I’m falling in love with Globals

James Mitchell

j_mitchell@wargaming.net

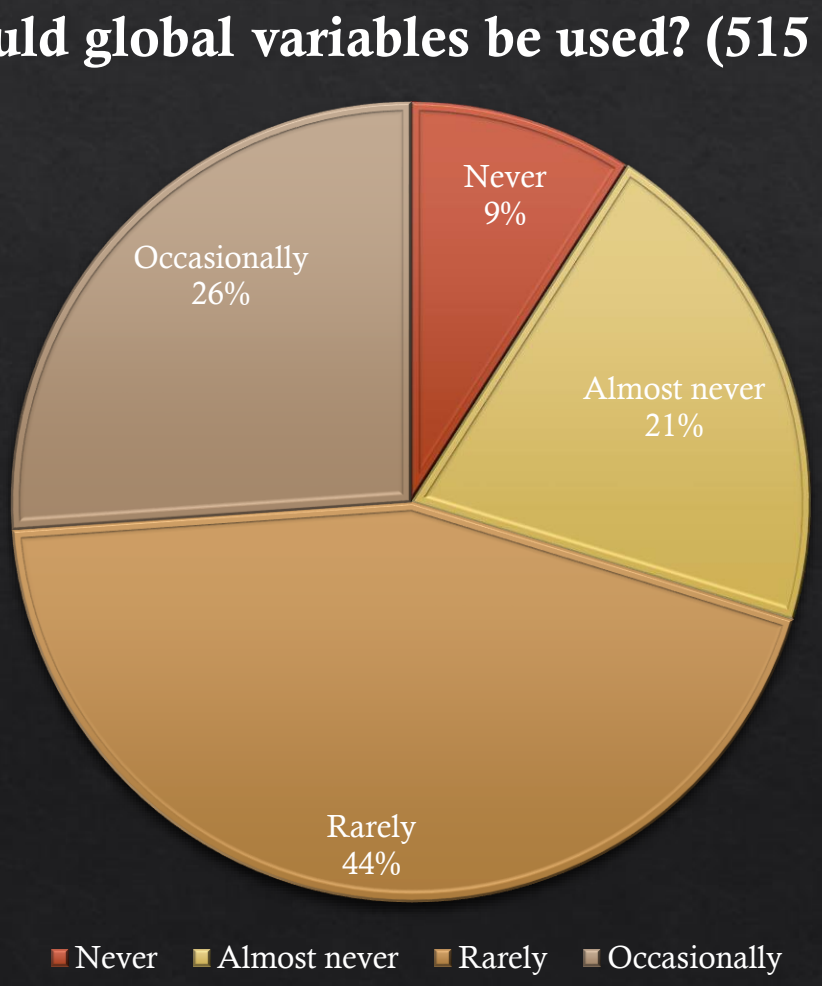
Expert Software Engineer

Wargaming Sydney

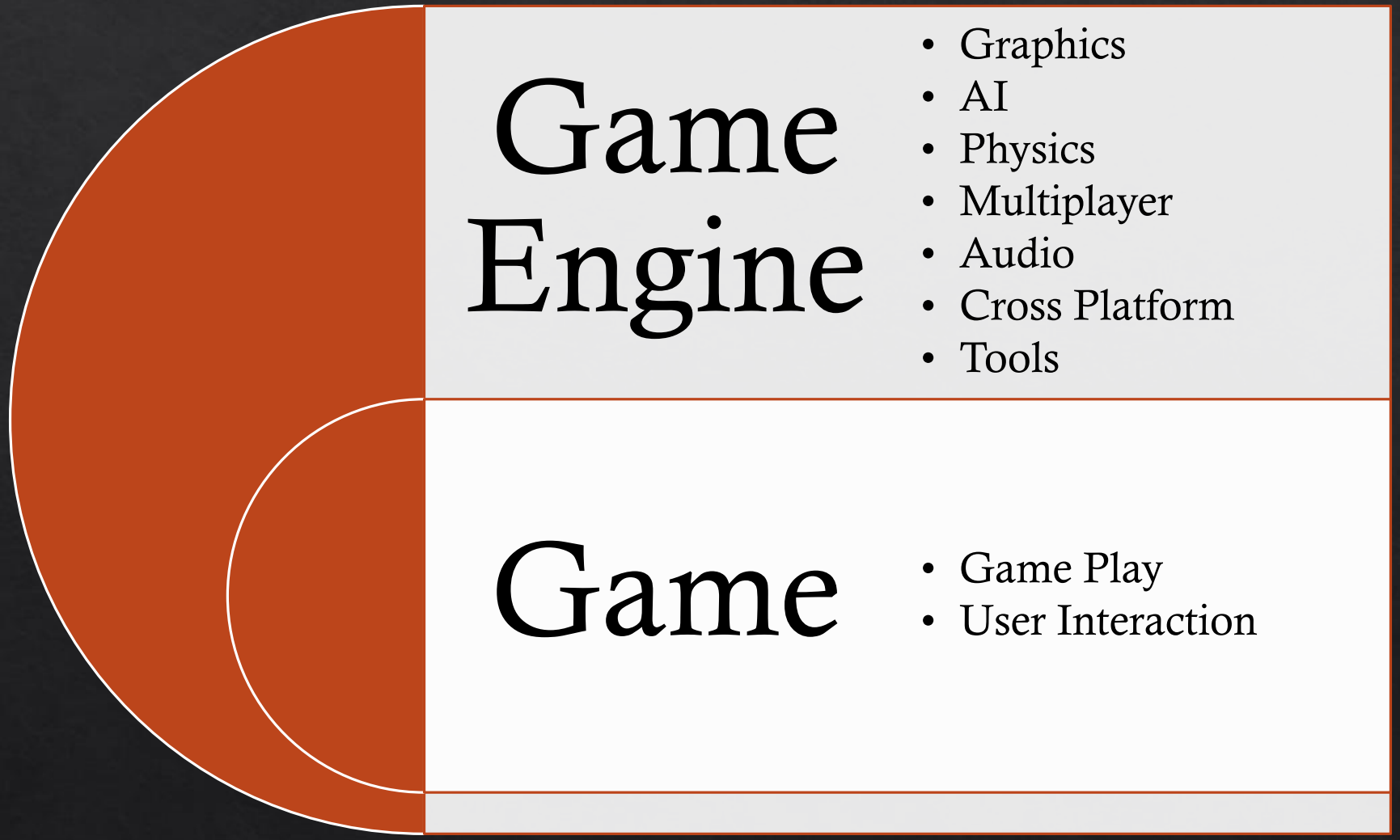
Survey: When should global variables be used?

- Never, ban them from the language (Or only for constants)
- Almost never, only as a workaround for poor API or shortage in language functionality (e.g. test case registration/discovery, etc)
- Rarely, only very limited use cases (e.g. logging, etc)
- Occasionally, used when something always exists or being global makes easier development (e.g. event dispatchers, thread pools, io, etc)

The survey results from the internet



What I work on: Game Engines



Agenda

- Globals are everywhere
  - The unavoidable globals
  - The globals you should use
  - The globals you might use
- The other tools in the toolbox
- The issues with Globals
- Guidelines for using Globals
- Questions

Why you are using Globals already

Some global variables are unavoidable, a committee decided them for you.

Standard I/O

```
std::cout << “Hello World”;
std::cerr << “Hello World”;
std::clog << “Hello World”;
```

Localization

```
std::setlocale(LC_ALL, "de_DE.UTF-8");

// 3,14
std::wprintf(L"%.2f\n", 3.14);


std::locale::global(std::locale("de_DE.UTF-8"));
std::wostringstream ss;
ss << 3.14;
std::wcout << ss.str();
```

std::error_code

```
std::error_code ErrorCode(ernno, std::generic_category());
std::error_code ErrorCode(::GetLastError(), std::system_category());
```

C APIs

```
double NotANumber = std::log(-1.0);
if (errno == EDOM)
   std::cout << std::strerror(errno);
```

Because you want to use futures

```
std::promise<int> Promise;
std::future<int> Future = Promise.get_future();
Promise.set_value_at_thread_exit(123);
```

Polymorphic allocators

```
MyMemoryResource MemResource;
std::pmr::set_default_resource(&MemResource);
```

Why you should use Globals

Why giving guns to your developers can make life easier

Magic numbers

```
constexpr double PI = 3.14159265358979323846264338327950288;

double VolumeOfSphere(float Radius) {
    return 4.0 / 3.0 * PI * Radius * Radius * Radius;
}
```

Reflection

```
struct AutoReg {
    using TestT = void (*)();
    inline static std::vector<TestT> Tests;
    AutoReg(TestT test) { Tests.push_back( test ); }
};

#define CONCAT_TRICK(arg1, arg2) arg1##arg2
#define CONCAT(arg1, arg2) CONCAT_TRICK(arg1, arg2)
#define UNAME(Name) CONCAT(Name, __LINE__)

#define TEST_CASE()                          \
    static void UNAME(TestCase)();           \
    AutoReg UNAME(TestReg){&UNAME(TestCase)}; \
    static void UNAME(TestCase)()
```

```
// Test discovery
TEST_CASE() {
    CHECK( false );
}
```

Logging

```
struct GlobalLogger : ILogger {
  void Console(string Message);
};
GlobalLogger Log;

Log.Console("Hello World");
```

Platform Abstractions

```
struct RawFileSystem : IFileSystem {
    FilePtr Open(string Name);
};
RawFileSystem FileSystem;

FileSystem.Open( “Hello World” );
```

Event Dispatcher

```
struct EventDispatcher  {
    future<void> WaitFor(chrono::ms ms);
    future<void> Post();
};
EventDispatcher MainLoop;

void DoSomething() {
     MainLoop.WaitFor(100ms).then( []{
        Log.Console("100ms later");
    }
}
```

```
void DoSomethingCoro() {
    co_await MainLoop.WaitFor(100ms);
    Log.Console("100ms later");
}

void Thread() {
    MainLoop.Post().then( [] {
        Log.Console("Back to main
thread");
    }
}
```

Configuration

```
// NOTE: Can be changed by the debugger, or something else at runtime
int VerboseLevel = 0;
bool IsDevMode = false;

if (VerboseLevel > 0)
    Log.Console("Doing something");

if (IsDevMode) {
    // validate the data more
    …
}
```

Global App

```
struct Application {
    void AutoSave();
    void Warn(string Message);
}
Application GApp;

// About to use some untrusted API that can break everything,
// trigger an autosafe now for safety
GApp.AutoSave();

GApp.Warn("This feature is deprecated");
```

Reflected configuration

```
// NOTE: Can be populated at init time
Config<int> DefaultThreadPoolSize("Threading/DefaultPoolSize", 2);
```

Metrics

```
Metric<int> PacketsLost("PacketsLost");
Metric<int> PacketsResent("PacketsResent");


void DoSomething()
{
    ++PacketsLost;
}
```

Intrusive profiling

```
struct Profiler {
    void Begin(string name);
    void End();
};
thread_local<Profiler> Profiler;
```

```
struct Profile : AutoRegProfiler {
    Profile(string name) { Profiler.Begin(name); }
    ~Profile() { Profiler.End(); }
};

{
    Profile profile("DoSomething");
    …
}
```

Shared libraries

```
struct ExampleAPI
{
   ~ExampleAPI() { dlclose(Handle); }
   void * Handle = dlopen("libsomething.so");
   void (*SomeMethod)() = dlsym(Handle, "SomeMethod");
} Example;

Example.SomeMethod();
```

Third-party API

```
if (PyErr_Occured())
   PyErr_Print();
```

Why you might use Globals

Entering the danger zone of Global variables usage, just because you can doesn’t mean you should.

Handles/IDs for global objects

```
unordered_map<int, Entity> Entities;


void OnSomeEvent(int EntityID)
{
    Entities[id].DoSomething();
}
```

Track info per stack frame

```
struct AutoRegStackMetaData {
    static thread_local<vector<StackMetaData&>> MetaData;
};
struct StackMetaData : AutoRegStackMetaData {
    StackMetaData(…);
    void Add(string Key, string Value);
};

StackMetaData md {
    {"FrameNum", … }
};
Log.Console(…); // Captures stack meta data
```

Command line

```
vector<string> CommandLine;
bool HasArgument(string Arg);

void Something() {
   if (HasArgument(“-server”))
      …
}
```

Locality defined specializations

```
ConsoleLogger Log;
void GlobalFunc();

struct Entity {
   void LocalFunc();
   AnnotatedLogger Log(::Log);
};
```

```
void GlobalFunc() {
   Log.Console("Hello World");
}

void Entity::LocalFunc() {
   Log.Console("Hello World");
}
```

The other tools in the toolbox

Some approaches to avoiding Globals can make code worse

Alternative : Explicit Context Objects

```
struct Context {
    Logger Logger;
};
// or
struct Context {
    Logger & Logger;
};
// or
struct Context {
    Logger & Logger() const;
};
```

Explicit Context

- Example: Context.Logger() or Context.Logger
- ❌ Lifetime of references to members can outlive members if not using shared pointers
- ❌ Requires all members to be available regardless of usage
- ❌ Must rebuild everything if the context is not a reference
- ❌ If members are not references then null checks must occur
- ❌ Another object to pass around
- = Ownership and usage of members is unknown from the interface
- ✓ Only context members are potentially used, no others

Alternative : Implicit Context Objects

```
struct Context {
    T & Get() const;
};
```

Implicit Context

- Example: Context.Get<Logger>
- ✗ Lifetime management becomes much harder (is there one ownership type?)
- ✗ Unclear if variable is available, always requires checking
- ✗ Inability to have multiple objects with the same type
- ✗ More expensive lookup
- ✗ Dependencies of items in context have their own initialization ordering issues
- ✗ Debugging lack of object presence can be painful
- ✗ Another object to pass around
- = Ownership and usage of members is unknown from the interface

Alternative : Dependency injection libraries

```
auto Log = []{}; auto Err = []{};
struct Test {
    BOOST_DI_INJECT(Test, (named=Log) ostream&
log, (named=Err) ostream& err) {
       log << "Hello"; err << "World";
    }
};
int main() {
 const auto injector = di::make_injector(
  di::bind<ostream>.named(Log).to(std::cout),
  di::bind<ostream>.named(Err).to(std::cerr));
 injector.create<Test>();
}
```

- Similar to Implicit context objects
- ❌ Unclear if variable is available, always requires checking
- ❌ Inability to have multiple objects with the same type without a lot of fiddling
- ❌ More expensive lookup
- ❌ Dependencies of items in DI have their own initialization ordering issues
- ❌ Syntax is awful to understand and read
- ❌ Another object to pass around
- = Ownership and usage of members is unknown from the interface
- ✓ Depending on DI library initialization can be ‘easier’

Alternative : Singletons are worse then globals

```
struct ConsoleLogger final {
    static ConsoleLogger & get() { return Instance; }
    void Write(string);
private:
    static ConsoleLogger Instance;
    ConsoleLogger();
    ConsoleLogger(const ConsoleLogger&) = delete;
};

ConsoleLogger::get().Write("Hello World");
```

- ✓ Enforcing only one item, no room for conflict or duplication
- ✓ Strongly typed object
- Only use when it can not be moved, copied or constructed multiple times, otherwise it defeats the purpose of enforcing one item.

The issues with Globals

The honest truth about the problems you might face

Issue : Locality reasoning

```
void Obj::ConnectTo(string Url) {
    …
    // What is Socket?
    Socket.Connect(Url);
}
```

Where does Socket live:

- It could be on the stack
- It could be in Obj
- It could be a Global
- It could be a reference from any of those locations, to another location.

Locality reasoning is something you should be concerned about, but removing Globals does not fix your problems, other solutions must be used instead such as naming conventions or an IDE that has semantic highlighting

Issue : Namespaces conflicts

```
// fileA.cpp
int Global;
void Func() {}

// fileB.cpp
int Global;
void Func() {}
```

- All symbols can cause a conflicts – functions, classes, etc
- Solution: Put everything in a namespace
- If you must use extern “C” then prefix your variables

Issue : Initialization issues

```
// logger.h
extern Logger Log;

// filesystem.h
extern FileSystem FS;

// globals.cpp
FileSystem FS;
Logger Log {FS.Open("File")};
```

- Try to keep global initialization together, this helps enforce ordering
- Try having one file to initialize all globals
- Avoid using global variables inside of types which have a chance of becoming a global instance

Issue : Concurrency

```
Logger Log;

std::thread([]() {
   Log.Console(“Thread1”);
});

std::thread([]() {
   Log.Console(“Thread2”);
});
```

- Concurrency issues can exist regardless of the shared state and how it is shared, Globals make it easier to share state
- Globals give lifetime safety for threads (Globals exist prior to threads starting, Globals exist longer then threads)
- If your application is multi-threaded then you should typically make all Global variables thread safe, or if you only expect the global for a single thread then add necessary checks to ensure this
- Depending on your Global variable, one approach to making a global thread safe is to thread local, this will make the state unique per thread

Issue : Testability

The impact on testing depends highly on the purpose of the global variable and where its used.

Guidelines for safe global usage which should reduce impact on testing.

Type of Global	Impact on Testability	Use in core library	Use in app library	Use in application
Static-Init registration	Low	✓ Allowed	✓ Allowed	✓ Allowed
Debugging and Monitoring (e.g. Logging, Profiling)	Low	Grey Area	✓ Allowed	✓ Allowed
Configuration	Low	Grey Area	✓ Allowed	✓ Allowed
Environment and Platform	Medium	✗ Denied	✓ Allowed	✓ Allowed
Domain specific subsystems	Medium	✗ Denied	✓ Allowed	✓ Allowed
Application Global/God Object	High	✗ Denied	✗ Denied	✓ Allowed

Conclusion

When and why should you use a global variable?

The guidelines for Globals

- Always think heavily before introducing a global
- If its not a constant, discuss it with your team/code stake holders
- Always consider the impact on testing, this is where you can and will get hit
- Always consider the impact on concurrency, you don’t know who will use it and when
- Prefer Globals in a namespace to avoid conflict
  - Or nested in a class
- “Global variables can be thought of as being member variables of the application. If you're not responsible for the application as an entity, you can't make anything global.” - /u/Dalzhim on reddit

Globals make it hard to test

The impact on testing depends highly on the purpose of the global variable and where its used.

Guidelines for safe global usage which should reduce impact on testing.

Type of Global	Impact on Testability	Use in core library	Use in app library	Use in application
Static-Init registration	Low	✓ Allowed	✓ Allowed	✓ Allowed
Debugging and Monitoring (e.g. Logging, Profiling)	Low	Grey Area	✓ Allowed	✓ Allowed
Configuration	Low	Grey Area	✓ Allowed	✓ Allowed
Environment and Platform	Medium	✗ Denied	✓ Allowed	✓ Allowed
Domain specific subsystems	Medium	✗ Denied	✓ Allowed	✓ Allowed
Application Global/God Object	High	✗ Denied	✗ Denied	✓ Allowed

Final Survey

Have I convinced you to reconsider using global variables?

The End

Twitter: @reductor

Email: j_mitchell@wargaming.net

The standard has global state

- std::cin, std::cout, std::cerr, std::clog, std::wcin, std::wcout, std::wcerr, std::clog
- std::set_new_handler
- std::set_terminate
- std::promise::set_exception_at_thread_exit
- std::set_unexpected (Removed)
- std::atexit, std::at_quick_exit
- std::locale::global, std::set_locale
- std::error_code::category, std::generic_category, std::system_category, std::iostream_category, std::future_category
- errno