- Um es in einem Projekt anderen oder neuen Mitarbeitern zu ermöglichen den eigenen Code zu verstehen, oder bei Ausscheidung aus dem Projekt einem eventuellen Nachfolger die Möglichkeit zu geben, das Projekt weiter zu führen ist es unbedingt notwendig, den Code so lesbar und verständlich wie möglich zu halten. Kaum ein Code wird nur einmal programmiert und dann nie wieder angerührt, und oftmals auch nicht nur von einem selbst verwendet. Deshalb ist es wichtig sich an gewisse Gepflogenheiten und Notationen zu halten die das lesen des Codes vereinfachen und verständlich halten.
  Die Seite an sich ist nicht wirklich übersichtlich gehalten und sehr satirisch und ironisch geschrieben. Das meiste was auf dieser Seite steht sollte einem Programmierer eigentlich klar sein und spiegelt nur die "worst-ca„e"-Szenarien wieder

    - *Creative Miss-spelling*
      If you must use descriptive variable and function names, misspell them. By misspelling in some function and variable names, and spelling it correctly in others (such as SetPintleOpening SetPintalClosing) we effectively negate the use of grep or IDE (Integrated Development Environment) search techniques. It works amazingly well. Add an international flavor by spelling tory or tori in different theatres/theaters.
      Man kann nie wissen, ob jemand aus versehen Schreibfehler verursacht hat oder einfach nur versucht durch Schreibfehler die Aufmerksamkeit auf etwas zu ziehen, oder es von z.B. einer ähnlichen Variable zu differenzieren.

    - *Names From Other LanguagesUse foreign language dictionaries as a source for variable names. For example, use the German* punkt *for* point. *Maintenance coders, without your firm grasp of German, will enjoy the multicultural experience of deciphering the meaning.*
      Viele Leute ziehen es vor Namen aus ihrer eigenen Sprache zu verwenden (zB deutsch), was gerade für Programmierer mit anderen Sprachkenntnissen international hinderlich ist.

    - *Document How Not Why*
      Document only the details of what a program does, not what it is attempting to accomplish. That way, if there is a bug, the fixer will have no clue what the code should be doing.
      Es genügt nicht nur zu schreiben, wie das Programm funktionieren sollte, wenn es nicht funktioniert muss auch klar sein was das Programm überhaupt machen sollte. Dies ist auch gerade dann wichtig wenn ein Code teil ersetzt werden muss.

    - *Obsolete Code*
      Be sure to comment out unused code instead of deleting it and relying on version control to bring it back if necessary. In no way document whether the new code was intended to supplement or completely replace the old code, or

whether the old code worked at all, what was wrong with it, why it was replaced etc. Comment it out with a lead / and trail */ rather than a // on each line. That way it might more easily be mistaken for live code and partially maintained.*
Es ist verdammt unübersichtlich wenn man fast genausoviel code als Kommentar wie tatsächlichen Code hat, von dem es unmöglich ist herauszufinden, ob er zur Erweiterung oder als Ersatz dient.

- *Never Validate*
  Never check input data for any kind of correctness or discrepancies. It will demonstrate that you absolutely trust the company's equipment as well as that you are a perfect team player who trusts all project partners and system operators. Always return reasonable values even when data inputs are questionable or erroneous.
  Rein aus Faulheit oder Angst vor Komplexität keine Validierung oder Errormanagement durchzuführen, kann ein komplettes Programm unbrauchbar machen, ohne dass man den Fehler erkennen kann. Grundsätzliche gibt es keine Eingaben denen man komplett vertrauen sollte, nicht einmal seinen eigenen.

- In C++ gibt es schon beachtlich viele verschiedene Libraries für jegliche Anwendungsgebiete. So ist es selten notwendig eine eigene zu schreiben. Falls man allerdings doch einmal in den Genuss kommen möchte ist dieser Leitfaden sehr übersichtlich und verständlich gehalten.

  - Wiederverwendung von Libraries, base libraries
    Wie oben schon erwähnt, sollte man sich sehr genau erkundigen, welche Libraries schon bestehen und was schon verwendet wurde. Es ist Zeit und Ressourcen sparender eine Library zu verwenden oder zu erweitern als etwas komplett neu zu schreiben, dass es in anderer Form vielleicht sogar noch besser gibt.

  - Dokumentation
    Klare API-Dokumentation über Speicherverwaltung, Parametermanagement, Threadfähigkeit und eine gute Anleitung wie diese genutzt werden können is unerlässlich.

  - Asynchronität
    Operationen, die einen immensen Zeitaufwand benötigen sollten grundsätzlich auf Asynchronität zurückgreifen. Die Anwendung könnte sonst extrem user-unfreundlich ablaufen, da man immer auf die Bereitschaft des Systems warten müsste bis man etwas anderes starten könnte.
  - Namens-Notationen
    In C werden keine Namespaces zur Verfügung gestellt. Deshalb ist es sehr

wichtig darauf zu achten, dass entsprechende Präfixe verwendet werden, welche auf die entsprechende Library verweisen.

- Helpers & Deamons
  Je nach Anwendungsgebiet der Library ist es sinnvoll einen Deamon oder ein Helperprogramm zu nutzen, um zu gewährleisten, dass der Code auf keinen Fall durch eine andere Programmiersprache interpretiert werden kann.

Es hat für unsere Arbeitet einen eher geringeren Nutzen, so lange man nicht vorhat eigene Engines o.ä. zu schreiben. Für das Entwickeln von Games reicht es im aus schon bestehende Libraries zu verwenden.

- Eine Zusammenfassung die man als Entwickler sehr gut zu Rate ziehen kann, da sie sehr einfach gehalten, kurz und prägnant ist und außerdem über eine angenehme Übersichtlichkeit verfügt.
  - Immer daran denken GPL zu benutzen, so stellt man sicher, dass zwar jeder deinen Code verwenden kann, aber keiner die Möglichkeit hat ihn zu verändern, sodass der die Rechte des Originalcodes immer bei dir liegen.
  - Auch hier ist es immens wichtig ein festes Präfix für alle exportet Symbols festzulegen, um Probleme mit den Namespaces zu vermeiden und eine genaue Zuordnung zu gewährleisten.
  - Man sollte auch immer darauf achten, dass Callbacks in der API vermieden werden, da die Language Bindings Iteratoren benutzen. Außerdem ist es Komfortabler für die Entwickler.
  - Immer darauf achten, dass Libraries Errorcodes zurückgeben. Niemals dürfen exit() oder abort() aufgerufen werden, da die Libraries auch während kritischen Prozessen aufgerufen werden müssen, die sich von Fehlern erholen.
  - Man sollte immer sicherstellen , dass der Code darauf vorbereitet ist zu jederzeit mit einem unerwarteten Systemabbruch zurecht zu kommen. Es sollten bei einer solchen Situation keine „unsauberen“ oder temporären Dateien zurückbleiben. Daran muss frühzeitig gedacht werden, da es schwer ist sich im Nachhinein darum zu kümmern. GCC Destruktoren sind hier keine Lösung.

Da wir kein gesteigertes Interesse daran haben den Linuxkernel umzuschreiben, und C/C++ hauptsächlich als Werkzeug in Engines benutzen, ist es für unsere Arbeit eher Irrelevant.

6)
a)
256 kb / 64 byte = 4096 Blöcke / 8 fach-assoziativ = 512 Blöcke = 9 bit index
64 byte = 2 ^ 6 = 6 bit offset
64 adressbits - 9 bit index - 6 bit offset = 49 bit tag bits

b)
Anzahl Sets im Cache:
256 kb / (8 fach-assoziativ * 64 byte) = 512 Sets

c)

Anzahl Cache Lines
256 kb / 64 byte = 4096 Cache Lines