C++ *fun*damentals

A probably, not so common, look at fundamentals.

Peter Lorimer

Common Fundamentals

- Pointers
- Syntax
- Memory Management
- OOP
- STL
- Templating

Basically, those things that differ from other higher level languages.

Disclaimer

I'm calling this talk ***a not so common look at fundamentals***. My set of fundamentals focuses more on design, than language specifics. Where I still believe that understanding language features and the standard template library are foundational to C++ development. I feel that one can understand those, and still write incredibly bad C++ code. This set of fundamentals aims to be applied to any size application successfully, and allow it to scale as it inevitably grows. They'll help you write *good* C++, regardless of the size of your application.

So if you're looking for an esoteric language tutorial, you won't get that. You have my advanced apologies.

Also, I apologize for my code colour scheme. ☺

- PL

Disclaimer to the Disclaimer

By no means am I stating that this is the ***only*** way to write good code. I'm also not stating that following it to a tee means your code will be good.

We need to use our critical thinking and judgement first and foremost. There is no silver bullet.

This is merely a collection of ideas from many people in the C++ world, who agree. When these practices are applied to code, it is safer, faster, and more easy to reason about.

- PL

Peter's *fun*damentals

1. Responsibility first
2. Ownership and lifetime
3. Call signatures
4. Error handling
5. *Templates (Generic Programming)*

The goal for my fundamentals.

Enable any level programmer to write simple, safe, clean, fast code, requiring minimal documentation.

0: Responsibility first.

We want to design components that are self-contained: independent, and with a single, well-defined purpose

— The Pragmatic Programmer

Why does this matter in C++

- Comprehensibility and maintainability.
- Compile times & potential optimization.
- Component reusability.

Coupling & Cohesion

“We strive for low coupling, and high cohesion.”

“We want highly cohesive classes, that are loosely coupled.”

“Let’s write tight code.”

You had 1 job!

- A job for everything, and everything has one job.

What’s that smell?

- Doing more than one thing. Doing nothing*.
- Large amounts of state.
- Lots of dependencies.
- Thousand line functions.

Scenario

- See interpreter0.cpp

1: Ownership & Lifetime

You should be able to explain, who owns what object, and how long it lives.

Why are these important in C++?

- To write clean code, we need to be able to reason about ownership.
- To write fast code, we need the compiler to be able to reason about lifetime.
- To write safe code, we need to be able to reason about both.
- It’s very hard to retrofit these two things “after”.

What is an Object in C++

From Jason Turner's 'Surprises in Object Lifetime' & the C++ standard.

"An object is a type (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void."

What is Object Lifetime?

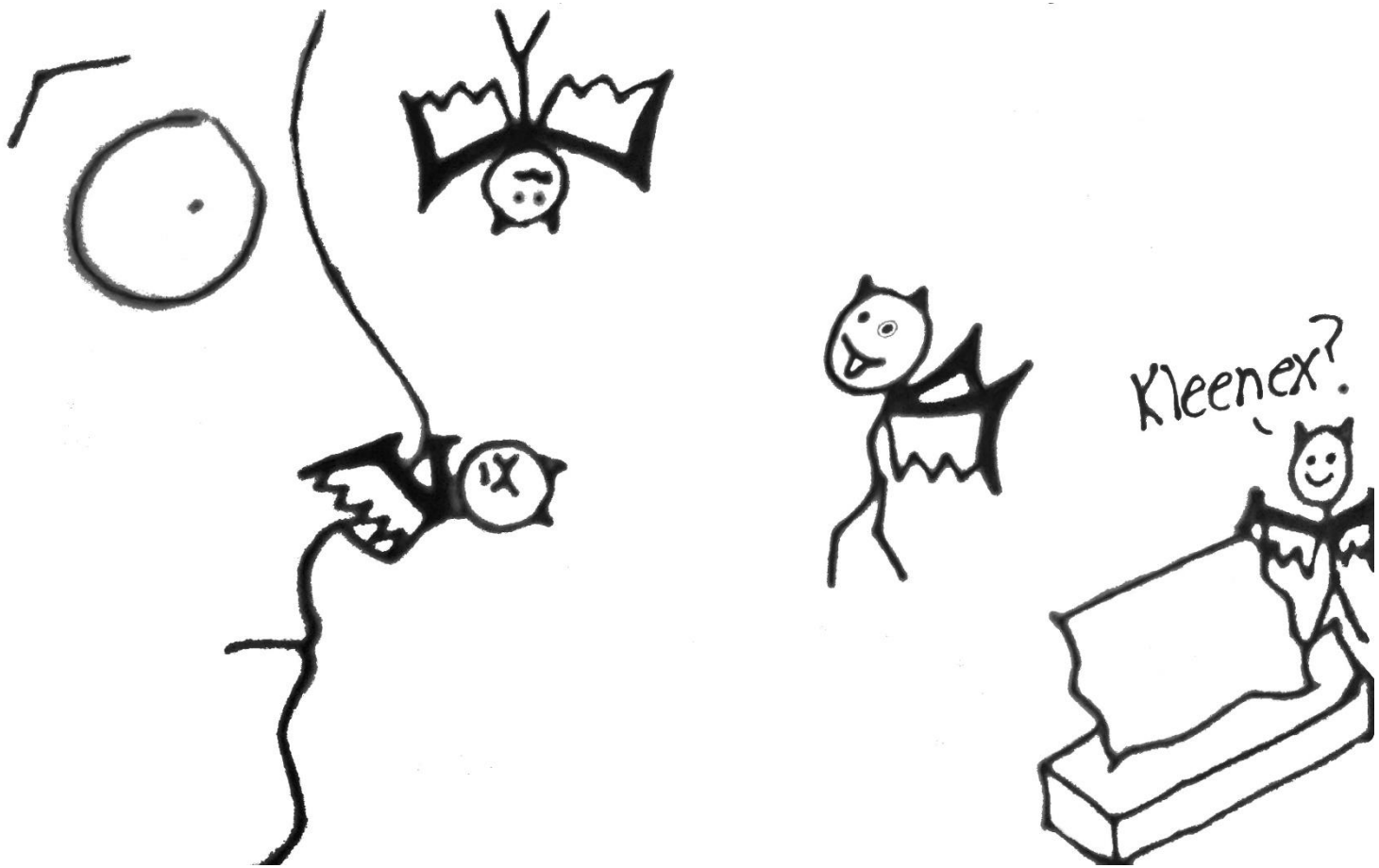
From the time an object is constructed, to the time it is destructed.*

*See Jason Turner’s talk & C++ standard for a better explanation.

Why should I care about lifetime?

```
const int & get_value()
{
    int value = 5;
    return value;
}
```

Undefined Behaviour



```
struct foo
{
    void bar();
};

struct foo_factory
{
    static foo* make_foo();
    static void free_foo(foo *f);
};
```

```
// What could go wrong?
void can_something_go_wrong()
{
    foo *f = new foo();

    f->bar();

    delete f;
}
// Mustn't forget to free
void who_does_what()
{
    foo * f = foo_factory::make_foo();
    if(!f->bar())
        return;

    foo_factory::free_foo(f);
}
```


Memory Leaks!!

RAII

- Resource Allocation Is Initialization

```
class SafeFoo
{
private:
    foo *f_;
public:

    SafeFoo()
    {
        f_ = foo_factory::make_foo();
    }

    ~SafeFoo()
    {
        foo_factory::free_foo(f_);
    }

    SafeFoo(const SafeFoo &)=delete;
    SafeFoo& operator=(const SafeFoo&)=delete;

    SafeFoo(SafeFoo &&)=delete;
    SafeFoo& operator=(SafeFoo&&)=delete;

    void bar()
    {
        f_->bar();
    }
};
```

1st note on “Self Documenting” code

RAII removes the need to document allocation and deallocation.

Composition over Inheritance

```
class helper_base
{
protected:
    void help_me_obiwan();
    int help_me_oprah();
};

class concrete_1 : public helper_base
{
public:
    void compute_something()
    {
        if(help_me_oprah() == 0)
        {
            help_me_obiwan();
        }
    }
};
```

```
class helper_base
{
protected:
    void help_me_obiwan();
    int help_me_oprah();
};


class concrete_1
{
    helper_base helper_;

public:
    void compute_something()
    {
        if(helper_.help_me_oprah() == 0)
        {
            helper_.help_me_obiwan();
        }
    }
};
```

Dependency Injection

```
class concrete_2
{
    helper_base &helper_;

public:
    concrete_2(helper_base &helper)
        :helper_(helper)
    {
    }

};
```

Signatures: They say a lot

```
// by value
void call_me_maybe(foo f);

// by pointer
void call_me_maybe(foo *f);

// by non-const reference
void call_me_maybe(foo& f);

// by rvalue reference
void call_me_maybe(foo &&f);

// by const pointer
void call_me_maybe(const foo *f);

// by const reference
void call_me_maybe(const foo &f);

// by const value
void call_me_maybe(const foo f);
```

In / Out Parameters

- An in parameter… is a read only parameter, used an input into a function.
- An out parameter… is used as a target for a function to put data into.

In - Pass by reference to const or by value

```
// by const reference
void call_me_maybe(const foo &f);
// by value
void call_me_maybe(foo f);
```

F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const

Reason

Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues.

What is "cheap to copy" depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value. When copying is cheap, nothing beats the simplicity and safety of copying, and for small objects (up to two or three words) it is also faster than passing by reference because it does not require an extra indirection to access from the function.

A note about passing by value.

CP.31: Pass small amounts of data between threads by value, rather than by reference or pointer

Reason

Copying a small amount of data is cheaper to copy and access than to share it using some locking mechanism. Copying naturally gives unique ownership (simplifies code) and eliminates the possibility of data races.

Note

- Defining "small amount" precisely is impossible.

In / Out – Pass by non-const &

```
// by non-const reference
void call_me_maybe(foo& f);
```

F.17: For "in-out" parameters, pass by reference to non-const

Reason

This makes it clear to callers that the object is assumed to be modified.

Will-move-from or sink parameters

```
// by rvalue reference
void call_me_maybe(foo &&f);
```

F.18: For "will-move-from" parameters, pass by X&& and std::move the parameter

Reason

It's efficient and eliminates bugs at the call site: X&& binds to rvalues, which requires an explicit std::move at the call site if passing an lvalue.

Out parameters… Don’t do it.

- Prefer return values.
- For multiple returns, use a complex type or tuple.

2nd note on self documenting code.

Your call signature matters, a lot. When done right, it *is* the documentation for the function parameters.

Summary

- Start by passing by reference to const (const &) or value
- If modification is required, pass by non-const ref
- When ownership changes, pass by rvalue (&&) ref

So what about pointers?

1. Use pointers to denote a single object. Not a list.
2. Raw pointers are non-owning.
3. Use T* to denote that “no value”, is an okay input. Always, always, always check for nullptr.

Return values

- Return by const & where object lifetime is known, and a copy is undesirable. Never ever ever return a local by &.
- Return locals by value
- Return a raw-pointer to indicate position, NOT ownership.

3 Error Handling

The greatest of faults, I should say, is to be conscious of none.

– **Thomas Carlyle (1795-1881)**

Error Handling Techniques

1. Exceptions
2. Error Codes
3. Hybrid

Exceptions

- Default C++ mechanism for reporting errors.
- Works together with lifetime system, to ensure no-leaks. Stack unwinding.
- Is pervasive / invasive.
- Will terminate if unhandled.

Error Codes

- Old school, used by C APIs.
- Can be beneficial in asynchronous patterns, where stack unwinding is not helpful.
- Easy to not check.
- Can lead to UB if left unchecked.

Prefer Exceptions.

1. It’s built in. So why not take advantage of that?
2. If rules are followed, we can guarantee 0 memory/resource leaks, without a garbage collector.
3. Violent reporting, cannot go unchecked.
4. Allow for clean separation between normal code, and error handling code.

Tips for Exception Handling

1. Fail fast.
   - Always prefer the fastest method to catch an error.
2. Build error categories.
   - Use custom exceptions for error classification.
3. Handle at the highest level possible.
   - Handle exceptions where you know you can recover.
   - Throw by value, catch by reference.
4. Exceptions are for “Exceptional cases”.
   - **Do not** use exceptions for flow control.

Fast Failure

- Prefer compile time checks where possible.
- Report errors immediately when they occur. (Where possible)
- Die if you have to.
- Use RAII.

(Show exception_handling.cpp)

Error Categories

- Custom errors to denote application specific errors.
- logic_error vs. runtime_error
- Avoid using std::exception directly.

Handling Exceptions

- Handle exceptions where you can recover.
- Prefer to handle at the highest possible level.
- Use a top-level exception handler.
- Avoid try/catch where unnecessary.

Summary

- Fail fast.
  - Compiler errors; throw at the failure point, die if you have to.
- Use exceptions; throw by value; catch by reference.
- Design your strategy upfront.
- Catch only when you can do something about it.
- Don’t use exceptions for flow control.

4 Templating

- Template programming is a different language!
- All the rules still apply. (and even more so)
- Prefer standard C++ without templates, first.
- Prefer STL where possible.
- In application code, TMP is useful ~1% of the time. **

** This is a completely made up statistic. But nevertheless you need TMP very rarely in application code.

Template Meta Programming

- C++ Template Language is a Turing complete language.
  - *This was an accident.*
- C++ Templates are used for code generation.
  - This means, you are programming, your program. It’s so meta…

Question.

What is a programming paradigm, that lets you separate an algorithm from the concrete types on which it acts?

- Polymorphism.

Compile Time Polymorphism

"Concepts"

Template Specialization

- Allows you to customize code based on specific types.
- Code paths which are chosen at compile time.

Smells like TMP spirit

- You’re crafting multiple instances of your class, only varying them by type.
  - `MyInt, MyDouble, MyString`
- You’re writing a library, that needs to be generic.
- You’ve got type specializations you know at compile time. (Though, you can prefer overloading first.)
- You’ve measured, and know that you’re paying an unnecessary runtime cost that could be computed at compile time shaving off mere microseconds.
- You’re building a program to run at compile time. (Compile time tetris.)

Tips for TMP

- It’s equally as important to know when ***not*** to use TMP, as it is when to use it.
- Start with the simplest tool first, and work your way to more complicated. Sometimes a for loop will work, and you don’t need templating.
- Practice TMP for fun, not in production.

Thanks for attending!

- Peter Lorimer
- mvpete@unparalleledadventure.com
- https://unparalleledadventure.com
- https://github.com/mvpete

References

- https://www.youtube.com/watch?v=uQyT-5iWUow
- https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
- https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=vs-2019
- https://devblogs.microsoft.com/cppblog/c20-concepts-are-here-in-visual-studio-2019-version-16-3/