Dealing with pointers in modern C++

Who am I?

- Alexander Hirsch
    - E-Mail: 1zeeky@gmail.com
    - Jabber/XMPP: z33ky@jabber.ccc.de
- Computer Science student at TU Darmstadt
- hobby-programmer
- looked into many languages and programming paradigms
- most experience with C++ (started getting proficient around 2007)

Disclaimer

What I state as facts might actually be untrue (though they are to my knowledge).

What I present as something "you should do", might not actually me a common idiom and just my personal opinion.

Examples are not necessarily in good coding style. They are just intended to show specific features and hint at use cases.

They should be valid C++, apart from missing `#include`s and some code should be inside a function, albeit it not being shown this way. Except of course where noted that the example would produce an error.

A shout out

A shout out to these websites I find very helpful for getting information about C++:

- `https://en.wikipedia.org/wiki/C%2B%2B11`
- `http://cppreference.com`
- `https://cppandbeyond.com`
- `https://stackoverflow.com`
- `https://isocpp.org`
- `http://meetingcpp.com`

The Standard document is obviously also a very good source. The (free) draft is mostly fine, too.

Dealing with pointers in modern C++

Introduction: Pointers

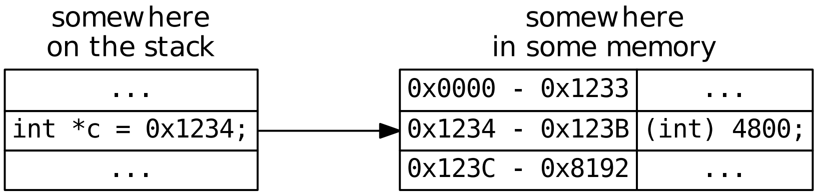
what, why and how

Stack & Heap

- Stack $\rightarrow$ program stack
  - local variables
  - function arguments
  - return address
- Heap $\rightarrow$ "memory pool"
  - dynamic lifetime
  - large data-structures (prevent stack overflow)
  - objects, whose size is only known at run-time (data hiding, polymorphism)

Pointers

Many people seem to have trouble **understanding what pointers are**.



Why do we need pointers?

Stack addressing done via offset to frame-pointer (done by compiler).
$\rightarrow$ Heap addressing
Other use-cases:

- optional arguments/return values
- returning or passing big objects as arguments without copy

Allocating objects

Stack

```
int  a;
char b[4];
a = 42;
b[0] = '0';
```

detail: Arrays implemented as pointers to the first element $\rightarrow$ `a[i]` is syntactic sugar for `*(a + i)`

Heap

```
int  *a   = new int;
char  b[] = new char[4];
//wrong: a = 42; -> set address to 42
//       C++'s type system would also disallow this
//       without explicit casting
*a = 42;
//note: same as in stack example
b[0] = '0';
```

`int` and `char[]` on the Heap, pointers on the Stack

Deallocating objects

Stack

```
{
    int  a;
    char b[4];
    //a and b implicitly popped off the stack
}
```

Heap

```
{
    int  *a   = new int;
    char  b[] = new char[4];
    delete   a;
    delete[] b;
    //a and b implicitly popped off the stack
}
```

The problem with pointers

So, what's the **problem** with pointers?
How do we need to "deal" with them?

- don't forget to **delete** your pointer
  - operator delete?
  - free(void*)?
  - DestroyObject(Object*)?
- but also only do it **once**
- delete your pointer in **all paths**, also when exceptions are thrown

The problem with pointers

Am I even responsible for deleting the pointer?

- `SDL_Window* SDL_CreateWindow(...)`[1]
  → you can: `SDL_DestroyWindow`
- `char** PHYSFS_enumerateFiles(...)`[2]
  → yes: `PHYSFS_freeList`
- `GList* g_list_reverse(...)`[3]
  → kind of: `g_list_reverse` operates in-place
- `char* strdup(...)`[4]
  → yes, `free`
- `char* strtok(...)`[4]
  → no, it uses the memory you passed in

[1] SDL 2.0.3, http://www.libsdl.org
[2] PhysicsFS 2.0.3, https://icculus.org/physicsfs/
[3] GLib 2.38.2, https://developer.gnome.org/glib
[4] standard C library (C99)

Avoiding Pointers

Avoid dealing with pointers
by avoiding them in the first place

Avoiding pointers - avoid `container<T*>`

I've seen containers like `std::vector<T*>` or `std::list<T*>` for no reason.
`std::vector<T>` and `std::list<T>` usually work just as well.

Reasons you really do want a `T*` container:

- polymorphism (consider intrusive lists → less indirection)
- need to copy the elements, but `T` is non-copyable or very expensive to copy (consider Ranges[1])
- container of references to existing objects

Tip: `boost::ptr_vector` (or `boost::ptr_list`, or ...); handles deletion, supports (customizable) deep-copies

[1]e.g. Boost[2].Range

[2]reputable, peer-reviewed collection of C++ libraries for general and various specific use cases

Avoiding pointers - lvalue-references

A reference is an alias to another object.
Compared to pointers:

- cannot be reassigned (assignment assigns to variable, not reference)
- there is no "`nullptr`-reference"
- semantics: you refer to an object (but don't own it)

```
void byCopy(int  a) {  a = 1; }
void byPtr (int *a) { *a = 2; }
void byRef (int &a) {  a = 3; }
int main() {
    int i = 0;
    byCopy(i); //i still 0
    byPtr(&i); //i now 2
    byRef(i);  //i now 3
}
```

Note: `T&`-containers (also `T&`-arrays) are impossible.
Consider `std::reference_wrapper`.

Avoiding pointers - rvalue-references vs lvalue-references

lvalue:	non-temporary object
	`x` in `int x;`
(p[1])rvalue	a temporary
	the result of `x + y`

With lvalue-references, you create
an **alias for something already named**.

With rvalue-references, you create
a **name for something unnamed**.

[1] pure

Avoiding pointers - rvalue-references

When you take a rvalue-reference you **extend the lifetime** of the temporary.
That also means, that a rvalue-reference is a lvalue.
When the rvalue-reference goes out of scope, the destructor of the referenced object is called.
This is not the case for lvalue-references, for which it would be disastrous since the reference is just an alias to something existing somewhere else.

Avoiding pointers - rvalue-references

When we copy an object, we must **duplicate all its data** (think copy-constructor).

If we knew that the object we're copying is a **temporary** that won't be used anymore, we could **cannibalize** it.

An rvalue-reference tells us that it references a temporary.

Avoiding pointers - move constructor

A constructor taking an rvalue-reference of the type that is constructed is called **move-constructor**.

It is called that, because the process of making a rvalue from a lvalue is called **moving**.

The utility function `std::move` was created for this.

Avoiding pointers - rvalue-references examples

```
struct BigMatrix
{
    //std::vector already supports rvalue-references,
    //that wouldn’t be interesting
    int *Data;
};

//move constructor taking a rvalue-reference as argument
BigMatrix::BigMatrix(BigMatrix &&that)
     //no need to allocate our own memory
    :Data(that.Data)
{
    //make sure that.~BigMatrix() doesn’t free our data!
    that.Data = nullptr;
}

BigMatrix::~BigMatrix()
{
    delete[] Data;
}
```

Avoiding pointers - rvalue-references examples cont'

```
//don't get the idea to return a rvalue-reference
BigMatrix operator*(const BigMatrix &a, const BigMatrix &b)
{
    BigMatrix result;
    ...
    //std::move makes a rvalue-reference
    //from a lvalue-reference
    return std::move(result);
}

int main()
{
    BigMatrix a(...), b(...);
    //move-constructed (or not with compiler optimization)
    BigMatrix c(a * b);
    //will not be move (or copy) constructed, uses temporary
    BigMatrix &&d = a * b;
}
```

Avoiding pointers - avoid C-style arrays

Prefer `std::array` or `std::vector` to C-style arrays
(`T array[size]; T *array = new T[size];`)

- `std::array` → compile-time known sizes
  beware of stack-overflow
- `std::vector` → dynamic sizable

Raw access (`T*`):

- via `data()`
- elements guaranteed to be stored continuously → interface with C-style APIs

Avoiding pointers - avoid C-style arrays - benefits

You can easily switch the underlying data-structure and usually not worry about anything more.
E.g. vector, list, deque

You can get bounds-checking.

STL containers provide a convenient interface for various operations, including

- adding and removing elements
- resizing or pre-allocating (reserving) memory
- copying the container

Some containers won't provide the functions that don't make sense for them (e.g. resizing `std::array`).
The shared interface is defined in various C++ Container concepts.

Avoiding pointers - avoid C-style arrays - benefits

Algorithms designed for usage with the container-interface also implemented by `std::vector`, like

- sorting
- set operations
- transformations
- creating partial copies

You can use the range-based `for` loop.
Note: Stack allocated C-style arrays work too,
but heap allocated ones do not.

Avoiding pointers - avoid C-style arrays examples

```cpp
std::vector<int> iv = { 1, 2, 3, 4, 5 };
int *ip = iv.data();
assert(ip[0] == 1);
++ip[0];
assert(iv[0] == 2);

std::vector<int> cv(ip, ip + 5);
assert(std::equal(iv, cv));
assert(iv.data() != cv.data());

std::vector<int> mv(std::move(iv));
assert(mv.data() == ip);
assert(iv.size() == 0);
```

Avoiding pointers - (Array TS[1]) `std::dynarray`

Post C++14 `std::dynarray`:

- size set at run-time
- (unlike `std::vector`) don't grow or shrink after that
- either on stack or heap (cannot specify manually)

Provides similar interface to `std::array` and `std::vector` for raw access to continuous data.

[1] Technical Specification

Avoiding pointers - `boost::optional`

`boost::optional`[1]

- like `option` in Standard ML and OCaml
- or like `Maybe` in Haskell

- `boost::optional` either has a value or is uninitialized
- uninitialized state is a well defined state (like a `nullptr`)
  invalid access leads to undefined behavior

Usage for optional arguments and return values.

[1] `std::optional` in Library Fundamentals TS

Avoiding pointers - `boost::optional`

```
boost::optional<int> a;
assert(!a);
a = 5;
assert(a && *a == 5);
a.reset(); //same as a = boost::none;
assert(!a);
```

Smart pointers

Wrapping pointers in a safer shell

Smart pointers

Smart pointers are classes that **wrap around a pointer**-member. By carrying **ownership-semantics**, they enable implicit release of resources (like memory).

C++98 has `std::auto_ptr`. I've rarely seen it in the wild.

C++TR1 suggested `std::shared_ptr` and `std::weak_ptr`. They got in C++11.

C++11 replaced `std::auto_ptr` with `std::unique_ptr`, which makes use of new language features.

Boost has `boost::intrusive_ptr`.

`std::shared_ptr`

`std::shared_ptr` is a **reference-counting** pointer.
Only reference-counting is thread-safe.

- start with count $==$ 1
- when copied, $++$count
- when destroyed, --count
- when count $==$ 0, delete pointer “automatically”

Default delete via `operator delete` (or `operator delete[]` for arrays), but you can specify a custom deleter.

std::shared_ptr examples

```
//make_shared creates a shared_ptr
auto i = std::make_shared<int>(5);
assert(i.unique() && i.use_count() == 1);
auto j = i;
assert(!i.unique() && i.use_count() == 2);
assert(i == j);
auto k = std::make_shared<int>(*i);
i.swap(k);
assert(i.unique());
assert(j == k);
```

```
auto window = std::shared_ptr<SDL_Window>(
    SDL_CreateWindow(...), &SDL_DestroyWindow);
//shared_ptr::get() obtains a raw pointer
SDL_GetWindowID(window.get());
//shared_ptr::reset deletes the pointer
window.reset();
assert(!window);
```

```
auto dup = std::shared_ptr<const char>(strdup(...), &free);
```

`std::weak_ptr`

The `std::weak_ptr` is a **weak reference** to a `std::shared_ptr`.

- does not modify reference-count
- still refers to an object
- in contrast to raw pointer: can know if object was freed (by `std::shared_ptr`)

To operate on an object a `std::weak_ptr` is pointing to, you must create a `std::shared_ptr` from it.
This way you have the guarantee, that the object remains valid during your operation.

std::weak_ptr examples

```
auto i = std::make_shared<int>(23);
auto weak_i = std::weak_ptr<int>(i);

//weak_ptr can also check the reference-count
assert(weak_i.use_count() == 1);
//lock() obtains a shared_ptr
assert(weak_i.lock() != nullptr);

*weak_i.lock() *= -1;
assert(*i == -23);

i.reset();
assert(weak_i.use_count() == 0
    && weak_i.expired()
    && !weak_i.lock());
```

`std::make_shared` and `std::allocate_shared`

`std::make_shared` is **more efficient** than using the `std::shared_ptr(T*)` constructor,
because it can allocate the reference-count right next to the `T`.
`std::make_shared` uses `operator new` and `operator delete`,
but you can supply a **custom Allocator**[1] with `std::allocate_shared`.

Note that when the `std::shared_ptr(T*)` constructor is used, the object is destroyed when the **last `std::shared_ptr` pointing to it** gets destroyed,
while `std::make_shared` (and `std::allocate_shared`) also require **no `std::weak_ptr`** to be referring to it.

[1]template-class with functions to allocate memory and optionally customizable construction and destruction behavior

`std::unique_ptr`

A `std::unique_ptr` is (should be) the **only owner** of an object.

- cannot be copied
- **can** be moved
- object freed when pointer goes out of scope

One can optionally supply a custom deleter, like with `std::shared_ptr`.

std::unique_ptr examples

```
//note: make_unique is coming in C++14;
//missing in C++11 (library defect)
auto i = std::unique_ptr<int>(new int(5));
//auto j = i; //not possible (cannot copy)
auto j = std::move(i);
assert(*j == 5 && !i);
```

```
auto window = std::unique_ptr<SDL_Window>(
    SDL_CreateWindow(...), &SDL_DestroyWindow);
//unique_ptr::get() obtains a raw pointer
SDL_GetWindowID(window.get());
//unique_ptr::reset deletes the pointer
window.reset();
//window no longer manages an object,
//so the following assertion holds
assert(!window);
```

```
auto dup = std::unique_ptr<char*>(strdup("abcdefg", &free);
//unique_ptr::release releases ownership
free(dup.release());
assert(!dup);
```

`boost::intrusive_ptr`

A `boost::intrusive_ptr` ties smart pointer interface into **existing reference-counting** mechanisms.

- basic workings like `std::shared_ptr`
- doesn't provide its own reference-count,
- provides hooks instead

This is mostly for interfacing with embedded reference counting. Note that intrusive reference counting can have performance and memory usage advantages compared to just using `std::shared_ptr`.

boost::intrusive_ptr examples

```
//automatically picked up by Boost, selected via signature
void intrusive_ptr_add_ref(GMappedFile *p)
{
    g_mapped_file_ref(p);
}
void intrusive_ptr_release(GMappedFile *p)
{
    g_mapped_file_unref(p);
}

//...

//false -> don't increase ref count (default true)
auto file = boost::intrusive_ptr<GMappedFile>(
    g_mapped_file_open(...), false);
auto file_copy = file;
file.reset();
assert(!file && file_copy);
```

The `boost::intrusive_ptr` does not introduce thread-safety.
Hooks are not called with a `nullptr`.

`std::smart_ptr(new T())` - beware of memory leak

```
f(std::smart_ptr<T>(new T()), g());
```

Compiler might generate something equivalent to this:

```
auto temp0 = new T();
auto temp1 = g();
auto temp2 = std::smart_ptr<T>(temp0);

f(temp2, temp1);
```

If `g()` throws an exception, you have a memory leak.
Using `std::make_shared` or `std::make_unique` would solve this problem.

Recap

- Avoiding pointers
  - `std::array`, `std::vector` and `std::dynarray`$^{\text{(Array TS)}}$ instead of C-style arrays
  - avoid `container<T*>`: store plain `T`, consider `boost::ptr_container` or intrusive lists
  - references instead of pointers to avoid copies
  - `boost::optional` (`std::optional`$^{\text{(Library Fundamentals TS)}}$) for optional values (instead of `nullptr`)
- Smart pointers
  - `std::shared_ptr` and `std::weak_ptr` - shared ownership
  - `std::unique_ptr` - single ownership
  - `boost::intrusive_ptr` - wrap existing reference-counting