Review

Review: C++ Programming Tips

Why C ?

- Code is FAST; compiler is FAST; often only little slower than hand-written assembly language code
- Lingua Franca of computing
- Portable. C compilers are available on all systems
- Compilers/interpreters for new languages are often written in C
- Easier to know what the assembly will look like when reading C

Why C++?

- C + classes + templates: FAST code + coding CONVENIENCE + SAFETY
- You are still in total control, unlike Java or C#

With great power comes great responsibility (Spiderman's Uncle)

From C to C++

Use `const` and `inline` instead of `#define`

- Macros are not type-safe
- Macros may have unwanted side effects. Use templates instead.

Prefer C++ library I/O over C library I/O

- C's `fprintf` and friends are unsafe and not extensible.
  - Like the syntax `"%6.2f"`? Use `boost::format`
  - This has been added as `std::format` to C++20
- C++ iostream class safe and extensible
- `iostream` speed has caught up, so speed is hardly a reason anymore for choosing C-library I/O

From C to C++

Prefer C++ style casts — easy to find with grep

Distinguish between pointers and references

- References always point to existing objects, no arithmetic, safer

Memory Management

Use the same form in corresponding calls to new and `delete`

- `int *p = new Foo; ... delete p;`
- `int *p = new Foo[100]; ... delete [] p;`

For each new there must be at least one corresponding `delete`

- Delete pointer members in destructors
- Otherwise you are creating memory leaks

No need for checking the return value of new

- It throws an exception if no memory available (in an ideal world)

`delete p` with `p = nullptr` is OK (ignored, no `!= nullptr` check required)

Memory Management

Beware of double deletes $\longrightarrow$ undefined behaviour

- Make sure all objects have sole owners
- For debugging consider adding `p = nullptr` after `delete p` or use template function:

```
template <typename T>
void destroy(T* &p) {
    delete p;
#ifndef NDEBUG
    p = nullptr;    // code created in debug mode
#endif
}

int *p = new int;
...
destroy(p);
*p = 0;             // error caught in debug mode
```

Better yet: say good-bye to raw pointers, new and `delete`, and use C++11 smart pointers and `make_*` functions instead!

The Big-4 (or 6)

When designing new classes decide which operators you have to define: constructor, destructor, CC, AO, MC, MAO

Things to consider: Do I want to risk undefined variables for gaining a little bit of speed for not initializing all components? Do I allocate resources like memory or file descriptors?

Define the Copy and Assignment operators when resources are dynamically allocated

- Default component-wise copy is often insufficient in this case

Make destructors virtual in base classes

- Otherwise base class pointers can't call the right destructor

The Big-4 (or 6)

Have the AO return reference to `*this`

- For iterated assignments `a = b = c ...`

Assign to all data members in the AO, and check for self-assignment!

```
if (this == &rhs) {return *this;}
```

Operators for which you know that the default implementation the compiler provides is wrong and you don't want to implement need to be made inaccessible by using `= delete` (or by making them private)

C++11 added move-semantics. If for your class X moving is faster than copying, implement the move-constructor `X(X &&)` and move-assignment `X &X::operator=(X &&)` which bind to rvalue references.

Operators

Never overload `&` `&&` `||` `,`

Distinguish between prefix and postfix forms of `++` `--`

- They (should) return different types:
  - `++i` : returns reference to `i`
  - `i++` : returns value of temporary object (can be slower!)

Be consistent

- `+` `+=` `prefix++` `postfix++` should have related semantics

Class/Function Design

Guard header files against multiple inclusion

`#ifndef ClassName_H_ ...` or `#pragma once`

Strive for complete and minimal public interfaces

- complete: users can do anything they need to do
- minimal: as few functions as possible, no overlaps

Minimize compilation dependencies between files

- Consider forward declaration in conjunction with pointers/references to minimize file dependencies:

```cpp
class Address;

class Person { ... Address *address; ... };
```

Why is it now no longer requires to include ***`Address`*** *or* ***`Person`*** *headers?*

Class/Function Design

Never use `using namespace X;` in header files

- it forces users of your class to use the same namespace, even if they don't want to

Avoid data members in public/protected interfaces

- Use get/set functions — more flexible and safer

Use const/constexpr whenever possible

Pass and return objects by reference if you can

- But don't return references to vanishing objects such as local variables!

Avoid returning writable "*handles*" to internal data from const member functions

- Otherwise constant objects can be altered from the outside

Inheritance

Make sure public inheritance models "*is a*"

- Never redefine an inherited non-virtual function
- Different results for `pBase->f()` and `pDeriv->f()`

Never redefine an inherited default parameter value

- Virtual functions are dynamically bound
- Default parameters are statically bound

Avoid casting down the inheritance hierarchy (base to derived class)

- Use virtual functions instead

Exceptions

Prefer exceptions over C-style error codes

Use destructors to prevent resource leaks

- Say "*good-bye*" to pointers that manipulate local resources — use smart pointers instead

Prevent resource leaks in constructors

- Destructors are only called for fully constructed objects

Prevent exceptions from leaving destructors

- Exceptions within exceptions terminate program and unwinding exceptions calls destructors ...

Catch exceptions by reference

- All alternatives create problems

Efficiency

Choose suitable data structures and efficient algorithms

Consider the empirical "80-20" rule:

- 80% of the resources are used by 20% of the code
- Focus your optimization efforts by using profilers (e.g. gprof, perf)

Avoid frequent heap memory allocation, prefer stack variables

Know how to save space

- bits, bytes, unions, home-brewed memory allocators

If necessary, optimize memory access patterns and data alignment to benefit from fast cache memory architectures

Understand costs of virtual functions, multiple inheritance, exception handling

STL Tips

Choose your containers wisely

- sequence vs. associative?
- tree-based vs. hash-based?
- speed vs. memory consumption?

Prefer C++ arrays over C-arrays. C++ arrays can check for index violations and know their size

- If speed matters, use C++ arrays, vectors, or hashed associative containers

Be careful when storing pointers in containers

- if the container owns the objects they have to be destroyed before the container is destroyed
- possible dangling pointers to vanished objects

Make sure comparison functors implement strict weak orderings

STL Tips

Note which algorithms expect sorted ranges

Have realistic expectations about thread safety of STL containers:

- **YOU** need to lock containers

Call `empty()` instead of checking `size()` against 0. It may be faster.

Make element copies cheap and correct

- STL copies elements often

More tips in Scott Meyers'

- Effective Modern C++ (C++14)
- Effective C++ (C++98/03 --- but still relevant)
- More Effective C++
- Effective STL