**Coding Standards & Practices for GAP**

Last Updated: 1/4/2024

Motivation

To provide a set of standards for how code is written within the GAP program. As a professional software engineer, you will need to conform to the standards set forth by the company you work with. These standards are based on common professional practices.

Note that any of these conventions may be changed for any given class. For example, if you’re working in Unreal 5, your instructor may require you to follow their naming conventions. You will be told on the first day of class if this is the case.

A note on versioning: We are using C++ 20. All features of C++ 11, 14, 17, and 20 are fair game (except as noted below) and, in many cases, are required. You may be able to use some C++ 23 features, but always ask your instructor first. They can allow or disallow any C++ 23 features they wish. C++ 26 and beyond are not allowed.

The Golden Rules

This document can’t possibly cover every single case and even if it could, every rule has exceptions. You will need to exercise your own judgment when deciding when to bend or break a rule, or how to format something not directly covered here. These golden rules provide a guide for you to make those decisions.

These golden rules supersede everything else in this document. If any other rule would violate one of the golden rules, ignore it and apply the golden rule instead.

1. Code must be as readable as possible.
2. Be consistent. If you come up with your own rule or convention, follow it consistently where appropriate.

Naming Conventions

Classes, namespaces, functions, typedef’s, aliases, concepts,modules, and enum names should be named using CamelCase, where the first letter is capitalized:

class FooBar;

namespace FooBar { }

enum class FooBar { };

typedef std::vector<int> IntVector; // typedef’s should be avoided

using FloatVector = std::vector<float>;

void DoTheThing();

export module Foo;

Filenames should be named in the same way:

FooBar.h

FooBar.cpp

FooBar.ixx

Local variables should be named in camel case, but with the first letter lowercase:

int fooBar;

Member variables, global variables, and static variables must have a single letter prefix with an underscore denoting their scope. Local variables (including local static variables) have no prefix.

int m\_fooBar; // member variable

int g\_fooBar; // global variable

static int s\_fooBar; // static variable (non-local)

int fooBar; // local variable

static int fooBar; // local static variable

Additionally, pointers should be prefixed with a ‘p’:

int\* m\_pFooBar; // member pointer variable

int\* g\_pFooBar; // global pointer variable

int\* s\_pFooBar; // static pointer variable (class or non-class)

int\* pFooBar; // local pointer variable

Constant variables that are essentially used as tuning variables (including all constexpr variables) should be prefixed with a k. This includes enum values as well:

const char\* kpFooBar = “baz”;

enum class FooBar

{

kFooBar,

kFooBarBaz

};

constexpr unsigned int kFoo = 0;

This overrides the scoping prefixes. So, for example, a member constant is named as such:

class Foo

{

static constexpr int kSomeConst = 100;

};

The exception to using k is if you have a variable that looks like any other variable but it just happens to not change. For example:

void Square(int val)

{

const int result = val \* val; // Treated like any other variable,

// it just happens to not change.

return result;

}

There are two general placement styles of const: west const and east const. We use west const.

const int foo = 10; // west const

int const foo = 10; // east const

#define’s should be all caps, with underscores between each word:

#define FOO\_BAR

Bracing & Formatting

Bracing & indentation should adhere to the Allman style:

<https://en.wikipedia.org/wiki/Indent_style#Allman_style>

while (x == y)

{

Foo();

FooBar();

}

if (x > y)

{

Foo();

}

else

{

FooBar();

}

FooBarBaz();

Indentation should be set to 4 spaces. Make sure you are inserting spaces rather than inserting the tab character. *In Visual Studio, you can set this by going to Tools -> Options -> Text Editor -> C/C++ -> Tabs. Change the Tab size to 4 and make sure “Insert spaces” is set.*

Binary operators should always be surrounded by a single space on either end, while unary operators should not have any spaces:

int foo = bar + baz;

++foo;

Control structures should have a space between the control keyword and the opening parentheses:

if (foo) // yes

Bar();

if(foo) // NO

Bar();

Never use single-line if statements or loops:

if (foo) FooBar(); // NO

while (foo) FooBar(); // NO

Instead, put it on the next line:

if (foo)

FooBar();

while (foo)

FooBar();

Multiple statements on the same line are also not allowed:

x = 10; y = 15; // NO

Single-line functions are allowed only if they are trivial accessors or mutators:

class Foo

{

int m\_bar;

public:

int GetBar() const { return m\_bar; }

void SetBar(int bar) { m\_bar = bar; }

// NO. This function has two statements, so it can’t be

// a single line. It’s also no longer a trivial getter

// since it has additional side effects.

int IncrementAndGetBar() { ++m\_bar; return m\_bar; }

};

It’s okay to omit the braces on an if statement or loop if there can be absolutely no confusion that it’s the only statement within the body. You should avoid single-line bodies that take up multiple lines:

// avoid this -- use braces instead

if (foo)

while (bar)

if (baz)

DoTheThing();

// this is ok

if (foo)

DoTheThing();

When in doubt, use braces. It is always okay to use braces, even if the body is trivial:

// OK

if (foo)

{

++bar;

}

In general, prefer to use init statements in if’s in order to prevent “leaking” the variable to the outer scope:

// Avoid this when you can:

const int value = GetValue();

if (value > 0)

DoThing();

// Do this instead:

if (const int value = GetValue(); value > 0)

DoThing();

Switch statements have the following form:

switch (foo)

{  
 case 1:

Bar();

break;

case 2:

Baz();

break;

default:

Error(“Default reached”);

return false;

}

Alternatively, you may add braces to the case statements so that each case contains a local scope. You must either have braces around all of the cases or none of the cases. You may not mix braced and unbraced cases.

switch (foo)

{  
 case 1:

{

Bar();

break;

}

case 2:

{

Baz();

break;

}

default:

{

Error(“Default reached”);

return false;

}

}

Either of the above forms are fine, but keep in mind that not including braces in a case statement can cause subtle errors because they will all share the same scope.

It is also permissible to have single-line case statements as long as they are only a single simple line, plus either a break or return statement:

switch (foo)

{

case 1: Bar(); break;

case 2: Baz(); break;

default: Error(“Default reached”); return false; // return OK

}

All case statements must have a break or a return statement at the end. There are two exceptions to this. The first is if the case is complete empty (essentially allowing “OR” logic):  
  
switch (foo)

{

// 1, 2, or 3

case 1:

case 2:

case 3:

Baz();

break;

}

The second is if you explicitly use C++ 17’s [[fallthrough]] attribute:

switch (foo)

{

case 1:

Bar();

[[fallthrough]]; // also do case 2

case 2:

Baz()

break;

}

In general, try to avoid this as much as you can. Instead, consider using if/else logic.

The ternary operator is allowed if it improves code readability:

const int x = (pFoo ? pFoo->GetX() : 0);

Nesting ternary operators is never allowed:

// NO

const int x = (pFoo ? pBar && pBaz ? pBar->GetX(pFoo, pBaz) : 2 : 0);

One final note is to avoid magic numbers or literals as case tests. Prefer constexpr variables instead.

Classes & Structs

In C++, classes and structs are exactly the same except for two things:

1. Classes default to private access while structs default to public.
2. Classes default to private inheritance while structs default to public inheritance.

However, classes and structs communicate two very different things. Structs should be used for pure data types. In general, a struct should never define any functions except for one or more constructors, the copy constructor, the copy assignment operator, the move constructor, the move assignment operator, and the destructor. If other functions are required, it’s time to promote the struct into a class.

There are two general ways to structure classes. The first is as follows:

class FooBar

{

public:

// Nested classes, enums, and typedefs go here. These might

// be private or protected instead of public.

private:

// private data goes here

protected:

// protected data goes here

public:

// public functions goes here

// simple accessors & mutators go here

protected:

// protected member functions go here

private:

// private member functions go here

};

The second is like this:

class FooBar

{

public:

// the public interface is here

protected:

// protected functions go here

private:

// private functions go here

// private data goes here

protected:

// protected data goes here

};

Either method is fine, or some third method could be valid. This is a very loose guideline, but the following cases must be true:

1. Data and functions must be separate. Do not mix them.
2. The public interface (i.e. public functions and data) must all be grouped together. This does not include public static constexpr data.
3. You must be as consistent as possible.

Your goal must always be readability.

Generally avoid public data in classes. In other words, you should not have a public member variable. Use accessors and mutators to manipulate variables from outside of the object. The exception is for structs or certain “primitive” mathematical types (Vector3 for instance). In these instances, the m\_ prefix is omitted as well:

struct Foo

{

int bar;

float baz;

};

There are three different places where a member variable can be initialized:

// class initialization

class Foo

{

int m\_bar = 0;

};

// constructor initializer list

class Foo

{

int m\_bar;

public:

Foo() : m\_bar(0) {}

};

// constructor body

class Foo

{

int m\_bar;

public:

Foo() { m\_bar = 0; }

};

Class initialization is allowed (and often preferred) if the value being initialized is a literal or constexpr value. This sets up a default initialization value which can make having multiple constructors easier to manage. Initializer lists or constructor bodies can override the in-class initialization, but if they don’t, the value will still be valid. The disadvantage is that initialization is broken up into separate locations in the source code. It’s up to you; do the thing that makes sense.

Initializing in the constructor body should be avoided as much as possible. The problem is that the object isn’t being constructed here, it’s being assigned to. This can cause subtle issues and performance problems. For example, consider this:

class Foo

{  
 std::unordered\_set<int> m\_bar;

public:

Foo(std::unordered\_set<int>&& bar)

{

m\_bar = std::move(bar);

}

};

The constructor above will heap-allocate a single sentinel node even though we’re performing a move because m\_bar’s default constructor is called (Visual Studio’s unordered\_set implementation heap allocates in the default constructor).

When using an initializer list, it should be formatted like this:

Foo::Foo()

: m\_bar(0)

, m\_baz(0)

{

//

}

This allows you to easily comment out a single line in the initializer list.

Follow the Rule of 5, which states that if you create a custom destructor, copy constructor, copy assignment operator, move constructor, or move assignment operator, you should implement all five of the magic functions C++ might define for you. In this case, if you want the default behavior, you should explicitly force it.

class Foo

{

public:

Foo();

~Foo(); // destructor defined, so we need to define the others

Foo(const Foo&) = default;

Foo& operator=(const Foo&) = default;

Foo(Foo&& right);

Foo& operator=(Foo&& right);

};

Read about the rule of 5 here:

<https://en.cppreference.com/w/cpp/language/rule_of_three>

Do not have empty constructors or destructors, use = default or omit them entirely instead:

// NO

class Foo

{

public:

Foo() { }

~Foo() { }

};

// YES

class Foo

{

public:

Foo() = default;

~Foo() = default;

};

// YES

class Foo

{

// empty

};

The use of override and final are required where appropriate:

class Foo

{

public:

virtual void DoTheThing() = 0;

virtual void DoSomethingElse() = 0;

};

class Bar : public Foo

{

public:

virtual void DoTheThing() override;

virtual void DoSomethingElse() override final;

};

In particular, use final where you can because it can improve performance in some situations.

Using the virtual keyword on overridden functions is optional. You can omit it if you want, but make sure you have the override keyword.

Prefer C++’s new cross-platform libraries over the platform-specific ones. For example, use std::thread instead of Windows’ threading API or pthreads. Use std::chrono instead of GetTickCount(). Use std::filesystem instead of OS-specific functions.

Function Objects & Lambdas

Lambdas may be used in any context that makes sense, as long as they don’t make the code harder to decipher. If you are storing a lambda somewhere (or any callable object), it should be named like a variable:

// lambda

auto lambda = []() { cout << "foo\n"; };

// functor

struct Functor // definition is named like a class

{

void operator()() { cout << “foo\n”; }

};

Functor functor; // instantiation named like a variable

// C function pointer, named like a pointer variable

void (\*pFoo)() = &Foo;

For lambdas, captures are okay, but do not use default captures of any kind:

auto foo = [&x, y]() { cout << "foo\n"; }; // YES

auto bar = [&]() { cout << "foo\n"; }; // NO

auto baz = [=]() { cout << "foo\n"; }; // NO

Enums

Scoped enums are preferred to non-scoped enums:

// scoped enum

enum class Foo

{

kBar,

kBaz,

};

// non-scoped enum

enum Foo

{

kBar,

kBaz,

};

Concepts & Templates

Prefer using concepts instead of traditional templates for anything other than truly generic types.

Prefer auto over explicit template syntax unless it would obfuscate the meaning, if you need to explicitly refer to a particular template type, or if you need to ensure that the types are truly the same:

// Old version

template <class Type>

Type Square(Type val)

{

return val \* val;

}

// Better C++ 20 version

auto Square(auto val)

{

return val \* val;

}

Note that you are always allowed to use explicit templates, but the auto types should generally be preferred.

Commenting

You should add comments in the follow circumstances:

1. At the top of a function.
   1. This comment should explain the purpose of the function.
   2. It should describe the inputs, outputs, and any assumptions.
2. At the top of a class.
   1. Describe the purpose of the class and how it should be used.
   2. Explain the main public interface.
   3. It’s not necessary to explain simple accessors or mutators.
3. For each block of code.
   1. There should be a short comment at the top of each block of code describing what that section does.
4. For anything that’s not extremely obvious.
   1. If you’re doing something tricky, write a big comment explaining it.
   2. If you had trouble figuring something out, write out your solution.
   3. If there’s a bit of confusing code, write a comment that explains it.
5. For any bug fixes or optimizations that weren’t obvious.
   1. If you changed something that seems like it should have worked, write a comment explaining what you changed and why.

Headers

For header guards, there is a choice between this:

#pragma once

and this:

#ifndef FOO\_BAR\_H

#define FOO\_BAR\_H

…

#endif // FOO\_BAR\_H

Use the first method. All modern compilers support this method and it is faster to compile in Visual Studio.

Header files should be self-contained. They should have header guards and include all the other headers necessary to compile them. In general, try to avoid adding #include’s inside header files where possible. Use forward declarations when you can.

Modules

If you are using modules, prefer them over headers as much as possible. This isn’t always possible when dealing with the standard library, 3rd party code, or even some legacy code, but you should generally prefer them for code you write.

Do not import header files as header units:

import <iostream>; // NO

Instead, just #include the file as normal in the global module fragment:

// Foo.ixx - a module file

module; // we’re now in the global module fragment

#include <iostream> // include the file

export module Foo; // we’re now in the exported Foo module fragment

Modules are not namespaces. It is not uncommon to have a namespace inside a module with the same name, but correct use of namespaces should be independent of modules.

Modules should only export symbols (functions, classes, types, etc.) that are actually required and used by clients of the module. Follow the same best practices here as with public/private functions in classes.

Ranges

Prefer the updated algorithms that don’t require an iterator range if running it across an entire container:

std::vector<int> foo = { 2, 4, 6, 8, 10 };

int result = std::accumulate(foo.begin(), foo.end(), 0); // NO

int result = std::accumulate(foo, 0); // YES

Ranges are allowed as long as they don’t obfuscate the code, though you should prefer loops instead of ranges for high-performance code. Ranges tend to perform worse than the equivalent hand-written loop.

main()

There are two forms of main() that are valid in standard C/C++:

// Form 1: No arguments, returns an int

int main()

{

return 0; // this indicates normal program termination

}

// Form 2: command-line arguments, returns an int

int main(int argc, char\* args[])

{

return 0; // this indicates normal program termination

}

Either of these forms are valid. There is a third form you will see sometimes that has no return value:

// Form 3: No return. This is forbidden.

void main() // NO

{

//

}

This will compile just fine in Visual Studio, but it is NOT considered valid C++. Because of this, it is forbidden. You must use Form 1 or Form 2.

Note that in C & C++, the return 0 at the end of main() is not required. main() implicitly returns 0. Because it’s so common, you are not required to explicitly return from main(), but it is still preferred since it keeps things consistent.

Misc

Avoid the postfix increment and decrement operators. Always use the prefix version unless there’s a real reason not too (though there almost never is). The reason is that postfix increment/decrement will create a temp variable. This is almost always optimized out for primitives, but for first-class objects (like some STL iterators) it can create a minor performance concern.

// yes

++foo;

--bar;

// no

foo++;

bar--;

Use parentheses whenever there’s any confusion over the order of operations. Don’t rely on other programmers memorizing the operator precedence tables. For example:

// NO

if (foo && bar || baz)

DoTheThing();

// YES

if (foo && (bar || baz))

DoTheThing();

Always use nullptr instead of NULL.

Prefer using std::byte\* instead of unsigned char\* for raw data.

Prefer constexpr over const whenever you can.

Use consteval instead constexpr for functions that you expect to always be compile-time.

Prefer aliases over typedefs.

Range-based for loops are encouraged but never required:

std::vector<GameObject\*> gameObjects; // assume this is filled out

for (GameObject\* pGameObject : gameObjects)

{

pGameObject->Update();

}

Structured bindings should be used to improve readability for things like tuples, pairs, and map iterators:

std::map<Id, GameObject\*> gameObjects; // assume this is filled out

for (const auto&[id, pGameObject] : gameObjects)

{

pGameObject->Update();

}

using WeaponData = std::tuple<int, float, int, int>;

WeaponData GetWeaponData();

auto[damage, quality, attackBonus, parryBonus] = GetWeaponData();

The auto keyword may be used anytime that it doesn’t obfuscate the meaning. A common example is when assigning iterators, but other uses are permitted as well:

auto it = gameObjects.begin();

Prefer lambdas to std::bind.

Coroutines are discouraged. They tend to significantly obfuscate the code, they require a considerable amount of boilerplate for basic functionality.

**C++ exceptions should not be used for any reason unless you are learning about exceptions.**

C++ Versions

C++ 20 is the language we are officially using, though most classes were written against C++ 17 so most of the examples you see will be in C++ 17. Regardless, all C++ 20 features are fair game and it is encouraged that you make full use of them. Note that as of this writing, Visual Studio still defaults to C++ 14 so you will need to manually change the language to support C++ 20 if you want to use any of those features. Make sure you change it for all configurations and platforms.

The C++ 23 standard is fully written, but no major compiler fully supports it. You are encouraged to read up on C++ 23 and play with the features it offers, but C++ 23 is not currently allowed on any official project or assignment.

C++ 26 is still in the development phase and is subject to change. You are welcome and encouraged to read about it, but it is forbidden to use in code.

C#

Properties should be named like a local variable, except that a prefix is not necessary. If it is an accessor, it should share the same name as the variable it’s tied to. Note that all the rules for single-line functions apply to properties. Example:

public class Foo

{

private int m\_bar = 0;

public int bar

{

// This is a single-statement getter, so it’s okay to be

// one line:

get { return m\_bar; }

// Not a single-statement setter, so it must be multiple

// lines:

set

{

if (value >= 0 && value < 100)

m\_bar = value;

}

}

}

All functions and variables must have an explicit access modifier rather than relying on the default of everything being private:

private int m\_foo = 0; // YES

int m\_foo = 0; // NO

private void DoThing() // YES

{

//

}

void DoThing() // NO

{

//

}

In C#, there is a very significant difference between classes and structs. The use of the m\_ prefix is not limited to structs in C#, but to any class or struct that philosophically follows the “pure data” guidelines set above:

public class Foo

{  
 public int x = 0;

public int y = 0;

}

Lua

Tables that are being used as classes should be named like classes. Other tables should be named as variables appropriately.

Lua has no concept of public or private, but those concepts are still important. Variables and functions that are prefixed with a leading underscore are considered private and should be treated as such:

function Foo:\_Bar()

--

end

Since Lua requires the self keyword to access member functions and variables, it is not necessary to use the ‘m\_’ prefix. Likewise, there are no static variables. Global variables should still be marked with a ‘g\_’ prefix.