Best Practices

This page is intended to provide some best practice, at the very best with some code snippets.

- C++ related stuff
- CMake related stuff
- SVN and VC related stuff
- Assorted
- thread conditions
- Automatic Resource Management

C++

Everything related to C++ is written down here.

C++ calling conventions

C++ provides (as far as I know) two different kinds of parameter passing conventions:

- call by value
- call by reference
- call by pointer is the same as call by reference, therefore I only use two conventions.

What I often see is code like the following, which may result from Java experiences or other "only pointer" programming languages:

```
std::string fun(std::string astring) {
   std::string retVal = astring + "\n";
   return retVal;
}
```

The intention of the function call is clear but what does a compiler do internally? The compiler understands it as a "call-by-value" function call which results in a copy of the passed value. So if i have code like:

```
   #include <string>
   ...
   std::string in("hello world");
   std::string out = fun(s);
```

The compiler instantiates a new copy of the string stored in the variable `s`, resulting in a constructor call and the required operations to copy the internals of `s` (allocation of heap space in that case and a copy of the character array "hello world"). This is not directly visible to the programmer. This copy is placed on the stack, i.e. we need `sizeof(std::string)` additional stack space. To avoid that and to make sure that the compiler does not introduce some additional stuff we do not want, we can rewrite the function's signature to:

```
std::string fun(const std::string &astring);
```

As you can see, we replaced the paramater with a reference to a constant `std::string`. References in C++ are handled just like pointers, the only visible difference is that you don't need to dereference the variable to access the actual data. Pointers, as well as references, fit into one register since they address some memory location. The call of the function `fun` is reduced to put the address of variable `s` onto the stack (or into one of the function call registers depending on the hardware architecture and assembler calling conventions), i.e. we only need `sizeof(void*)` stack space. But more interestingly we avoid an unnecessary copy constructor call, depending on the variable that is being passed/copied, this can result in quite an improvement. The visible behaviour has not been changed at all. The only thing to make sure is that the implementation of the function can not assign a new value to the passed string but that is only a minor drawback which can easily be worked around.

Namespaces

Namespaces in C++ fullfill one main aspect: making "same" names globally available with different names. So for example you are going to implement a class called `vector` and you put into a namespace called `algebra` and include the standard STL `vector` header file also, you still can use both: your vector (algebra::vector) and the STL vector (std::vector). To make those names available without namespace prefix one can use the `using` keyword to import all symbols defined in a namespace. So far so good. But to avoid name clashes in otherwise completely unrelated code, one should be very careful with the `using` statements.

Putting `using namespace std;` into a header file which can and probably will be included by unknown clients will transparently import the whole std namespace into that client which can result in quite funny compiler errors. There is no problem however if you put such a using statement into a compilation unit, i.e. into a .cpp file because it is only visible to that compilation. If you want to be perfectly clear you can also put those statements at the top of functions.

Constructors and the `explicit` keyword

C++ is sometimes really inventive when it comes to automatically transform a value to a different type ;-) When you write a class and a constructor which accepts only one parameter - this also holds for constructors that take more than one parameter but with default arguments - C++ may transparently transform any occurrence of the parameter to the new type (if it think that could be a good idea - which in some cases can actually be the case).

```
class Month {
public:
  Month(int month) : month_(month) {}
  // ...
private:
  int month_;
};
```

The class above wraps an integer into a `Month` object. The problem here is, that any integer may transparently be made into a `Month` object which can result in undesired behaviour. To avoid that, C++ provides the `explicit` keyword, which means you

have to explicitly state that you really want a `Month` object. Suppose for example the following `Date` object:

```
class Date {
public:
  Date(int year, int month, int day) : year_(year), month_(month), day_
  // ...
};
```

As a programmer you have to make sure that you pass the right integers in the correct order, depending on the length of the argument list that can be quite tricky and will most of the time result in an API lookup or worse some bad behaviour at runtime, i.e. Date d(10, 6, 2008) (we wanted: Date d(2008, 6, 10)) will create an incorrect `Date` object, but the compiler will not complain about it. What one wants to have in that case is a compile time error if you pass the wrong arguments. No take a look at the `Month` class again, in the current state it doesn't help but if we write it like:

```
class Month {
public:
   explicit Month(int month) : month_(month) {}
   // ...
};
class Year //...
class Day //...
```

We can rewrite our `Date` class as:

```
class Date {
public:
  Date(const Year &y, const Month &m, const Day &d) // ...
};
```

What we gained now is the ability to get compile time errors when we create a Date object with wrong arguments.

So make sure that every constructor that only takes one parameter is `explicit` or double check that you know what you do. WARNING: don't make the copy constructor explicit!

Parentheses, curly braces, spaces

Make your code readable for (other) humans. Try to avoid any ambiguity.

- Always use parentheses to group boolean expressions.
- Use curly braces to make block borders explicitly visible.
- Always separate operators by spaces.

Do not write

```
return place != NULL ? place->getID() : "";
```

Write instead

```
return (place != NULL) ? place->getID() : "";
```

or even better

```
return ((place != NULL) ? place->getID() : "");
```

Do not write

```
if (data==NULL) _type = TYPE_EMPTY;
```

We are using macros. What, if TYPE_EMPTY is a macro? You have to look at the defining position of TYPE_EMPTY to be sure.

Write instead

```
if (data == NULL) { _type = TYPE_EMPTY; }
```

or even better

```
if (data == NULL)
{
  _type = TYPE_EMPTY;
}
```

No, this is no waste of space.

Do not write

```
  XMLCh* s = el->getAttribute(X("type"));
  if(XMLString::stringLen(s) > 0) type = string(S(s));
  else type = "";
  s = el->getAttribute(X("operationName"));
  if(XMLString::stringLen(s) > 0) operationName = string(S(s));
  else operationName = "";
  s = el->getAttribute(X("resourceName"));
  if(XMLString::stringLen(s) > 0) resourceName = string(S(s));
  else resourceName = "";
  s = el->getAttribute(X("selected"));
  if(XMLString::compareString(s,X("true")) == 0) selected = true;
  else selected = false;
  s = el->getAttribute(X("quality"));
  quality = XMLString::stringLen(s) != 0 ? atof(S(s)) : -1;
```

Write at least something like

```
XMLCh* s = el->getAttribute(X("type"));

if (XMLString::stringLen(s) > 0)
{
  type = string(S(s));
}
else
{
  type = "";
}

s = el->getAttribute(X("operationName"));

if (XMLString::stringLen(s) > 0)
{
  operationName = string(S(s));
}
else
{
  operationName = "";
}

...

quality = ((XMLString::stringLen(s) != 0) ? atof(S(s)) : (-1));
```

(BTW, are you sure, that `atof` is the right choice? What about defaulting to zero? What about Nan?)

Now, when it becomes clear, what you are doing, you better write

```
XMLCh* s = el->getAttribute(X("type"));

type = ((XMLString::stringLen(s) > 0) ? string(S(s)) : "");

s = el->getAttribute(X("operationName"));

operationName = ((XMLString::stringLen(s) > 0) ? string(S(s)) : "");

...
```

Maybe you want to factor out the boilerplate code, but this is painful in C++, so this would be a reasonable code fragment.

Proper member initialization with initialization lists

Browse the web for that topic.

Very short: Instead of

```
class C
{
 private:
  type1 x;
  type2 y;
 public:
  C (type1 x_, type2 y_) { x = x_; y = y_; }
}
```

write

```
class C
{
 private:
  type1 x;
  type2 y;
 public:
  C (type1 x_, type2 y_) : x(x_), y(y_) {}
}
```

Reason: The first version creates the object with the standard constructor and then assigns the values to x and y. The second one omits this assignment.

Very nice: There is not need to invent new names, this is valid and legal

```
class C
{
 private:
  type1 x;
  type2 y;
 public:
  C (type1 x, type2 y) : x(x), y(y) {}
}
```

There is a small pitfall using initialization lists: The order of initialization equals to the order of declaration of the class members. So

```
  C (type1 x, type2 y) : x(x), y(y) {}
```

and

```
  C (type1 x, type2 y) : y(y), x(x) {}
```

are both initializing first x and then y, since the declaration of x is before the declaration of y.

Comment (Alex): I checked this because i was not aware of that and it seems to work. But a funny remark is that i get a compiler warning (-Wall) if i try to do the latter case ;-)

```
// main.cpp
#include <iostream>

class A {
  public:
    A() { std::cout << "A() called" << std::endl; }
    A(int v) : v_(v) { std::cout << "A(" << v << ") called" << std::en
    int v() const { return v_; }
  private:
    int v_;
};

class B {
  public:
    B() { std::cout << "B() called" << std::endl; }
    B(int v) : v_(v) { std::cout << "B(" << v << ") called" << std::en
    int v() const { return v_; }
  private:
    int v_;
};

class FooAB {
  public:
    FooAB(int a, int b) : a_(a), b_(b) { }

    int a() const { return a_.v(); }
    int b() const { return b_.v(); }
  private:
    A a_;
    B b_;
};

class FooBA {
  public:
    FooBA(int a, int b) : b_(b), a_(a) { }

    int a() const { return a_.v(); }
    int b() const { return b_.v(); }
  private:
    A a_;
    B b_;
};

int main(int argc, char **argv) {
  std::cout << "initializing FooAB" << std::endl;
  FooAB fooAB(1, 2);
  std::cout << std::endl;
  std::cout << "initializing FooBA" << std::endl;
  FooBA fooBA(1, 2);
  std::cout << std::endl;
}
```

Size type, iterators

The type of `vector<T>::size()` is neither `int`, nor `unsigned int`. It is `vector<T>::size_type`!

So do not write

```
vector<T> v;
for (unsigned int i = 0; i < v.size(); ++i)
{
  f (v[i]);
}
```

but instead

```
vector<T> v;
for (vector<T>::size_type i = 0; i < v.size(); ++i)
{
  f (v[i]);
}
```

or even better use an iterator (if possible a const_iterator):

```
vector<T> v;
for (vector<T>::iterator it = v.begin(); it != v.end(); ++it)
{
  f (*it);
}
```

-Wall -Werror

Use this flags! If you really sure, that certain warnings are no errors, then disable this specific warning. Prefer local annotations to do so. For instance, to avoid the "unused parameter" warning, annotate the unused parameter with the `__attribute__((unused))`!

You can introduce these flags to CMake using:

```
ADD_DEFINITIONS(-Wall -Werror)
```

Increment, Pre- or Post- or what?

In general pre-increment is faster!

Reason: For post-increment the object must increment itself and then return a temporary containing its old value. Try to implement both, the pre- and the post-increment!

```
class C {
  // pre
  C& operator++() { ++some_internal_counter; return *this; }
  // post
  const C operator++(int) {
    C tmp = *this; // save a copy, a lot of memcpy
    ++*this; // increment
    return tmp; // return the old value
  }
};
```

So write

```
for (iterator it(v.begin); it != v.end(); ++it)
```

even for simple types like 'int' write

```
for (int i(0); i < N; ++i)
```

Rule of thumb: Never ever use post-increments.

Measurement:

```
rahn@lts017:~> cat inc.cpp; g++ -Wall -Werror inc.cpp -o inc.bin; time ./
#include <iostream>
#include <vector>

int main(int argc, char ** argv)
{
  const std::vector<int> v(1<<28);

  if (std::string("pre") == std::string(argv[1]))
    {
      std::cout << "pre" << std::endl;
      for (std::vector<int>::const_iterator i(v.begin()); i != v.end(); +
    }
  else if (std::string("post") == std::string(argv[1]))
    {
      std::cout << "post" << std::endl;
      for (std::vector<int>::const_iterator i(v.begin()); i != v.end(); i
    }
  else
    {
      std::cout << "construction only" << std::endl;
    }
  return EXIT_SUCCESS;
}
construction only

real    0m1.495s
user    0m0.796s
sys     0m0.692s
pre

real    0m6.024s
user    0m5.324s
sys     0m0.700s
post

real    0m9.061s
user    0m8.297s
sys     0m0.756s
```

Note, that the time ratio is not 6/9 == 2/3, it is (6-3/2)/(9-3/2) == 3/5!

Another thing with this fancy increment operators is to avoid something like

```
x = arr[++j];
```

Write instead

```
++j;
x = arr[j];
```

There are two reasons for this: a) Separation of concerns, namely of incrementing a counter and accessing an element of arr and b) Readability. Don't forget: We write Codes for other humans to read not for programs like compilers or interpreters. A

third reason is: Try to access the j-th element of some array arr2 in both versions.

CMake

When searching package via pkgconfig the command

```
   pkg_check_modules (<PACKAGE> REQUIRED package>' >= 1.12)
```

sets some variables that are used in the CMakeLists.txt files

```
<PACKAGE>_FOUND
<PACKAGE>_INCLUDE_DIRS

<PACKAGE>_LIBRARY_DIRS
<PACKAGE>_LIBRARIES

<PACKAGE>_STATIC_INCLUDE_DIRS
<PACKAGE>_STATIC_LIBRARIES
<PACKAGE>_STATIC_LIBRARY_DIRS
```

When we write our own Find<PACKAGE>.cmake file, we should define the same variables.

SVN and VC best practices

In this section I will shortly describe what nice things can be done with a version control system like SVN. Additionally I will also show you one can use 'git-svn' on top of an existing SVN-based project.

- SVN RedBook as a single HTML page
- SVN RedBook as a PDF document

SVN or Subversion is a version control system, that means it is able to keep track of any modifications within a given directory structure. It is able to recognize file or directory modification, creation and deletion (and several things more). The developers of SVN suggest to layout the project into three different directories: `trunk`, `branches` and `tags`. All three of them are just "random" names for directories but they are used with special meanings. The `trunk` directory contains the main develop line of a project, while `branches` and `tags` are directories containing "copies" of the trunk at a specific point in time (such a point in time is also called a "Revision").

Directory layout and their meanings

As described above the project should be split into three top-level directories:

- `trunk`
  - contains the main-development branch, i.e. the current status of the project
  - this "branch" should always be kept in a "good state", i.e. it should compile and test cases should be (mostly) working, see below for a reason
- `branches`

    - contain several subdirectories where each of them should be a copy of the trunk
    - those directories are used for developing new features or bugfixing
    - so, if you want to implement a new feature, like "A configuration system", create yourself a branch with
        - `svn copy <url-to-trunk> <url-to-tags>/config-system-implementation`
    - you have to make sure that the branch does not already exist, though
- `tags`
    - this directory is nearly the same as the `branches` directory but it is supposed to be "read-only"
    - its only purpose is to give a point-in-time (revision) a meaningful name
    - for example, if you are releasing a new version of your project and you are sufficiently happy with its status, give the current revision a name
        - `svn copy <url-to-trunk> <url-to-tags>/v1.0.0`
        - this creates a very space efficient copy of the trunk

Development cycle

This section discusses the main development cycle of a bug-fix or a new feature. The purpose of doing it that way is, that you won't disturb any other people working on the same project as you and you can still have the full potential of a version control system, i.e. you can check in your changesets, go back if you did something wrong and so on.

In our case, we have the following project layout:

- SDPA
    - trunk
        - sdpa
            - trunk
            - branches
            - tags
        - gwes
            - trunk
            - branches
            - tags
        - other sub projects
    - branches
    - tags

The first thing todo is to create a new branch of the trunk for the thing you want to implement. As stated above, the trunk should always be kept in a 'known good state', i.e. if you branch from it, you should already get a compilable version of the project, it is not nice to have broken code when starting a new branch. You create a new branch for the implementation of the configuration stuff for the 'sdpa' by issuing the following command:

- `svn copy <url-to-SDPA>/trunk/sdpa/trunk <url-to-SDPA>/trunk/sdpa/branches/config-system-implementation`
- now either checkout a fresh version of the branch:
    - `svn checkout <url-to-SDPA>.../config-system-implemenation sdpa-config`

- or switch your existing checkout to the branch
  - `svn switch <url-to-SDPA>.../branches/config-system-implementation`
- you should remember the revision-number the trunk had at which you made the copy, for example within the commit message of the 'copy' command, let's say it was Revision 42.

You now have access to a completely independent development branch and can go ahead hacking together the new feature. In regular occasions you should however integrate the changes that have happened to the main development (i.e. trunk) - this removes a lot of surprises at the end. But it is completely up to you when you incorporate those changes. To integrate the current version of the trunk into your branch, you do the following while being in your working copy of the branch:

- `svn merge <url-to-sdpa-trunk>`
- a special note here: this "easy" syntax only works with Subversion 1.5 and upwards, before that you were required to note down the exact revision number the trunk had when you branched, either by remembering it, or by nailing it down with:
  - `svn log --stop-on-copy`
  - this is 1. error-prone and 2. inconvenient, just make sure that you use a reasonable 'new' version of Subversion
- the second remark is, older versions of subversion did not track those "merge" operations, i.e. if you performed the latter command, you are required to remember what you merged. Suppose the trunk was at revision 80 by now and you know that you branched at revision 42, you are merging the differences between 42 and 80, i.e. `svn merge -r 42:80 <url-to-trunk>`. In your commit message for your branch you should note that down because you will need it later, i.e. "merged trunk to my development branch revisions 42 through 80". This is not required anymore with subversion 1.5 or later!

As stated above, to keep surprises low, you should do that step regularly. Fix any problems that have been introduced by the merge and commit your changes.

When you are completely finished with your implementation and implemented enough unit tests and merged the latest trunk revision into your branch, you are able to update the trunk with your code. Get hands on a clean checkout of the trunk, by either checking it out, by switching your working directory to the trunk or updating an already existing checkout, just make sure that the directory is completely clean and does not carry any local modifications:

- `svn status` should result in an empty output (check the '?' signs if anything is wrong)
- perform the merge:
  - `svn merge --reintegrate <url-to-your-branch>`
- compile and test it again, nothing should be wrong here, otherwise you did something wrong
  - commit your changes and state that you merged your branch into the main development tree
- delete the special branch

WARNING: The documentation states about the '--reintegrate' that you should not use that branch for anything afterwards, i.e. no merging from trunk and no

reintegration. Thus, delete the branch and recreate it again with the same name. A way more detailed description of this topic can be found in the redbook chapter 4.

Commit early, Commit often

A very good thing todo is to commit very often and in a logical way. That means, do a commit always when you are done with a logical code change. For example adding a new function to a class results in a modification of the header file and the source file, those two changes should be contained in a single commit describing that particular changeset. With current versions of svn it is possible to modify more files and only commit "some" of them related to a single changeset. This is done using the new "svn changelist". The following command adds a list of files to a new "changelist":

```
$ svn changelist <name> <files>+
```

When you do a **svn status** you will see the output split into those different changelists. **svn commit** supports also a new parameter called **--changelist <name>** that allows you to commit only those files related to a given changeset. Those two things are especially important when you edited more files and the edited files belong to different logical changes, you don't have to "throw" your changes away, you can just selectively commit them in a logical way.

Adding new files to the repository for example should always be contained in a single commit, i.e. only the addition of those files. Usage of the new files can be commited afterwards. This makes it possible to selectively merge the addition of the new files into a different branch etc.

`svn diff` and layout changes

Always do a local `svn diff` before committing. Be sure, that your changes affect one single concern only, that is well described in the LOG message.

Do not mix concerns!

Especially do not mix layout changes with content changes. It makes `svn diff` useless if 5 lines of real changes are interspersed with 500 lines of indentation changes.

Best is to avoid layout changes completely! If your development environment supports context sensitive layout, then please teach it to **display** your favorite layout style, but not to **save** it. Others may use other environments.

Special words on distributed VC systems

I am currently testing the "Git" version control system on top of the SDPA-SVN. It differs in some fundamental things from a centralized approach as SVN is. Git is one of the newer open-source distributed version control systems. The basic difference is, that when you make a "checkout" you actually clone the complete history of a version controlled directory with all changes and logs and whatnot. After that you do not need any communication with the source of you clone, you are completely disconnected from it - however, it is of course possible to reintegrate your changes back to the source. The advantages in short:

- no central server component, anyone's repository can serve as a source for cloning, merging etc.
- absolutely no communication required to perform branching/tagging etc.
- there are a lot more advantages but those are the ones i currently see relevant for our development
    - I usually program on the laptop or somewhere and not do not always have an internet connection available but i still want to use all features related to branching and merging, it is impossible to do that with SVN or any client-server based approach.

Git (and probably some of the other systems, but I did not verify that) has a client which is able to be backed by an SVN repository, that means:

- you perform a usual clone of an SVN based repository
- get a complete git repository out of it
- can do anything you like, branch around, do everything git provides
- at some later point you may want to commit your changes back to the trunk or some branch of the SVN repository

Let's get started

You create the clone of the SVN repository with, warning most of the information applies only to recent versions of git/git-svn, i used git-1.6.3.2:

- `git svn clone -s https://svn.itwm.fhg.de/svn/SDPA/trunk/sdpa`
- this clones the 'sdpa' subproject using the standard SVN layout (specified by -s), i.e. it uses trunk, tags, branches
- you now end up with a new directory called 'sdpa' that contains a '.git' directory keeping all the information required for git

It is also possible to clone the whole SDPA project and tell git-svn about the subprojects afterwards. The steps are a little bit more complicated:

- `git svn clone --add-author-from --use-log-author --username=<your-login> --stdlayout https://svn.itwm.fhg.de/svn/SDPA`
- this can take quite a while to complete (make sure you have enough coffee around)
- `git branch -a` does not show much since we don't have any branches or tags on the top level, we need to introduce our subprojects to git
- enter the just created git repository in SDPA and edit the file `.git/config`
- its contents should look somehow look like the following:

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  autocrlf = false

[svn-remote "svn"]
  url = https://svn.itwm.fhg.de/svn/SDPA
  fetch = trunk:refs/remotes/trunk
  branches = branches/*:refs/remotes/*
  tags = tags/*:refs/remotes/tags/*

# up to here git-svn created the entries the following stuff must be
# for each subproject add a svn-remote section and tell git-svn abou

[svn-remote "sdpa"]
  url = https://svn.itwm.fhg.de/svn/SDPA/trunk/sdpa
  fetch = trunk:refs/remotes/sdpa/trunk
  branches = branches/*:refs/remotes/sdpa/*
  tags = tags/*:refs/remotes/sdpa/tags/*

[svn-remote "gwes"]
  url = https://svn.itwm.fhg.de/svn/SDPA/trunk/gwes
  fetch = trunk:refs/remotes/gwes/trunk
  branches = branches/*:refs/remotes/gwes/*
  tags = tags/*:refs/remotes/gwes/tags/*

[svn-remote "fhglog"]
  url = https://svn.itwm.fhg.de/svn/SDPA/trunk/fhglog
  fetch = trunk:refs/remotes/fhglog/trunk
  branches = branches/*:refs/remotes/fhglog/*
  tags = tags/*:refs/remotes/fhglog/tags/*
```

- now you can do a `git svn fetch` to update your git repository
- if you do a `git branch -a` you should get something like:

```
* master
  remotes/fhglog/trunk
  remotes/gwes/trunk
  remotes/sdpa/krueger
  remotes/sdpa/machado
  remotes/sdpa/rotaru
  remotes/sdpa/trunk
  remotes/trunk
```

- that means, git knows about all those subproject branches
- if you do an `ls` in the current directory, you see the whole project, you definitely do not want that ;-)
- to work with one of those branches there, do a `git co -b my-branch-name remotes/$project/trunk` and replace the last trunk with the branch you want to work with.
- an `ls` should now show only the files that belong to that branch, perfect.

- an `git svn dcommit` then goes to the specific branch you are working with, see below
- open problems:
  - one can create remote svn branches using the command `git svn branch foo`
  - i have no idea how to remove that when one is finished with it ;-(

Standard work-cycle

The standard procedure if you do some changes are nearly the same, one difference is that git automatically keeps track of 'changesets' and you manually have to specify which files you actually want to commit:

- `git status` prints a list of files that are "tracked", i.e. under version control and "untracked" files
- the 'tracked' files are further split into 'modified but not staged' and 'staged changes'
- git differentiates your checkout into the working directory and the so-called "staging area", to add a file to your next commit, you do a `git add <path>`, to see what your commit will be about, call `git diff --cached` and you will only see the diff output of those files that have been staged. When you are satisfied with your changeset, commit it with `git commit`. Proceed the same way with the remaining unstaged/untracked files.
- all those things happen completely local in respect to the underlying svn repository!

To synchronize your git repository with the SVN repository you have to get those changes:

- `git svn fetch` will get all commit made to the repository (to all branches)
- `git svn rebase` will move your git HEAD to the svn HEAD

To commit your accumulated changes back to the SVN repository:

- `git svn dcommit` this will perform single commits for each single commit you have performed

Interactive changeset creation

A very nice thing (but i did not play with it yet) is the possibility to interactively create your changeset. You can pass the `--patch` option to the `git add` call and your editor will pop up to present you a list of changes (hunks), now you can selectively add changes that are related to a logical modification to the staging area and perform a commit afterwards. It is even possible to interactively modify and split the hunks into smaller changes. I think this is a very convenient feature if you made changes to a single file that are completely unrelated (i.e. belong to different logical changes) but you still want to commit in a proper way. Another and more powerful version of `--patch` is `--interactive`, i cannot tell anything about it yet though.

Branching with git

If you want to know what branches exist for the current project or if you want to

create a new branch, you use the `branch` command:

- `git branch` shows the current branch (marked by an asterisk) you are on and all available local branches
- `git branch -a` shows all branches as well as the remote branches
- `git branch <name> [<start>]` creates a new branch with the given name and bases it on <start> which is per default the current head and can be either a changeset, tag, another branch
    - to create a new branch and immediately check it out, you use:
        - `git checkout -b <branch-name>`
- with git svn you cannot just checkout a remote svn branch, the list you get via `git branch -a` contains all branches (local and remote). To checkout one of the remote branches, you create a local branch from it: `git checkout -b <local-branch-name> remote-branch`, for example, when i want to checkout the stuff Tiberiu does, i do `git checkout -b rotaru-br rotaru`.

Merging with git

Git knows a lot more about merging than Subversion does.

- to merge the "trunk" into your current development, type:
    - `git merge master`
    - this results in a list of modified and conflicting files
    - fix the problems, test and do the commit

Stashing away your work

A very nice feature of git is to `stash` work. You sometimes code a bit and figure out that there is some bug you want to fix or a small feature you could implement right now. What do you do with the modifications you already done? In svn you would be required to commit the incomplete changes i guess and remember what you wanted to do etc. With git, you just push your changes on a stash, it's easy like that. After the stash operation (`git stash save` you end up with a complete clean working index pointing to the HEAD of your branch, all your modifications have been saved away. You are now able to merge some other changes, fix the bug, implement the feature etc. When you are done with your small change, you can easily perform a `git stash pop` to get your prior work back.

Testing partial commits

You can use git stash save --keep-index when you want to make two or more commits out of the changes in the work tree, and you want to test each change before committing (excerpt from `git help stash`):

```
# ... hack hack hack ...
$ git add --patch foo            # add just first part to the index
$ git stash save --keep-index    # save all other changes to the stash
$ edit/build/test first part
$ git commit -m ´First part´     # commit fully tested change
$ git stash pop                  # prepare to work on all other changes
# ... repeat above five steps until one commit remains ...
$ edit/build/test remaining parts
$ git commit foo -m ´Remaining parts´
```

Pitfalls

empty directories

Git does only track files, so if you have an empty directory, it will not be tracked by git in any ways, either you add some empty dotfile into the directory or just ignore it, there can occur some problems in relation to svn, because svn tracks empty directories but they will not appear in your git checkout.

traces from other branches etc

Sometimes you may run into problems when switching between branches where git does not automatically files belonging to the branch you just switched away from. In that case a `git status` will report those files and directories as untracked, have an eye on them, maybe add some of them to the index with `git add`, any files (and directories) you don't want to have, can be removed with `git clean` and some flags to it. `git clean -n -d -x` performs a dry run `-n` and will show you which directories (`-d`) and otherwise ignored files `-x` will be removed - if you are confident, replace the `-n` with an `-f` to force the operation. Usually git is configured to require the `-f` flag.

Interesting links

- http://git.or.cz/course/svn.html
- http://blog.nuclearsquid.com/writings/git-add
- http://www.kernel.org/pub/software/scm/git/docs/user-manual.html
- http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html

Assorted

Make debug output parsable

Ensure, your debug output is easily parsable by standard tools like `awk`, `sort`, `perl`. Normally such tools use spaces as field separator. So separate fields by spaces!

Instead of

```
Class.method(param1,param2,param3)="Value"
```

produce something like

```
Class.method ( param1 , param2 , param3 ) = "Value"
```

Instead of

```
Workflow with id="name of workflow" is not available!
```

produce

```
Workflow with id = "name of workflow" is not available!
```

or even omit the double quotes and produce

```
Workflow with id = name of workflow is not available!
```

The name of the workflow is everything between `id =` and `is not available`, so the double quotes are superfluous. In fact, omitting the double quotes makes the code more readable.

thread conditions

```
#include <pthread.h>
class bla {
  public:
   bla() :
      m_ThreadCondition(PTHREAD_COND_INITIALIZER)
   {
     pthread_mutex_init(&m_qmutex, NULL);
   }

   ~bla() {
     pthread_mutex_destroy(&m_qmutex);
   }
    void Lock() { pthread_mutex_lock(&m_qmutex); }
    void Unlock() { pthread_mutex_unlock(&m_qmutex); }

    pthread_mutex_t& GetMutex() { return m_qmutex; }

    pthread_mutex_t& GetConditionVariable() { return m_ThreadCondition;;

  private:
    pthread_mutex_t m_qmutex;          /**< Thread mutex */

    pthread_cond_t m_ThreadCondition;  /**< Thread Condition variable */
};




bla cBla;
const struct timespec *pWaitTime;


// Thread 1
// Wait for signal of other tread
cBla.Lock();
while (!condition) {
  pthread_cond_wait( &cBla.GetConditionVariable(), &cBla.GetMutex() );
  // or
  pthread_cond_timedwait( &cBla.GetConditionVariable(), &cBla.GetMutex()
  // check return value (especially for ETIMEDOUT)
}
// mutex is locked again
{
  critical section
}
cBla.Unlock();

// Thread 2
// send signal to other thread
pthread_cond_signal( &cBla.GetConditionVariable() );
```

Automatic Resource Management

The following describes a very simple exploit of constructor/destructor usage to provide an automatic management of resources. As an example, I chose the management of a "lock" resource, but it can be applied to just any kind of resource which has a meaning of "acquire" and "release".

The motivation for this automatic management is simply the following: within some function, you get hands on some resource and you are supposed to release it when you exit the function. This might sound very easy to accomplish, but always remember: what happens if your function exits at a point you don't expect (a subroutine might throw an exception for example), but lets put this into code. The first example shows the problem:

```
class C {
  public:
    explicit C(Resource *r) : res_(r) {}

    void m() {
       res_->acquire();

       // do something

       if (have_to_call_f) {
          f();
       }

       // do something

       res_->release();
    }
  private:
    void f() {
      throw std::runtime_error("I am a very bad function!");
    }

    Resource *res_;
};
```

As you can see, we have a very simple class with just one public method. This function acquires some resource, does some computation and then decides whether to call another function, f, or not. Unfortuanately, this function f has quite some problems and throws an exception which we did not expect! What happens? Right, we exit our function m without getting to the point where we release the resource - typically this results in a deadlock or an "out-of-resources" error. The worst part however is, that this behaviour only occurs, **when** we have to call f() (this might only be the case in 0.01% of all cases, a really hard to find bug...).

So, what is the solution? Easy: let the compiler do all the dirty work for us ;-)

```
class ResourceMgr {
  public:
  explicit ResourceMgr(Resource *resource) : res_(resource)
  {
     res_->acquire();
  }

  ~ResourceMgr()
  {
     res_->release();
  }
  private:
    Resource *res_;
};

// modification of class C
class C {
  public:
    void m() {
       ResourceMgr resource_mgr(res_);

       // do something

       if (have_to_call_f) {
          f();
       }

       // do something
    }
};
```

That's all we have to do ;-) What will happen now? When f throws an exception, the compiler was so kind to add code for the "clean-up" stuff he has to do, like cleaning up the stack of the "m" function call - including the call to ResourceMgr::~ResourceMgr? which will release our resource.

Benifits: clearly less errors, less code, less thinking.

Checking gcc compiler version

GCC comes with some macros to identifie himself:

```
__GNUC__
__GNUC_MINOR__
__GNUC_PATCHLEVEL__
```

e.g. gcc 3.2.1

```
__GNUC__ = 3
__GNUC_MINOR__ = 2
__GNUC_PATCHLEVEL__ = 1
```