Developing SimObjects in gem5

Let's begin by building gem5

Let's build gem5 in the base **gem5** directory while we go through some basics. Do it by running the following commands.

```
cd gem5
scons build/NULL/gem5.opt -j$(nproc)
```

Some basics on gem5's build system

gem5's build system is *complicated*.

As we'll see in the coming sections, gem5 has a number of domain-specific languages and source-to-source compilers.

This means two things:

1. It's not always obvious what the final code is going to look like.
2. There are many, many options for setting up the gem5 build.

Configuring the gem5 build

As we've seen, there are multiple ways to configure the gem5 build and this results in different binaries.

There are two categories of options:

1. Build-time configuration (e.g., what models to include in the binary)
2. Compiler configuration (e.g., optimizations, debug flags, etc.)

If you forget any of this,

> `scons --help` will explain the targets and Kconfig tools.

Compiler configuration

This is specified by the *suffix* of the gem5 binary you build. For example, `gem5.opt` is built with the "opt" configuration. The options are:

- `fast`: All optimizations, no debug symbols, most assertions are disabled.
  - Use this only after you have fully debugged your models.
  - Significant speedup over `opt` because the binary is smaller (~50 MiB).
- `opt`: Optimized build with debug symbols and all panics, assertions, etc. enabled.
  - This is the most common build target, but it's very large (~500 MiB).
  - Can use this with `gdb` but sometimes it will say "that variable is optimized away".
- `debug`: Minimal optimizations (`-O1`) and all debug symbols.
  - Use this when you need to debug the gem5 code itself and `opt` doesn't work for you.
  - *Much, much* slower than `opt` (order 5-10x slower).

> Remember, these options are for the *gem5* binary, not the simulated system. Choosing *fast* or *debug* will not affect the output of the simulator (unless there are bugs, of course).

Build-time configuration

There are many options for configuring the gem5 build.

Two ways to do this:

1. Use the defaults found in `gem5/build_opts`
2. Configure with `Kconfig` (the same tool the Linux kernel uses)

Build_opts

```
ALL                          GCN3_X86                    NULL_MOESI_hammer   X86_MESI_Two_Level
ARM                          MIPS                        POWER               X86_MI_example
ARM_MESI_Three_Level         NULL                        RISCV               X86_MOESI_AMD_Base
ARM_MESI_Three_Level_HTM     NULL_MESI_Two_Level         SPARC
ARM_MOESI_hammer             NULL_MOESI_CMP_directory    VEGA_X86
Garnet_standalone            NULL_MOESI_CMP_token        X86
```

In `build_opts` you'll find a number of default options. Most of them are named `<ISA>_<PROTOCOL>`.

For example, `X86_MESI_Two_Level` is the build option for the X86 ISA and with the MESI_Two_Level protocol.

You can also build multiple ISAs into a single binary (e.g., `ALL`), but you **cannot** build multiple Ruby coherence protocols into one binary.

Using Kconfig

Kconfig is a tool that allows you to configure the gem5 build interactively.

When using Kconfig, you first must create a build directory.
Note: This directory can have any name and live *anywhere* on your system.

Common practice is to create a directory called `build` in the gem5 source directory and to use a default from `build_opts`.

```
scons defconfig build/my_gem5/ build_opts/ALL
```

In this case, we are using `build/my_gem5` as the build directory and `build_opts/ALL` as the default configuration.

Using Kconfig (cont.)

Once you have created the build directory, you can run `scons` with the `menuconfig` target to get an interactive configuration tool.

```
scons menuconfig build/my_gem5/
```

<script src="https://asciinema.org/a/nMSV0wVOKNavHSJEt3I77jxyu.js" id="asciicast-nMSV0wVOKNavHSJEt3I77jxyu" async="true"></script>

Putting it all together

To build gem5, once you have a build directory set up and configured, you can run the following command.

```
scons build/my_gem5/gem5.opt -j$(nproc)
```

This will build the gem5 binary with the configuration you have set up.
It will build the "opt" binary.

Note: gem5 takes a long time to build, so using multiple cores is important.
I don't know how many cores you have, so I've used `-j$(nproc)` to use all of them.
You may want to use fewer cores if you're doing other things on your system.

gem5's Scons build system

There are two main types of files used to set up gem5's build:

- `SConstruct`: Contains definitions of types of build targets.
  - All of the `SConstruct` files are executed first.
  - Some code is also in `gem5/build_tools`
  - To be honest, this code is confusing and not easy to trace.
- `SConscript`: Contains the build instructions for a file.
  - Defines *what* to build (e.g., which C++ files to compile).
  - You will mostly interact with these files.

We support *most* common OSes and *most* modern compilers. Fixing compiler errors in the SCons build is not straightforward.

> We strongly encourage you to use **supported** compilers/OSes or use docker to build gem5.
>
> You will *not* find the SCons documentation helpful. gem5 has customized it *way* too much.

SimObjects

What is a SimObject?

`SimObject` is gem5's name for a simulated model. We use `SimObject` and its children classes (e.g. `ClockedObject`) to model computer hardware components. `SimObject` facilitates the following in gem5:

- Defining a model: e.g. a cache
- Parameterizing a model: e.g. cache size, associativity
- Gathering statistics: e.g. number of hits, number of accesses

SimObject in Code

In a gem5 build, each `SimObject` based class has 4 related files.

- `SimObject` declaration file: Python(ish) script (.py extension):
  - Represents the model at the highest level. Allows instantiation of the model and interfacing with the C++ backend. It defines the sets of parameters for the model.
  - **CAUTION**: You should not change parameter values (which we will learn about in the future) in this file if what you want to do is to reconfigure your `SimObject`.
- `SimObject` header file: C++ header file (.hh extension):
  - Declares the `SimObject` class in C++.
    Strongly tied to `SimObject` definition file.
- `SimObject` source file: C++ source file (.cc extension):
  - Implements the `SimObject` functionalities.
- `SimObjectParams` header file: **Auto-generated** C++ header file (.hh) from `SimObject` definition:
  - Declares a C++ struct storing all the parameters of the `SimObject`.

Exercise: HelloSimObject

We will start building our first `SimObject` called `HelloSimObject`, and we will look at one of the `SimObject` files.

We will start with the following steps.

1. Write a definition file.
2. Write a header file.
3. Write a source file.
4. Write a `SConscript`.
5. Compile.
6. Write a configuration script and run it.

Later, we'll do the following steps.

7. Add a parameter to the definition file.
8. Update the source file.
9. Compile.
10. Write a second configuration script and run it.

Step 1: Simple SimObject

SimObject Definition File: Creating the Files

Let's create a python file for our `SimObject` under:
src/bootcamp/hello-sim-object/HelloSimObject.py

Since gem5 is still compiling, start by opening a new terminal.

Then, run the following commands in the base **gem5** directory:

```
cd gem5
mkdir src/bootcamp
mkdir src/bootcamp/hello-sim-object
touch src/bootcamp/hello-sim-object/HelloSimObject.py
```

SimObject Definition File: Importing and Defining

Open src/bootcamp/hello-sim-object/HelloSimObject.py in your editor of choice.

In `HelloSimObject.py`, we will define a new class that represents our `HelloSimObject`.
We need to import the definition of `SimObject` from `m5.objects.SimObject`.
Add the following line to `HelloSimObject.py` to import the definition for `SimObject`.

```
from m5.objects.SimObject import SimObject
```

Let's add the definition for our new `SimObject`.

```
class HelloSimObject(SimObject):
    type = "HelloSimObject"
    cxx_header = "bootcamp/hello-sim-object/hello_sim_object.hh"
    cxx_class = "gem5::HelloSimObject"
```

SimObject Definition File: Deeper Look at What We Have Done

Let's take a deeper look at the few lines of code we have.

```
class HelloSimObject(SimObject):
    type = "HelloSimObject"
    cxx_header = "bootcamp/hello-sim-object/hello_sim_object.hh"
    cxx_class = "gem5::HelloSimObject"
```

- `type` is the type name for the `SimObject` in Python.
- `cxx_header` denotes the path to the C++ header file that declares the `SimObject` in C++. **IMPORTANT**: This path should be specified relative to `gem5/src`.
- `cxx_class` is the name of your `SimObject` class in C++.

`type`, `cxx_header`, and `cxx_class` are keywords defined by the `MetaSimObject` metaclass. For a complete list of these keywords, look at src/python/m5/SimObject::MetaSimObject. Some (if not all) of these keyword variables can be skipped. However, I strongly encourage you to at least define `type`, `cxx_header`, `cxx_class`.

Word to the Wise and A Little Peek into the Future

- I strongly recommend setting `type` to the name of the `SimObject` class in Python. I also recommend making sure that the C++ class name is the same as the Python class. You will see throughout the gem5 codebase that this is *not* always the case. However, I strongly recommend following this rule to rid yourself of any compilation headaches.

- We will see later that, when gem5 is built, there will be an **auto-generated** struct definition that stores the parameters of that class. The name of the struct will be determined by the name of the `SimObject` itself. For example, if the name of the `SimObject` is `HelloSimObject`, the struct storing its parameters will be `HelloSimObjectParams`. This definition will be in a file under params/HelloSimObject.hh in the build directory. This struct is used when instantiating an object of a `SimObject` in C++.

SimObject Header File: Creating the Files

Now, let's start building our `SimObject` in C++. First, let's create a file for our `SimObject` by running the following commands in the base **gem5** directory. **REMEMBER**: We set `cxx_header` to `bootcamp/hello-sim-object/hello_sim_object.hh`. Therefore, we need to add the definition for `HelloSimObject` in a file with that same path.

```
touch src/bootcamp/hello-sim-object/hello_sim_object.hh
```

VERY IMPORTANT: If a `SimObject` class inherits from another `SimObject` class in Python, it should do the same in C++. For example, `HelloSimObject` inherits from `SimObject` in Python, so in C++, `HelloSimObject` should inherit from `SimObject`.
VERY IMPORTANT: `SimObject` parameter structs are inherited in the same way as the `SimObject` itself. For example, if `HelloSimObject` inherits from `SimObject`, `HelloSimObjectParams` inherits from `SimObjectParams`.

SimObject Header File: First Few Lines

Open src/bootcamp/hello-sim-object/hello_sim_object.hh in your editor of choice and add the following code to it.

```
#ifndef __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__
#define __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__

#include "params/HelloSimObject.hh"
#include "sim/sim_object.hh"

namespace gem5
{

class HelloSimObject: public SimObject
{
  public:
    HelloSimObject(const HelloSimObjectParams& params);
};

} // namespace gem5

#endif // __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__
```

SimObject Header File: Deeper Look into the First Few Lines

Things to note:

- `__BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__` is an include guard to prevent double includes and cyclic includes. gem5's convention is that the name should reflect the location of the header file relative to the `gem5/src` directory, with `_` being the separator.
- `sim/sim_object.hh` holds the definition for class `SimObject` in C++.
- As mentioned previously, `params/HelloSimObject.hh` is auto-generated and declares a struct named `HelloSimObjectParams`.
- Every `SimObject` should be declared/defined inside the `namespace gem5`. Different categories of `SimObjects` may have their own specific namespace such as `gem5::memory`.
- Class `HelloSimObject` (C++ counterpart for `HelloSimObject` in Python) should inherit from class `SimObject` (C++ counterpart for `SimObject` in Python).
- Every `SimObject` class needs to define a constructor that takes exactly one argument. This argument must be a constant reference object of its parameter struct. Later on, we will look at gem5's internal process that instantiates objects from `SimObject` classes.

SimObject Source File: All the Code

Let's create a source file for `HelloSimObject` under: src/bootcamp/hello-sim-object/hello_sim_object.cc.

```
touch src/bootcamp/hello-sim-object/hello_sim_object.cc
```

Open src/bootcamp/hello-sim-object/hello_sim_object.cc in your editor of choice and add the following code to it.

```
#include "bootcamp/hello-sim-object/hello_sim_object.hh"

#include <iostream>

namespace gem5
{

HelloSimObject::HelloSimObject(const HelloSimObjectParams& params):
    SimObject(params)
{
    std::cout << "Hello from HelloSimObject's constructor!" << std::endl;
}

} // namespace gem5
```

SimObject Source File: Deeper Look

Things to note:

- gem5's convention for the order of include statements is as follows.
  - the header for the `SimObject`.
  - C++ libraries in alphabetical order.
  - other gem5 header files in alphabetical order.
- We only define `HelloSimObject's` constructor since that's the only function it has so far.
- The `params` object passed to the `HelloSimObject::HelloSimObject` constructor is an object of `HelloSimObjectParams` which inherits from `SimObjectParams`. This means `params` can then be passed on to the `SimObject::SimObject` constructor.

Let's Start Building: SConscript

We need to register our `SimObject` with gem5 for it to be built into the gem5 executable. At build time, `scons` (gem5's build system) will look through the gem5 directory searching for files named `SConscript`. `SConscript` files include instructions on what needs to be built. We will simply create a file called `SConscript` (inside our `SimObject` directory) by running the following command in the base **gem5** directory.

```
touch src/bootcamp/hello-sim-object/SConscript
```

Add the following to the SConscript.

```
Import("*")

SimObject("HelloSimObject.py", sim_objects=["HelloSimObject"])

Source("hello_sim_object.cc")
```

Let's Start Building: Deeper Look at the SConscript

Things to note:

- `SimObject("HelloSimObject.py", sim_objects=["HelloSimObject"])` registers `HelloSimObject` as a `SimObject`. The first argument denotes the name of the submodule that will be created under `m5.objects`. All the `SimObjects` listed under `sim_objects` will be added to that submodule. In this example, we will be able to import `HelloSimObject` as `m5.objects.HelloSimObject.HelloSimObject`. It is possible to define more than one `SimObject` in one Python script. Only `SimObjects` listed under `sim_objects` will be built.
- `Source("hello_sim_object.cc")` adds `hello_sim_object.cc` as a source file to be compiled.

Let's Compile

Now, the only thing left to do before we can use `HelloSimObject` in our configuration script is to recompile gem5. Run the following command in the base **gem5** directory to recompile gem5.

```
scons build/NULL/gem5.opt -j$(nproc)
```

While we wait for gem5 to build, we will create a configuration script that uses `HelloSimObject`. In a separate terminal, let's create that script inside gem5/configs. First, let's create a directory structure for our scripts. Run the following set of commands in the base **gem5** directory to create a clean structure.

```
mkdir configs/bootcamp
mkdir configs/bootcamp/hello-sim-object
touch configs/bootcamp/hello-sim-object/first-hello-example.py
```

Configuration Script: First Hello Example: m5 and Root

Open configs/bootcamp/first-hello-example.py in your editor of choice.

To run a simulation, we will need to interface with gem5's backend. `m5` will allow us to call on the C++ backend to instantiate `SimObjects` in C++ and simulate them. To import `m5` into your configuration script, add the following to your code.

```
import m5
```

Every configuration script in gem5 has to instantiate an object of class `Root`. This object represents the root of the device tree in the computer system that gem5 is simulating. To import `Root` into your configuration, add the following line to your script.

```
from m5.objects.Root import Root
```

Configuration Script: First Hello Example: Creating Instances in Python

We will also need to import `HelloSimObject` into our configuration script. To do that, add the following line to your configuration script.

```
from m5.objects.HelloSimObject import HelloSimObject
```

The next thing we need to do is create a `Root` object and a `HelloSimObject` object. We can just add our `HelloSimObject` object as a child of the `root` object by using the `.` operator. Add the following lines to your configuration to do that.

```
root = Root(full_system=False)
root.hello = HelloSimObject()
```

NOTE: We are passing `full_system=False` to `Root` because we are going to simulate in `SE` mode.

Configuration Script: First Hello Example: Instantiation in C++ and Simulation

Next, let's tell gem5 to instantiate our `SimObjects` in C++ by calling `instantiate` from `m5`. Add the following line to your code to do that.

```
m5.instantiate()
```

Now that we have instantiated our `SimObjects`, we can tell gem5 to start the simulation. We do that by calling `simulate` from `m5`. Add the following line to your code to do that.

```
exit_event = m5.simulate()
```

At this point, the simulation will start. It will return an object that holds the status of the simulation. We can see why the simulation exits by calling `getCause` from `exit_event`. Add the following line to your code to due that.

```
print(f"Exited simulation because: {exit_event.getCause()}.")
```

Everything Everywhere All at Once

Here is the complete version of our configuration script.

```
import m5
from m5.objects.Root import Root
from m5.objects.HelloSimObject import HelloSimObject

root = Root(full_system=False)
root.hello = HelloSimObject()

m5.instantiate()
exit_event = m5.simulate()

print(f"Exited simulation because: {exit_event.getCause()}.")
```

Simulate: First Hello Example

Run with the following command in the base **gem5** directory.

```
./build/NULL/gem5.opt ./configs/bootcamp/hello-sim-object/first-hello-example.py
```

<script src="https://asciinema.org/a/ffjsHBq6mPCR1DPxT15WCkm58.js" id="asciicast-ffjsHBq6mPCR1DPxT15WCkm58" async="true"></script>

Overview of exercise

- We created a `SimObject` called `HelloSimObject`.
  - A SimObject is the base for all models in gem5.
  - SimObjects are implemented in C++ but exposed to the python configuration scripts.
  - SimObjects can enqueue events and have parameters (as we'll see soon).
- We defined the `HelloSimObject` in Python and C++.
  - Each SimObject needs a definition file, a header file, a source file, and a to be declared in the `SConscript` file.
  - The Python definition file is used to interface with the C++ backend and exposes parameters, the header and C++ source files implement the model, and the `SConscript` file tells gem5 to build the model.
- We created a configuration script that uses `HelloSimObject`.
  - So far, all SimObjects have been hidden in the stdlib
  - We saw how to create a `Root` object and instantiate a `SimObject` in a configuration script.
  - We also saw how to run the simulation "manually" without the stdlib.

A short Detour: m5.instantiate

Detour: m5.instantiate: SimObject Constructors and Connecting Ports

Below is a snippet of code from the definition of `m5.instantiate`:

```
# Create the C++ sim objects and connect ports
    for obj in root.descendants():
        obj.createCCObject()
    for obj in root.descendants():
        obj.connectPorts()
```

When you call `m5.instantiate`, first, all the `SimObjects` are created (i.e. their C++ constructors are called). Then, all the `port` connections are created. If you don't know what a `Port` is, don't worry. We will get to that in the later slides. For now, think of `ports` as a way for `SimObjects` to send each other data.

Detour: m5.instantiate: SimObject::init

Here is a later snippet of code in `instantiate`.

```
    # Do a second pass to finish initializing the sim objects
    for obj in root.descendants():
        obj.init()
```

In this step, gem5 will call the `init` function from every `SimObject`. `init` is a virtual function defined by the `SimObject` class. Every `SimObject` based class can override this function. The purpose of the `init` function is similar to the constructor. However, it is guaranteed that when the `init` function from any `SimObject` is called, all the `SimObjects` are created (i.e. their constructors are called).

Below is the declaration for `init` in `src/sim/sim_object.hh`.

```
    /* init() is called after all C++ SimObjects have been created and
    *  all ports are connected.  Initializations that are independent
    *  of unserialization but rely on a fully instantiated and
    *  connected SimObject graph should be done here. */
    virtual void init();
```

Detour: m5.instantiate: SimObject::initState, SimObject::loadState

Below shows another snippet from instantiate:

```
# Restore checkpoint (if any)
    if ckpt_dir:
        _drain_manager.preCheckpointRestore()
        ckpt = _m5.core.getCheckpoint(ckpt_dir)
        for obj in root.descendants():
            obj.loadState(ckpt)
    else:
        for obj in root.descendants():
            obj.initState()
```

`initState` and `loadState` are the last step of initializing `SimObjects`. However, only one of them is called for every simulation. `loadState` is called to unserialize a `SimObject's` state from a checkpoint and `initState` is only called when starting a new simulation (i.e. not from a checkpoint).

Continued in next page.

Detour: m5.instantiate: SimObject::initState, SimObject::loadState: C++

Below is the declaration for `initState` and `loadState` in `src/sim/sim_object.hh`.

```
    /* loadState() is called on each SimObject when restoring from a
    *  checkpoint.  The default implementation simply calls
    *  unserialize() if there is a corresponding section in the
    *  checkpoint.  However, objects can override loadState() to get
    *  other behaviors, e.g., doing other programmed initializations
    *  after unserialize(), or complaining if no checkpoint section is
    *  found. */
    virtual void loadState(CheckpointIn &cp);
    /* initState() is called on each SimObject when *not* restoring
    *  from a checkpoint.  This provides a hook for state
    *  initializations that are only required for a "cold start". */
    virtual void initState();
```

We Will See Later

You might have noticed that we also call `m5.simulate` in our configuration script. For now, `HelloSimObject` does nothing interesting during simulation. We will look into the details of simulate later.

Params

Exercise adding SimObject Params

In this exercise, we will add a parameter to our `HelloSimObject` and use it in the constructor.
We will add a parameter called `num_hellos` to our `HelloSimObject` and use it to print `Hello from HelloSimObject's constructor!` multiple times.

We will modify the following files:

- The SimObject definition file.
- The implementation of the SimObject in C++.
- The configuration script.

SimObject parameters are where gem5's software architecture shows its strength. These provide an easy way to parameterize your models and control the values via python.

Let's Talk About Params: Model vs Params

As we mentioned earlier, gem5 allows us to parameterize our models. The whole set of parameter classes in gem5 is defined under `m5.params`, so let's go ahead and import everything from `m5.params` into our `SimObject` definition file. Open src/bootcamp/hello-sim-object/HelloSimObject.py in your editor of choice and add the following line to it.

```
from m5.params import *
```

Now we just need to define a parameter for our `HelloSimObject`. Add the following line to the same file (the `HelloSimObject` definition) to do that. You should add this line under the definition of `class HelloSimObject`.

```
num_hellos = Param.Int("Number of times to say Hello.")
```

Make sure to take a look at src/python/m5/params.py for more information on different parameter classes and how you can add a parameter.
CAUTION: `Params` allow you to define a default value for them. I strongly recommend that you don't define defaults unless you really have to.

HelloSimObject Definition File Now

Here is what your `HelloSimObject` definition file should look like after the changes.

```
from m5.objects.SimObject import SimObject
from m5.params import *

class HelloSimObject(SimObject):
    type = "HelloSimObject"
    cxx_header = "bootcamp/hello-sim-object/hello_sim_object.hh"
    cxx_class = "gem5::HelloSimObject"

    num_hellos = Param.Int("Number times to say Hello.")
```

NOTE: This change to `HelloSimObject.py` will now add an attribute to the `HelloSimObjectParams` the next time you compile gem5. This means that we can now access this parameter in the C++ code.

Using num_hellos

Now, we're going to use `num_hellos` to print `Hello from ...` multiple times in the constructor of the `HelloSimObject`. Open src/bootcamp/hello-sim-object/hello_sim_object.cc in your editor of choice.

Change `HelloSimObject::HelloSimObject` like below:

```
HelloSimObject::HelloSimObject(const HelloSimObjectParams& params):
    SimObject(params)
{
    for (int i = 0; i < params.num_hellos; i++) {
        std::cout << "i: " << i << ", Hello from HelloSimObject's constructor!" << std::endl;
    }
}
```

Make sure you don't delete `include` statments and any lines containing `namespace gem5`

RECOMPILE: All we need to do now is just recompile gem5. Simply do that by running the following command in the base **gem5** directory.

```
scons build/NULL/gem5.opt -j$(nproc)
```

params/HelloSimObject.hh

As we mentioned before, the parameters of a `SimObject` are defined in an auto-generated header file with the `SimObject's` name.

Now that we have added a parameter to `HelloSimObject`, it should now be defined under `HelloSimObjectParams` in build/NULL/params/HelloSimObject.hh.

If you look at the header file, you should see something like this.

```
#ifndef __PARAMS__HelloSimObject__
#define __PARAMS__HelloSimObject__

namespace gem5 {
class HelloSimObject;
} // namespace gem5
#include <cstddef>
#include "base/types.hh"

#include "params/SimObject.hh"

namespace gem5
{
struct HelloSimObjectParams
    : public SimObjectParams
{
    gem5::HelloSimObject * create() const;
    int num_hellos;
};

} // namespace gem5

#endif // __PARAMS__HelloSimObject__
```

Configuration Script: Second Hello Example

Let's create a copy of first-hello-example.py named second-hello-example.py. Just run the following command in the base **gem5** directory to do this.

```
cp configs/bootcamp/hello-sim-object/first-hello-example.py configs/bootcamp/hello-sim-object/second-hello-example.py
```

Now, open second-hello-example.py in your editor of choice and change the code so that it passes a value for `num_hellos` when you instantiate a `HelloSimObject`. Below is a full example of this.

```
import m5
from m5.objects.Root import Root
from m5.objects.HelloSimObject import HelloSimObject

root = Root(full_system=False)
root.hello = HelloSimObject(num_hellos=5)

m5.instantiate()
exit_event = m5.simulate()

print(f"Exited simulation because: {exit_event.getCause()}.")
```

Simulate: Second Hello Example

Run with the following command in the base **gem5** directory.

```
./build/NULL/gem5.opt ./configs/bootcamp/hello-sim-object/second-hello-example.py
```

<script src="https://asciinema.org/a/P1nULfk7VRZGvQURZJryl7mAK.js" id="asciicast-P1nULfk7VRZGvQURZJryl7mAK" async="true"></script>

Summary of adding parameters

- Parameters are a powerful tool in gem5. They allow you to easily change the behavior of your models via python (instead of having to recompile the C++ each time).
- We saw an example of an integer parameter, but there are many other types of parameters you can use.
  - Strings, floats, enums, and more.
  - Addresses, address ranges
  - Vectors of other parameters
  - Ports
  - We'll also see how you can use other SimObjects as parameters as well.
- See `params.py` for more information on the different types of parameters you can use.

Summary of Steps

- **Creating a basic `SimObject`**
  - `SimObject` definition file (.py)
    - Defines the sets of parameters for the model.
  - `SimObject` header file (.hh)
    - Declares the `SimObject` class in C++.
  - `SimObject` source file (.cc extension):
    - Implements the `SimObject` functionalities.
  - `SConscript`
    - Register our `SimObject` with gem5.
  - Auto-generated `SimObjectParams` header file (.hh)
    - Declares a C++ struct storing all the parameters of the `SimObject`.
  - Configuration file (.py)
    - Instantiate `SimObject` and run the simulation.
- **Adding a parameter (`num_hellos`)**
  - Update the definition file and the source file.
  - Write a new configuration file.
  - Re-compile and re-run.