terminated char array as file name
  $\Longrightarrow$ use `c_str()` member of `string` to use it.
  Since C++11, you can use a `std::string` directly.

File example: cp

```
ifstream in("in"); // "in" has type const char *
if (!in) {
    cerr << "Could not open file 'in' for reading!"
         << endl;
    return 1;
}

ofstream out("out");
if (!out) {
    cerr << "Could not open file 'out' for writing!"
         << endl;
    return 1;
}

string s;
while (getline(in, s))
    out << s << endl;

return 0;
```

File example: cat

```
int main(int argc, char *argv[]) {
    int fail_count = 0;
    // For each file in the input list
    for (int i = 1; i < argc; ++i) {
        ifstream in(argv[i]);
        // If it exists, write its contents.
        // Otherwise generate an error message.
        if (in) {
            string s;
            while (getline(in, s))
                cout << s << endl;
        } else {
            cerr << "Cannot open file " << argv[i] << endl;
            ++fail_count;
        }
    }
    return fail_count;
}
```

Memory management: automatic and static

Local variables are allocated when encountered.
Destroyed at end of block where defined.

```
int* invalid_pointer() {
    int x;
    return &x; // Don't do this at home!
}
```

Static variables are created on first use (or before) and live until the end of the program.

```
int* pointer_to_static() {
    static int x;
    return &x; // This is (somewhat) fine
}
```

Memory management: dynamic

If `T` is object type, `new T` allocates a default-initialized object and returns pointer to it.

`new T(val)` initializes the object to value `val`.

Objects so created lives until:

- end of program;
- `delete p` where `p` is pointer to object created by `new`:
  - `p` becomes invalid pointer;
  - deleting 0 has no effect;
  - deleting `p` twice is disastrous!

```cpp
int* p = new int(42);
++*p;     // *p is 43
delete p; // RIP p

int* pointer_to_dynamic() {
    return new int(0); // Caller is now responsible for cleanup
}
```

Memory management: arrays

new T[n] array of n default-initialised elements.

delete[] p deallocates a dynamic array.

Arrays with zero elements are permitted – simplifies code

- in this case **new** returns valid *off-the-end* pointer.

```
// Works fine even if n is zero
T* p = new T[n];
vector<T> v(p, p + n);
delete[] p;
```

```
char* duplicate_chars(const char* p) {
  size_t length = strlen(p) + 1; // `strlen` does not count '\0'
  char* result = new char[length];
  copy(p, p + length, result);
  return result;
}
```

Avoid new and delete in modern C++

Ownership is not explicit which is very **error-prone**.

```
char* get_a_pointer();

char* p = get_a_pointer();
```

Should I `delete p`? Should I copy its value?

What to do then?

Avoid new and delete in modern C++

Ownership is not explicit which is very **error-prone**.

```
char* get_a_pointer();

char* p = get_a_pointer();
```

Should I `delete p`? Should I copy its value?

What to do then?

- Hide `new` and `delete` in **proxy** classes:
  - A `vector` is only 24 bytes on my machine.
  - Can be copied easily.
  - Will keep track of backing buffer, and free it when vector goes out of scope.
  - Requires overriding assignment and copy constructors (TBD).

Avoid **new** and **delete** in modern C++

Ownership is not explicit which is very **error-prone**.

```
char* get_a_pointer();

char* p = get_a_pointer();
```

Should I `delete` p? Should I copy its value?

What to do then?

- Hide `new` and `delete` in **proxy** classes:
  - A `vector` is only 24 bytes on my machine.
  - Can be copied easily.
  - Will keep track of backing buffer, and free it when vector goes out of scope.
  - Requires overriding assignment and copy constructors (TBD).
- Or better: use C++11 **smart pointers**.

unique_ptr for exclusive ownership

- It represents **exclusive ownership**.
- Very light-weight wrapper, no performance cost.
- Used like a regular pointer.
- Defined in <memory>.
- Don't use auto_ptr, which has problems and is deprecated.

```
unique_ptr<int> p1(new int(42));
unique_ptr<int> p2 = p1; // Error: cannot copy unique pointer
unique_ptr<int> p3 = move(p1);
// p1 is now nullptr, and should not be used anymore
// Memory will be released when p3 goes out of scope

// Safer and cleaner alternative with C++14
auto p = make_unique<int>(42);

unique_ptr<char> get_a_pointer();
// Caller becomes owner, and compiler will delete automatically
```

`share_ptr` allows shared ownership

- It uses **reference counting** to know when to delete the pointed-to object.
- Always use `make_shared` to create shared pointers (also in C++11).
- You can use `weak_ptr` to break cycles. A `weak_ptr` keeps a reference to the object, but won't prevent deletion.
- When using a `weak_ptr`, call `lock()` to transform it into a `share_ptr` (avoid premature deletion).

Modern C++ avoids `new`/`delete`

The smart pointers can replace most, if not all use cases for explicit `new` and `delete`.