
Configurando seu ambiente de trabalho¹²

1 Controle de versão com git

O versionamento com `git` oferece quatro benefícios principais:

- A possibilidade de desfazer alterações no código que possam ter introduzido bugs.
- O backup contínuo do seu código em um repositório remoto.
- A possibilidade de gerenciar seu repositório localmente quando não se tem acesso à internet.
- A facilitação do trabalho em equipe a partir de um único repositório.

Para usar o `git`, você pode versionar um diretório já existente ou criar um repositório a partir de um diretório vazio. Nesta disciplina, utilizaremos a ferramenta GitLab para hospedar e gerenciar nossos repositórios.

1.1 Comandos básicos do git

O `git` é um sistema de controle de versão complexo e poderoso, mas nesta disciplina nos ateremos aos seus comandos mais simples.

git clone <url> <dir> Copia um repositório existente no endereço <url>, colocando seu conteúdo em um diretório de nome <dir>. Utilizado quando não se possui arquivos locais antes da criação do repositório.

git status Informa o status dos arquivos sob versionamento do subdiretório onde você estiver:

1. Fora de versionamento (*Untracked*), quando o arquivo ainda não está sob o controle de versão.
2. Novos arquivos (*New file*), quando novos arquivos foram adicionados desde o último *commit*.
3. Modificados (*Modified*), arquivos sob controle de versão que foram alterados desde o último *commit*.

git add <arquivo> Adiciona um arquivo ao controle de versão. Note que este arquivo só será consolidado após um *commit*.

git commit <dir> -m “<mensagem>” Realiza um *commit* (consolidação local) do subdiretório <dir>, com uma mensagem de log <mensagem>.

git checkout <arquivo> Reverte as alterações feitas a um arquivo desde o último commit.

git push <nome_repositorio> <branch> Envia os commits do ramo <branch> realizados localmente desde a última sincronização para o repositório identificado pelo nome <nome_repositorio>. Nesta disciplina, trabalharemos sempre com um ramo único (*master*) e um único repositório (*origin*), isto é, use sempre o comando **git push origin master**.

git pull Busca os commits efetuados por outros usuários (ou em diferentes máquinas) no repositório.

¹Roteiro de estudo fornecido na disciplina de Linguagem de Programação 1 (LP1) da Universidade Federal do Rio Grande do Norte (UFRN).

²Autor: Leonardo Bezerra.

1.2 Criando um novo repositório no GitLab do IMD

Criando um grupo de acesso

1. Acesse <http://projetos.imd.ufrn.br> e faça seu login (ou cadastro).
2. Clique em **Groups**, no menu lateral e, em seguida, no botão **New group**, na barra superior.
3. No campo **Group path**, digite *Nome-Sobrenome-LP1-2017-1-T05*.
4. Nas opções **Visibility level**, escolha a opção *Private*.
5. Clique no botão **Create group**.

Configurando permissões do grupo

6. Clique em **Members**, no menu lateral.
7. No campo **People**, digite *leobezerra*.
8. No campo **Group access**, escolha a opção *Reporter*.
9. Clique no botão **Add users to group**.

Criando um novo projeto

10. Clique em **Group**, no menu lateral (ou, se você estiver em outra página, acesse a página do grupo).
11. Clique no botão **New project**.
12. No campo **Project path**, digite um nome para seu repositório (neste exercício, use *lab00*).
13. Nas opções **Visibility level**, escolha a opção *Private*.
14. Clique no botão **Create project**.
15. Siga as instruções que aparecem na página inicial do projeto (**Command line instructions**) Note que você deverá seguir as instruções para quem vai começar com um diretório vazio (*Create a new repository*).

2 Organizando os arquivos do seu projeto

A estruturação de projetos *open source* em C++ costuma seguir algumas regras, separando por exemplo as assinaturas das funções (arquivos-cabeçalho **.h**, ou *headers*) das implementações das funções (arquivos-fonte **.cpp**, ou *sources*). Os principais subdiretórios utilizados para uma boa organização de projetos C++ são:

- **include**: headers.
- **src**: sources.
- **application**: source da sua aplicação.
- **bin**: arquivos executáveis.
- **lib**: bibliotecas adicionais.
- **build**: arquivos-objeto (**.o**).
- **doc**: documentação do código.
- **test**: testes unitários.

2.1 Baixando e versionando o projeto

Para baixar o arquivo zip contendo o projeto modelo, acesse a turma virtual do SIGAA e localize os arquivos adicionados ao tópico “*Aulas 03 e 04*”. Descompacte este arquivo no diretório que contém seu repositório git, criado na etapa anterior. Note que os diretórios **doc** e **lib** não foram utilizados neste modelo, uma vez que ainda não estamos trabalhando com documentação e bibliotecas externas. O exemplo utilizado é o do primeiro exercício do *Laboratório 0*, sendo que os arquivos source e da aplicação (*lab00.cpp*) estão incompletos.

Para adicionar os novos arquivos ao controle de versão, execute os seguintes comandos no terminal. Note que você deve estar no diretório raiz do seu repositório:

1. Renomeie a pasta modelo.

```
mv modelo conversor
```

2. Adicione a pasta conversor e seu conteúdo ao versionamento.

```
git add conversor
```

3. Confirme que o diretório conversor e seu conteúdo foram adicionados.

```
git status
```

4. Crie um commit (consolidação local):

```
git commit . -m "conversor: projeto modelo do lab00."
```

5. Envie as alterações do log de commits pro GitLab (origin).

```
git push origin master
```

2.2 Completando o projeto modelo

Seguindo a estrutura de arquivos apresentada, implemente as funções que estão faltando nos arquivos source e application. Você pode (deve) reutilizar o código enviado pelo SIGAA no Laboratório 0.

3 Conhecendo o compilador g++

O compilador *open source* mais utilizado em ambientes Unix é o GCC (*GNU Compiler Collection*). Especificamente para C++, utilizamos a versão g++. As opções (*flags*) mais utilizadas são descritas abaixo:

Opções de saída

- o <bin>: gera um arquivo binário de nome bin. Note que qualquer caminho (diretório) já existente pode ser especificado para receber este arquivo binário. Caso nenhum nome de binário seja especificado, é gerado um arquivo a.out.
- c: compila os arquivos-fonte (*sources*), mas não gera um arquivo executável (não faz a link-edição). O produto da compilação são arquivos-objeto (.o), havendo um para cada arquivo source. Esta opção pode ser usada conjuntamente com a opção -o, mas neste caso apenas para especificar o nome do arquivo-objeto de saída.

Opções de inclusão

- I <dir>: considera que existem arquivos-cabeçalho (*headers*) no diretório especificado. É necessário quando se possui headers locais que se deseja incluir, mas elas não estão no mesmo diretório onde se invoca o g++.
- l <lib>: inclui a biblioteca lib no processo de link-edição. Caso esta biblioteca não esteja em um caminho padrão do sistema, é necessário utilizar também a opção -L <dir>, detalhada abaixo.
- L <dir>: considera que existem bibliotecas (*libs*) no diretório especificado. É necessário quando se possui bibliotecas locais que se deseja incluir, mas elas não estão no mesmo diretório onde se invoca o g++.

Notem que, para este primeiro momento do curso, não será necessário utilizar a opção -L.

Opções de versão

- std=c++XX: configura o g++ para a versão XX do C++. A flag -std=c++11, por exemplo, configura o compilador para trabalhar com o C++11.
- ansi: forma alternativa de configurar o compilador para a versão C++99. É equivalente à opção -std=c++99.

Opções de alertas

- Wall: ativa todos os alertas (*warnings*) de compilação. Útil para evitar erros em tempo de execução.
- pedantic: caso a versão ANSI (C++99) esteja sendo adotada, ativa warnings mais exigentes para reduzir possíveis erros em tempo de execução.

Opções de otimização

- Ox: configura o g++ para utilizar o nível de otimização X. Os níveis de otimização variam de 0 (-O0, sem otimização) a 3 (-O3). Níveis de otimização mais altos levam a tempos de execução menores, ao custo de um maior tempo de compilação.

3.1 Compilando o projeto modelo

Existem diferentes formas de compilar o projeto modelo. As principais estão detalhadas abaixo:

```
g++ -o bin/lab00 application/lab00.cpp src/conversor.cpp -I include -Wall -O3 -ansi -pedantic
```

- executa os processos de compilação e link-edição. Os arquivos objetos gerados durante o processo são removidos automaticamente após a conclusão da link-edição.

```
g++ -c -o build/conversor.o src/conversor.cpp -I include -Wall -O3 -ansi -pedantic
g++ -c -o build/lab00.o application/lab00.cpp -I include -Wall -O3 -ansi -pedantic
g++ -o bin/lab00 build/*.o -Wall -O3 -ansi -pedantic
```

- separa a compilação em arquivos-objeto do processo de link-edição. As duas primeiras chamadas ao `g++` executam apenas a compilação em arquivos-objeto. A segunda chamada faz a link-edição, gerando o executável a partir dos arquivos-objeto fornecidos.

3.2 Compilando com um Makefile

Projetos grandes rapidamente se tornam de difícil gerenciamento para compilação manual. Dentre as principais ferramentas utilizadas para automação da compilação, o `make` é um programa simples porém suficiente poderoso para projetos de pequena escala.

Para compilar um projeto usando `make`, basta criar um arquivo `Makefile` descrevendo as regras de compilação do projeto e executar o comando `make`. Nesta disciplina, um arquivo `Makefile` genérico o suficiente foi disponibilizado junto ao projeto modelo.

Extra! – aqueles que explicarem o arquivo `Makefile` do modelo até a próxima aula ganharão 1,0 ponto extra nesta unidade.

4 Testes de software

Para aumentar a confiabilidade do seu código, é importante executar testes que verifiquem se o comportamento do seu código é o esperado. Dentre os diversos tipos de testes possíveis, nesta disciplina utilizaremos predominantemente os testes de **unidade** e de **aceitação**.

Testes de unidade são testes para verificar a confiabilidade de unidades de código (procedimentos). No projeto modelo, existe um arquivo `test/t_<arquivo>.cpp` para cada source `src/<arquivo>.cpp`. Para executar os testes de unidade, use `make test`.

Testes de aceitação são testes que verificam se o código satisfaz os requisitos apresentados na especificação do software. Nesta disciplina, serão disponibilizados *scripts* para testes de aceitação a cada exercício de implementação.