# **Code Design**

**Correctness, simplicity, and clarity come first**

Keep It Simple Software:

* Correct is better than fast.
* Simple is better than complex.
* Clear is better than cute.
* Safe is better than insecure.

**Give one entity one cohesive responsibility.**

Focus on one thing at a time: Prefer to give each entity (variable, class, function, namespace, module, library) **one well-defined responsibility**. As an entity grows, its scope of responsibility naturally increases, but its responsibility should not diverge.

A good business idea, they say, can be explained in one sentence. Similarly, each program **entity should have one clear purpose.**An entity with several disparate purposes is generally disproportionately harder to use, because it carries more than the sum of the intellectual overhead, complexity, and bugs of its parts. Such an entity is larger (often without good reason) and harder to use and reuse.

Avoid collecting several low-level abstractions into a larger low-level conglomerate. Implementing a complex behavior out of several simple ones is easier than the reverse.

**Don’t optimize prematurely.**

**Premature optimization is as addictive as it is unproductive**. The first rule of optimization is: **Don’t do it**. The second rule of optimization (for experts only) is: Don’t do it yet. Measure twice, optimize once.

We define premature optimization as making designs or code more complex, and so less readable, in the name of performance when the effort is not justified by a proven performance need (such as actual measurement and comparison against goals) and thus by definition adds no proven value to your program. All too often, unneeded and unmeasured optimization efforts don’t even make the program any faster.

So, by default, don’t focus on making code fast; focus first on making code as clear and readable as possible. Clear code is easier to write correctly, easier to understand, easier to refactor—and easier to optimize. Complications, including optimizations, can always be introduced later—and only if necessary.

**Don’t pessimize prematurely.**

Easy on yourself, easy on the code: All other things being equal, notably code complexity and readability, certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pessimized alternatives. This is not premature optimization; it is avoiding gratuitous pessimization.

Avoiding premature optimization does not imply gratuitously hurting efficiency. By premature pessimization we mean writing such gratuitous potential inefficiencies as:

* Defining pass-by-value parameters when pass-by-reference is appropriate.
* Using postfix ++ when the prefix version is just as good, postfix uses additional memory
* Using assignment inside constructors instead of the initializer list

**WRONG:**

|  |
| --- |
| Object::Object() {  m\_Member1 = 0;  m\_Member2 = false;  ...... } |

**GOOD:**

|  |
| --- |
| Object::Object(): m\_Member1(0), m\_Member2(false) {  ...... } |

* Not reducing spurious temporary copies of objects (especially in inner loops)
* Making repeated calls instead of a single call where possible (especially when that call is heavy)

|  |
| --- |
| // let's suppose we have a map which holds other maps as values  map["something"]["key1"] = "a value";  map["something"]["key2"] = "another value";  map["something"] ...  // You should take in consideration calling map["something"] is not free |

* Leaving code prone to memory access violation (like not checking pointers for NULL), again you should be avoiding pointers and preferring references.

Avoiding premature pessimization becomes particularly important when you are writing a library. You typically can’t know all contexts in which your library will be used, so you will want to strike a**balance** that leans more toward **efficiency and reusability** in mind, while at the same time not exaggerating efficiency for the benefit of a small fraction of potential callers. Drawing the line is your task, but the bigger fish to focus on is scalability and not a little cycle-squeezing.

As a side note here, when modifying existing code try not to leave garbage around, some old operations might be no longer needed because of the new change, you might also be tricked into following the old code and repeating the same mistakes the last developer has made.

**Know when and how to code for scalability.**

Beware of explosive data growth: Without optimizing prematurely, keep an eye on asymptotic complexity. Algorithms that work on user data should take a predictable, and preferably no worse than linear, time with the amount of data processed. When optimization is provably necessary and important, and especially if it’s because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.

This item illustrates one significant balance point between **“don’t optimize prematurely”** and **“don’t pessimize prematurely”**.

**Here’s the balance**: It would clearly be wrong to optimize prematurely by using a less clear algorithm in anticipation of large data volumes that may never materialize. But it would equally clearly be wrong to pessimize prematurely by turning a blind eye to algorithmic complexity, a.k.a. “big-Oh” complexity, namely the cost of the computation as a function of the number of elements of data being worked on.

**Minimize global and shared data.**

Sharing causes contention: Avoid shared data, especially global data. Shared data increases coupling, which reduces maintainability and often performance.

Avoid data with external linkage at namespace scope or as static class members. These complicate program logic and cause tighter coupling between different (and, worse, distant) parts of the program. Shared data weakens unit testing because the correctness of a piece of code that uses shared data is conditioned on the history of changes to the data, and further conditions the functioning of acres of yet-unknown code that subsequently uses the data further.

Objects that are at namespace scope, static members, or shared across threads or processes will reduce parallelism in multithreaded and multiprocessor environments and are a frequent source of performance and scalability bottlenecks. Strive for “shared-nothing;” prefer communication (e.g., message queues) over data sharing. Prefer low coupling and minimized interactions between classes.

**Ensure resources are owned by objects. Use explicit RAII and smart pointers.**

Don’t saw by hand when you have power tools: C++’s “resource acquisition is initiali- zation” (RAII) idiom is the power tool for correct resource handling. RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms.

Make sure that all resources are owned by objects. **Prefer to hold dynamically allocated resources via smart pointers instead of raw pointers**. Also, perform every explicit resource allocation (e.g., new) in its own statement that immediately gives the allocated resource to a manager object such as std::unique\_ptr (or shared\_ptr if you want to share it) otherwise, you can leak resources because the order of evaluation of a function’s parameters is undefined.

Avoid writing non leak-proof code in general, learn how to avoid common mistakes, there are a lot of features in the standard library starting with C++11 to help you with this.

For example:

**WRONG:**

|  |
| --- |
| class MyClass  {          ...          ObjectType1 \* m\_MemberPointer1;          ObjectType1 \* m\_MemberPointer2;          ObjectType1 \* m\_MemberPointer3;          ...          MyClass() // Constructor          {              m\_MemberPointer1 = new ObjectType1(...);              m\_MemberPointer2 = new ObjectType2(...);              m\_MemberPointer3 = new ObjectType3(...); // let's suppose this new fails          }            ~MyClass() // Deconstructor          {              delete m\_MemberPointer1;              delete m\_MemberPointer2;              delete m\_MemberPointer3;          }  } |

If the new in the example falis, then *std::bad\_alloc* exception will be thrown, the destructor of*MyClass* will not be called, **m\_MemberPointer1** and **m\_MemberPointer2** will leak because they have already been allocated. We could catch the exception and treat it, making sure we deallocate the first two members, but there is a nicer way with smart pointers (plus maybe you want to disable exceptions) :

We will use **std::unique\_ptr** to avoid leaking cases. What std::unique\_ptr does is documented in the <http://en.cppreference.com/w/cpp/memory/unique_ptr>, but we will describe it here as well, the s*td::unique\_ptr* keeps a reference to an object on the heap and acts as a normal pointer and is able to retains sole ownership and destroys that object when the unique\_ptr goes out of scope.

**GOOD:**

|  |
| --- |
| // OK - std::unique\_ptr exists since c++11 and has been added to the standard since tr1  class MyClass  {      ...      std::unique\_ptr<ObjectType1> m\_MemberPointer1;      std::unique\_ptr<ObjectType1> m\_MemberPointer2;      std::unique\_ptr<ObjectType1> m\_MemberPointer3;      ...      MyClass() // Constructor      {          //make\_unique allocates the object and returns a unique\_ptr.          m\_MemberPointer1 = make\_unique<ObjectType1>(...);          m\_MemberPointer1 = make\_unique<ObjectType1>(...);          m\_MemberPointer1 = make\_unique<ObjectType1>(...); // let's suppose this fails      }        ~MyClass() // Deconstructor      {          // these are no longer needed          //delete m\_MemberPointer1;          //delete m\_MemberPointer2;          //delete m\_MemberPointer3;      }  } |

If the allocation fails, the first two members will deallocate because their afferent unique\_ptr’s will be deconstructed.

# Code Structure

### **Namespaces**

Use them sparingly, they are useful to avoid name clashes between libraries but can make it more confusing if you use too many levels of them, so rule should be:

* + Keep only one namespace per main module/library ( don’t put more than 2 levels of namespaces ).
  + Make sure all classes belong to a namespace ( avoid global classes that could clash with other libraries )
  + **Never use "using namespace yournamespace"**, it promotes ambiguity in your code and might lead to problems in the future. I**f is used in a public header is even worse!**

### **One class per .h /.cpp file**

Use an explicit name for .h and .cpp file : NameOfTheClass.h / NameOfTheClass.cpp (**With some very few exceptions for small utility classes, so keep in mind not always**)

### **Use forward declaration to save compile time.**

Do not #include from a header file unless strictly necessary, every #include statement requires a file hit on hard drive and slows down you compilation process. Moreover it adds lot of dependencies between headers which can make your whole project to have to be recompiled when modifying a single header.

This is MyClassA.h

**WRONG:**

|  |
| --- |
| #include "MyClassB.h"  #include "MyClassC.h"  class MyClassA  {      public:          void SomeFunction(const& MyClassB);        protected:          MyClassC\* m\_Member;  }; |

**GOOD:**

|  |
| --- |
| //Use forward declaration  class MyClassB;  class MyClassC;  class MyClassA  {      public:          void SomeFunction(const& MyClassB);        protected:          MyClassC\* m\_Member;  }; |

This is MyClassA.cpp:

**GOOD:**

|  |
| --- |
| //it has to include headers though  #include "MyClassA.h"  #include "MyClassB.h"  #include "MyClassC.h"  void MyClassA::SomeFunction(const& MyClassB)  {          ...  } |

**When can I forward declare a type?**

Simple Rule – When you don’t need its **size** or **members**.

* + You need type’s **members** when you access them explicitly
  + You need **size** when **inheriting** or when you use it **as a member of your class**

**GOOD:**

|  |
| --- |
| // OK - forward declaration of IncompleteClass  class IncompleteClass;  class Foo  {      unsigned m\_Bar;      std::unique\_ptr<IncompleteClass> m\_Works; // using in smart pointers ok        // using as parameter, reference or pointer is ok      void FunctionOne(IncompleteClass iValueParam, IncompleteClass& iRefParam, const IncompleteClass\* iPtrParam);      // using as function return is ok      IncompleteClass FunctionTwo();        /\* When defining the above two functions in their source files, you will have to include header files \*/  }; |

### **Headers order**

Headers in each source file go in this order:

My current file: MyClassA.cpp

**GOOD:**

|  |
| --- |
| // my current file: MyClassA.cpp    // if you use precompiled header  #include "precomp.h" // Module header / Precompilation header    #include "OtherClass1.h" // other headers needed  #include "OtherClass2.h"  #include "OtherClass3.h"    #include <string> // System headers    /\* including current cpp header of the file after rest of the dependencies  will allow you not to include them inside it the header as well \*/    #include "MyClassA.h" // current cpp file header |

### **Statements**

* + One instruction per line otherwise your code will be harder to read and harder to debug.

**WRONG:**

|  |
| --- |
| Function1(); Function2(); Function3();  if (cond) function1() else function2(); |

* + Always delimit blocks by curly bracers (never ommit when block has one instruction)

**WRONG:**

|  |
| --- |
| if (cond)  DoSomething(); |

**GOOD:**

|  |
| --- |
| if (cond)  {      DoSomething();  } |

### **Passing arguments**

* + Prefer **passing arguments by reference** (unless PODs, Plain Old Data Types), don’t create temporary copies unless you have to
  + Use const for all your arguments passed by reference that should not be modified
  + Prefer passing arguments by reference then by pointers, pointers can point to bad memory (such as deleted memory but not set to NULL on pointer)

### **Structure/Class Memory Alignment**

Always try to declare Structure/Class members in the ascending order of their sizeof value, this will guarantee the most optimal memory alignment in regards of size for object instances.

**WRONG:**

|  |
| --- |
| // WRONG - members are not ordered (total size 24 chars)  class MyClass  {      int m\_Member1; // sizeof(int) = 4 (on most platforms)      long long m\_Member2; // sizeof(long long) = 8      int m\_Member3; // sizeof(int) = still 4  } |

**GOOD:**

|  |
| --- |
| // OK - members are ordered (total size 16 chars)  class MyClass  {      int m\_Member1; // sizeof(int) = 4 (on most platforms)      int m\_Member3; // sizeof(int) = still 4      long long m\_Member2; // sizeof(long long) = 8  } |

The above example explains for a single architecture, aligning everything can be hard to get right for all the platforms you are compiling for, take in mind sizeof() values vary, but it is good practice, overall it saves a lot of memory usage when taking lots of instances into account.

### **Function bodies and returns**

* + Avoid very long functions and/or with deep nested instructions.
  + Keep the life span of your variables short inside a function. Use braces to limit the variables lifetime.
  + If the function returns an existing member, return it by const reference.
  + In C++11 you can use move semantics to return temporary objects (rvalues), more about move semantics in another chapter (C++11 introduction).

**GOOD:**

|  |
| --- |
| // OK - move semantics (C++11 only)    std::string Function()  {      std::string stackString("something");      return stackString; // this will be moved, not copied.  } |

* + For performance reasons, avoid too many arguments inside a function ( above 3 arguments it is better to think of merging arguments inside a structure or splitting the function, depending on the platform, arguments can be quickly passed through CPU registers or require to be pushed on the stack )

### **Constants**

* + Do not use magic numbers or hardcoded value in the code – **NEVER** . If this is a constant declare it as “static const” ( not with a #define ).

**WRONG:**

|  |
| --- |
| newSpeed = oldSpeed \* 2.54f; // What is 2.54 ???? |

**GOOD:**

|  |
| --- |
| // OK - Declaring the constant in GlobalSettings.h    static const float k\_SpeedTurboFactor;  // In GlobalSettings.cpp  static const float k\_SpeedTurboFactor = 2.54f;    // In game code  newSpeed = oldSpeed \* k\_SpeedTurboFactor; |

* + Same thing as above can apply to strings

**WRONG:**

|  |
| --- |
| // WRONG - Do not hardcode strings that you repeat throughout the code.  objRoot["tile"] = ...  objRoot["tiIe"] = ... // Something wrong here...  objRoot["key"] = ... |

You are relying on the compiler to remove duplicate strings for memory improvement and you can easily mistype something without realizing it.

* + Use enumerations for integer values such as error codes or masks ( flags )

**GOOD:**

|  |
| --- |
| enum class EErrorsCodes  {      eOk = 0,      eFileNotFound,      eEndOfFile,      eCannotWriteToFile,      // etc...  };    enum class FClearMask  {      fClearBack = 1<<0,      fClearZ = 1<<1,      fClearStencil = 1<<2,      fClearAll = fClearBack | fClearZ | fClearStencil  }; |

### **Preprocessor Macros**

Preprocessor macros have this danger that they don’t handle namespacing and can be changed ( #undef/#define ) at any time making it very hard to detect… **AVOID** them unless needed for very specific case

Follow the following rules:

* + Guard your header files for duplicates

**GOOD:**

|  |
| --- |
| #ifndef \_\_MYFILE\_H\_  #define \_\_MYFILE\_H\_    // Here if the content of your file  #endif // \_\_MYFILE\_H\_ |

* + Use preprocessor MACRO functions as least as possible if at all, they can be inefficient or make debugging harder

**WRONG:**

|  |
| --- |
| #define MAX(x,y)&nbsp; ( x > y ? x : y)    // If this macro is called with a function, one of them will be called twice !    uint32 height = MAX(ComputeLeftHeight(),ComputeRightHeight());    // becomes  uint32 height = ComputeLeftHeight() > ComputeRightHeight() ? ComputeLeftHeight() : ComputeRightHeight();    // which is inefficient because of an useless second call |

* + Do not use preprocessor for constants ( number / strings / etc. ), use constants instead.
  + Do not #define/#undef in the middle of a source file (.cpp), defines should be at the top and should be consistent across the whole project.

# Code Style

**Prefer compile-time and link-time errors to run-time errors.**

**Don’t put off ‘til run time what you can do at build time**: Prefer to write code that uses the compiler to check for invariants during compilation, instead of checking them at run time. Run-time checks are control dependent and data dependent, which means you’ll seldom know whether they are exhaustive. In contrast, **compile-time checking is not control dependent or data dependent and typically offers higher degrees of confidence**.

The C++ language offers many opportunities to “accelerate” error detection by pushing it to compilation time. Exploiting these static checking capabilities offers you many advantages, including the following:

* Static checks are data- and flow-independent: Static checking offers guarantees that are independent of the program inputs or execution flow. In contrast, to make sure that your run-time checking is strong enough, you need to test it for a representative sample of all inputs. This is a daunting task for all but the most trivial systems.
* Statically expressed models are stronger: Oftentimes, a program that relies less on run-time checks and more on compile-time checks reflects a better design be cause the model the program creates is properly expressed using C++’s type system. This way, you and the compiler are partners having a consistent view of the program’s invariants; run-time checks are often a fallback to do checking that could be done statically but cannot be expressed precisely in the language.
* Static checks don’t incur run-time overhead: With static checks replacing dynamic checks, the resulting executable will be faster without sacrificing correctness.

**Examples**

1. Compile-time Boolean conditions. If you are testing for compile-time Boolean conditions such as sizeof(int) > = 8, use static assertions instead of run-time tests.
2. Compile-time polymorphism. Consider replacing run-time polymorphism (virtual functions) with compile-time polymorphism (templates) when defining generic functions or types. The latter yields code that is better checked statically.
3. Downcasting. If you frequently use **dynamic\_cast** (or, worse, an unchecked **static\_cast**) to perform downcasting, it can be a sign that your base classes offer too little functionality. Consider redesigning your interfaces so that your program can express computation in terms of the base class.

Some conditions cannot be checked at compile time and require run-time checks. For these, prefer to use assertions to detect internal programming errors.

**Use const proactively.**

**const** is your friend: Immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible and make const your default choice when you define a value: It’s safe, it’s checked at compile time, and it’s integrated with C++’s type system. Don’t cast away const except to call a const-incorrect function.

|  |
| --- |
| void Fun( vector<int>& v)  {      //...      const size\_t len = v.size();      //... 30 more lines ...  } |

When seeing **len**‘s definition above, you gain instant confidence about **len**‘s semantics throughout its scope (assuming the code doesn’t cast away **const**, which it should not do; see below): It’s a snapshot of v’s length at a specific point. Just by looking up one line of code, you know **len**‘s semantics over its whole scope. Without the **const**, **len** might be later modified, either directly or through an alias. Best of all, the compiler will help you ensure that this truth remains true.

**Don’t replace enums with strings**

String are there to help display human readable information, they are not intended to use as states for your application, parsing strings is not easy on the cpu. Always use **enums** for internal states of the program.

**Always initialize variables.**

Uninitialized variables are a common source of bugs in C and C++ programs. Avoid such bugs by being disciplined about cleaning memory before you use it; initialize variables upon definition.

In the low-level efficiency tradition of C and C++ alike, the compiler is often not required to initialize variables unless you do it explicitly (local variables, forgotten members omitted from constructor initializer lists). Do it explicitly. There are few reasons to ever leave a variable uninitialized. None is serious enough to justify the hazard of undefined behavior.

**A common misconception** about uninitialized variables is that they will crash the program, so that those few uninitialized variables lying around here and there will be quickly revealed by simple testing. On the contrary, programs with uninitialized variables can run flawlessly for years if the bits in the memory happen to match the program’s needs. Later, a call from a different context, a recompilation, or some change in another part of the program will cause failures ranging from inexplicable behavior to intermittent crashes.

**Declare variables as locally as possible.**

Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible.

Variables whose lifetimes are longer than necessary have several drawbacks:

* **They can’t always be sensibly initialized**: Never declare a variable before you can initialize it sensibly. Uninitialized variables are a pervasive source of bugs (see above about initializing variables)
* **They pollute their context with their name**
* **They make the program harder to understand and maintain**

**Avoid inheriting from classes that were not designed to be base classes.**

Some people don’t want to have kids: Classes meant to be used standalone obey a different blueprint than base classes. Using a standalone class as a base is a serious design error and should be avoided. To add behavior, prefer to add nonmember functions instead of member functions. To add state, prefer composition instead of inheritance. Avoid inheriting from concrete base classes.

**Prefer composition to inheritance.**

**Inheritance is the second-tightest coupling relationship** in C++, second only to **friendship**. **Tight coupling is undesirable** and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.

**Inheritance** is often overused, even by experienced developers. A sound rule of software engineering is to minimize coupling: If a relationship can be expressed in more than one way, **use the weakest relationship that’s practical**.

**Composition** has important advantages over **inheritance**:

* **Greater flexibility without affecting calling code** - A private data member is under your control. You can switch from holding it by value to holding by (smart) pointer without breaking client code.
* **Greater compile-time insulation, shorter compile times** - Holding an object by smart pointer, rather than as a direct member or base class, can also allow you to reduce header dependencies because declaring a pointer to an object doesn’t require that object’s full class definition.
* **Wider applicability**: Some classes were not designed to be bases in the first place. Most classes, however, can fulfil the role of a member.

**Inheritance** affords a great deal of power, including substitutability and/or the ability to override virtual functions. But **don’t pay for what you don’t need**.

**Prefer initialization to assignment in constructors.**

|  |
| --- |
| // WRONG class A  {      std::string s1;      std::string s2;      public:      A()      {          s1\_ = "Hello, ";          s2\_ = "world";      }      //...  };    // In reality, constructor's code is generated as if you wrote: class A  {      std::string s1;      std::string s2;        public:      A() : s1\_(), s2\_()      {          s1\_ = "Hello, ";          s2\_ = "world";      }      //...  };    // OK class A  {      std::string s1;      std::string s2;        public:      A() : s1\_("Hello, "), s2\_("world")      {      }      //...  }; |

\* For basic types (int, bool, double, etc...), since c++11 (on vs2012 does not work) you can assign a value directly in class declaration.

|  |
| --- |
| class Test  {      public:      int a = 10;  }; |

# Naming Conventions

## Code Formatting

    By code formatting we imply things like: number of spaces or tabs for indentation.

    Whether the opening bracket for a block lies on the same line with its instruction.

    How many spaces are between function arguments and parenthesis etc.

**It is recommended to use the same format in all Online Libraries**. This makes the code more easy on the eye.

## Naming Conventions

Before we dive in:

**CamelCase**  = capitalLetterForEachNewWord

**PascalCase** = LikeCamelCaseOnlyFirstLetterIsCapitalized

PascalCase is subset of CamelCase.

### **Language - one for all, English**

Use only English for all source code

|  |
| --- |
| // All the code, comments, variable and resources should be named in English.  // If you wish to add a comment in another language then also include the English translation. |

### **All struct/class/typedef have to be in PascalCase.**

**WRONG:**

|  |
| --- |
| class myClass { |

**GOOD:**

|  |
| --- |
| class MyClass { |

### **Use variable prefixes to denote scope in this format**

* + **g\_** for global variables (try to avoid using global variables)
  + **k\_** for global constant values
  + **s\_** for static class member variable ( which you should only use sparingly)
  + **m\_** for other class member variable
  + **i\_** and **o\_:** optionally, you can use only for function parameters i\_ for input parameter and o\_ for output parameter

Valid exceptions are very simple structures/math patterns ( vec3, rect ,etc..) where it is ok to have simple member names ( position.x , color.r etc. )

### Tabs vs. spaces

If you have to edit an old file, please use the file indentation (4 spaces or tabs).  
For new files please use tabs for indentation.  
**Do not change the indentation of an entire file! this will stop us to use svn blame easily.**

### **Name booleans as positive questions, avoid negative names.**

**WRONG:**

|  |
| --- |
| bool m\_NotMainCharacter;  bool m\_Redraw; |

**GOOD:**

|  |
| --- |
| bool m\_IsMainCharacter;  bool m\_HasToRedraw; |

### **Local variables have to start with lower case (CamelCase).**

**WRONG:**

|  |
| --- |
| void Function(...)  {      float MyLocal1;      int CurrentStep = 0;      ...  } |

**GOOD:**

|  |
| --- |
| void Function(...)  {      float myLocal1 = 0.0f;      int currentStep = 0;      ...  } |

### **Functions have to respect the following rules:**

* + Function naming has to be done in **PascalCase**.
  + Functions should try to have a very explicit name and **mandatory** some clear **comments** on what it does
  + If the function does any allocation, it has to descriptive name (**Allocate**, **Create**, etc ...)

**WRONG:**

|  |
| --- |
| Data& GetData()  {      //allocation happens here      ...  } |

**GOOD:**

|  |
| --- |
| /\*\* This function is used to allocate data for .....  \* @return a reference to the allocated data. \*/  Data& AllocateData()  {      //allocation happens here      ...  } |

* + Name functions as **Get/Set** when accessing a member directly
  + Do not name functions as **Get** if the operation is cumbersome, name **Find**/**Compute**instead.
  + Use complementary name for complementary functions such as **Start**/**Stop**, **Show**/**Hide**etc...
  + Do not return useless booleans, return booleans only if the function can fail in some way.

**USELESS:**

|  |
| --- |
| bool Player::IncreaseHealth(u32 iHealth)  {      m\_Health += iHealth;      return true;  } |

**GOOD:**

|  |
| --- |
| void Player::IncreaseHealth(u32 iHealth)  {      m\_Health += iHealth;  } |

* + **Function arguments** have to be named in using **CamelCase** and must not have any prefixes.

**OK:**

|  |
| --- |
| void Function1(int firstArgument, Object& secondArgument, const OtherObject& thirdArgument) |

## Important Notes About Naming:

### **Consistent Naming**

Remember to always be consistent with the naming standard we have set, do not forget badly named variables inside your code, try to maintain consistency at all times.

**WRONG:**

|  |
| --- |
| class MyClass  {      int m\_SomeOffset;      bool iDontKnowWhatImDoing;      int otherOffset;      ...  }; |

### **Do not make enums/constants/defines confusing.**

You should try not to make enums/constants/defines confusing because of their names, do not mix them together for the same purpose.

As it will be said in the next chapter, use enums more often, use them when you have to identify things (such as error codes) or to create masks, constants you should be using for different values (magic numbers) and finally **defines you should be avoiding**.