Metaprogramming

ABC CSE Winter School

What is metaprogramming?

Metaprogramming is a term that is used to describe everything aside coding that is utilized during development or makes the process of your work more efficient

Metaprogramming includes (not limited to): managing dependencies, build systems, testing, continuous integration systems etc.

Build systems

If you develop a C++ program and you want to see its output what do you do? You run many compilation commands and see the output. Is there a way to make this process faster?

Yes, build systems are “built” specifically for that purpose. There are many types of build systems. At their core, they’re all similar.

You define **target** (thing you want to build), **dependencies** (things target needs to be built), and **rules** (commands that do something with dependencies to create target)

Let’s see an example of make

make

```
target: dependency1 dependency2
	command1
	command2
	...
```

In make, the first target also defines the default goal. If you run make with no arguments, this is the target it will build.

make is a programming language like bash-script, but slightly different

Dependency management

Complex software have dependencies that are themselves projects.

You might depend on installed programs (like python), system packages (like openssl), or libraries within your programming language (like matplotlib). These days, most dependencies will be available through a repository that hosts a large number of such dependencies in a single place, and provides a convenient mechanism for installing them. (apt, snap in ubuntu, PyPi for python libraries etc.)

Versioning

Most projects that other projects depend on issue a version number with every release. Usually something like 8.1.3 or 64.1.20192004. They are often, but not always, numerical. Version numbers serve many purposes, and one of the most important of them is to ensure that software keeps working.

For example, when I use a particular library in my project and the library owner renames some function, in that case my project is no longer runnable. Versions help avoiding such issues and allow me to state “I am using that version for my project, let me use that version when I build my project”

Versioning

There are many ways to specify version, one of them is semantic versioning.

It has following syntax (uses numbers): MAJOR.MINOR.PATCH

MAJOR version when you make incompatible API changes,
MINOR version when you add functionality in a backwards compatible manner, and
PATCH version when you make backwards compatible bug fixes.

Versioning

When working with dependency management systems, you may also come across the notion of **lock files**. A **lock file** is simply a file that lists the exact version you are *currently* depending on of each dependency. Usually, you need to explicitly run an update program to upgrade to newer versions of your dependencies.

An extreme version of this kind of dependency locking is **vendoring**, which is where you copy all the code of your dependencies into your own project. That gives you total control over any changes to it, and lets you introduce your own changes to it, but also means you have to explicitly pull in any updates from the upstream maintainers over time.

Continuous integration

Continuous integration, or CI, is an umbrella term for “stuff that runs whenever your code changes”

It may include: upload a new version of the documentation, upload a compiled version somewhere, release the code to pypi, run your test suite, and all sort of other things.

A company that offers CI is Github Actions (you will probably use it in one of your courses)

Testing

You will probably not encounter many test generation during your undergraduate years, but for your future career it is good to know the following terms:

Test suite: a collective term for all the tests

Unit test: a “micro-test” that tests a specific feature in isolation

Integration test: a “macro-test” that runs a larger part of the system to check that different feature or components work together.

Regression test: a test that implements a particular pattern that previously caused a bug to ensure that the bug does not resurface.