Lecture 1: Intro

- std::cerr vs. std::cout
  - ↳ std::cerr is not buffered
- use == on std::string values
  - ↳ strcmp for char* values
  - char* can be cast/auto-converted to std::string
- $ c++ main.cpp sum.cpp -o sum
- $ ./sum
- extern keyword: optional
  - ↳ extends functions visibility to the whole program

Lecture 1: Makefiles

source files → compiler → object files → linker

C++ -c [options] [source files]

linker → executable → ./output_file

C++ [options] [source files] [object files] -o output_file

libraries

Makefiles: General structure

1. Header: comments...
2. Rules: describes dependencies

files that "all" target depends on

target name

all: main.cpp file1.cpp file2.cpp file1.h file2.h

⟶ C++ main.cpp file1.cpp file2.cpp -o hello

command to execute

required tab

only runs if any of the dependencies have changes since last time you ran make

Macros:

CXX = c++

CFLAGS = -std=c++11

CCSOURCE = file1.cpp file2.cpp

program: $(CCSOURCE)

$(CXX) $(CFLAGS) -o program $(CCSOURCE)

automatic variables

$@ Target filename of the current rule

$^ All the filenames of all the prerequisites

$< First prerequisite filename in the list.

$? All out of date prerequisites

```
program: file1.cpp file2.cpp
	C++ $^ -o $@

program: file1.cpp file2.cpp
	C++ file.cpp file2.cpp -o program
```

Lecture 2: Classes

- forward Declaration: tells the compiler that there is a declaration of an entity before defining that entity
  - ↳ used with functions, user defined types
  - ↳ prob no header file
- In c++ create objects using pointers

```
Posn *obj1 = new Posn(2,3)
```

use → not . to access member variables and methods

- use virtual keyword before member functions that need to be overriden in subclasses
- NO interfaces in C++, we will use abstract classes
- C++ supports multiple inheritance
- Java uses "garbage collection" to free unreachable objects
- what is dynamic casting?

runtime mechanism to safely convert pointers/references of base class type to derived class types

```
bool equals(Animal *a) {
    Tiger *t = dynamic_cast<Tiger*>(a);
```

Lecture 3: Version control

- Main components
- Git: distributed version control system for source code management
- Github: web based hosting service to store the data remotely

"shows history" → git log or git checkout id

- create a work branch

```
$ git branch -d work // Delete existing work branch
$ git branch work // Creates new branch work
$ git checkout work // Switches HEAD to work branch
```

- Make changes and periodically save work

```
$ git add . // Stage ALL changes
$ git commit -m "..." // Creates new commit
$ git push origin work // push work to remote repo
```

- At a working state, switch back to

```
$ git checkout main
$ git merge --squash work // stages all work changes in main
// avoids cluttering main branch with multiple small commits
$ git commit -m "..."
```

git fetch: retrieve remote changes w/o merging

git status: checks for conflicts

git pull --rebase: integrate changes from remote repository into your local branch while rebasing your local commits on top of the remote changes

- if error → git status → edit conflicting files → git add . → git rebase --continue

Lecture 4: Testing

Testing principles

- Systematic
  - Arbitrary testing: likely to miss bugs
  - Exhaustive testing: often impossible
- Early and Often
  - Late testing would lead to painful debugging at later stages
  - Testing for each method/function/class
  - Confident when making changes
- Automatic
  - eliminate user input while testing

- Representative inputs for array parameters
  - Include boundries, empty lists, lists with one element, unique lists, duplicate elements in the list, sorted and unsorted lists, odd/even number of elements
- Representative inputs for file parameters
  - Empty files, file does not exist, files with small/medium/large sizes
- Testing program with i/o streams
  - 1. pass string as parameter then test
  - 2 pass i/o streams as parameters

```
Static void say_hello(std::istream &in,
  std::ostream &out) {
    std::string name;
    in >> name;
    out << "Hello, " << name;
}

int main(int argc, char *argv[]){
    say_hello(std::cin, std::cout)
}
```

& is like * but w/o changing:

1- . to →

2- in << ... to (*in) <<...

Step 2: create string streams

How to reduce repeated code in tests?

- test helper method

```
Static std::string say_hello_string(std::string s) {
    std::stringstream in(s);
    std::stringstream out("");
    say_hello(in, out);
    return out.str();
}

TEST_CASE("hello") {
CHECK(say_hello_string("Dolly") == "Hello, Dolly!")
```

String streams: associates a string object with a stream allowing to read from the string as if it were a stream

Lecture 5: Program design

- Express how to represent data
- write down a statement of purpose, a signature and a header

what it takes and returns

- examples: given — expect
- template: contains mainly para

```
Float celsius_to_fahrenheit(float a){
    ..... n ....
}
```

Replace body with expression that computes what purpose statement promises

- Run tests

Fix if tests fail

↳ incorrect example samples

↳ Error in function

Lecture 6: Debugging

Difficulties in debugging

- Symptoms may not give clear indications about the cause
- Symptoms may be difficult to reproduce
- Errors may be correlated
- Fixing an error may introduce new errors

Debugging Strategies

- Incremental and Bottom up program development
- Backtracking: trace your steps
- Binary search: ⟵

Git tools - git diff, git bisect

- Problem Simplification

↳ cut down the data

Debugging Methods

- Instrument program to log information

↳ print statements

Better → use debuggers

Instrument program with assertions

git diff: shows changes between commits

git bisect: binary search to find commit that introduced bug

Lecture 6: Debugging (cont.)

LLDB: Default debugger in Xcode

clang++ -g -O0 main.cpp ...

lldb <executable name>

-g Builds executable with debugging symbols

-O0 Optimization level 0 (No optimization, default)

$ (lldb) breakpoint set --name divint

$ (lldb) b file1.cpp:12 ← (line 12)

$ (lldb) breakpoint delete

command: frame variable (fv)

↳ shows the arguments and local variables for the current frame

command: target variable (ta v)

↳ Shows the global/static variables defined in the current source file

command: bt or thread backtrace

↳ prints the stack trace of the current thread

command: frame info

↳ lists information about currently selected frame

Address Sanitizer or ASAN

fast memory error to detect:

- Out of bounds access to heap, stack, globals, Use-after-free, Use-after-scope, double-free, invalid free, memory leaks

Lecture 7: Defensive Programming

Assertions vs Error Handling

- Use assertions for assumptions/conditions that should never occur
- Use Error-handling checks for circumstances that might not occur often
- Assertions → boolean
- Code passes invalid parameters to another function
  - Null pointers
  - Return value not within allowed range

Errors need to be handled

Input values are not within some range

- File missing/not able to open
- Socket gets disconnected
- out of range, Division by 0

How to handle errors that are expected to occur?

- Return a neutral value
- keep reading until next valid data
- log a warning message to a file
- Return an error code (throw exception)

std::exception — Parent class of all the standard C++ exceptions

logic_error — Exception happens in the internal logic of a program

- invalid argument... out of range... length_error

runtime_error - Exception happens during runtime

- range_error... overflow_error... underflow_error

Why different exception types?

- Make code simpler + cleaner
- Easier to investigate a specific category rather than a general one
- Allow programmer to recover from an error

Lecture 8: Code Coverage

→ Describes how much of your code is executed while testing

- "goodness" of test cases
- Many metrics/notions used

1. Statement/Line coverage → Every line runs at least once
2. Branch → Both true/false cases of each condition execute
3. Path → All possible execution paths run
4. Function → each function is called at least once
5. Loop → ensures loops behave correctly in all cases

```
int returnInput(int input, bool cond1,
  bool cond2, bool cond3) {
    int x = input
    int y = 0
    if (cond1)
        x++,          if !cond = false
                      Statement: 66%
    if (cond2)
        x--,
    if (cond3)
        y=x
    return y
```

CHECK(returnInput(2, true, true, true) == 2)

CHECK(returnInput(2, true, true, true) == 3)

Statement → 100%. path 1/8

Branch → 50%.

Test coverage → measures how much of the feature set is covered

types: feature/risk/requirements

Continuous Integration: practice of running those tests traditionally performed during "integration" little and often throughout the development process, rather than waiting til code is complete

Lecture 9: Documentation

Keys to effective comments

- User styles that don't breakdown or discourage modification
- Use the design recipe steps to reduce commenting time
- Integrate commenting into your development style
- Performance is not a good is not a good reason to avoid commenting

Comments and Guidelines

- Source files with the main function
  - the purpose of the program, the version number
- Other source files or classes (header files)
  - the author, revision history, A description of the class
- Functions
  - Explanation of the function usage, how the function works
  - Description of parameters, return value and special return values, exceptions
- Member functions of classes — dependencies if any
- Statement Blocks - Explanation of non obvious functionality, Provide insight into subtleties in the algorithm, why statements do what they do

Lecture 10: Parsing

- Parsing is the task of turning text into Expr objects
- Parentheses are not in Expr, because the Expr tree structure already handles grouping: its abstract syntax
- The parser deals with characters in text, which is concrete syntax

Lecture 12: Power of variables

- Avoid magic numbers - for (i=0, i<100, i++)
- Naming conventions

ClassName, TypeName, localVariable, routineParameter, RoutineName(), m_ClassVariable, g_globalVariable, CONSTANT

Lecture 12: Include files - Libraries

One way to organize source files

- separate header files from cpp files
- Can we compile? what happens to .cpp files?

→ updating all .cpp files with new header location-time consuming

Solution: include the new path while compiling (-I flag)

↳ update only make file

```
project/
├── src/
│   ├── main.cpp
│   └── foo.cpp
└── include/
    └── foo.h
```

CXXFLAGS = -Iinclude

$(TARGET): $(SRCS)

$(CXX) $(CXXFLAGS) $(SRCS)...

Libraries → code that can be reused by programs

↳ Static Library (or archive) vs Dynamic Library (shared)

↳ Code linked at compile time. Executable → code shared by multiple programs and loaded to memory at runtime

File keeps its own copy of library code

namespace: container for identifiers (e.g functions, classes, vars)

- The C++ standard library is a collection of libraries
- the std namespace is used to organize Standard Library identifiers

Lecture 13: Test automation

- Generating random inputs → results of output are compared against expected output

Needs — a way to generate inputs, a driver to send the inputs and recieve outputs, a way to decide whether the output was good (Test oracle)

Benefits - saves time, quick to find bug (and date: couple mins), use another implementation differential testing

ensures consistency and reduces human error, reduces variance in test quality, run tests more frequently and anytime

When to use it → test cases which are executed repeatedly, tests which are difficult to perform manually, time consuming tests

When to NOT use → Test cases which are faster to test manually rather than develop automated tests for them

FUZZING - type of test generation, provides random generated input to a program

Lecture 14: Design Patterns

- Patterns give a design common vocabulary
- Classifying Patterns
  - creational patterns (how objects can be created)
  - Structural patterns (how objects/classes can be combined)
  - Behavioral Patterns (how methods can be implemented)

Singleton: Ensures a class has only one instance and provides a global point of access to it. Useful for resources like databases or configuration settings where only one instance should exist

Builder: Allows for the step by step creation of a complex object. Separates the construction of an object from its representation, making it easier to create complex objects

Facade: Provides a simplified interface to a complex subsystem. It hides the complexities of the system and provides a higher level interface for clients to use

Observer: Defines a one-to-many dependency between objects. When one object (subject) changes state, all dependent objects (observers) are notified and updated automatically

Interpreter: Defines a grammatical representation for a language and provides an interpreter to evaluate sentences in the language. Used in situations where complex parsing is needed

Visitor: Allows adding new operations to existing class hierarchies w/o modifying the classes. The visitor pattern is used to perform operations on elements of an object structure

```
std::string Expr::to_string() {
    std::stringstream st("");
    this->printExp(st);
    return st.str(); }
std::string Expr::to_pretty_string() {
    std::stringstream st;
    this->pretty_print(st, prec_none);
    return st.str(); }
void Expr::pretty_print_at(std::ostream& ot, precedence_t prec,
std::streampos& last_newline_pos) { printExp(ot); }
void Expr::pretty_print(std::ostream& ot, precedence_t prec) {
    std::streampos last_newline_pos = ot.tellp();
    pretty_print_at(ot, prec, last_newline_pos); }
NumExpr::NumExpr(int value) { this->value = value; }
int NumExpr::interp() { return value; }
bool NumExpr::has_variable() { return false; }
Expr* NumExpr::subst(const std::string& var, Expr* replacement) {
    return this; }
bool NumExpr::equals(const Expr* e) {
    const NumExpr* numExpr = dynamic_cast<const NumExpr*>(e);
    return numExpr && this->value == numExpr->value; }
void NumExpr::printExp(std::ostream& ot) { ot << value; }
AddExpr::AddExpr(Expr* lhs, Expr* rhs) {
    this->lhs = lhs;
    this->rhs = rhs; }
int AddExpr::interp() { return lhs->interp() + rhs->interp(); }
bool AddExpr::has_variable() {
    return lhs->has_variable() || rhs->has_variable(); }
Expr* AddExpr::subst(const std::string& var, Expr* replacement) {
    return new AddExpr(lhs->subst(var, replacement), rhs->subst(var,
replacement)); }
bool AddExpr::equals(const Expr* e) {
    const AddExpr* addExpr = dynamic_cast<const AddExpr*>(e);
    return addExpr && lhs->equals(addExpr->lhs) &&
           rhs->equals(addExpr->rhs); }
void AddExpr::printExp(std::ostream& ot) {
    ot << "(";
    lhs->printExp(ot);
    ot << "+";
    rhs->printExp(ot);
    ot << ")"; }
void AddExpr::pretty_print_at(std::ostream& ot, precedence_t prec,
std::streampos& last_newline_pos) {
    bool use_parentheses = (prec >= prec_add);
    if (use_parentheses) { ot << "("; }
    lhs->pretty_print_at(ot, prec_add, last_newline_pos);
    ot << " + ";
    rhs->pretty_print_at(ot, prec_none, last_newline_pos);
    if (use_parentheses) { ot << ")"; } }
MultExpr::MultExpr(Expr* lhs, Expr* rhs) {
    this->lhs = lhs;
    this->rhs = rhs; }
int MultExpr::interp() { return lhs->interp() * rhs->interp(); }
bool MultExpr::has_variable() {
    return lhs->has_variable() || rhs->has_variable(); }
Expr* MultExpr::subst(const std::string& var, Expr* replacement) {
    return new MultExpr(lhs->subst(var, replacement), rhs->subst(var,
replacement)); }
bool MultExpr::equals(const Expr* e) {
    const MultExpr* multExpr = dynamic_cast<const MultExpr*>(e);
    return multExpr && lhs->equals(multExpr->lhs) &&
           rhs->equals(multExpr->rhs); }
void MultExpr::printExp(std::ostream& ot) {
    ot << "(";
    lhs->printExp(ot);
    ot << "*";
    rhs->printExp(ot);
    ot << ")"; }
void MultExpr::pretty_print_at(std::ostream& ot, precedence_t prec,
std::streampos& last_newline_pos) {
    bool use_parentheses = (prec >= prec_mult);
    if (use_parentheses) { ot << "(" }
    lhs->pretty_print_at(ot, prec_mult, last_newline_pos);
    ot << " * ";
    rhs->pretty_print_at(ot, prec_add, last_newline_pos);
    if (use_parentheses) { ot << ")"; } }
VarExpr::VarExpr(const std::string& name) {
    this->name = name; }
int VarExpr::interp() { throw std::runtime_error("Variable has no
value"); }
bool VarExpr::has_variable() { return true; }
Expr* VarExpr::subst(const std::string& var, Expr* replacement) {
    if (this->name == var) { return replacement; }
    return this; }
bool VarExpr::equals(const Expr* e) {
    const VarExpr* varExpr = dynamic_cast<const VarExpr*>(e);
    return varExpr && this->name == varExpr->name; }
void VarExpr::printExp(std::ostream& ot) { ot << name; }
LetExpr::LetExpr(const std::string& var, Expr* rhs, Expr* body) {
    this->var = var;
    this->rhs = rhs;
    this->body = body; }
bool LetExpr::equals(const Expr* e) {
    const LetExpr* letExpr = dynamic_cast<const LetExpr*>(e);
    return letExpr && this->var == letExpr->var &&
        this->rhs->equals(letExpr->rhs) &&
        this->body->equals(letExpr->body); }
int LetExpr::interp() {
    int rhsValue = rhs->interp();
    Expr* substitutedBody = body->subst(var, new NumExpr(rhsValue));
    return substitutedBody->interp(); }
bool LetExpr::has_variable() {
    return rhs->has_variable() || body->has_variable(); }
Expr* LetExpr::subst(const std::string& var, Expr* replacement) {
    if (this->var == var) {
        return new LetExpr(this->var, rhs->subst(var, replacement),
body); }
    return new LetExpr(this->var, rhs->subst(var, replacement),
                       body->subst(var, replacement)); }
void LetExpr::printExp(std::ostream& ot) {
    ot << "(_let " << var << "=";
    rhs->printExp(ot);
    ot << " _in ";
    body->printExp(ot);
    ot << ")"; }
void LetExpr::pretty_print(std::ostream& ot, precedence_t prec) {
    std::streampos last_newline_pos = ot.tellp();
    pretty_print_at(ot, prec, last_newline_pos); }
void LetExpr::pretty_print_at(std::ostream& ot, precedence_t prec,
std::streampos& last_newline_pos) {
    bool needs_parentheses = (prec != prec_none);
    if (needs_parentheses) { ot << "("; }
    std::streampos position1 = last_newline_pos;
    std::streampos current_pos = ot.tellp();
    ot << "_let " << var << " = ";
    rhs->pretty_print(ot, prec_none);
    ot << "\n";
    last_newline_pos = ot.tellp();
    for (int i = position1; i < current_pos; i++) {
      ot << " "; }
    ot << "_in  ";
    body->pretty_print_at(ot, prec_none, last_newline_pos);
    if (needs_parentheses) { ot << ")"; } }
```

```
// Purpose: Parses command-line arguments and returns the corresponding run mode.
// Parameters:
//   - argc: The number of command-line arguments.
//   - argv: An array of C-style strings representing the command-line arguments.
// Returns: A run_mode_t value indicating the mode of operation (e.g., do_test,
do_interp..
run_mode_t use_arguments(int argc, char **argv) {
  // Check if the number of arguments is not equal to 2.
  // The program expects exactly 2 arguments: the program name and a flag.
  if (argc != 2) {
    std::cerr << "Usage: msdscript [--test | --interp | --print | --pretty-print]\n";
    exit(1); } // Exit the program with a non-zero status code (1) to indicate an error.
  // Extract the second argument (the flag) from the argv array.
  // argv[0] is the program name, and argv[1] is the flag.
  std::string flag = argv[1];
  // Check the value of the flag and return the corresponding run mode.
  if (flag == "--test") {
    // If the flag is "--test", return do_test to indicate test mode.
    return do_test;
  } else if (flag == "--interp") { return do_interp; }
  else if (flag == "--print") { return do_print; }
  else if (flag == "--pretty-print") { return do_pretty_print; }
  else { // If the flag is not recognized, print an error message to standard error.
    std::cerr << "Invalid flag. Use --test, --interp, --print, or --pretty-print\n";
    exit(1); } } // Exit the program with a non-zero status code (1) to indicate an error.

// Define the CATCH_CONFIG_RUNNER macro to enable the Catch2 test framework.
// This allows the program to run tests when the --test flag is provided.
#define CATCH_CONFIG_RUNNER
int main(int argc, char* argv[]) {
  try { // Parse command-line arguments and determine the run mode
    run_mode_t mode = use_arguments(argc, argv);
    if (mode == do_test) { // If the mode is do_test, run the Catch2 test suite
      int result = Catch::Session().run();  // Create a Catch2 session and run the tests
      return result; } // Return the test result (0 for success, non-zero for failures)
    std::string input;
    std::getline(std::cin, input); // Read input from standard input
    std::stringstream ss(input); // Create a string stream from the input for parsing
    Expr* expr = parse_expr(ss); // Parse the input expression into an Expr object
    switch (mode) { // Handle the flag based on the run mode
      case do_interp: {
        // If the mode is do_interp, interpret the expression and print the result
        int result = expr->interp();
        std::cout << result << "\n";
        break; }
      case do_print: { // If the mode is do_print, print the expression as a string
        std::cout << expr->to_string() << "\n";
        break; }
      case do_pretty_print: {
        // If the mode is do_pretty_print, pretty-print the expression
        std::cout << expr->to_pretty_string() << "\n";
        break; }
      default:
        std::cerr << "Invalid mode.\n"; // If the mode is invalid, print an error message
        return 1; } // Exit with a non-zero status code to indicate an error
    return 0; // Exit with a status code of 0 to indicate success
  } catch (const std::exception& e) {
    // Catch any exceptions that occur during execution
    // Print the error message to standard error
    std::cerr << "Error: " << e.what() << "\n";
    return 1; } } // Exit with a non-zero status code to indicate an error
```

```
// Purpose: Consumes a specific character from the input stream.
//   - expect: The expected character to consume.
// Throws: std::runtime_error if the next character does not match the expected character.
static void consume(std::istream& in, int expect) {
    int c = in.get();  // Read the next character from the input stream
    if (c != expect) { // Check if the character matches the expected one
      throw std::runtime_error("bad input");} } // Throw an error if it doesn't match
void skip_whitespace(std::istream& in) {
    while (true) {
      int c = in.peek(); // Peek at the next character without consuming it
      if (!isspace(c)) break; // Exit the loop if the character is not whitespace
      consume(in, c); } // Consume the whitespace character } }
// Purpose: Parses a number (integer) from the input stream.
// Returns: A pointer to a NumExpr object representing the parsed number.
// Throws: std::runtime_error if the input is invalid (e.g., no digit after '-').
Expr* parse_num(std::istream& in) {
    int n = 0;          // Initialize the number to 0
    bool negative = false; // Flag to indicate if the number is negative
    if (in.peek() == '-') {  // Check if the number is negative
      negative = true; // Set the negative flag
      consume(in, '-'); // Consume the '-' character
      if (!isdigit(in.peek())) { // Ensure there is a digit after the '-'
        throw std::runtime_error("invalid input"); } }// Throw an error if no digit follows
    while (true) {   // Parse the digits of the number
      int c = in.peek(); // Peek at the next character
      if (isdigit(c)) {  // If it's a digit, consume it and add to the number
        consume(in, c);
        n = n * 10 + (c - '0'); // Convert the character to a digit and add to the number
      } else { break;} } // Exit the loop if a non-digit character is encountered
    if (negative) {    // Apply the negative sign if necessary
      n = -n; }
    return new NumExpr(n);} // Return a new NumExpr object representing the parsed
number
// Purpose: Parses a variable name from the input stream.
// Throws: std::runtime_error if the variable name contains invalid characters (e.g., '_').
Expr* parse_var(std::istream& in) {
    std::string name; // Initialize an empty string to store the variable name
    while (true) {
      int c = in.peek(); // Peek at the next character
      if (isalpha(c)) {  // If it's an alphabetic character, consume it and add to the name
        consume(in, c);
        name += static_cast<char>(c);
      } else if (c == '_') {        // If an underscore is encountered, throw an exception
        throw std::runtime_error("invalid input");
      } else { break;} } // Exit the loop if a non-alphabetic character is encountered
    if (name.empty()) {    // Ensure the variable name is not empty
      throw std::runtime_error("invalid input"); }
    // Return a new VarExpr object representing the parsed variable
    return new VarExpr(name); }
// Purpose: Parses a multicand (number, variable, or parenthesized expression).
// Throws: std::runtime_error if the input is invalid.
Expr* parse_multicand(std::istream& in) {
    skip_whitespace(in); // Skip any leading whitespace
    int c = in.peek();   // Peek at the next character
    if ((c == '-') || isdigit(c)) {   // Handle numbers (including negative numbers)
      return parse_num(in); }
    else if (c == '(') {  // Handle parenthesized expressions
      consume(in, '('); // Consume the '(' character
      Expr* e = parse_expr(in); // Parse the inner expression
      skip_whitespace(in); // Skip any trailing whitespace
      consume(in, ')'); // Consume the ')' character
      return e;}
    else if (isalpha(c)) {   // Handle variables
      return parse_var(in);}
    else {    // Handle invalid input
      throw std::runtime_error("bad input");} }
// Purpose: Parses an addend (multicand or multiplication expression).
Expr* parse_addend(std::istream& in) {
    Expr* lhs = parse_multicand(in); // Parse the left-hand side of the multiplicatic
    skip_whitespace(in); // Skip any whitespace
    int c = in.peek();   // Peek at the next character
    if (c == '*') {   // Handle multiplication
      consume(in, '*'); // Consume the '*' character
      Expr* rhs = parse_addend(in); // Parse the right-hand side of the multiplica
      return new MultExpr(lhs, rhs);} // Return a new MultExpr object
    return lhs;}    // If no multiplication, return the left-hand side
// Purpose: Parses an expression (addend or addition expression).
// Returns: A pointer to an Expr object representing the parsed expression.
Expr* parse_expr(std::istream& in) {
    Expr* lhs = parse_addend(in); // Parse the left-hand side of the addition
    skip_whitespace(in); // Skip any whitespace
    int c = in.peek();   // Peek at the next character
    if (c == '+') {   // Handle addition
      consume(in, '+'); // Consume the '+' character
      Expr* rhs = parse_expr(in); // Parse the right-hand side of the addition
      return new AddExpr(lhs, rhs);} // Return a new AddExpr object
    return lhs;}    // If no addition, return the left-hand side
// Purpose: Main parse function that parses an expression from the input stream.
// Returns: A pointer to an Expr object representing the parsed expression.
Expr* parse(std::istream& in) {
    return parse_expr(in);} // Delegate to parse_expr
// Purpose: Wrapper for testing that parses a string into an expression.
Expr* parse_str(const std::string& s) {
    std::stringstream ss(s); // Create a string stream from the input string
    return parse(ss);} // Parse the expression from the string stream
```

```
# Compiler and compiler flags
CXX = g++ # Use g++ as the C++ compiler
CXXFLAGS = -Wall -Wextra -std=c++17  # Compiler flags:
# -Wall: Enable all warnings
# -Wextra: Enable extra warnings
# -std=c++17: Use the C++17 standard
# Target executables
TARGET = msdscript # Name of the main executable
TEST_TARGET = test_msdscript  # Name of the test executable
# Source and object files for the main program
SRCS = main.cpp expr.cpp cmdline.cpp tests.cpp parse.cpp  # List of
source files
OBJS = $(SRCS:.cpp=.o) # Generate object file names by replacing .cpp
with .o
# Source and object files for the test program
TEST_SRCS = test_msdscript.cpp exec.cpp  # List of source files for the
test program
TEST_OBJS = $(TEST_SRCS:.cpp=.o)  # Generate object file names by
replacing .cpp with .o
all: $(TARGET) # Default target: build the main executable
# Rule to build the main executable
$(TARGET): $(OBJS) # The target depends on the object files
    $(CXX) $(CXXFLAGS) -o $@ $^  # Link the object files into the executable
# $@: The target (msdscript)
# $^: All dependencies (object files)
# Rule to build the test executable
$(TEST_TARGET): $(TEST_OBJS)  # The target depends on the test object
files
    $(CXX) $(CXXFLAGS) -o $@ $^  # Link the test object files into the test
executable
# $@: The target (test_msdscript)
# $^: All dependencies (test object files)
clean:
    rm -f $(OBJS) $(TARGET)   # Remove object files and the main
executable
# -f: Force removal (ignore errors if files don't exist)
# Phony targets (targets that are not actual files)
.PHONY: all clean test
# Target to run tests
test: $(TARGET) # The test target depends on the main executable
    ./$(TARGET) --test # Run the main executable with the --test flag
# Target to generate documentation
doc: msdscript # The doc target depends on the main executable
    cd documentation && doxygen  # Change to the documentation directory
and run Doxygen
```

Trying Generated Expressions

Overall:

- Generate an expression string
- Send string as input to **msdscript**
- Check **msdscript** output and exit code

Inside the **msdscript** implementation:

- We take control of input and output using **std::istream&** and **std::ostream&** arguments

From the **outside**:

- We need a way to run a program
- Send it input to **std::cin**,
- and capture its output to **std::count**

Generate **Expr** values or strings?

By generating strings, we can make the test generator more separate from the code it's trying to test

Simple Generator

```
std::string random_expr_string() {
    if ((rand() % 10) < 6)
        return std::to_string(rand()); // 40% of the time
    else
        return random_expr_string() + "+" + random_expr_string(); // 40% of the time
}
```

Even without tracking the expected sum, but could check:

- **--interp** mode prints *some* number
- **--print** mode prints some expression that interps to the same number
- **--pretty-print** mode prints some expression that interps to the same number and pretty-prints exactly the same
- exit code is always 0

Parsing Let

How about variables and **_let**?

```
⟨expr⟩      = ⟨addend⟩
            | ⟨addend⟩ + ⟨expr⟩
⟨addend⟩    = ⟨multicand⟩
            | ⟨multicand⟩ * ⟨addend⟩
⟨multicand⟩ = ⟨number⟩
            | ( ⟨expr⟩ )
            | ⟨variable⟩
            | _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Should not allow immediate **_let**...

... but **parse_expr** will consume *, anyway

Static library: Adding and Linking

- Create the library
  - Generate the object files
  - Use ar (a Linux *ar*chive utility tool) to create the library file
- Linking the library
  - Specify library path using: `-L flag`
  - `-lname` is equivalent to `libname.a`
- With Cmake:
  - `add_library(LibraryName STATIC simple_lib.cpp)`

Dynamic library: Adding and Linking

- Linux
  - Create the library
    - Generate the object files
    - Use -shared flag with clang++: `clang++ -shared -o libmy_library.so my_library.o`
  - Linking the library
    - Specify library path using: `-L flag` with `-lname` as in static library
    - Use `-rpath` flag to specify the shared library path when building the executable.
- With Cmake:
  - `add_library(LibraryName SHARED simple_lib.cpp)`

```
_let x = 5
_in   _let y = 6
      _in  x + 1
```

Substitute x with 5 in

```
_let y = 6
_in  x + 1
```

should change to

```
_let y = 6
_in  5 + 1
```

Substitution of ⟨variable⟩ with ⟨expr⟩ at **_let**:

- bind *same* ⟨variable⟩: don't substitute in the *body*
- bind *different* ⟨variable⟩: substitute in the *body*

Let Grammar

```
⟨expr⟩ = ⟨number⟩
       | ⟨expr⟩ + ⟨expr⟩
       | ⟨expr⟩ * ⟨expr⟩
       | ⟨variable⟩
       | _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Right-hand side RHS

body