# 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 ____1 << n____; // (**)
}

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:** `gcc -o prog power.c`

> **Solution:**
> Since '`power.c`' is a ***C program***, it is standard practice to use the '`gcc`' compiler to compile it. On Linux, the command is `gcc power.c -o power`; however, equivalent commands such as `gcc power.c -o prog` or `gcc power.c -o prog.exe` (commonly used on Windows) are also valid.
> It is worth noting that other **C compilers**, such as *clang*, can also be used.Foe instance, the command `clang power.c -o prog` and its equivalents are all acceptable.
> Furthermore, the widely used **C++ compiler** ***g++*** can also compile C programs by treating the source file as a **C++ program**. For example, the command `g++ power.c -o power` and its equivalent variations are also correct.

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

**Command:** `./prog`

> **Solution:**
> Any other equivalent command is also acceptable, such as `.\prog` or `./prog.exe`.

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:** `./prog < 1.in > 1.out`

**Solution:**
Any equivalent command is also correct, such as `.\prog > 1.out < 1.in`. Additionally, since the method of outputting to the file is not specified, if the file `1.out` exists, we can either use `>` to overwrite the file or `>>` to append to it. Therefore, commands such as `./prog < 1.in >> 1.out` and `./prog >> 1.out < 1.in` are also acceptable.

(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.

**Solution:**
`scanf("%d%d", &x, &y);` It is also correct to include any sequence of whitespace characters between the two `%d`'s, but not after the second `%d`.

(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)`.

**Solution:**
Bitwise shifting: `1 << n`.

**2. (9 points) Pointers and Classes Basics**

```
#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(__Food(x, y)__); // (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.

**Solution:**

- [**3 points**] `Food(x, y)` or `{x, y}` is completely correct.
- [**2 points**] `std::move(Food(x, y))` is syntactically correct, but since `Food(x, y)` originally returns an **rvalue**, using `std::move` is unnecessary.

(2) (3') **Type** of `food` in (2): `const Food &` or `Food const&`.

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

**Solution:**
No, it will not work because using auto food creates a copy of the element in the vector. Instead of binding to the original element, the loop creates a new temporary object through the copy constructor, which is independent of the original vector element. As a result, any modifications made to food will not affect the original elements in the vector.

**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]`

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

**Solution:**
[Undefined behavior (2 points)] Dereferencing a pointer of type `float` and attempting to access a variable of type `int` results in an undefined behavior. (2 points)

(2) (4') `[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);
}
```

**Solution:**
[Undefined behavior (2 points)] Undefined bahavior occurs because `temp` is deallocated before it is used in the subsequent iteration. (2 points)

(3) (4') `[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);
}
```

**Solution:**
[Undefined behavior (2 points)] The code exhibits **undefined behavior** due to returning the address of a local variable, which is invalid after the scope of the local variable ends. Specifically, the function `createSnakeNode` returns a pointer to a local variable `newNode`, which will be deallocated upon function exit, leaving the returned pointer dangling. (2 points)

**Note:** In the original code, `x` is incorrectly used instead of `pos_x`. This typo causes the code to fail compilation. (2 points) You will **receive full points** for identifying this issue as a **"[Compile error (2 points)]"** and explaining it clearly.

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

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

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

**Solution:**
[Compile error (2 points)] `std::cout` does not directly support outputting objects of type `std::vector`. (2 points)

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

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

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

**Solution:**
[Correct (4 points)] The `+` operator is overloaded for the `std::string` class, enabling the concatenation of string objects. Additionally, the `<<` operator is overloaded for the `std::ostream` class, facilitating the output of string data to the standard output stream.

**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

**Solution:**

1. `Student(const std::string &name, ... ) : m_name(name), ...`

   **Lvalue:** Binds, then copies. **Rvalue:** Moves, then copies (inefficient). **[1 + 1 points]**

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

   **Lvalue:** Binds, then copies. **Rvalue:** Moves, then copies (inefficient). **[1 + 1 points]**

   **Explanation:**

   `std::move()` can be used on a `const` lvalue, but it will **NOT** enable the object to be moved. This is because the move constructor of `std::string` accepts an rvalue reference to a non-const object. When applied to a const lvalue, `std::move()` results in a `const &&`, which cannot be passed to the move constructor. As a result, the object is copied rather than moved.

3. `Student(std::string name, ... ) : m_name(name), ...`

   **Lvalue:** Copies twice (inefficient). **Rvalue:** Moves, then copies (inefficient). **[1 + 1 points]**

4. `Student(std::string name, ... ) : m_name(std::move(name)), ...`

   ***The most efficient constructor for both lvalues and rvalues.*** **[2 point]**
   **Lvalue:** Copies, then moves (efficient). **Rvalue:** Moves twice (efficient). **[1 + 1 points]**

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

```cpp
#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`.

**Solution:**
Note that the 'getters', `getName()` and `getAge()`, have been provided for you. Therefore, it is **not necessary** to declare the function as a `friend`.

```cpp
// At (4)
std::ostream& operator<<(std::ostream &os, const Student &student) {
  return os << "Student " << student.getName()
            << "'s age is: " << student.getAge() << std::endl;
}
```

---

Alternatively, if you choose to forgo the use of the provided `getters`, there is no penalty for declaring the function as `friend` in order to access the private data members, `m_name` and `m_age`, directly.

```cpp
// At (2)
friend std::ostream& operator<<(std::ostream &os, const Student &student);
```

```
// At (4)
std::ostream& operator<<(std::ostream &os, const Student &student) {
  return os << "Student " << student.m_name
            << "'s age is: " << student.m_age << std::endl;
}
```

**Note:** Any other valid implementation is considered correct. The grading breakdown is as follows:

- **2 points** for a correct function declaration (parameters, return values),
- **2 points** for returning the passed-in reference `os`.

(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.

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

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

  std::sort(students.begin(), students.end());  // (*): compile error
}
```

**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***.

**Solution:** You can define the operator `<` as a member function of `Student`:

```
// At (3)
bool operator<(const Student &other) const {
  return m_age < other.m_age;  // using the getter getAge() is also correct
}
```

Alternatively, you can define it as a non-member function:

```
// At (5)
bool operator<(const Student &lhs, const Student &rhs) {
  return lhs.getAge() < rhs.getAge();
}
```

---

**Note**: Any equivalent implementation is acceptable, including declaring the function within the class and defining it outside, or declaring it as a friend function. **1 point** will be awarded for the correct function declaration and **2 points** for the accurate comparison.

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:

**Solution:**

```
std::sort(students.begin(), students.end(), []
    (const Student& lhs, const Student& rhs)
    -> bool { return lhs.getAge() < rhs.getAge(); });
```

You may omit `->bool` return type specification in the lambda expression. However, it is crucial to avoid directly accessing the **data members** of the `Student` class within the lambda body. Failure to adhere to this restriction will result in a deduction of **2 points**.

Furthermore, the comparison should utilize the `<` operator rather than `<=`, as `std::sort` requires a comparison function object that adheres to the ***Compare*** concept defined by the C++ language standard. According to the standard, the return value of the comparison function, when converted to `bool`, must evaluate to `true` if the first argument precedes the second in a strict weak ordering. While using `<=` will not result in a **point deduction**, it does not conform to the requirements specified by the language standard for the comparison function object expected by `std::sort`.

(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.

```
// 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:**

**Solution:**

```
bool func(const Student &lhs, const Student &rhs) {
  return lhs.getAge() > rhs.getAge();
}
```

Alternatively, you can define it with >=:

```
bool func(const Student &lhs, const Student &rhs) {
  return lhs.getAge() >= rhs.getAge();
}
```

You will lose **2 points** if the comparison operator's direction is reversed.

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**.

**Solution:**

```
mySort(students.begin(), students.end(), &func);
```

The `&` before the function name `func` may be omitted, as a function name can implicitly be converted to a pointer to its address. You will lose **2 points** if the direction of the comparison operator is reversed.

**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:

```cpp
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.

**Solution:**

Yes, the destructor of `List` can be defaulted. [2 points]

When the destructor of `List` is defaulted, it will invoke the destructor of `std::unique_ptr` on the `head` pointer, which will automatically destroy the `Node` that it points to. Furthermore, the destructor of `Node` will recursively call the destructor of `std::unique_ptr` on its `next` member, resulting in the destruction of the entire list. This recursive destruction is handled efficiently through the smart pointer mechanism, ensuring proper memory management without manual intervention. [3 points]

If the answer were "No", the only acceptable explanation would be that recursion may lead to stack overflow in the case of deeply nested or excessively large linked lists. This risk arises from the fact that each recursive destructor call consumes stack space, which could exhaust the available stack memory if the list is too large.

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

A. The copy constructor

B. The copy assignment operator

C. The move constructor

D. The move assignment operator

**Solution:** The `std::unique_ptr` is designed to ensure that ownership of the pointed-to object is unique and cannot be shared. As a result, it deletes both the copy constructor and the copy assignment operator to prevent copying of the unique pointer.

**Note:** You will receive 2 points if you select either option A or option B; any other selection will result in 0 points.

(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) {
    tmp->next = std::move(head);
    head = std::move(tmp);
    ________________________
  }
  else {
    head = std::move(tmp);
  }
}
```

**Solution:** You will earn 4 points for correctly updating `tmp->next` and 3 points for correctly updating `head`. Failure to properly move the unique pointer `tmp` will result in a deduction of up to 3 points.

(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:

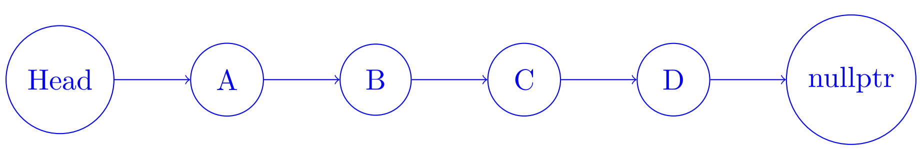
```
bool List::contains(int val) {
  if (!head)
    return false; // The list is empty.

  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.
      auto newHead = std::move(prev->next);
      prev->next = std::move(newHead->next);
      newHead->next = std::move(head);
      head = std::move(newHead);
      return true;

    }
    prev = current;
    current = prev->next.get();
  }
  return false; // No matched val.
}
```

**Solution:**

The statement return true; is awarded 4 points, while the remaining statements contribute a total of 8 points. It is crucial to emphasize that creating new nodes when modifying the order of the linked list is not permitted, as this may lead to significant semantic inconsistencies and negatively impact computational complexity. Should this approach be implemented, 4 out of the 8 points will be deducted.

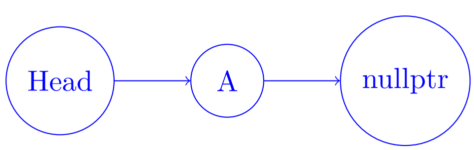
The functionality of the code can be visualized through the following steps:

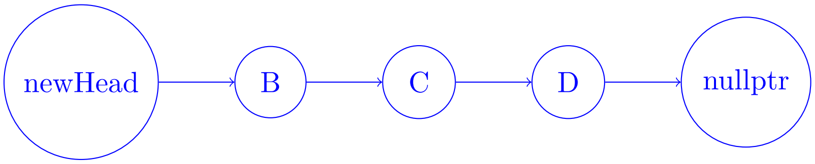
1. Initialization:



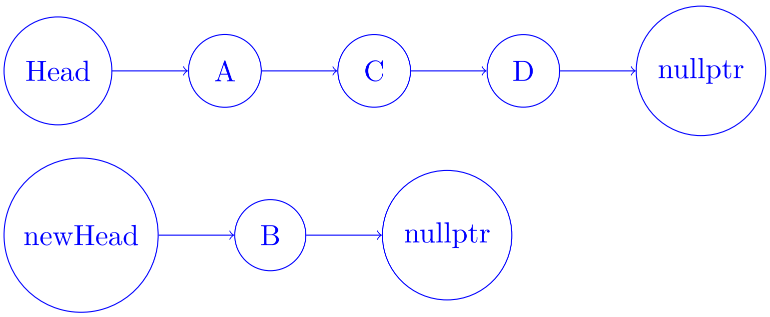

**Assume *Node B* is the target node.**

2. Operation: `auto newHead = std::move(prev->next);`

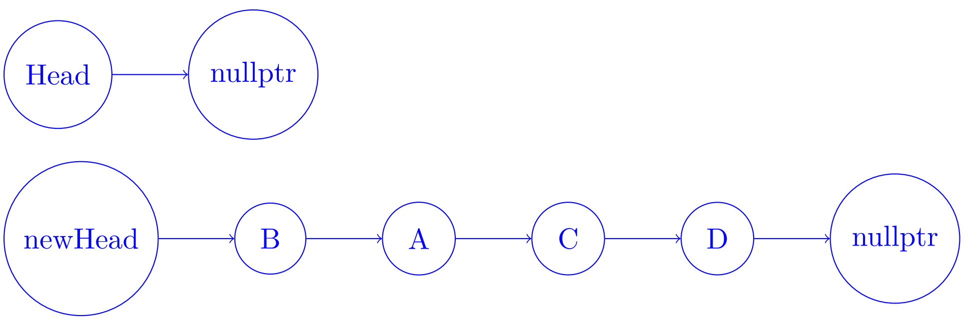




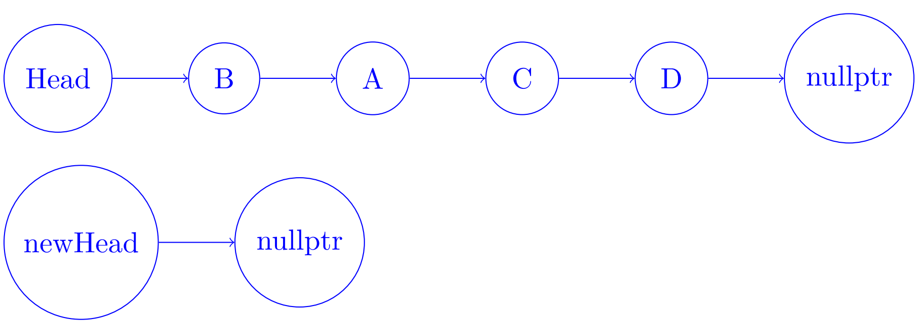

3. Operation: `prev->next = std::move(newHead->next);`




4. Operation: `newHead->next = std::move(head);`




5. Operation: `head = std::move(newHead);`



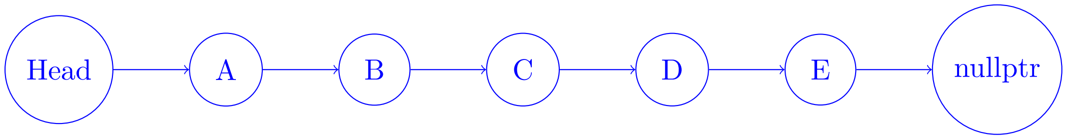

---

Additionally, you may use `std::swap` to simplify the above steps:

```
std::swap(head, prev->next);
std::swap(current->next, prev->next);
```

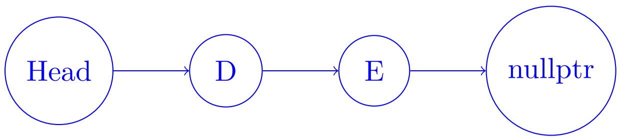
This can be depicted with the following diagram:

1. Initialization:

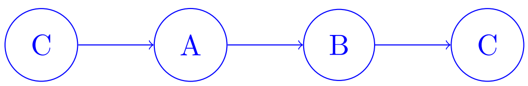



**Assume *`Node D`* is the target node.**, then `prev` points to Node `C` and `current` points to Node `D`.

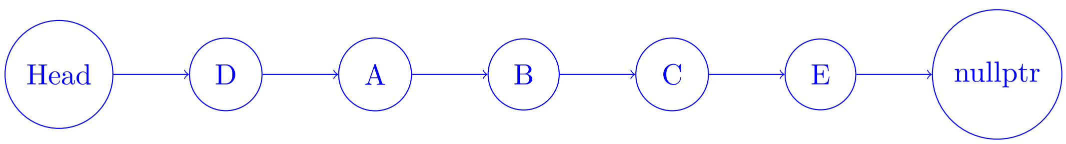
2. Operation: `std::swap(head, prev->next);`




The remaining nodes now form a cycle:



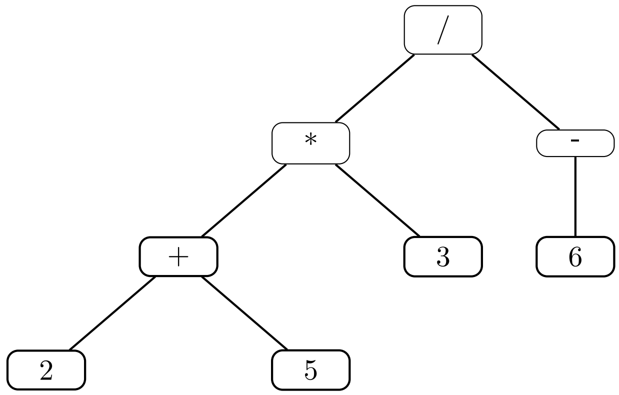

3. Operation: `std::swap(current->next, prev->next);`




The advantage of using `std::swap` is that the resources managed by unique pointers are exchanged, ensuring that each object is pointed to by a unique pointer at any given point in time. This eliminates concerns related to object destruction or other memory management issues.

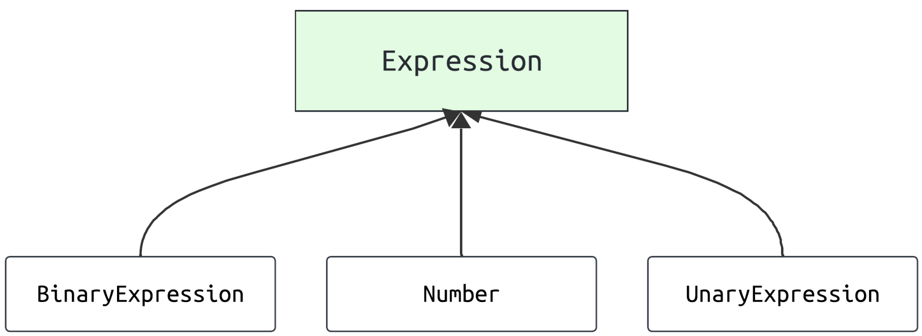
**6. (31 points) Expression tree**

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), '*');`

**Solution:** $10 \times 20$. [You will lose 1 point if you fail to provide an **expression**].

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

**Solution:** $-42$. [You will lose 1 point if you fail to provide an **expression**].

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

**Solution:** $2 \times (-3)$. [You will lose 1 point if you fail to provide an **expression**].

(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:
  / (2 points) Expression() = default;
  virtual int evaluate() const = 0; (3 points)
  virtual std::string toString() const = 0; (3 points)
  virtual ~Expression() = default; (3 points)
  ______________________________
  ______________________________
};
```

**Solution:** You will lose 1 point for each pure virtual function that is not declared as pure virtual. However, if it is implemented incorrectly, you will lose all 3 points for each declaration.

(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.

**Solution:** The destructor of Expression should be declared as virtual. [2 points] Consider the following code snippet:

```
Expression *e = new Number(6);
delete e;
```

Although the static type of `e` is `Expression *`, it points to an object of type `Number`. If the destructor of `Expression` is not virtual, `delete e` will not invoke the destructor of `Number`, leading to undefined behavior. [4 points]