Dyninst can run in three execution modes:

- **Rewriter Mode**: Open a binary, instrument and write to a new binary.
- **Attach Mode**: Attach to a running process, instrument and continue execution.
- **Create Mode**: Create a new process, instrument and continue execution.

Instrumentation is in the form of Abstract Syntax Trees (ASTs):

Instrumentation calls an external library function.

Dyninst can parse & analyze binary code:

Find all the functions in a binary.

Generate control flow graph (CFG) for the functions and find all the basic blocks in the CFG.

Find all the load and store instructions in the basic blocks.

Dyninst can insert instrumentation at any point in the binary code:

Insert instrumentation at every load and store instruction.

Dyninst can monitor & control processes:

Continue execution of a stopped process.

Check if the process has terminated.

Wait for the process to change execution status.

---

**Anatomy of a Dyninst Program**

**A Memory Tracing Tool**

---

```cpp
#include files left out for brevity

int main (int argc, const char * argv[]) {
  BPatch *bp = new BPatch;  // Initialize Dyninst environment

  // Set options for dynamic instrumentation
  bp->setOption("-ludebug", "true");

  // Open a new binary or create/attach to a process
  if (runmode == binary_rewriter) {
    // Open a new binary
    addSpace = bp->openBinary(mutatee);
  } else if (runmode == attach) {
    // Attach to a running process
    addSpace = bp->processAttach(mutatee, atoi(argv[3]));
  } else if (runmode == create) {
    // Create a new process
    addSpace = bp->processCreate(mutatee, argv[3]);
  }

  // Find all functions in the binary
  BPatch_vector<BPatch_function> *funcs = addSpace->getFunctions();

  // Load libcTraceMemory.so
  BPatch_vector<BPatch_function> *libTraceMemory = new BPatch_function;

  // Find function calls that will be instrumented at load and store instructions
  for (BPatch_vector<BPatch_function>::iterator i = funcs->begin(); i != funcs->end(); i++) {
    BPatch_set<BPatch_code> loadOpcode;  // Load opcode set
    BPatch_set<BPatch_code> storeOpcode;  // Store opcode set
    BPatch_set<BPatch_code> instCallFuncs;  // Instruction call function set
    BPatch_set<BPatch_code> instLoadPoints;  // Instruction load points
    BPatch_set<BPatch_code> instStoreCallFunc;  // Instruction store call function

    // Load opcode
    if (i->getOpcode() == BPatch_opLoad) {
      loadOpcode.insert(*i);
    }

    // Store opcode
    if (i->getOpcode() == BPatch_opStore) {
      storeOpcode.insert(*i);
    }

    // Instruction call function
    if (i->getCallType() == BPatch_funcCallExpr) {
      instCallFuncs.insert(*i);
    }

    // Instruction load points
    if (i->getLoadPointType() == BPatch_loadPoint) {
      instLoadPoints.insert(*i);
    }

    // Instruction store call function
    if (i->getStoreCallType() == BPatch_storeCallFunc) {
      instStoreCallFunc.insert(*i);
    }

    // Insert snippet
    bp->callSnippet(callLoadPrint, instLoadPoints);  // Call load print snippet
    bp->callSnippet(callStorePrint, instStoreCallFunc);  // Call store call function
  }

  // Print memory access
  bp->printMemoryAccess();

  // Process tracing
  bp->processTracing(argc, argv);

  // Wait for the process to terminate
  bp->waitForegroundColorChange();

  return 0;
}
```

---

**A mutator program for printing a trace of all memory accesses in a mutatee program**

```bash
> ./traceMemory -b hello hello-rewritten
> ./hello-rewritten
```

---

**Hello World**

```bash
Hello World
```