CS100 Introduction to Programming
Spring 2025
Midterm Exam

Instructors: Lan Xu, Yuexin Ma

Time: 21 May 2025, 8:15–10:15

INSTRUCTIONS

Please read and follow the following instructions:

1. The examination paper comprises **four major questions**. The level of difficulty **does not necessarily increase** in sequential order. Please **manage your time wisely**.
2. **This exam is relatively lengthy. For each question, concise answers are expected. If you encounter unfamiliar material, move on promptly rather than spending excessive time. Detailed explanations are not required; grading will be based strictly on key language points.**
3. Each of the four primary questions is preceded by a sign [**C**] or [**C++**] indicating whether the question pertains to the C or C++ programming language, respectively.
4. You have **120 minutes** to answer the questions.
5. You are permitted to bring **at most three (3) standard-sized A4 cheatsheets**, which may be **printed or hand-written**. Cheatsheets that do not meet these requirements will be **confiscated**.
6. You should write the answer to every problem in the dedicated box **clearly**.
7. You should write **your name and your student ID** as indicated on the top of **each page** of the exam sheet.
8. You are **not allowed to bring any electronic devices**, including regular calculators and electronic watches.
9. You are **not allowed to discuss or share anything** with others during the exam.

Name	
Student ID	
School Email	

(This page is intentionally left blank.)

1. (14 points) [C] Input and Output

Imagine you are the instrcutor for a programming exam. Before students can continue, you want each one to honestly promise not to cheat. You write the following C program, which asks the student to type a specific sentence and enter their student number:

```
#include <stdio.h>

int main(void) {
  int num;
  scanf("I promise not to cheat in this exam. My student number is %d.", &num);
  printf("Student %d finishes his/her promise.\n", num);
  return 0;
}
```

The key part of this program is how `scanf` requires the input to exactly match the format string. If you are not familiar with this behavior, here is an excerpt from *en.cppreference.com*:

```
int scanf(const char *format, ...);
```

Reads data from `stdin`, matches it to the format string, and stores the results in the provided variables.

Parameters

- **format** — a pointer to a null-terminated string describing the expected input format
- **...** — the variables that will receive the input values

The format string may include:

- Non-whitespace characters except `%`: **each character in the format string must exactly match the input, or the function fails.**
- Whitespace characters
- Conversion specifications (like `%d`, `%s`)

Return value

The number of input items successfully assigned (may be zero if no matches occurred), or EOF if an input failure occurs before any assignment.

(1) Answer the following questions to explore how this program behaves. You may consult the "Parameters" section in the excerpt above, which explains the meaning and use of each argument in the function prototypes.

i. (1') What is the exact input you must enter for this program to work correctly? Use `'\n'` to indicate where you should press Enter. *Write your answer exactly as you would type it, but replace the sample student number with your own.*

ii. (1') After you enter the above, what will the program print? Use `'\n'` to represent a newline in your answers.

(2) (2') This simple program is not reliable: if the student mistypes even one character, the program may not behave as expected. The improved version below will keep asking until the student enters the sentence correctly. Fill in the blanks to check for errors and ignore bad input. (Hint: check the return value of `scanf`.)

```c
#include <stdio.h>

int main(void) {
  int num, matched, c;
  do {
    matched = scanf(
        "I promise not to cheat in this exam. My student number is %d.", &num);
    if (__________________________) {

      /* Discard the rest of the line */
      while ((c = getchar()) != '\n' && c != EOF);

      printf("Oops—there was a typo. Please try again!\n");
    }
  } while (__________________________);

  printf("Thank you! Student %d has promised honestly.\n", num);
  return 0;
}
```

(3) Now imagine you are the instructor at the front of the classroom. Before you let any student make their promise, you must show you are really the instructor. After you log in, you will call

up each student, one at a time, and keep track of how many students have made their promise.

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

/* Instructor login check */
bool verify_instructor(void);

/* Student promise routine */
void check_promise(void);

int main(void) {
  /* 1. Check that you are the instructor */
  if (!verify_instructor()) {
    printf("Access denied. You shall not pass!\n");
    return 1;
  }

  /* 2. Ask each student to make their promise */
  int count = 0;
  char choice;
  do {
    check_promise();
    count++;
    printf("Next student, please? (y/n): ");
    // the leading space skips leftover newlines
    if (scanf(" %c", &choice) != 1)
      break;
  } while (choice == 'y' || choice == 'Y');

  /* 3. Print the total number of students */
  printf("All done! Total students: %d\n", count);

  return 0;
}
```

The only missing part is the instructor login. Before any student can make a promise, you must type your username and secret passkey. The function must:

1. Read your input safely
2. Remove the trailing newline character
3. Check that the input matches an approved username and passkey

Important: DO NOT attempt to fill in the blanks in the code below until you have carefully read and followed the steps and hints provided in the sub-questions and documentation excerpts.

```
1  bool verify_instructor(void) {
2    char user[50], pass[50];
3
4    printf("Username: ");
5    if (!________________________________________)
6      return false;
7    user[______________________________] = '\0';
8
9    printf("Passkey: ");
10   if (!________________________________________)
11     return false;
12   pass[______________________________] = '\0';
13
14   const char *names[] = {"Yuexin Ma", "Lan Xu"};
15   const char *keys[]  = {"8888", "6666"};
16
17   unsigned n = sizeof(names) / sizeof (char *);
18   for (unsigned i = 0; i != n; ++i) {
19     if (______________________________ && ______________________________)
20       return true;
21   }
22   return false;
23 }
```

i. (2') To read user input into a buffer safely, use the function `fgets`. Below is an excerpt from *en.cppreference.com* describing the usage of `fgets`. Consult this documentation and fill in the missing arguments on **lines 5 and 10** above.

```
char *fgets(char *str, int count, FILE *stream);
```

Reads up to count - 1 characters from the file stream into the string array str. Stops if it reads a newline character (which is also stored in str), or if it reaches the end of the file. On success, puts a null character after the last character in str.

Parameters

- `str` — pointer to a char array
- `count` — maximum number of characters to write (usually the array length)
- `stream` — file stream to read from

Return value

- Returns `str` on success, or a null pointer on failure.

ii. (2') To ensure the buffers are null-terminated, use `strcspn` as illustrated. Consult the excerpt from *en.cppreference.com* below, which describes `strcspn`, and fill in the blanks on **lines 7 and 12** above accordingly.

```
size_t strcspn(const char *dest, const char *src);
```

Returns the length of the initial part of `dest` that does not have any character from `src`.

Parameters

- `dest` — pointer to the string to check
- `src` — pointer to the string of characters to stop at

Return value

The length of the initial segment that has only characters not found in `src`.

Example

```
#include <string.h>
#include <stdio.h>
int main(void) {
  const char *string = "abcde312$#@";
  const char *invalid = "*$#";
  size_t valid_len = strcspn(string, invalid);
  if(valid_len != strlen(string))
    printf("'%s' contains invalid chars starting at position %zu\n",
           string, valid_len);
}
```

Output:

```
'abcde312$#@' contains invalid chars starting at position 8
```

iii. (2') Finally, compare the username and passkey using `strcmp`. Refer to the excerpt from *en.cppreference.com* below, which explains the function `strcmp`, and fill in the appropriate expression on **line 19** above.

```
int strcmp(const char *lhs, const char *rhs );
```

Compares two null-terminated byte strings lexicographically. The sign of the result is the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the strings being compared. The behavior is undefined if lhs or rhs are not pointers to null-terminated byte strings.

Parameters

- `lhs, rhs` - pointers to the null-terminated byte strings to compare

---

Return value

- Negative value if lhs appears before rhs in lexicographical order.
- Zero if lhs and rhs compare equal.
- Positive value if lhs appears after rhs in lexicographical order.

(4) Now, let us combine the student promise logic and the counter into a single function. By declaring the counter variable as `static` within `check_promise()`, its value is preserved across multiple function calls. This approach simplifies the code and makes it easier to manage:

```
1  void check_promise(bool finalize) {
2    _____ _____ count = 0;  // persists across calls
3    if (finalize) {
4      printf("Total students processed: %d\n", count);
5      return;
6    }
7    count++;
8    // other parts
9  }
10 int main(void) {
11   // ...
12   do {
13     check_promise(__________);
14     // ...
15   } while ( /* user chooses to continue */ );
16   check_promise(__________);
17   return 0;
18 }
```

i. (1') Fill in the blanks on **line 2** above with the correct keywords to declare the counter variable so that its value is preserved across multiple calls to the function.

ii. (1') When does the variable `count` get its first value? Select only **ONE** correct answer.
   A. Before the program starts.
   B. After the program starts but before `main()` begins.
   C. The first time `check_promise()` runs.
   D. Each time the program reaches **line 2**.

iii. (2') The boolean parameter `finalize` is used to determine when the total number of students should be printed. Fill in the blanks on **line 13** and **line 16** above so that, after processing all students, the total count is displayed.

2. (37 points) [C, C++] Pointers and Arrays

The department's lightweight *GradeBook* utility maintains a list of student names and their corresponding quiz grades. These are currently stored in two separate arrays.

(1) Below is an example of an initial roster.

```
char *roster[] = { "Alice", "Bob", "Cara", NULL };
```

The structure of this array is similar to a null-terminated string. Please answer the following questions regarding the C string representation.

i. (1') Choose **ALL** the correct statements about the declaration `char str[] = "string";`:

A. It fails to initialize `str`.

B. It initializes `str` to a fixed-size array with length 6.

C. It initializes `str` to a fixed-size array with length 7.

D. It initializes `str` to a fixed-size array whose size is unknown.

E. It initializes `str` to a dynamically allocated array.

ii. (3') Additionally, string literals can be bound in the following ways: `char *str1 = "string1";` or `const char *str2 = "string2";`. Which of these implementations (if any) allows modification of the string? Which implementation (`str1` or `str2`) is better? Please provide a brief explanation.

(2) To determine the number of students in the roster when the size of the array is unknown, we can use the `sizeof` operator to calculate the byte size of the array. Alternatively, we can add a `NULL` pointer at the end of the array (e.g., `roster[3] == NULL`) and traverse it. Please answer the following questions about the function `count_names`.

```
size_t count_names(char *const a[]) {
  size_t n = 0;
  while (a[n] != NULL)
    n++;
  return n;
}
```

i. (2') The type `size_t` is used to store the result of the `sizeof` operator. Is `size_t` a typedef (alias) for an existing type, or is it a separate, distinct type? Is it signed or unsigned?

ii. (1') If `count_names` is called with the `roster` array defined as above, what value will the function return?

iii. (6') Please specify the types of the following expressions. Write the type exactly as it appears in C, instead of vague descriptions such as "a pointer" or "a C language statement."

α) `n++`

α) ______

β) `roster`

β) ______

γ) `roster[0]`

γ) ______

δ) `&(roster[0])`

δ) ______

ϵ) `a`

ϵ) ______

ζ) `a[0]`

ζ) ______

(3) Since the original `roster` array is designed to remain immutable, a writable copy is required. Below is an auxiliary function that performs a deep copy of the roster on the heap, along with an excerpt from *en.cppreference.com* explaining the C string library function `strcpy` for reference.

```
#include <stdlib.h>

char **clone_roster(char *const src[]) {
  size_t n = count_names(src);
  char **dst = malloc(________________________________________);
  for (size_t i = 0; i < n; i++) {
    dst[i] = malloc(________________________________________);
    strcpy(dst[i], src[i]);
  }
  dst[n] = NULL;
  return dst;
}
```

```c
char *strcpy( char *dest, const char *src );
```

Copies the null-terminated byte string pointed to by `src`, including the null terminator, to the character array whose first element is pointed to by `dest`. Undefined behavior occurs if the `dest` array is not large enough, or if the strings overlap. Additionally, the behavior is undefined if either `dest` is not a pointer to a character array or `src` is not a pointer to a null-terminated byte string.

Parameters

- `dest` – pointer to the character array to write to
- `src` – pointer to the null-terminated byte string to copy from

---

Return value

Returns a pointer to `dest`.

i. In C, copying an array of pointers like `char *roster[]` can be done as a *shallow copy* or a *deep copy*. A **shallow copy** copies only the pointers, so both arrays point to the same strings. A **deep copy** creates new memory for each string and copies the contents, making the arrays fully independent. For each statement below about copying `roster`, indicate True (T) or False (F):

α) (1') A shallow copy of the `roster` array duplicates only the pointer values, not the strings themselves.

α) ________

β) (1') If both the original and the shallow-copied `roster` are later freed, this will always be safe and correct.

β) ________

γ) (1') A deep copy of the `roster` array allocates new storage for each string and duplicates their contents, which ensures that modifying or freeing one array has no impact on the other.

γ) ________

ii. (4') Fill in the blanks in the above code snippet for `clone_roster`. Below is the excerpt of `strlen` from *en.cppreference.com*, which may help when filling in the blanks:

```c
size_t strlen(const char *str);
```

Returns the length of the given null-terminated byte string, i.e., the number of characters in a character array whose first element is pointed to by `str`, up to but not including the first null character. Undefined behavior occurs if `str` is not a pointer to a null-terminated byte string.

Parameters

- `str` – pointer to the null-terminated byte string to be examined

---

Return value

The length of the null-terminated byte string `str`.

(4) Since the copied `roster` uses dynamically allocated memory, you must also write a function to free all the memory when it is no longer needed.

```
#include <stdlib.h>
/* Frees everything returned by clone_roster(): */
void free_roster(char **roster) {
  size_t n = count_names(src);
  for (size_t i = 0; i != n; ++i)
    ____________________;
  ____________________;
}
```

i. (2') Please fill in the blanks to complete the function implementation.

ii. (2') What will be the program's state if we mistakenly call `free_roster` on the original `roster`? Select only **ONE** correct answer.

A. Undefined behavior.

B. Unspecified behavior.

C. Compile-time error.

D. Runtime error.

E. The program runs normally.

(5) Recall that the roster functions as a mini *Gradebook*, which requires the ability to store student grades. To achieve this, we introduce an additional array, `score`, to record the grades for each student across all subjects:

```
int score[3][3] = { { 7, 9, 8 }, /* Alice */
                    {10, 6, 7 }, /* Bob */
                    {6, 8, 10 }}; /* Cara */
```

i. (1') What is the value of `score[1][2]`?

ii. (2') Select **ALL** expressions whose values are equivalent to the address of Bob's first quiz grade.

A. `&score[1][0]`

B. `score + 1`
C. `*(score + 1)`
D. `score[1]`

iii. (1') Write the pointer arithmetic equivalent of the expression `score[row][col]`, using `p`, where `p` is a pointer of type `int **` initialized as `int **p = (int **)score;`.

(6) **[C++]** The current approach for managing the student roster and their quiz scores is not yet complete. In real-world scenarios, the number of students cannot be predicted in advance. As a result, both `roster` and `score` should be dynamically allocated on the heap to accommodate varying sizes, which introduces memory management challenges. In contrast, C++ provides a simpler solution for such challenges through the use of STL containers. Below is the C++ equivalent representation for `roster` and `score`:

```
struct Student {
  std::string name;
  std::vector<int> scores;
};
```

i. (1') Initialize an object of `struct Student` named `stu_1` with the information for the student "Cara".

ii. (1') Initialize a `std::vector` called `roster` using the given `stu_1` object as its only element.

iii. (2') STL containers follow the RAII (Resource Allocation Is Initialization) paradigm, which simplifies manual memory management by acquiring all resources in constructors and releasing them in destructors. Please explain how this approach allows the binding of an object's lifetime to its scope and helps avoid memory-related issues, particularly memory leaks.

(7) (5') **[C++]** Many situations require the `score` array to work in coordination with the `roster`. For example, the following C function `praise_students` is designed to print the names of all students whose quiz scores are all greater than or equal to 7.

```c
#include <stdio.h>

void praise_students(char *const roster[], int **scores, size_t cols) {
  size_t rows = count_names(roster);
  for(size_t i = 0; i != rows; ++i){
    int all_seven_or_above = 1;
    for(size_t j = 0; j != cols; ++j){
      if(scores[i][j] < 7){
        all_seven_or_above = 0;
        break;
      }
    }
    if(all_seven_or_above)
      printf("Well done, %s! All your grades are 7 or above.\n", roster[i]);
  }
}
```

Please rewrite the `praise_students` function in C++ based on the `roster` data structure. Be sure to maintain const correctness and avoid unnecessary copies of the data.

```cpp
#include <iostream>

void praise_students( ______________________________ roster) {
  for (_________________________ : roster) {
    bool all_seven_or_above = true;
    for (______________________________) {
      if (____________________) {
        all_seven_or_above = false;
        break;
      }
    }

    if (all_seven_or_above) {
      std::cout << "Well done, " << _______________
                << "! All your grades are 7 or above.\n";
    }
  }
```

3. (20 points) [C++] Constructors and the Builder Pattern

You have previously encountered code structures similar to the following:

```
class Student {
private:
  std::string name_;
public:
  /* TODO: Implement the constructor here. */

  const std::string &name() const { return name_; }
  std::string &name() { return name_; }
};
```

Notice that an appropriate constructor for `Student` has yet to be implemented. There are two typical ways to implement such a constructor:

- **Option 1**

  `explicit Student(const std::string &name) : name_{name} {}`

- **Option 2**

  `explicit Student(std::string name) : name_{std::move(name)} {}`

(1) (2') Briefly explain the advantage of marking this constructor as `explicit`.

(2) (5') Analyze how each constructor implementation behaves when passed arguments of different value categories (such as lvalues and rvalues). Based on your analysis, determine which implementation is better.

(3) Once a suitable constructor is provided, you can instantiate `Student` objects as follows:

```
// Instance 1.
Student student0 = "Shaw";
// Instance 2.
Student student1 = student0;
// Instance 3.
Student student2 = std::move(student0);
```

i. In C++, expressions are classified as *lvalue*, *xvalue*, or *prvalue*. For the right-hand side of each assignment above (Instances 1–3), specify the value category by filling in the blank below.

α) (1') (Instance 1) `"Shaw"` ______________________

β) (1') (Instance 2) `student0` ______________________

γ) (1') (Instance 3) `std::move(student0)` ______________________

ii. For Instance 2 and Instance 3 above, write down the declarations of the constructors that are invoked.

α) (1') Instance 2 ____

β) (1') Instance 3 ____

(4) (2') Consider the scenario where the `Student` class is extended to include additional member variables such as ID number and email address:

```
class Student {
private:
  std::string name_;
  int id_;
  std::string email_;

public:
  Student(const std::string &name, int id, const std::string &email)
    : name_(name), id_(id), email_(email) {}

  const std::string &name() const { return name_; }
  std::string &name() { return name_; }
  const int &id() const { return id_; }
  int &id() { return id_; }
  const std::string &email() const { return email_; }
  std::string &email() { return email_; }
};
```

While a constructor with three parameters is still manageable, constructors can quickly become complicated and hard to use when a class has many data members. For example, imagine a `Student` class that stores not only a name, ID, and email, but also fields such as GPA, phone number, address, and exam scores:

```
class Student {
  std::string name_;
  int id_;
  std::string email_;
  double gpa_;
  std::string phone_;
  std::string address_;
  std::vector<int> exam_scores_;

public:
  Student(const std::string &name, int id, const std::string &email,
          double gpa, const std::string &dob, const std::string &phone,
          const std::string &address, const std::vector<int> &exam_scores)
    : name_(name), id_(id), email_(email), gpa_(gpa), phone_(phone),
      address_(address), exam_scores_(exam_scores) {}
};
```

To address this challenge, C++ developers commonly utilize the **Builder Pattern**, which mitigates the drawbacks of constructors with an excessive number of parameters. The objective is to support an interface that enables the following construction style:

```
Student student = Student{}.name("Shaw")
                           .id(2024533000)
                           .email("shaw1@shanghaitech.edu.cn");
```

In this example, a default-constructed `Student` object is created, and member functions `name(...)`, `id(...)`, and `email(...)` are called in sequence to assign values to the respective fields. The following code fragments are member functions of the `Student` class. Complete the code below so that this form of chained initialization is supported:

```
class Student {
  std::string name_;
  int id_{};
  std::string email_;
public:
  Student() = default;

  const std::string &name() const { return name_; }
  std::string &name() { return name_; }
  const int &id() const { return id_; }
```

```
  int &id() { return id_; }
  const std::string &email() const { return email_; }
  std::string &email() { return email_; }

  Student &name(const std::string &value) {
    name_ = value;
    return *this;
  }

  Student &id(int value) {
    ______________________;
    return ___________;
  }

  Student &email(const std::string &value) {
    ______________________;
    return ___________;
  }
};
```

(5) The current implementation lacks robustness: it does not guarantee that all required fields are initialized. For instance, a user might forget to set the `email` field, resulting in an incomplete or invalid `Student` object. To address this, we seek a mechanism that ensures all necessary fields are set before a `Student` instance can be created.

To achieve this, we introduce a separate `StudentBuilder` type, which manages the construction process and performs runtime validation to ensure every required field is assigned. The goal is to support the following usage:

```
Student student = StudentBuilder{}.name("Shaw")
                                  .id(2024533000)
                                  .email("shaw1@shanghaitech.edu.cn");
```

The `StudentBuilder` class is used to help create a `Student` object step by step. It allows users to set each required field individually, using a chain of method calls. Before a `Student` object can be created, `StudentBuilder` checks that all necessary information has been provided. This design ensures that no incomplete or invalid `Student` objects can be constructed, helping to prevent errors caused by missing data.

> **Important: DO NOT attempt to fill in the blanks in the code below until you have carefully read and followed the steps and hints provided in the sub-questions and documentation excerpts.**

```
class Student {
  _________________________________;  // Your declaration of StudentBuilder here
  std::string name_;
  int id_{};
```

```
5     std::string email_;
6
7   public:
8     Student() = default;
9
10    const std::string &name() const { return name_; }
11    std::string &name() { return name_; }
12    const int &id() const { return id_; }
13    int &id() { return id_; }
14    const std::string &email() const { return email_; }
15    std::string &email() { return email_; }
16  };
```

```
1   class StudentBuilder {
2     Student student_;
3   public:
4     ____________________ name(const std::string &value) {
5       ____________________;
6       return _____________;
7     }
8
9     // The implementations of id and email are similar and omitted for brevity.
10    ... id(int value) { ... }
11    ... email(const std::string &value) { ... }
12
13    operator Student() {
14      assert(_______________ &&
15             _______________ &&
16             _______________);
17
18      return std::move(_______________);
19    }
20  };
```

i. (1’) To allow `StudentBuilder` to access the private members of `Student`, declare `StudentBuilder` as a friend class of `Student`. Complete the declaration in the blank at **Line 2**.

ii. (2’) The `StudentBuilder` class supports method chaining for setting fields, similar to the member functions in `Student`. Complete the implementation of the `name` method in `StudentBuilder` by filling in the blanks from **Line 21** to **Line 24**.

iii. (3’) To enable conversion from a `StudentBuilder` object to a `Student` object, implement the type conversion operator. **In particular, ensure that the following invariants hold before construction:**
   - The `name` field must be a non-empty string.
   - The `id` field must not be zero.
   - The `email` field must be a non-empty string.

   Fill in the blanks for this operator in `StudentBuilder`, from **Line 31** to **Line 37**.

4. (29 points) [C++] Inheritance and Polymorphism

Expressions constitute the fundamental components of computation in programming languages by combining operators and operands to produce values. In C++, each expression is categorized by its value category (lvalue vs. rvalue) and by its **arity**, which specifies the number of operands an operator accepts. The most common arities are:

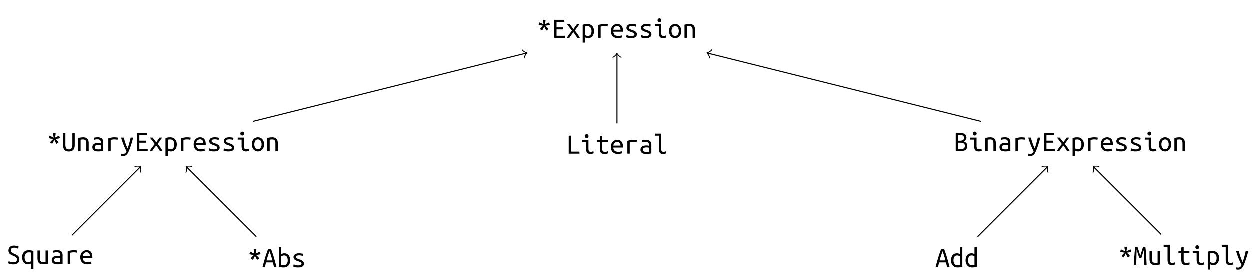
- **Literals** (arity 0): constants with fixed values (e.g. `42`).
- **Unary expressions** (arity 1): an operator applied to one operand (e.g. `-x`).
- **Binary expressions** (arity 2): an operator applied to two operands (e.g. `x + y`).

Although source code may seem to combine more than two operands—such as `a + b + c`—the compiler actually interprets these expressions as a series of nested binary operations:

$$\mathtt{a} + \mathtt{b} + \mathtt{c} \;\equiv\; (a + b) + c,$$

This nesting reflects the recursive nature of expressions: both unary and binary expressions take subexpressions as operands, which themselves can be literals, unary expressions, or binary expressions. Consequently, complex expressions are constructed by layering these components from literals at the leaves up to the root expression.

To model this naturally recursive structure, we define **an abstract base class `Expression`**, from which the following concrete subclasses inherit:



Please note the following details represented in the diagram:

- Each node corresponds to a C++ class in the expression hierarchy.
- Arrows indicate public inheritance, pointing from the derived class to its base class.
- Nodes marked with an asterisk (`*`) indicate classes whose definitions are incomplete; you may be asked to supply their implementations in the following questions.
- The class named `Abs` represents the mathematical "absolute value" operation, which returns the non-negative value of a number (e.g., $\text{Abs}(-3) = 3$ and $\text{Abs}(5) = 5$).

The following code blocks present the class definitions in the expression hierarchy. Please **read each definition carefully** before proceeding to the exercises below.

```
class Square : public UnaryExpression {
public:
```

```
  explicit Square(Expression *sub_exp) : UnaryExpression(sub_exp) {}

  ~Square() override {}

  int evaluate() const override {
    return exp->evaluate() * exp->evaluate();
  }
  std::string to_string() const override {
    return "Square(" + exp->to_string() + ")";
  }
  std::string getType() const override {
    return "Square";
  }
};
```

```
class BinaryExpression : public Expression {
protected:
  Expression *exp1;
  Expression *exp2;

public:
  BinaryExpression(Expression *sub_exp1, Expression *sub_exp2)
    : exp1(sub_exp1), exp2(sub_exp2) {}

  ~BinaryExpression() override {}
};
```

```
class Add : public BinaryExpression {
public:
  Add(Expression *sub_exp1, Expression *sub_exp2)
    : BinaryExpression(sub_exp1, sub_exp2) {}

  ~Add() override {}

  int evaluate() const override {
    return exp1->evaluate() + exp2->evaluate();
  }
  std::string to_string() const override {
    return "(" + exp1->to_string() + " + " + exp2->to_string() + ")";
  }
  std::string getType() const override {
    return "Add";
  }
};
```

```
class Literal : public Expression {
private:
  int num;

public:
  explicit Literal(int n) : num(n) {}

  ~Literal() override {}

  int evaluate() const override {
    return num;
  }
  std::string to_string() const override {
    return std::to_string(num);
  }
  std::string getType() const override {
    return "Literal";
  }
};
```

(1) Review the definition of class `Add` and indicate whether each statement below is True (T) or False (F):

i. (1') The `public` specifier on the second line is redundant, as members of a `class` default to public access.

i. ____________

ii. (1') Applying the `override` keyword to `Add::evaluate()`, `Add::to_string()`, and `Add::getType()` is optional but strongly recommended, as it clarifies that these methods override base-class virtual functions.

ii. ____________

iii. (1') In the body of `Add::evaluate()`, calling `exp1->evaluate()` (or `exp2->evaluate()`) will always cause infinite recursion.

iii. ____________

iv. (1') Both `Add::evaluate()` and `Add::to_string()` can access the protected members `exp1` and `exp2` declared in the base class `BinaryExpression`.

iv. ____________

v. (1') Since `~Add()` has an empty body and performs no work, it could be declared `const`.

v. ____________

(2) The following is a partial definition of class `Multiply`. As a sibling class of `Add` (both inherit from `BinaryExpression`), you may consult `Add`'s definition for reference while completing the exercises.

```
class Multiply : public BinaryExpression {
```

```
public:
  Multiply(Expression *sub_exp1, Expression *sub_exp2)
    : BinaryExpression(sub_exp1, sub_exp2) {}

  ~Multiply() override {}

  int evaluate() const override {
    return ______________________________________________;
  }
  std::string to_string() const override {
    return ______________________________________________;
  }

  std::string getType() const override {
    return "Multiply";
  }
};
```

i. (2') Fill in the blanks above to complete the implementations of `Multiply::evaluate()` and `Multiply::to_string()`.

ii. Determine whether each statement below is Correct or Incorrect. If you judge a statement as Incorrect, briefly explain why.

α) (2') Since `BinaryExpression` has its own constructor, you can create a `BinaryExpression` object directly.

β) (2') Because `Add` and `Multiply` do not add any new data members, their constructors are inherited from the base class.

γ) (2') The statement `Add add_object;` is valid and creates an instance of `Add`.

(3) Next, consider the class `UnaryExpression` and its subclasses.

i. (3') Provide the full definition of `UnaryExpression`, including its constructor, destructor, and any necessary member functions.
Hint: `UnaryExpression` is the counterpart of `BinaryExpression` in this hierarchy. Whereas `BinaryExpression` manages two operand expressions named `exp1` and `exp2`, `UnaryExpression` contains a single operand pointer named `exp`, consistent with the design shown in `Square`.

ii. (3') Similarly, provide the full definition of class `Abs`, implementing all necessary member functions as required, but do not introduce any additional data members.
Hint: Refer to `Square` as an example of a fully defined subclass of `UnaryExpression`.

(4) (4') Complete the definition of the abstract base class `Expression`. Considering the interface requirements imposed by `BinaryExpression`, `UnaryExpression`, and `Literal`, declare the appropriate member functions as pure virtual.

(5) Based on your completed hierarchy, answer the following:

i. (1') Choose **ALL** the abstract classes (those containing at least one pure virtual function).
A. `Expression` B. `BinaryExpression` C. `UnaryExpression` D. `Literal` E. `Add` F. `Multiply` G. `Square` H. `Abs`

ii. (1') Choose **ALL** the polymorphic classes (those declaring at least one virtual function).
A. `Expression` B. `BinaryExpression` C. `UnaryExpression` D. `Literal` E. `Add` F. `Multiply` G. `Square` H. `Abs`

(6) (4') Consider the following program, which uses `parseExpression()` to create an `Expression` object from its input and then outputs the results of `to_string()` and `evaluate()`:

```cpp
#include <iostream>

Expression *parseExpression(const std::string &);

int main(int argc, char *argv[]) {
  if (argc != 2)
    std::cout << "Illegal input." << std::endl;
  else {
    Expression *exp = parseExpression(argv[1]);
    if (exp != nullptr) {
      std::cout << exp->to_string() << " = "
                << exp->evaluate() << std::endl;
      delete exp;
    } else
      std::cout << "Parsing failed." << std::endl;
  }
  return 0;
}
```

Here, `parseExpression` is a function (implementation omitted) that takes a string representing an arithmetic expression and recursively constructs the corresponding expression tree using classes such as `Literal`, `Add`, `Multiply`, `Abs`, and `Square`. The function operates by:

1. Splitting the input by `'+'` or `'*'` operators not enclosed in parentheses, recursively parsing the parts into binary expressions.
2. Detecting unary functions like `Abs(...)` or `Square(...)` and recursively parsing their inner expressions.
3. Removing surrounding parentheses and parsing the enclosed expression.
4. Returning a `Literal` node if the string represents a number.

The examples below serve to clarify the operation of this complex function:

- Input `"3"` returns a `Literal` with value 3.
- Input `"Abs(-3)"` returns an `Abs` expression whose operand is a `Literal` of `-3`.
- Input `"2+3"` returns an `Add` expression with two `Literal` children 2 and 3.

Suppose the classes above have been compiled into an executable program named `Expression`. For each of the following function calls, write exactly what is printed to standard output. Be sure to include the full output line as it would appear.

i. `./Expression Square(5*5)`

ii. `./Expression 5*Abs(-18)+6`

This page is intentionally left blank.
You may detach and use it as your exam script.