CS100 Introduction to Programming
Fall 2024
Midterm Exam

Instructors: Ying Cao

Time: December 12nd 13:00 - 14:40

INSTRUCTIONS

Please read and follow the following instructions:

- You have 100 minutes to answer the questions.
- You are not allowed to bring any electronic devices including regular calculators.
- You are not allowed to discuss or share anything with others during the exam.
- You should write the answer to every problem in the dedicated box **clearly**.
- You should write **your name and your student ID** as indicated on the top of **each page** of the exam sheet.

Name	
Student ID	

1. (16 points) A Simple C Program

The following is the C program '`power.c`'.

```c
#include <stdio.h>

int power(int x, int y) {
  int result = 1;
  for (int i = 0; i < y; i++)
    result *= x;
  return result;
}

int power_of_two(int n) {
  return _____; // (**)
}

int main(void) {
  int x, y;
  scanf( _____ ); // (*)
  printf("%d\n", power(x, y));
  return 0;
}
```

(1) We will begin by compiling and executing this program. For each of the following steps, please write down the corresponding **terminal commands**. Please provide the commands applicable to one of the following platforms: **Windows**, **macOS**, or **Linux**.

i. (2') *Compile* the program '`power.c`' into an executable file named '`prog.exe`' (or '`prog`' on macOS/Linux) within **the current directory**.

Command: _____

ii. (2') *Execute* the compiled **executable file** obtained in step i.

Command: _____

iii. (4') *Execute* the compiled executable file obtained in step i, but **redirect** the program's input from the file '`1.in`' and output to the file '`1.out`', assuming both files are in the current directory (the same directory as the executable).

Command: _____

(2) (3') Fill in the blank marked with (*) so that the program can read two integers from the input, separated by any contiguous sequence of whitespace characters.

(3) (5') The function `power_of_two` accepts an integer `n` and returns 2^n, where `n` is guaranteed to be **positive**. Fill in the blank marked (**) with exactly one expression to complete that function. Your solution should be **faster and simpler** than calling `power(2, n)`.

2. (9 points) Pointers and Classes Basics

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
class Food {
  int pos_x, pos_y;
public:
  Food(int x, int y) : pos_x(x), pos_y(y) {}
  int getX() const { return pos_x; }
  int getY() const { return pos_y; }
};
class Game {
  std::vector<Food> foodList;
public:
  /*
   * @brief a function that creates a new food which isn't on the snake or foods
   */
  static Food createNewFood();
  void addFood(int x, int y) {
    foodList.push_back(_____________); // (1) Fill in the blank
  }
  bool isFoodInList(int x, int y) const {
    for (const auto &food : foodList) // (2)
      if (food.getX() == x && food.getY() == y)
        return true;
    return false;
  }
  bool ifSnakeEatFood(int x, int y) {
    for (auto &food : foodList) { // (3)
      if (food.getX() == x && food.getY() == y) {
        food = createNewFood();
        return true;
      }
    }
    return false;
  }
};
```

(1) (3') Fill in the blank (1) in the code above.

(2) (3') **Type** of `food` in (2): _____.

(3) (3') In (3), will using `auto food` instead of `auto &food` work? Explain why.

3. (20 points) Behaviors

For each of the following code snippets, determine whether it contains a compile error or undefined behavior. Write down the **type of the mistake** (either "**Compile error**" or "**Undefined behavior**"), and **explain why**. If there is no mistake, write "**Correct**" **without further explanation**.

Note that each code snippet contains at most one type of mistake.

The code snippets marked "`[C]`" are based on the C17 standard (ISO/IEC 9899:2018). The code snippets marked "`[C++]`" are based on the C++17 standard (ISO/IEC 14882:2017).

(1) (4') `[C]`

```c
int main(void) {
  int i = 42;
  float *fp = &i;
  ++*fp;
}
```

(2) (4') `[C]`

```c
typedef struct SnakeNode {
  int pos_x;
  int pos_y;
  struct SnakeNode *next;
} SnakeNode;

void freeSnake(SnakeNode *head) {   // free the SnakeNode list
  SnakeNode *temp = head;
  while (temp->next != NULL) {
    temp = temp->next;
    free(temp);
  }
  free(head);
}
```

(3) (4') `[C]`

```c
#include <stdio.h>

typedef struct SnakeNode {
  int pos_x;
  int pos_y;
```

```
  struct SnakeNode *next;
} SnakeNode;

SnakeNode *createSnakeNode(int x, int y) {
  SnakeNode newNode = {.pos_x = x, .pos_y = y};
  return &newNode;
}

int main(void) {
  SnakeNode* head = createSnakeNode(0, 0);
  printf("head->x = %d\n", head->pos_x);
  printf("head->y = %d\n", head->pos_y);
}
```

(4) (4') [C++]

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> array = {1,2,3};
  std::cout << array << std::endl;
}
```

(5) (4') [C++]

```
#include <iostream>
#include <string>

int main() {
  std::string hello{"hello"};
  std::string s = hello + "world" + "C";
  std::cout << s << std::endl;
}
```

4. (43 points) Sorting Algorithm

The following code presents the initial definition of the Student class. The numbered positions *(1)*, *(2)*, *(3)*, etc., indicate areas where modifications or questions may arise:

```
class Student {
private:
  std::string m_name;
  int m_age;

public:
  Student() = delete;
  ~Student() = default;

  Student(std::string name, int age): m_name(std::move(name)), m_age(age) {} //(1)

  const std::string& getName() const { return m_name; }

  void setAge(int age) { m_age = age; }

  int getAge() const { return m_age; }
  //(2)
  //(3)
};
//(4)
//(5)
```

(1) (10') Compare the following four constructor initializer list scenarios:

1. `Student(const std::string &name, ... ) : m_name(name), ...`
2. `Student(const std::string &name, ... ) : m_name(std::move(name)), ...`
3. `Student(std::string name, ... ) : m_name(name), ...`
4. `Student(std::string name, ... ) : m_name(std::move(name)), ...`

Identify which combination of input parameter types and operations (whether `std::move` is used or not) is the most efficient, and provide a detailed explanation for your reasoning

(2) (4') The following code demonstrates an attempt to directly output `Student` objects via `std::cout`.

```
#include <iostream>
#include <vector>

int main() {
  std::vector<Student> students = {{"Bob", 17}, {"Alice", 23}, {"John", 19}};

  for (const auto &student : students) {
    std::cout << student;
  }
}
```

Implement an **overload** for the `operator<<` to enable the output of **ALL** data members of the `Student` class when a `Student` object is passed to `std::cout`.

(3) Attempting to sort the `students` vector using `std::sort` results in a compilation error due to the absence of a defined comparison operator for the `Student` objects.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main() {
6    std::vector<Student> students = {{"Bob", 17}, {"Alice", 23}, {"John", 19}};
7
8    std::sort(students.begin(), students.end());  // (*): compile error
9  }
```

The error message produced by executing the command `g++ main.cpp -std=c++17` on macOS is as follows:

```
error: no match for 'operator<' (operand types are 'Student' and 'Student')
```

Please resolve the issue by performing the following steps:

i. (3') **Method 1: Overloading the Comparison Operator**

Enhance the `Student` class by overloading the `<` operator to enable comparisons between `Student` objects. This will allow sorting a vector of `Student` objects **ascending order by age**.

Provide your implementation below, which should include:

- The definition of the overloaded `<` operator.
- The appropriate location of the operator definition within the class declaration of `Student`, as shown on ***Page 6***.

ii. (6') **Method 2: Using a Lambda Expression**

Without overloading the `<` operator, it is still possible to enable the usage of `std::sort`. Implement a lambda expression with `std::sort` to sort the `students` vector in ascending order by age. Modify the line marked with an asterisk (*) in the following code snippet:

(4) (10') In C++, lambda expressions can serve as predicates for algorithms such as `std::sort`, providing a flexible and efficient mechanism for sorting. Similarly, C employs function pointers to pass functions as arguments, a feature that is also supported in C++ for custom algorithm behavior.

Below is a C-style function, `mySort`, that sorts `Student` instances based on a specified comparison policy.

```cpp
// C-style C++ function
void mySort(std::vector<Student>::iterator begin, std::vector<Student>::iterator end,
            bool(* policy)(const Student&, const Student&)) {
  for (auto it = begin; it != end - 1; ++it) {

    auto pivot = it;

    // Iterate through the unsorted portion
    for (auto it2 = it + 1; it2 != end; ++it2) {
      if (policy(*it2, *pivot)) {
        // Update pivot
        pivot = it2;
      }
    }

    // Move the pivot to its correct position
    std::swap(*it, *pivot);
  }
}
```

Task: Sort the `students` vector in **descending order** by age ***using the `mySort` function***. Your solution should include the following:

i. (6') **Definition of the auxiliary comparision function:**

ii. (4') **Usage:** Rewrite the invocation of `std::sort` in `(3)` using the `mySort` function along with the auxiliary comparison function. Provide the solution in **one line of code**.

5. (28 points) Singly Linked-List with Smart Pointers

In Homework 4, we implemented a game using the C programming language. One of the most fundamental aspects of the game's architecture was the use of the **linked list** data structure.

Now that we have introduced the concepts of **classes** and **smart pointers** in C++, we can leverage these features to redesign and implement a more robust version of the linked list.

Below are the class definitions for the linked list node and the linked list itself:

```
class Node {
  friend class List;

private:
  int value;
  std::unique_ptr<Node> next;

public:
  explicit Node(int val = 0): value(val), next(nullptr) {}

  ~Node() = default;
};


class List {
private:
  std::unique_ptr<Node> head{nullptr};

public:
  List() = default;

  ~List(); // (1) Can it be default?

  void push_front(int); // TODO
  bool contains(int);   // TODO
};
```

(1) (5') Can we define the destructor of `List` as defaulted (`= default`)? Explain your answer.

(2) (4') Which member function(s) of `std::unique_ptr<Node>` is/are deleted? _____

A. The copy constructor
B. The copy assignment operator
C. The move constructor
D. The move assignment operator

(3) (7') The task is to implement the `push_front` function, which inserts a new node at the head of the linked list. Please complete the code segments as indicated below. Note that certain sections may be left intentionally blank if they are not necessary in the given context.

```
void List::push_front (int val) {
  auto tmp = std::make_unique<Node>(val); // To construct a new node
  if (head) {
    _____
    _____
    _____
  }
  else {
    head = std::move(tmp);
  }
}
```

(4) (12') Implement the member function `contains`, which should

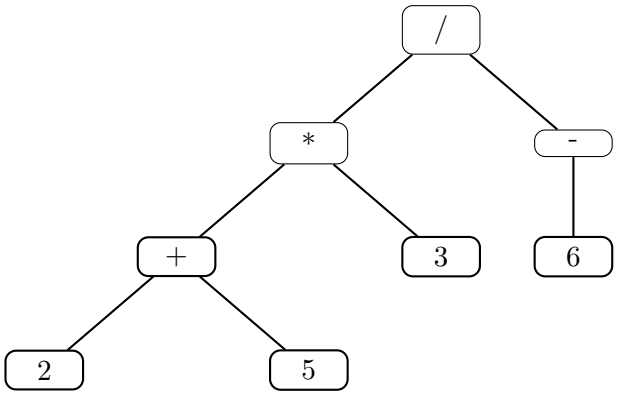
- Search for a given value `val` in the linked list.
- Return `true` if `val` is found; otherwise, return `false`.
- If found, move the node containing `val` to the front of the list, unless it's already at the front.
- In case there are multiple instances of `val`, only the first occurrence should be moved.

Fill in the code segments as indicated below:

```
bool List::contains(int val) {
  if (!head)
    return false; // The list is empty.

  if (head->value == val)
    return true; // The value is already at the front.

  // Define two raw pointers to traverse the list.
  Node *prev = head.get();
  Node *current = head->next.get();

  while (current) {
    if (current->value == val) { // Now we have found the val.
      _____
      _____
      _____
      _____
      _____
      _____
    }
    prev = current;
    current = prev->next.get();
  }
  return false; // No matched val.
}
```

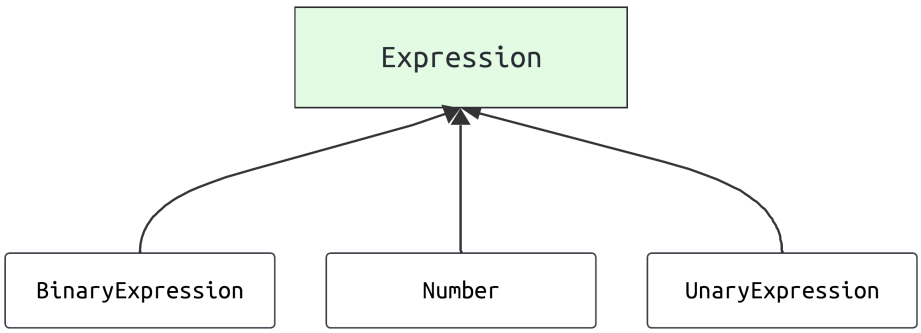
6. (31 points) Expression tree

An arithmetic expression has a tree structure. For example, the tree representing $(2+5)\times 3/(-6)$ is



Such a tree is quite useful, as it contains the structure of all subexpressions. We may perform some operations on the tree recursively, such as printing, evaluation, conversion to some special forms, etc. Now we will take a look at a class hierarchy representing these different kinds of nodes. Further exploration of this example will be left as homework assignments.

The hierarchy of the classes is shown below.



The classes `Number`, `UnaryExpression` and `BinaryExpression` are defined as follows.

```
class Number : public Expression {
  int value;
public:
  explicit Number(int val) : value(val) {}
  int evaluate() const override { return value; }
  std::string toString() const override { return std::to_string(value); }
};

class UnaryExpression : public Expression {
  const Expression *sub;
  char op;
public:
  UnaryExpression(const Expression *subExpr, char op) : sub(subExpr), op(op) {}
  int evaluate() const override {
```

```
    switch (op) {
    case '+': return +sub->evaluate();
    case '-': return -sub->evaluate();
    default:
      throw std::invalid_argument("Invalid operator!"); // report an error
    }
  }
  std::string toString() const override {
    return std::string("(") + op + sub->toString() + ')';
  }
  ~UnaryExpression() override { delete sub; }
};

class BinaryExpression : public Expression {
  const Expression *lhs;
  const Expression *rhs;
  char op;
public:
  BinaryExpression(const Expression *left, const Expression *right, char op)
      : lhs(left), rhs(right), op(op) {}
  int evaluate() const override {
    switch (op) {
    case '+': return lhs->evaluate() + rhs->evaluate();
    case '-': return lhs->evaluate() - rhs->evaluate();
    case '*': return lhs->evaluate() * rhs->evaluate();
    case '/': return lhs->evaluate() / rhs->evaluate();
    default:
      throw std::invalid_argument("Invalid operator!"); // report an error
    }
  }
  std::string toString() const override {
    return "(" + lhs->toString() + op + rhs->toString() + ")";
  }
  ~BinaryExpression() override { delete lhs; delete rhs; }
};
```

The definition of the base class `Expression` is not provided for now, but perhaps you can infer something from the definitions of these derived classes.

Answer the following questions.

(1) (7') For each of the following code snippets, write down the mathematical expression it models.

i. (2') `Expression *e1 = new BinaryExpression(new Number(10), new Number(20), '*');`

ii. (2') `Expression *e2 = new UnaryExpression(new Number(42), '-');`

iii. (3') `Expression *e3 = new BinaryExpression(new Number(2), new UnaryExpression(new Number(3), '-'), '*');`

(2) (11') Complete the definition of the base class `Expression` by filling in the blanks below. Note that certain sections may be left intentionally blank if they are not necessary in the given context.

```
class Expression {
public:
  _____ Expression() = default;
  _____________________________________________
  _____________________________________________
  _____________________________________________
  _____________________________________________
  _____________________________________________
};
```

(3) (6') Did you declare the destructor of `Expression` as virtual? If you did, provide an example where failing to do so would result in an error. If you did not, explain why this is unnecessary.