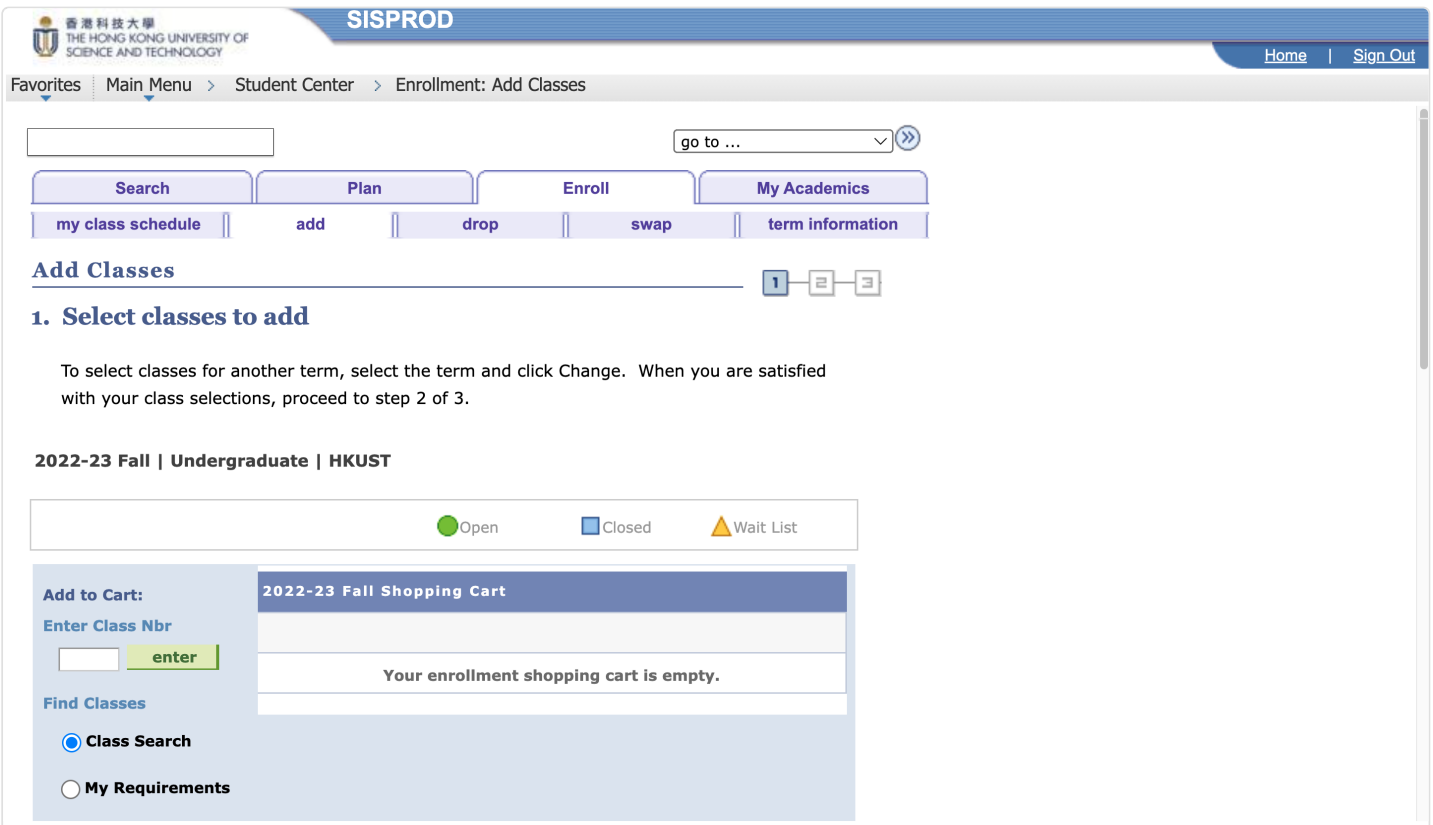
COMP 2012 Object-Oriented Programming and Data Structures

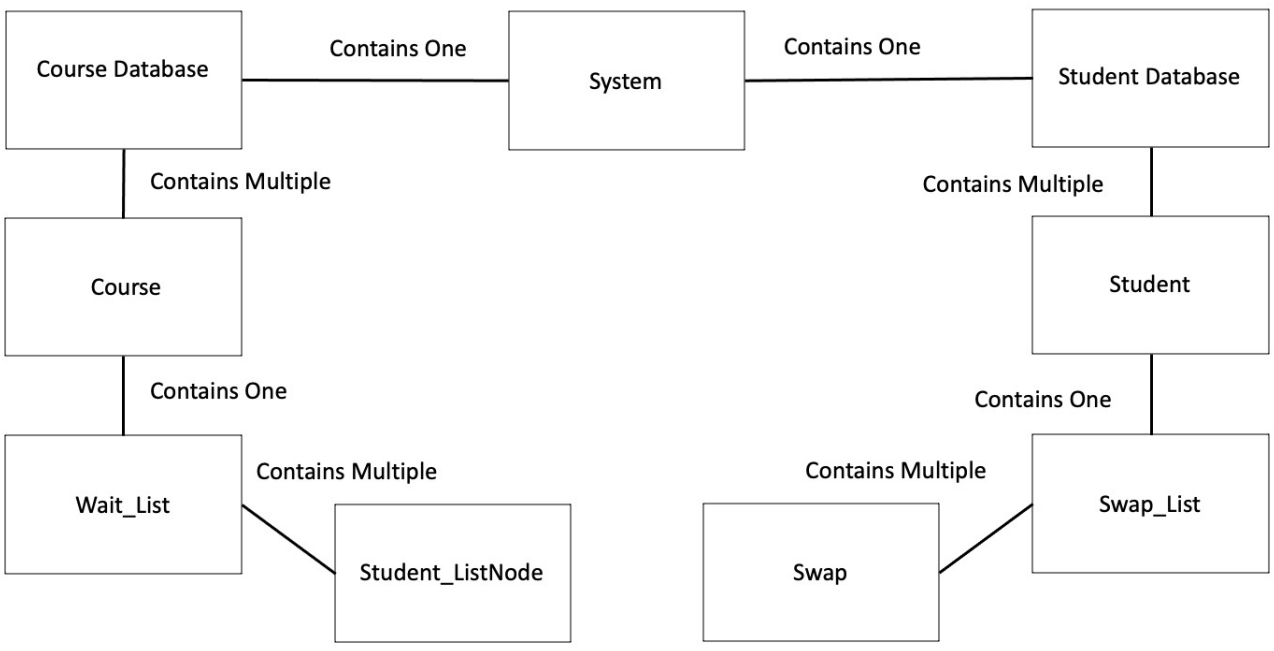
Assignment 1 Course Registration System



Introduction

Before the beginning of every semester, we spend quite some time registering for the courses. It is because we have a lot of classes to choose from, and a lot of operations (add, swap, drop) to be done to get the best timetable. In this programming assignment, you will implement a simplified version of the course registration system. We will test you heavily on dynamic object allocation and deallocation, class constructors and destructor, and different types of linked lists.

Class Structure Overview



Note: This diagram is for illustrative purposes only. It does not follow any standard for any kind of class diagram.

Each rectangle in the diagram represents one class. You can observe the following:

1. The `System` class contains 1 `Course_Database` and 1 `Student_Database`.
2. The `Course_Database` class contains multiple `Course`s.
3. The `Course` class contains 1 `Wait_List`.
4. The `Wait_List` class contains multiple `Student_ListNode`s.
5. The `Student_Database` class contains multiple `Student`s.
6. The `Student` class contains 1 `Swap_List`.
7. The `Swap_List` class contains multiple `Swap`s.

Menu

- Introduction
- Download
- Classes
- Mechanism
- Task
- Sample Output and Grading Scheme
- Submission & Deadline
- Change Log
- FAQ

Page maintained by

Ivan Choi
Email: wmchoiaa@connect.ust.hk

Last Modified: 10/02/2022 16:49:10

Homepage

Course Homepage

You may be confused by what `Swap_List` is. We will introduce it in detail later. In a nutshell, `Swap_List` stores all the pending `Swap` requests that are in the `Wait_List` of a course for a student.

Overview on System

Even though this is a simplified version of the course registration system in HKUST, we will still discuss the capabilities of our system, as there are some differences that we'll highlight between ours and HKUST's.

The class `System` provides a list of functions. The following is the definition of the class `System`.

```
class System {
  private:
  Course_Database* course_database;
  Student_Database* student_database;

  public:
  System(const int max_num_course, const int max_num_student);
  System(const System& system);
  ~System();

  // Student related functions
  void admit(const char* const name, const int student_id, const double gp
  bool apply_overload(const int student_id, const int request_credit);
  bool add(const int student_id, const char* const course_name);
  bool swap(const int student_id, const char* const original_course_name,
  void drop(const int student_id, const char* const course_name); // Assum

  // Course related functions
  void add_course(const char* const name, const int num_credit, const int

  void print_info() const;

  // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

The characteristics of the class `System` are as follows:

- It has a `course_database` and a `student_database` that contains all course registration data.
- It provides some student-related functions:

```
void admit(const char* const name, const int student_id, const double
```

Admit a student into the system, creating an entry in the student database. You can assume that all students are given different `student_id`. `gpa` refers to the gpa of the student and will be used for credit overload requests. You can assume `gpa` will always be between 0.0 to 4.3 inclusive.

```
bool apply_overload(const int student_id, const int request_credit);
```

Apply overload for the student with `student_id`, with the number of credits `request_credit`. The student has to obtain certain `gpa` to request a credit overload. The detailed requirements will be introduced in a later section.

```
bool add(const int student_id, const char* const course_name);
```

The student with `student_id` registers for course with `course_name`. If the course is full, the student will join the `wait_list` of the course. In this assignment, we do not have the concept of **Prerequisites/Co-requisites/Reserved Seats**, this means that any student can add any courses they want as long as they do not exceed their credit limit.

```
bool swap(const int student_id, const char* const original_course_name
```

The student with `student_id` applies to replace the course named `original_course_name` with the course named `target_course_name`. The student will drop the `original_course` immediately if there is a vacancy in the `target_course`. If there is no vacancy, the student will join the `wait_list` for the `target_course`.

Note that the student will **drop** the `original_course` only when he/she gets into the `target_course`. This means if the student joins the `wait_list` of the `target_course`, he/she will NOT drop the `original_course` immediately. Only when someone who was enrolled in the `target_course` drops it, and the student is in the first place on the `wait_list`, then the system will drop the `original_course` for the student. (This part is the same as the HKUST system.)

```
void drop(const int student_id, const char* const course_name);
```

The student with `student_id` applies to drop the course with `course_name`. In this assignment, we do not have the concept of **Pre-enrolled Courses**, which means that any student can drop any courses they want.
Note: in the HKUST system, you can drop a previous add/swap request. However, in this assignment, the system will not provide such functionality.

If a student drops a course, then the vacancy will be filled by the first student on the waitlist given the waitlist is non-empty. Something tricky about this part is whether the student, who gets into the course, joins the `wait_list` through `add` operation or `swap` operation. If the student joins the `wait_list` through the `swap` operation, then the student has to `drop` the `original_course` in the `swap_request`, as we mentioned above. This means that one drop operation could trigger a series of drop behaviours.

- The class also provides a course-related function:

  ```
  void add_course(const char* const name, const int num_credit, const ir
  ```

  Add a course into the system, creating an entry in the course database. `num_credit` refers to the credit count of the course and `max_capacity` refers to the maximum number of students this course can accommodate. You can assume that all courses are given a different `name`.

- and provides a debug function:

  ```
  void System::print_info() const;
  ```

  This function will print all the information stored in the databases.

  Note: In all the remaining classes, there will be a member function with the prefix "print". These functions are all used for printing the class information. All these functions have been implemented for you. They will be useful for debugging and for grading purposes.

Note: The Constructors and Destructor of the class `System` have already been implemented.

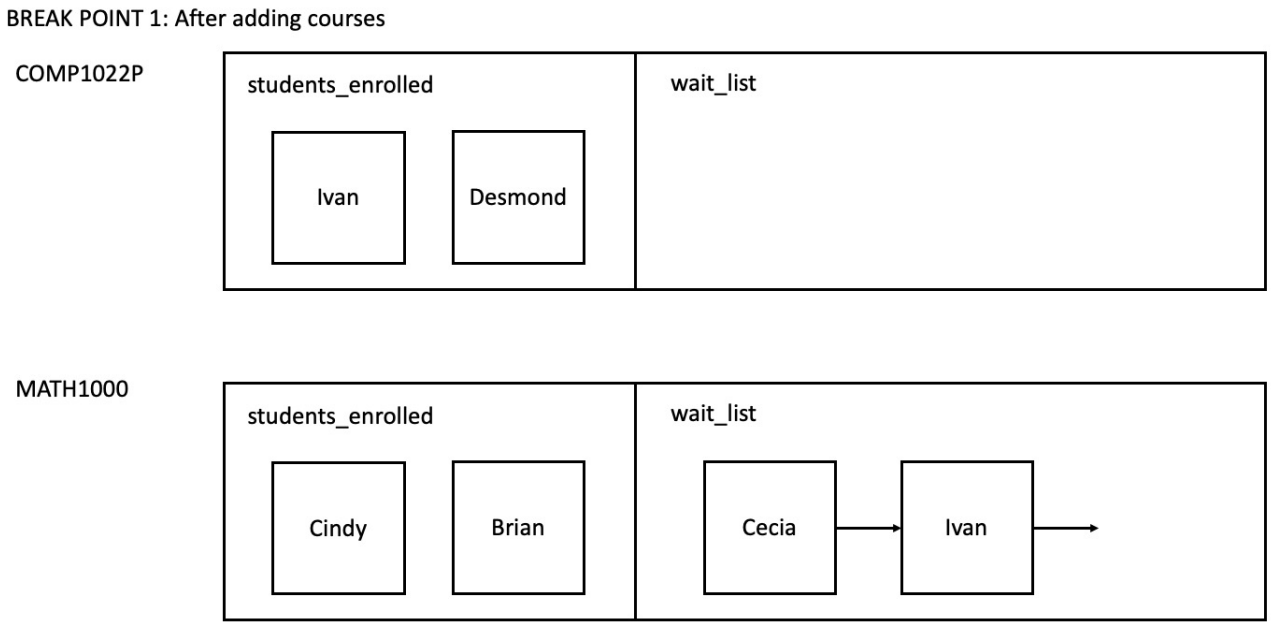
Example

An example explains better than million words, we will go through an example first. The uncommented sections are output messages, the commented sections are the function calls that print out the output messages. We set up 3 breakpoints to perform a detailed analysis. You can assume that there is already a `System` object with the name `system` created.

```
  Admit Ivan into the system.     // system->admit("Ivan", 1000, 2.0);
  Admit Desmond into the system.  // system->admit("Desmond", 1001, 4.3);
  Admit Cindy into the system.    // system->admit("Cindy", 1002, 4.3);
  Admit Cecia into the system.    // system->admit("Cecia", 1003, 4.3);
  Admit Brian into the system.    // system->admit("Brian", 1004, 4.3);
  Creating new course: COMP1022P  // system->add_course("COMP1022P", 3, 2);
  Creating new course: MATH1000   // system->add_course("MATH1000", 1, 2);

  Ivan gets enrolled on COMP1022P successfully.         // system->add(1000,
"COMP1022P");
  Desmond gets enrolled on COMP1022P successfully.      // system->add(1001,
"COMP1022P");
  Cindy gets enrolled on MATH1000 successfully.         // system->add(1002,
"MATH1000");
  Brian gets enrolled on MATH1000 successfully.         // system->add(1004,
"MATH1000");
  MATH1000 is full. Cecia is currently in the waitlist. // system->add(1003,
"MATH1000");
  MATH1000 is full. Ivan is currently in the waitlist.  // system->add(1000,
"MATH1000");

  // BREAK POINT 1
  //system->swap(1002, "MATH1000", "COMP1022P");
  Cindy perform swaps from MATH1000 to COMP1022P.
  COMP1022P is full. Cindy is currently in the waitlist.

  // BREAK POINT 2
  //system->drop(1000, "COMP1022P");
  Ivan drops COMP1022P.
  Cindy in waitlist gets enrolled on COMP1022P successfully.
  Cindy drops MATH1000.
  Cecia in waitlist gets enrolled on MATH1000 successfully.

  // BREAK POINT 3
```

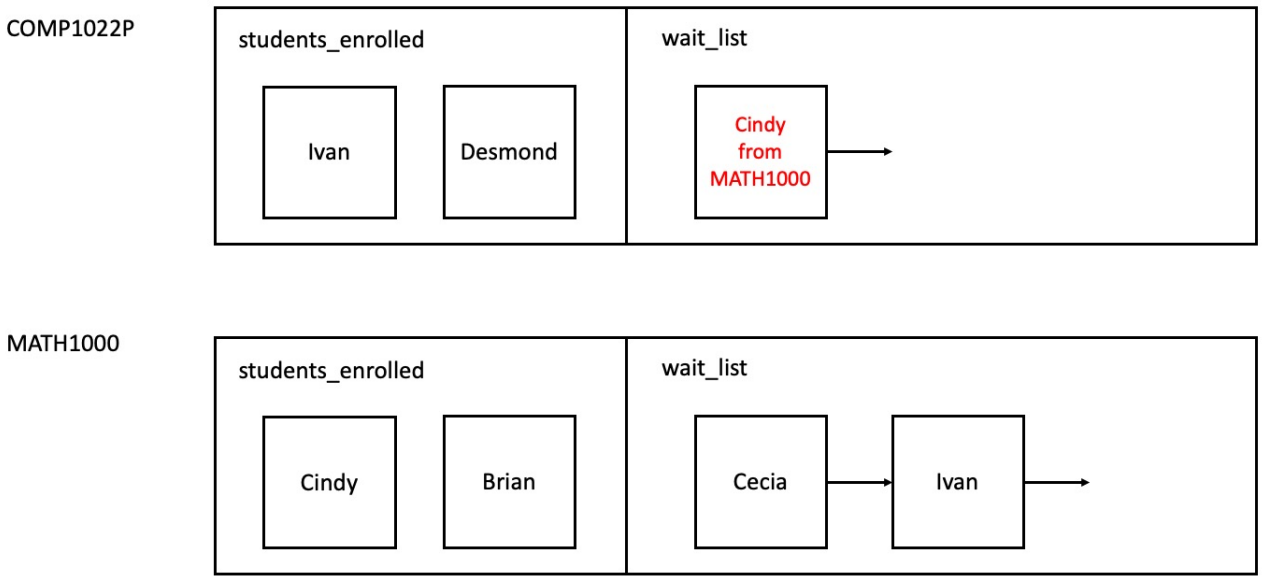
Note: The above text shows some messages corresponding to different function calls. However, these messages are for illustrative purposes only. You do NOT have to include these messages in your submission.

The following pictures will explain the example in detail, and the important parts are marked in red.



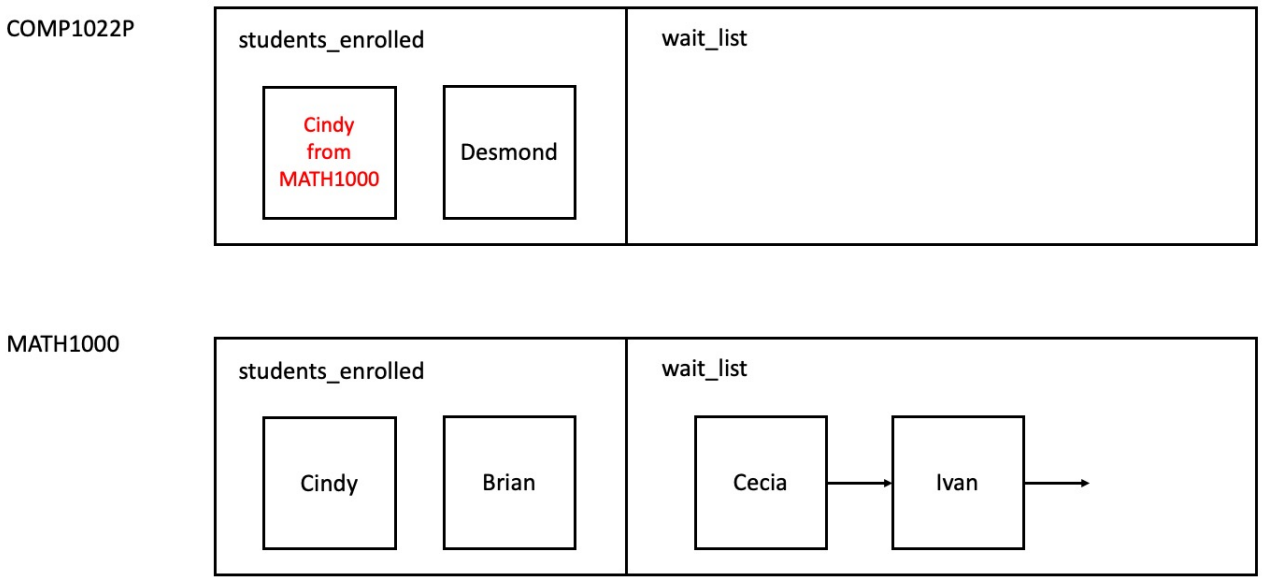
Here we can see the course status at breakpoint 1. Students who cannot get into the class join the end of the wait_list.

BREAK POINT 2: Cindy swap from MATH1000 to COMP1022P



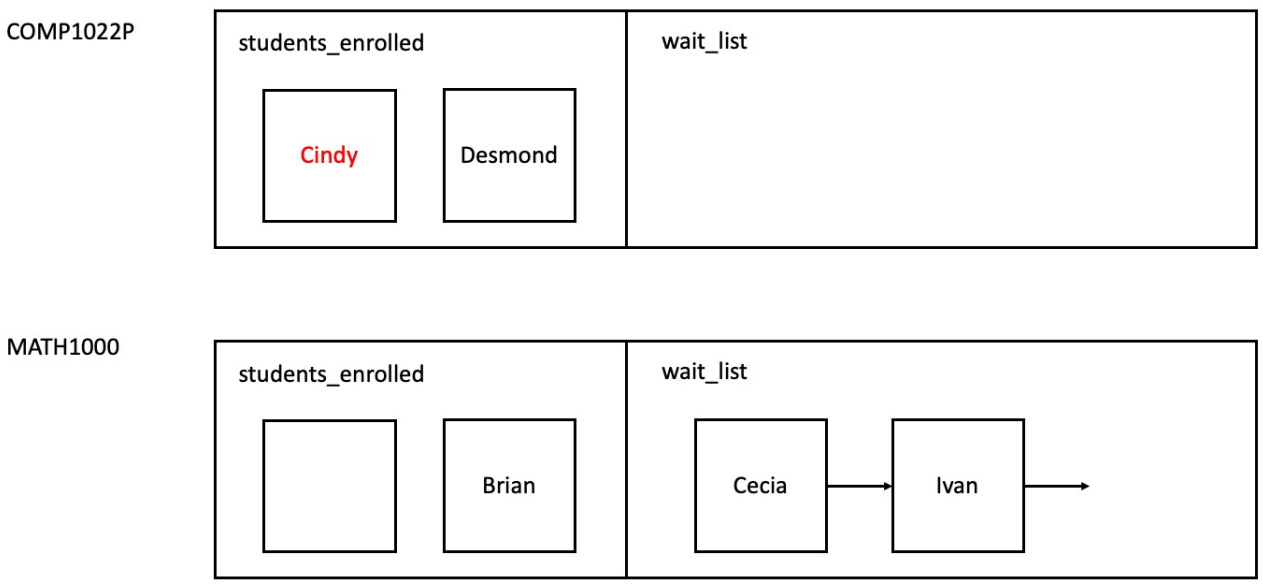
From breakpoint 1 to breakpoint 2, Cindy swaps from MATH1000 to COMP1022P. However, since there is no vacancy for her, she has to stay in the first place on the wait_list.

From BREAK POINT 2 to 3 (Slide 1): Ivan drops COMP1022P. Cindy gets into COMP1022P.



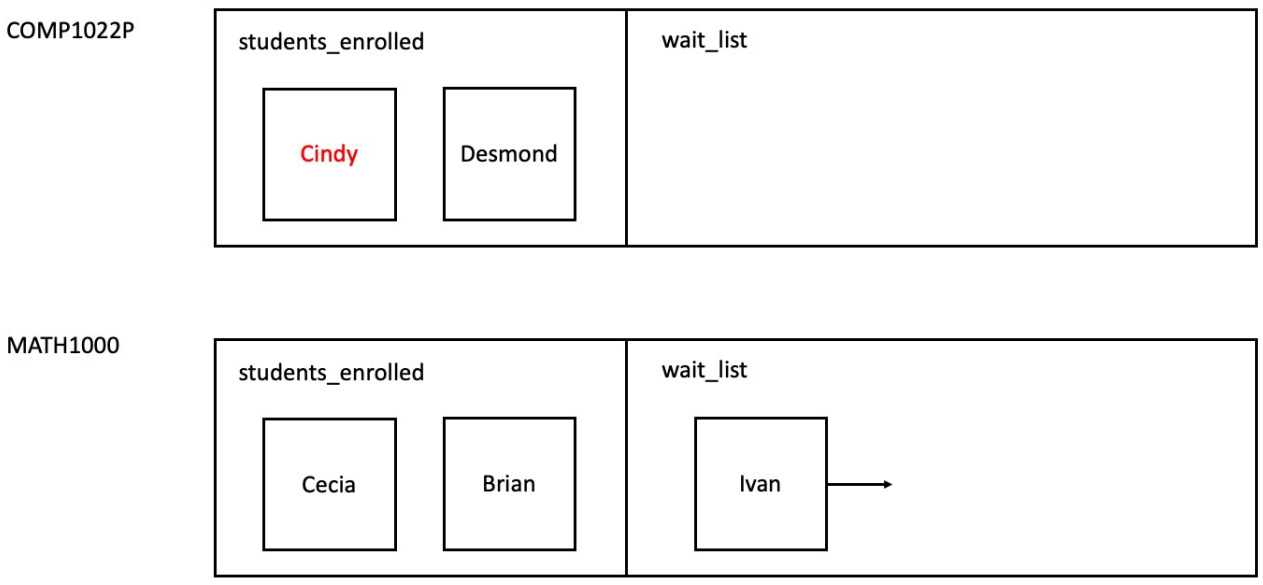
From breakpoint 2 to breakpoint 3, Ivan drops COMP1022P. This means now Cindy, who is in first place on the `wait_list` , can now get into course COMP1022P.

From BREAK POINT 2 to 3 (Slide 2): Cindy drops the original course MATH1000.



Since Cindy joined the wait_list through the `swap operation`, she has to drop the `original_course` MATH1000.

From BREAK POINT 2 to 3 (Slide 3): Cecia gets into course MATH1000.



There is a vacancy created after Cindy dropped MATH1000, which means Cecia, who is in first place on the wait_list, can now get into course MATH1000. Ivan now is in first place on the wait_list.

In this example, you can see that one drop operation (From Ivan), can trigger drop behaviour from others (Cindy).

Deadlock

One possible concern about this assignment will be deadlock. Deadlock, in our case, is a cycle of swap operations that will make all swap requests in the cycle pending forever. For example:

```
system->admit("Harry", 1000, 2.0);
system->admit("Ron", 1001, 2.0);

system->add_course("MAGIC1000", 1, 1); // MAGIC1000 with capacity of 1 stu
system->add_course("MAGIC1001", 1, 1); // MAGIC1001 with capacity of 1 stu

system->add(1000, "MAGIC1000"); // Harry gets into MAGIC1000
system->add(1001, "MAGIC1001"); // Ron gets into MAGIC1001

system->swap(1000, "MAGIC1000", "MAGIC1001"); // Harry try to swap from MA
system->swap(1001, "MAGIC1001", "MAGIC1000"); // Ron try to swap from MAGI
```

An example output message (For illustrative purposes):

```
Admit Harry into the system.
Admit Ron into the system.
Creating new course: MAGIC1000
Creating new course: MAGIC1001
Harry gets enrolled on MAGIC1000 successfully.
Ron gets enrolled on MAGIC1001 successfully.
Harry perform swaps from MAGIC1000 to MAGIC1001.
MAGIC1001 is full. Harry is currently in the waitlist.
Ron perform swaps from MAGIC1001 to MAGIC1000.
MAGIC1000 is full. Ron is currently in the waitlist.
```

In this piece of code, if Harry wants to get into MAGIC1001, he needs Ron to drop it first. However, to let Ron drops MAGIC1001, he needs to get into MAGIC1000, which in turn needs Harry to drop it.

In the end, even if both of them are willing to drop the original course, both swap operations will never be satisfied, as the system is designed not to drop the original_course when the student is still on the waitlist.

Such a phenomenon is called a **Deadlock**. Dealing with deadlock is a difficult task, as it can involve more than 2 entities (students), and multiple resources for each type of resource (courses with capacity > 1). Deadlock will be taught in COMP3511. You may find more information here if you are interested.

There is no evidence of whether the real HKUST system deals with deadlocks. However, in this assignment, you should **NOT** try any approach to tackle with deadlock problem in your submitted files but rather let the system goes into deadlock.

Download

- Skeleton code: skeleton.zip (Last Update: 2022-10-02 16:13)
- Test cases expected result: expected.zip (Last Update: 2022-10-02 16:13)

You may visit Change Log to view the changes.

Please download the skeleton code and read the code as you progress through the following sections.

We have written a Makefile for you. To generate the executable, run `make` in the terminal. To run the executable, run `./pa1.exe` in the terminal and input the test case number you want. More on the Submission and Deadline section.

Class Introduction

We will introduce each class in detail in this section. We will go through the student-related classes first.

Class Student

```
class Student {
  private:
  char* name;
  int student_id;
  double gpa;
  int max_credit;
  int curr_credit;

  int num_enrolled_course;
  char** enrolled_courses;

  int pending_credit;
  Swap_List* swap_list;

  public:
  Student(const char* const name, const int student_id, const double gpa);
  Student(const Student& student);
  ~Student();

  void print_info() const;

  // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

Data Members:

- `name`: name of the student, a pointer to a dynamically allocated array of char(s).
- `student_id`: student identifier, will be used to retrieve a student record from the student database.

- `gpa`: gpa of the student, will be used in credit overloading.
- `max_credit`: the maximum number of credits that the student can enroll. The default max_credit will be set to `STUDENT_INIT_MAX_CREDIT`, defined in `student.cpp`, with value 18.
- `curr_credit`: the current number of credits that the student has enrolled in.
- `num_enrolled_course`: the number of courses that the student currently has enrolled on.
- `enrolled_courses`: the name of the courses that the student has enrolled on. It is a pointer to a dynamically allocated array of `char` pointers, while each `char` pointer points to another dynamically allocated array of `char`(s), storing one course name. The capacity of the dynamic array will be `STUDENT_MAX_NUM_COURSE`, defined in `student.cpp`, with value 10.
- `pending_credit`: it is a special kind of credit designed for this assignment. It will be introduced later.
- `swap_list`: a pointer to a dynamically allocated `Swap_List` object. It will record all swap requests still waiting in the `wait_list` of the `target_course`. In order words, if the student gets into the `target_course` successfully without waiting, then the swap request will NOT be recorded. A swap request will be introduced in class `Swap_List` later.

Member Functions:

```
Student(const char* const name, const int student_id, const double gpa);
```

This is a constructor of the class `Student`. It will be called when the system admits a new student, which means the student has not enrolled on any courses. This constructor will initialize all the data members in the class.

- `this->name`: deep copy of parameter `name`, the size of the dynamically allocated array that `this->name` points to should be just enough to hold all the characters in the parameter `name`.
- `this->student_id`: copy assignment of parameter `student_id`.
- `this->gpa`: copy assignment of parameter `gpa`.
- `max_credit`: initialize to `STUDENT_INIT_MAX_CREDIT`.
- `curr_credit`: initialize to 0.
- `num_enrolled_course`: initialize to 0.
- `enrolled_courses`: create a dynamic array of char* with capacity STUDENT_MAX_NUM_COURSE.
- `pending_credit`: initialize to 0.
- `swap_list`: create an object of the `Swap_List` dynamically with its default constructor.

```
Student::Student(const Student& student);
```

This is a copy constructor of the class `Student`. It will perform a deep copy of the parameter `student`.

```
Student::~Student();
```

This is a destructor of the class `Student`. It will clean up all the dynamically allocated objects in this class.

Class Swap and Class Swap_List

Just now, we have talked about the data member `swap_list` in class `Student`. We will explain it in this section.

```
class Swap {
  public:
  char* original_course_name;
  char* target_course_name;
  Swap* next;

  Swap(const char* const original_course_name, const char* const target_co
  ~Swap();
};
```

A `Swap` is very similar to a node in a linked list. Every `Swap` object in this assignment is dynamically allocated. It will record a swap operation currently in the `wait_list` of the `target_course`. Once the student gets into the `target_course`, the `Swap` object will be deleted.

Data Members:

- `original_course_name`: a pointer to a dynamically allocated array of `char`(s), record the original_course_name of the swap operation.
- `target_course_name`: a pointer to a dynamically allocated array of `char`(s), record the target_course_name of the swap operation.
- `next`: a pointer that points to the next dynamically allocated `Swap` object.

Member Functions:

```
Swap(const char* const original_course_name, const char* const target_cour
```

This is a constructor of the class `Swap`. Similar to the data member `name` in class `Student`, it will perform a deep copy of the two parameters `original_course_name` and `target_course_name` to `this->original_course_name` and `this->target_course_name`, respectively, and also copy assignment of the `next` parameter.

Note: in this assignment, all the name-related variables (char) are distinct dynamically allocated arrays, which means that the construction of these variables relies on deep copying of the actual function parameter(s).*

```
Swap::~Swap();
```

This is a destructor of the class `Swap`.

Note: There is no copy constructor of the class `Swap` because a deep copy of a `Swap` object is not meaningful. (Think: how will we tackle the next pointer?)

To make the deep copy meaningful, we introduce the class `Swap_List`. `Swap_List` is a wrapper class of `Swap` and a linked list.

```
class Swap_List {
    private:
    Swap* head;

    Swap_List();
    Swap_List(const Swap_List& swap_list);
    ~Swap_List();

    void print_list() const;

    // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

Data Members:

- `head`: it is a pointer pointing to the head of the linked list containing `Swaps`.

Member Functions:

```
Swap_List::Swap_List();
```

This is a constructor of the class `Swap_List`. It will be called when the system admits a new student. Since a new student does not enroll on any course, `this->head` will be assigned with `nullptr`.

```
Swap_List::Swap_List(const Swap_List& swap_list);
```

This is a copy constructor of the class `Swap_List`. It will perform a deep copy of the whole `swap_list`, particularly, all `Swap` in the linked list.

```
Swap_List::~Swap_List();
```

This is a destructor of the class `Swap_List`.

Class Student_Database (Implemented)

```
class Student_Database {
  private:
  Student** students;
  int capacity;
  int size;

  public:
  Student_Database(const int capacity);
  Student_Database(const Student_Database& database);
  ~Student_Database();

  bool create_entry(const char* const name, const int student_id, const do
  Student* get_student_by_id(const int student_id) const;

  void print_all_students() const;

  // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

Data Members:

- `students`: a pointer to a dynamically allocated array of `Student` pointers, each of them points to a dynamically allocated `Student` object.
- `capacity`: the maximum number of students the database can contain.
- `size`: the current number of students the database stores.

Note: You may find "size" and "capacity" confusing because their meaning is very similar. However, these variable names come from a powerful array-liked container `vector` in C++ standard template library (STL). You will learn this STL container later in this course.

Member Functions:

```
Student_Database::Student_Database(const int capacity);
```

This is a constructor of the class `Student_Database`. It will be called when the constructor of the class `System` is called. It will do copy assignment for `capacity` and initialize `size` to 0. Then it will dynamically allocate an array of type `Student*`, of size `capacity`. Note that since there is no student in the system initially, all the entries of this array will be initialized as `nullptr`.

```
Student_Database::Student_Database(const Student_Database& database);
```

This is a copy constructor of the class `Student_Database`. It will perform a deep copy of the whole `database`.

```
Student_Database::~Student_Database();
```

This is a destructor of the class `Student_Database`.

```
bool create_entry(const char* const name, const int student_id, const doub
```

This is a function to create a student entry in the database.

```
Student* get_student_by_id(const int student_id) const;
```

This is a function to search for a `Student` object in the database that has the same `student_id` as the function parameter and returns a pointer pointing to the object.

Next, we will introduce course-related classes.

Class Course

```
class Course {
    private:
    char* name;
    int num_credit;
    int capacity;
    int size;
    Wait_List* wait_list;
    int* students_enrolled;

    public:
    Course(const char* const name, const int num_credit, const int course_
    Course(const Course& course);
    ~Course();

    void print_info() const;

    // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

Data Members:

- `name`: stores the name of the course, a pointer to a dynamically allocated array of type `char`.
- `num_credit`: the number of credits of this course.
- `capacity`: the maximum number of students this course can accommodate.
- `size`: the current number of students enrolled on the course.
- `wait_list`: a pointer to a dynamically allocated `Wait_List` object.
- `students_enrolled`: a pointer to a dynamically allocated array of type `int`, each entry stores the `student_id` of students that enrolled on the course.

Member Functions:

```
Course(const char* const name, const int num_credit, const int course_capa
```

This is a constructor of the class `Course`. It will be called when we add a new course to the system. This constructor will initialize all the data members in the class.

- `this->name`: perform a deep copy of the function parameter `name`.
- `this->num_credit`: copy assign the function parameter `num_credit`.
- `this->capacity`: copy assign the function parameter `course_capacity`.
- `size`: initialize to 0.
- `wait_list`: allocate a `Wait_List` object dynamically with the default constructor of `Wait_List`.
- `students_enrolled`: dynamically allocate an array of size `course_capacity`, with every entry to be 0 initialized.

```
Course::Course(const Course& course);
```

This is a copy constructor of the class `Course`. It will perform a deep copy of the whole `course`. You should take the dynamically allocated objects with extra care.

```
Course::~Course();
```

This is a destructor of the class `Course`.

Class Student_ListNode and Class Wait_List

This part is very similar to the class `Swap` and class `Swap_List` session.

```
class Student_ListNode {
  public:
  int student_id;
  Student_ListNode* next;

  Student_ListNode(const int student_id, Student_ListNode* const next);
  ~Student_ListNode() = default; // default means we do not clean up the c
};
```

`Student_ListNode` is a node inside the `Wait_List` linked list. Just like `Swap` in `Swap_List`, every `Student_ListNode` object in this assignment is dynamically allocated.

Data Members:

- `student_id`: stores the `student_id` of the student.
- `next`: a pointer that points to the next dynamically allocated `Student_ListNode` object.

Member Functions:

```
Student_ListNode(const int student_id, Student_ListNode* const next);
```

This is a constructor of the class `Student_ListNode`. It should perform copy assignments for all data members from the function parameters.

```
Student_ListNode::~Student_ListNode() = default;
```

This is a destructor of the class `Student_ListNode`. Note that we marked it default because we will rely on the destructor of the class `Wait_List` to perform the clean-up of all dynamically allocated `Student_ListNode`.

Note: There is no copy constructor of the class `Student_ListNode`, same as class `Swap`, for the same reason. Deep copy is not meaningful for this class.

```
class Wait_List {
  private:
  Student_ListNode* head;
  Student_ListNode* end;

  public:
  Wait_List();
  Wait_List(const Wait_List& wait_list);
  ~Wait_List();

  void print_list() const;

  // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

Data Members:

- `head`: it is a pointer pointing to the head of the linked list containing `Student_ListNode`.

- `end`: it is a pointer pointing to the end of the linked list containing `Student_ListNode`.

When a student joins a `wait_list`, he/she will be appended to the `end` of the list. When a student drops from a course and a vacancy appears, the student at the `head` position can get into the course. Therefore, we have both the `head` and `end` pointers for fast manipulation.

Member Functions:

```
Wait_List::Wait_List();
```

This is a constructor of the class `Wait_List`. It will be called when the system adds a new course. Since initially, no student enrolled on this course, `this->head` and `this->end` will be assigned with `nullptr`.

```
Wait_List::Wait_List(const Wait_List& wait_list);
```

This is a copy constructor of the class `Wait_List`. It will perform a deep copy of the whole `wait_list`, particularly, all `Student_ListNode` in the `wait_list`.

```
Wait_List::~Wait_List();
```

This is a destructor of the class `Wait_List`.

Class Course_Database (Implemented)

```
class Course_Database {
  private:
  Course** courses;
  int capacity;
  int size;

  public:
  Course_Database(const int capacity);
  Course_Database(const Course_Database& database);
  ~Course_Database();

  bool create_entry(const char* const name, const int num_credit, const int
  Course* get_course_by_name(const char* const course_name) const;

  void print_all_course() const;

  // Accessors and mutators - READ .CPP FILE FOR MORE INFO
};
```

Data Members:

- `courses`: a pointer to a dynamically allocated array of `Course` pointers, each entry points to a dynamically allocated `Student` object.
- `capacity`: the maximum number of courses the database can contain.
- `size`: the current number of courses the database stores.

Member Functions:

```
Course_Database(const int capacity);
```

This is a constructor of the class `Course_Database`. It will be called when the constructor of the class `System` is called. It will do copy assignment for `capacity` and initialize `size` to 0. Then it will dynamically allocate an array of type `Course*`, of size `capacity`. Note that since there is no student in the system initially, all the entries of this array pointed by `courses` will be initialized as `nullptr`.

```
Course_Database::Course_Database(const Course_Database& database);;
```

This is a copy constructor of the class `Course_Database`. It will perform a deep copy of the whole `database`. You should take dynamically allocated objects with extra care.

```
Course_Database::~Course_Database();
```

This is a destructor of the class `Course_Database`.

```
bool create_entry(const char* const name, const int num_credit, const int
```

This is a function to create a course entry in the database. We will talk about the mechanism of how record creation is handled in this class later.

```
Course* get_course_by_name(const char* const course_name) const;
```

This is a function to search for a `Course` object in the database that has the same `name` as the function parameter, and returns a pointer pointing to the object.

Quick Note on Accessor and Mutator Functions

Every class has a set of accessor and mutator functions to allow you to access to the data members from outside classes, or any helper function that you create. Please go through all of the .cpp files given to you to understand which accessor/mutator functions you have access to.

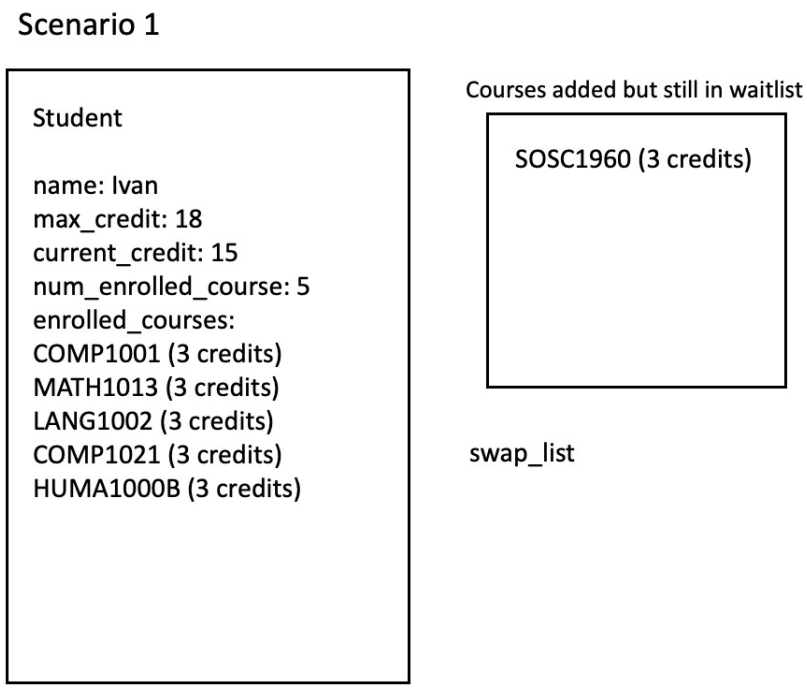
Mechanism

In this section, we will introduce some important mechanisms of the system. We will give each mechanism/policy a name so that when you have any questions and would like to post on Piazza, you may refer to them.

Worst Case Credit Control Policy

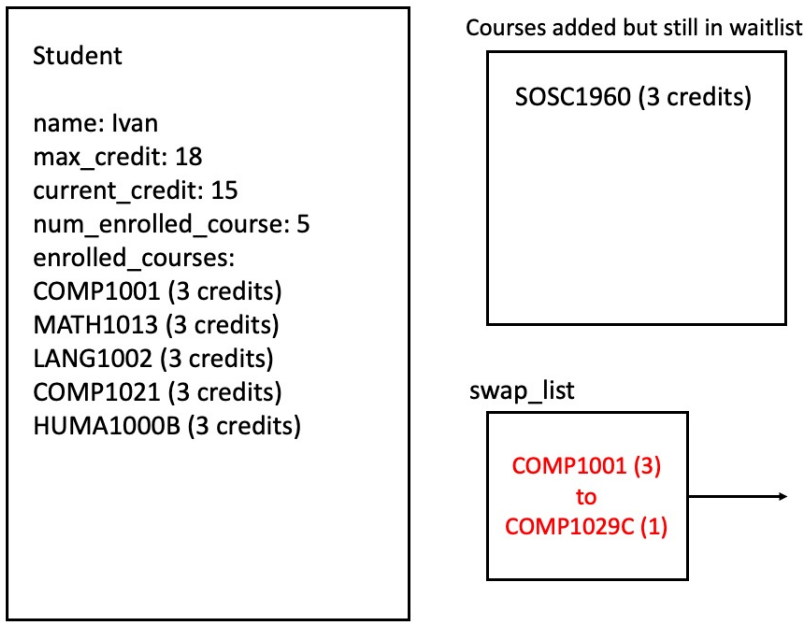
Do you still remember the class data member `pending_credit` in class `Student`? We will address its usage here.

In the real HKUST course registration system, students are allowed to add/swap many courses even if they have no credits left. However, in this assignment, you will implement a "worst-case credit control policy" to ensure that a student's `curr_credit` is always less than or equal to his/her `max_credit` in ANY possible scenario. The system will reject the student's add/swap request if the such request violates the policy. Let's start with an example.



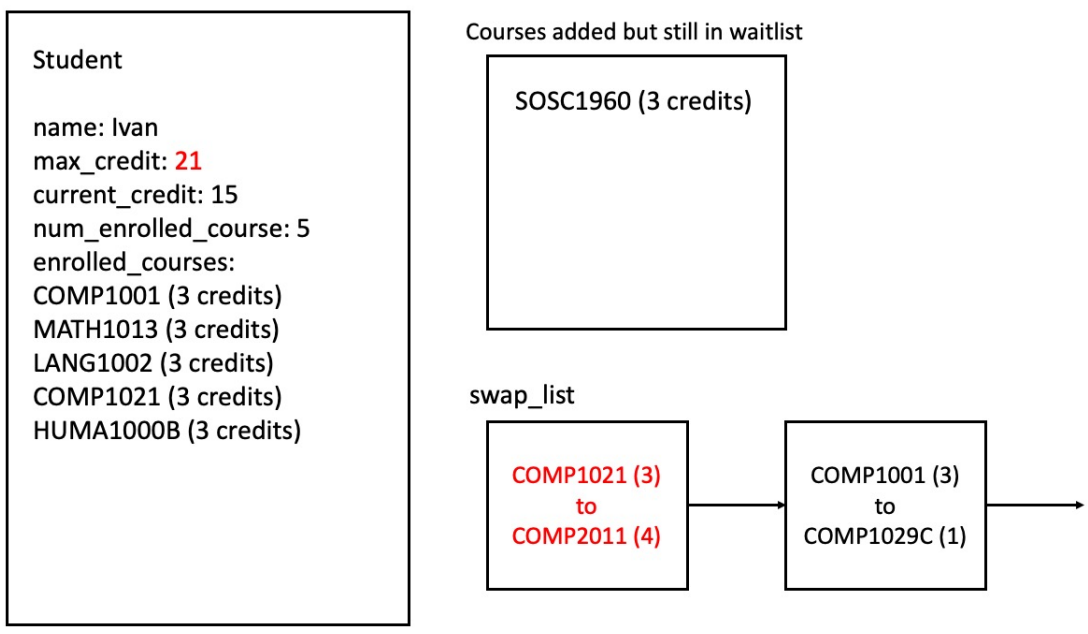
Although Ivan is currently enrolled for 15 credits, it is still possible for him to get into SOSC1960. Therefore, his credit in the worst-case scenario will be 18. That means right now Ivan cannot add any course with credit larger than 0, or submit a swap request to a course that introduces more credits to him.

Scenario 2



What if there was a swap request that could potentially reduce the number of credits? Indeed, if Ivan can get into COMP1029C through swapping, his `curr_credit` will be 13, and then you may say he can add a course with at most 2 credits. However, the swap request that stays in `swap_list` is also on the course's waitlist. In worst-case scenario, Ivan cannot get into COMP1029C, and he may get into SOSC1960 if someone else drops, meaning the worst-case credit count is still 18.

Scenario 3



Ivan decides to overload to 21 credits and it is approved. He found out that he should study COMP2011 (4 credits) rather than COMP1021. Therefore, he requests a swap. Now, in the worst-case scenario, he will get into SOSC1960, swap to COMP2011, failed to swap to COMP1029C, leading to a worst-case credit count of 19 credits. Therefore, he will be rejected if he tries to add a course of 3 credits or apply to a swap request that will possibly add 3 more credits to his workload.

Both `curr_credit`, and `max_credit` have different meanings. To enforce the policy, we introduce the third credit class data member `Student::pending_credit`, which represents the worst-case extra credits that the student could take. For example:

- Scenario 1: `pending_credit` = credit from SOSC1960 = 3
- Scenario 2: `pending_credit` = credit from SOSC1960 = 3
- Scenario 3: `pending_credit` = credit from SOSC1960 + increase of credit after swapping to COMP2011 = 3 + 1 = 4

The worst-case credit control policy can be simplified into a formula. Our target is always to keep the formula hold:
curr_credit + pending_credit <= max_credit

We will reject any add/swap request that will break the above formula. For example:

- In Scenario 1, Ivan's application to add COMP1029C (1 credit) will be rejected, but he can add COMP2633 (0 credit).
- In Scenario 3, Ivan's application to add HUMA1000B (3 credits) will be rejected, but he can add COMP3633 (2 credits).

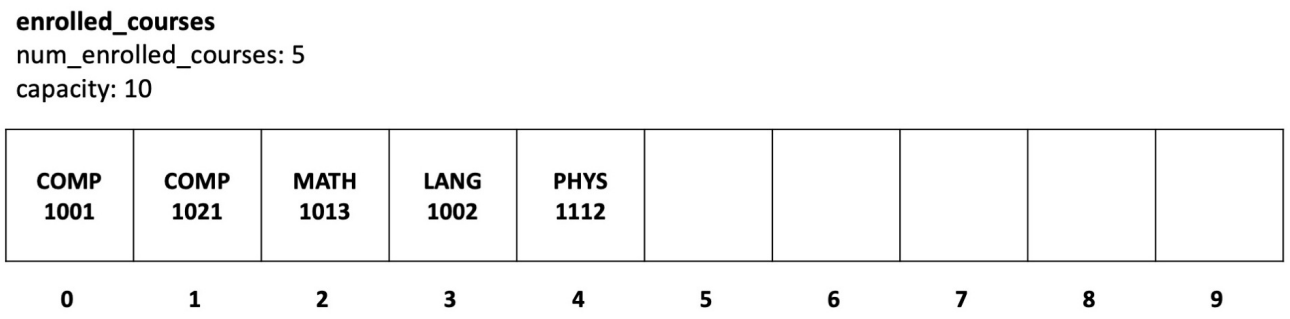
This policy will give us a nice property. Whenever someone drops a course, then the student at the first place on the wait_list will always have enough credit to get into the course (You may have to think for a moment to convince yourself). This greatly simplifies the drop operation as we don't have to go through the wait_list.

Array: Insertion and Deletion

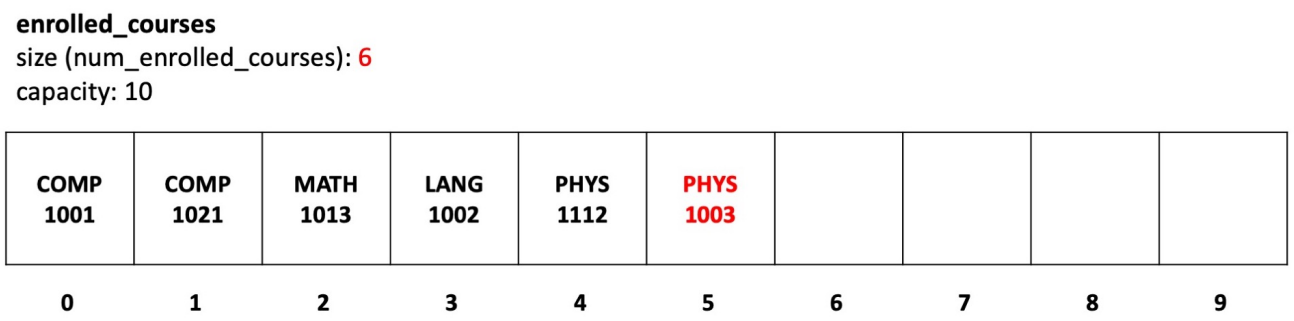
There are many dynamically allocated arrays used in this assignment. Here we will define a unified way to insert/delete an item.

We will use `Student::enrolled_courses` as an example. The important parts are marked in red.

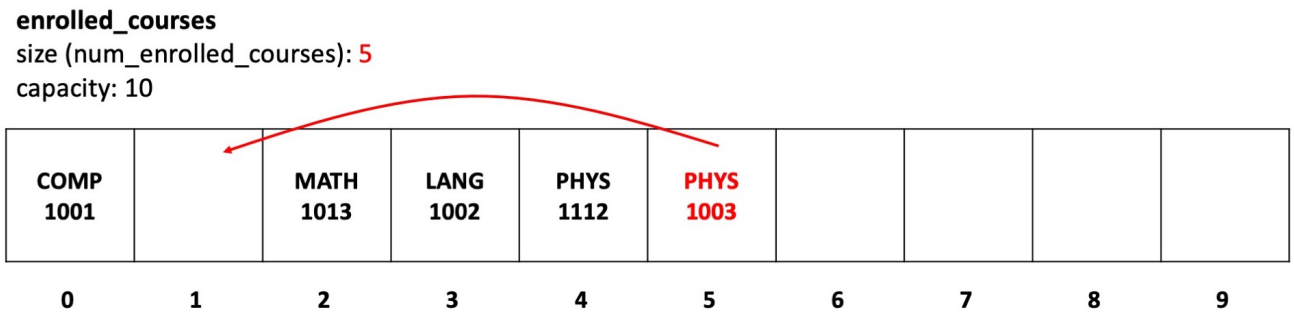
Assume now Ivan has enrolled on 5 courses, and the courses are stored in this format.



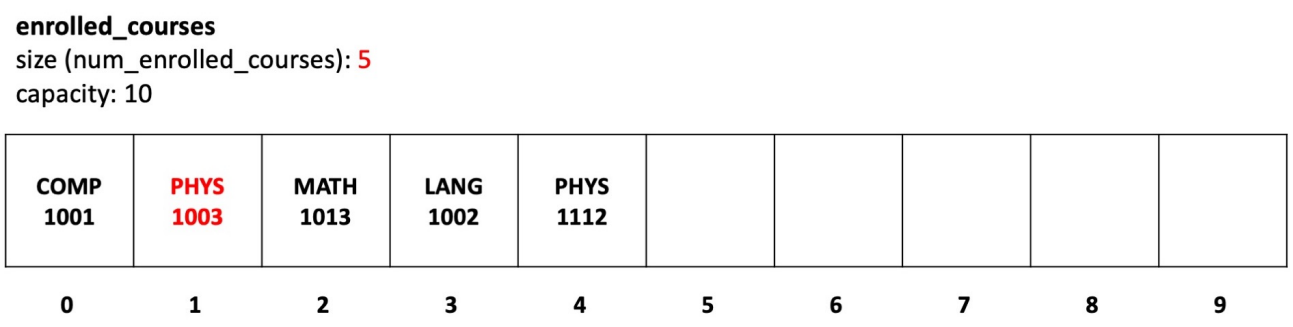
Ivan now registers for a new course PHYS1003. Here we will introduce a mechanism for insertion. We do NOT sort the array and any newly created item will be **appended to the end** of the array. We have to introduce the notion of `size` and `capacity` again. The `capacity` refers to the maximum number of objects this dynamic array can store, whereas `size` refers to the current number of objects stored in this dynamic array. The newly created item will always be assigned to the index `size`, i.e. 5 in this case. (It seems like magic, but why it works?) After that, we will increase the `size` counter by 1.



Ivan decided to drop COMP1021. Here we will introduce a mechanism for deletion. Since we do not sort the array, then we can just simply **replace it with the last** in the array.
Note: If the deleted item is already the last item in the array, then we do not have to move anything.



We will then decrease the `size` counter by 1.



The intention for these insertion/deletion mechanisms is to always store the items in the first `size` number of entries in the array. With that, we can have a very nice property: when we add a new item, we can always insert it at the index `size`.

Special Cases: How about when we have to add an entry and delete an entry simultaneously? Which operation goes first?

You will encounter this situation in 2 ways:

1. When someone drops a course and the course's waitlist is not empty, we have to remove 1 student_id and introduce the student_id of the student at the head of the course's waitlist to `Course::students_enrolled`. Under such a situation, we can replace the dropped `student_id` with the new `student_id`.

2. When someone swaps successfully, we have to remove the `original_course_name` and introduce the `target_course_name` to `Student::enrolled_courses`. We will insert `target_course_name` first and then perform a drop request to the `original_course`. This will help us replace the `original_course_name` with the last entry (i.e. `target_course_name`) through Array Deletion Mechanism. The final effect is similar to the above Case, and the `target_course_name` replaces the `original_course_name`.

Important Note: Things get complicated for the 2nd case. If you think deeply, it is possible that at a certain moment in the system, a student may "enrol" up to 11 courses (because we add first, then drop) which violates the `STUDENT_MAX_NUM_COURSE` (i.e. 10). However, we promise that there will be no test cases in which a `Student` can register for more than 9 courses at any moment. This means that you can strictly follow the above mechanisms without any boundary or size checking.

In this assignment, such a mechanism is applicable for:

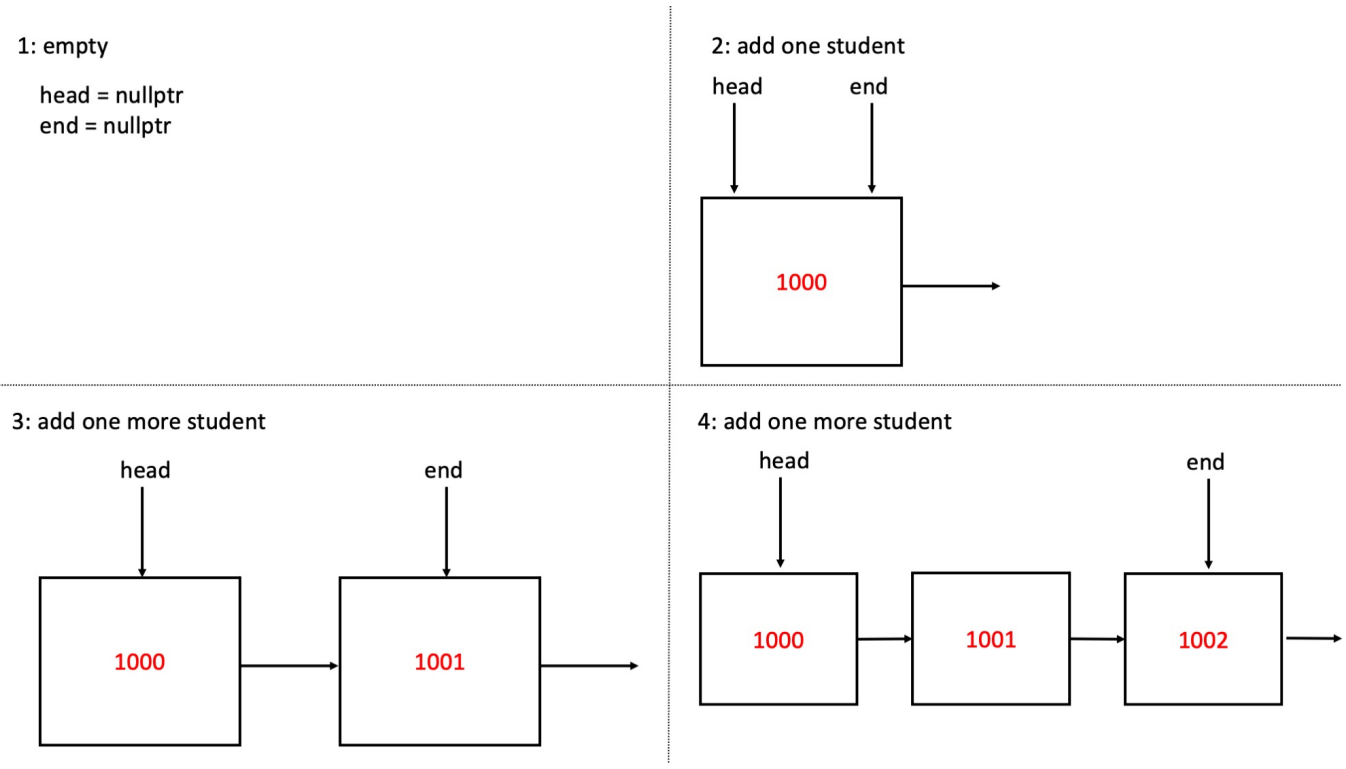
- `Course::students_enrolled`
- `Student::enrolled_courses`

You should strictly follow the mechanism for these class data members or else you will produce mismatched output results.

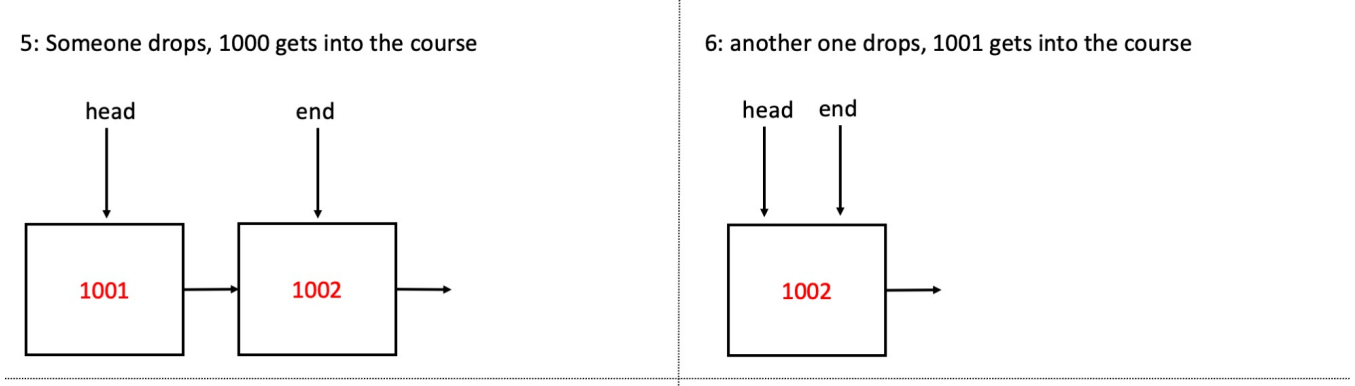
We will talk about the linked list next.

Wait_List

Class `Wait_List` has a `head` pointer and an `end` pointer. We will illustrate how to use them with an example.



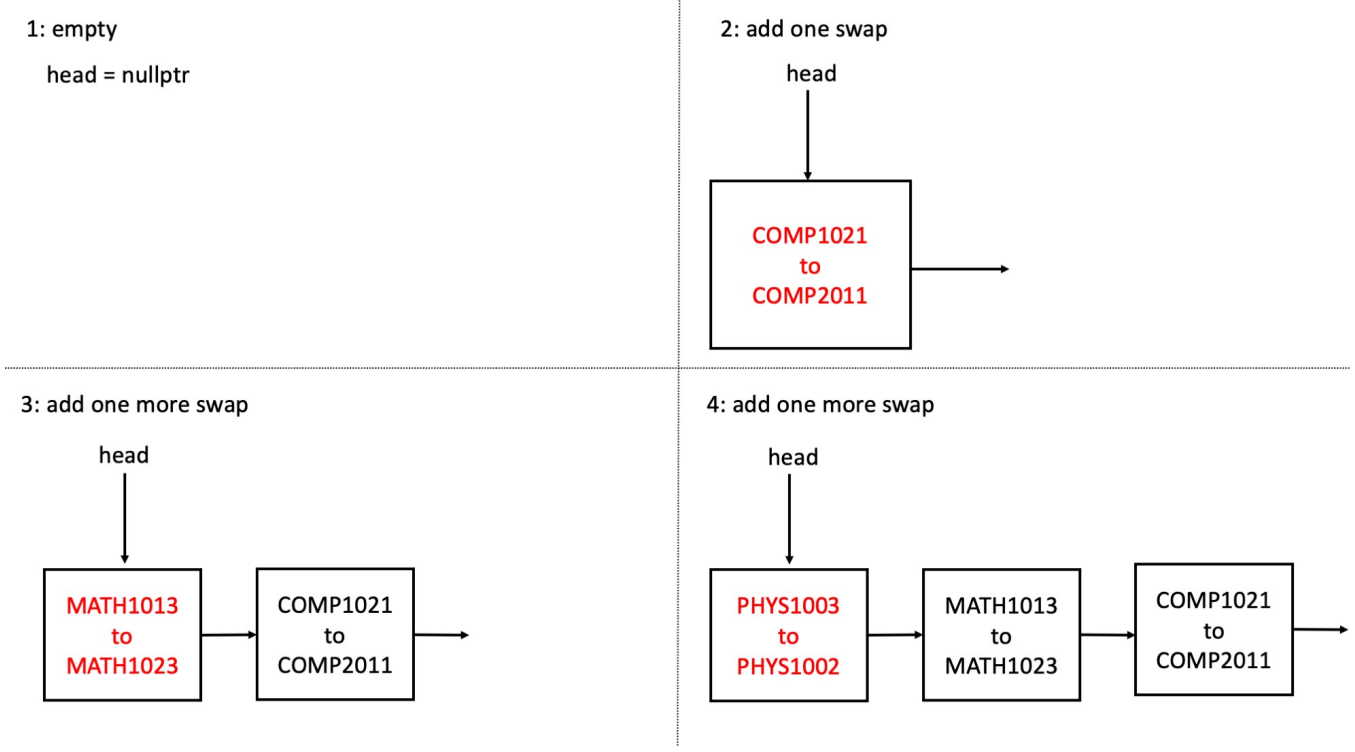
The above diagram shows the object creation process and what object should the pointers point to. You should be careful that when the list contains one element, both `head` and `end` are pointing to the same object. Since we have the `end` pointer, when someone joins the waitlist, we can append it to the end of the list very easily.



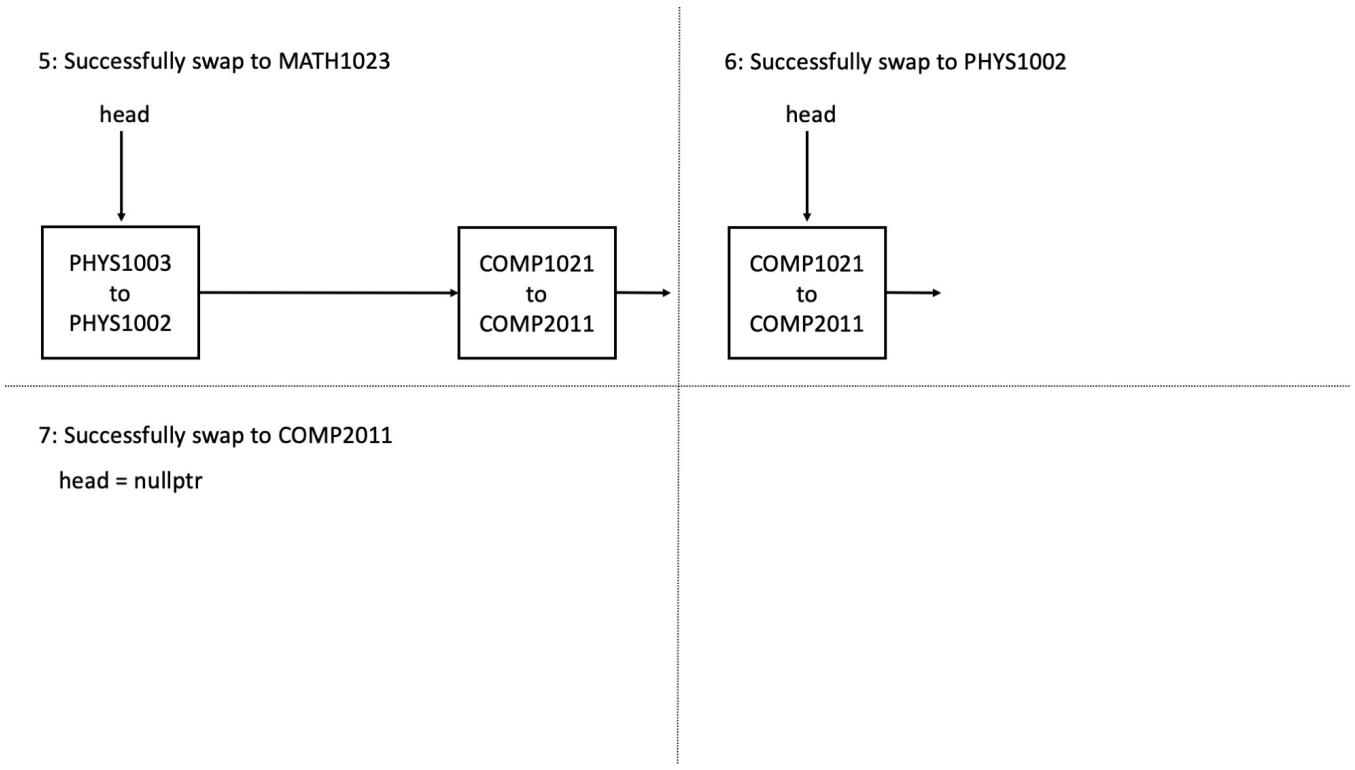
The above diagram shows the object deletion process and what object the pointers should point to.

Swap_List

Class `Swap_List` has a `head` pointer only. We will illustrate how to use them with an example.



The above diagram shows the object creation process and what object should the pointer should point to. You can observe that the newly created object is always **appended to the front** of the list.



The above diagram shows the object deletion process and what object should the pointers point to. It is very similar to the list node object deletion you have learned in COMP2011.

With all the above mechanisms explained, we ensure that we have a unique way to implement the system. Make sure that you strictly follow all the mechanisms discussed, or else you may have different output compared to the grading system.

Read the FAQ page for some common clarifications. You should check that a day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work.

If you need further clarification of the requirements, please feel free to post on the Piazza (via Canvas) with the pa1 tag. However, to avoid cluttering the forum with repeated/trivial questions, please **read all the given code, webpage description, sample output, and latest FAQ (refresh this page regularly) carefully before posting your questions**. Also, please be reminded that we won't debug any student's assignment for fairness.

Submission details are in the Submission and Deadline section.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to ensure you understand what is considered as plagiarism and the penalties. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, up to and including expulsion.

Task

In this section, we will describe what you need to do to complete this programming assignment.

Task 1: Implement all Conversion/Other Constructors and Destructors

As you may have to call other classes' constructors or destructors during the implementation, you are suggested to work on them in the following sequence:

1. class `Student`
2. class `Swap` and class `Swap_List`
3. class `Course`
4. class `Student_ListNode` and class `Wait_List`

Though the following classes' constructors and destructors are implemented for you. You should read them before working on Task 1.

1. class `System`
2. class `Student_Database`
3. class `Course_Database`

You are also suggested to study `main.cpp` to understand which functions/constructors/destructors are called.

Please refer to the section Classes for a detailed explanation of the implementations. Make sure you understand what each class data member stores.

Note: There exists some class data member that does not contain any objects when the object is constructed. For example, `Course::Wait_List`. However, they may contain dynamic objects after the system starts running. When writing the destructor, you should also take care of these class data members to prevent a memory leak.

Task 2: Implement all Copy Constructors

The most important principle of developing such a system is to do a regular backup. Here we have to implement all the copy constructors to create another copy of the system. Such copy should be a deep copy.

Similarly, you are encouraged to call other classes' copy constructors when writing one. For example, you may call the copy constructor of class `Swap_List` when writing the copy constructor of class `Student`. You are suggested to implement the copy constructors in the following sequence.

1. class `Swap_List`
2. class `Student`
3. class `Wait_List`
4. class `Course`

You should read the following implemented classes' copy constructor before working on Task 2.

1. class `System`
2. class `Student_Database`
3. class `Course_Database`

Please refer to the section Classes for a detailed explanation of the implementations.

Task 3: Implement all system functions

You have to implement the functionality provided by the system.

Note: You are suggested to implement the system as the following flow. However, it is not a must to follow as there could be other ways to produce the same output.

Note 2: Whenever you see "update the corresponding class data member", it indicates that it is suggested to modify some/all class data member that needs to be changed. However, you may do it earlier/later as long as they match with the final desired outcome.

```
bool apply_overload(const int student_id, const int request_credit);
```

The credit overload criteria are as follows:

1. Students can never request more than 30 credits, meaning that a student can study 30 credits at maximum.
2. If a student would like to overload to more than or equal to 24 credits, the student should have a GPA of at least 3.7.
3. If a student would like to overload to more than 18 credits, the student should have a GPA of at least 3.3.

The function should return **false** when:

1. The `request_credit` is more than 30 credits.
2. The `request_credit` is valid, but the student does not have enough GPA.

Otherwise, the function should return **true**, indicating the student is allowed to overload and the request is approved.

Note: You can assume that every apply_overload request is an "overload" request, meaning there will be no student asking for "overloading" to credit smaller or equal to 18.

```
bool add(const int student_id, const char* const course_name);
```

This function will perform the following in sequence:

1. Enforce the "Worst Case Credit Control Policy", and check whether the student can perform such an operation. Return **false** if such operation violates the policy.
2. Check if the course has any vacancies. If there is, enroll the student in the course. If not, add the student to the waitlist for the course.
3. Update the corresponding class data members of the student and the course. Return **true**.

This function will return **false** only when such a request violates the "Worst Case Credit Control Policy". Otherwise, the function should return **true**. This means even if the student gets on the waitlist for the course, the function will still return **true**.

```
bool swap(const int student_id, const char* const original_course_name, co
```

This function will perform the following in sequence:

1. Enforce the "Worst Case Credit Control Policy", and check whether the student can perform such an operation. Return **false** if such operation violates the policy.
2. Check if the course has any vacancies.

    - If there is,
        1. Enroll the student into the `target_course` and update the corresponding `Course` and `Student` class data members.
        2. Drop the `original_course` (You are encouraged to use the `System::drop` function for this).
           Note: Please refer to the Special Case under `Array: Insertion and Deletion` part.
    - If not, add the student to the waitlist of the course and create a dynamically allocated `Swap` object and add to the `Swap_List` of the student to record a pending `Swap` waiting in the waitlist.
3. Return **true**.

```
void drop(const int student_id, const char* const course_name);
```

You may notice that this function does not have any return type. It is because there is no restriction for students to drop any courses. This function will perform the following in sequence:

1. Drop the student from the course. Update the corresponding class data members of the student. You should delete the dynamically allocated memory corresponding to the dropped `course name` in `enrolled_courses` for the dropped student.
2. Check if anyone is waiting on the course waitlist.
    - If yes, the student in the first place on the waitlist can get into the course.
        1. The student must be able to get into the course (guaranteed by Worst Case Credit Control Policy). Enroll the student in the class and update the class data members of the course and the student. Do remember to delete the `Student_ListNode`.
        2. Check if the newly enrolled student gets onto the waitlist through add/swap operation by searching his/her `Swap_List`.
            - If there exists a `Swap` where the `target_course_name` is equal to `course_name`, we can be sure that the student gets onto the waitlist through the swap operation. Then, drop the `original_course` for the student and remove the `Swap` node from the `Swap_List`. Update the corresponding class data members of the student.
            - If not, the student gets enrolled through add operation. Update the corresponding class data members of the student.
    - If no, update the corresponding class data members of the course.

Hint: In this drop function, you have to clean up the dynamically allocated memory of a course name, possibly clean up dynamically allocated `Student_ListNode` or `Swap`.

Note: We provide several helper function declarations and explanations in the skeleton code in `system.cpp`. You may find them useful when doing task 3. However, there will NOT be any test cases testing these functions.

Note 2: You can assume these functions will be called in a well-behaved sequence, meaning that the student will only drop the course when he/she is enrolled on the course, The student will not add a course that already enrolled, the student has enrolled on the `original_course_name` in swap operation...

Hints

1. This assignment is complicated in terms of understanding the requirements. Read the documentation thoroughly especially for the Classes and the Mechanism part.
2. Another difficult point is the dependencies between classes. For every action performed in the code segment, check all the class data member variables and determine which has to be modified.
3. If you cannot pass some test cases, besides reviewing your code, do not overlook on the fact that you misunderstand the requirements.
4. Write more tests. It is very easy to create your own test cases and predict the final outcome once you understand the mechanisms and requirements.
5. Although you should not include debug messages in your submissions, you are encouraged to write debug messages for every function to trace your program better. Make sure you comment all of them before submission.
6. The functions in `system.cpp`, `Course_Database::get_course_by_name` and `Student_Database::get_student_by_id` are very useful because they can retrieve the entry in the database. You will have to use them in every system function.

Sample Output and Grading Scheme

Your finished program should produce the same output as our sample-output for all given test cases. User input, if any, is omitted in the files. Please note that sample output, naturally, does not show all possible cases. It is part of the assessment for you to design your test cases to test your program. Remember to remove any debugging message you might have added before submitting your code.

There are 20 given test cases which the code can be found in the given main function in `main.cpp`. These 20 test cases are first run without memory leak checking (numbered #1 - #20 on ZINC). Then, the same 20 test cases will be rerun, in the same order, with memory leak checking (those will be numbered #21 - #40 on ZINC). For example, test case #28 on ZINC is actually the given test case 8 (in the given main function) run with memory leak checking.

Each test case run without memory leak checking (i.e., #1 - #20 on ZINC) is worth 1 mark. The second run of each test case with memory leak checking (i.e., #21 - #40 on ZINC) is worth 0.25 mark. The maximum score you can get on ZINC before the deadline, will be 20*(1+0.25) = 25.

About memory leak and other potential errors
Memory leak checking is done via the `-fsanitize=address,leak,undefined` option (related documentation here) of a recent g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to the "Your Output" tab in the test case details popup) for errors such as memory leaks. Other errors/bugs such as out-of-bounds, use-after-free bugs, and some undefined-behavior-related bugs may also be detected. You will get a 0 mark for the test case if there is any error. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leaks yourself using the same options, you may follow our Checking for memory leak yourself guide.

After the deadline
We will have 24 additional test cases, which won't be revealed to you before the deadline. Together with the 20 given test cases, there will then be 44 test cases used to give you the final assignment grade. All 44 test cases will be run two times as well: once without memory leak checking and once with memory leak checking. The assignment total will therefore be 44*(1+0.25) = 55. Details will be provided in the marking scheme, which will be released after the deadline.

Here is a summary of the test cases for your information.

Main thing to test	**Number of test cases in main before deadline (given test cases)**
Student: constructor	2
Course: constructor	2
System: credit overload function	2
System: add function	3
System: drop function	3
System: swap function	3
Worst Case Credit Control Policy	2
Deep Copy	3

Functions in this assignment are correlated. For example, to drop a class, you must be able to add a class. Therefore, you are suggested to tackle the test cases in ascending order. Do note that the provided test cases only test on a `limited` functionality. In the hidden test cases, we will perform a comprehensive check for all features introduced (function implemented by you and all mechanisms).

Assumptions
The followings assumptions hold for ALL the test cases (including the hidden test cases):

1. A student's gpa is always between 0.0 to 4.3.

2. Since there is a precision problem in representing floating point numbers, we will not perform a boundary test for the credit overload function.
3. Each student is given a different student_id.
4. Every course has a unique course name.
5. The student will never request credit lower than or equal to 18.
6. At any moment, all students in the system will have enrolled at most 9 courses.
7. All kinds of credits must be non-negative.
8. All add, drop, swap requests are valid, i.e.:
    - The student will drop a course only when he/she has enrolled on the course.
    - The student will swap a course only when he/she has enrolled on the `original_course`.
    - The student will swap a course only when he/she has not enrolled on the `target_course`.
    - There will not be any course being the `target_course` of both an add operation and a swap operation from the same student.
    - The student will not use the same course as the `target_course` or `original_course` for 2 or more swap requests.
    - ...

Submission and Deadline

Deadline: 23:59:00 on Oct 16, 2022 (Sunday)

Please submit the following files to ZINC by zipping the following 5 files. ZINC usage instructions can be found here.

- `system.cpp`
- `course.cpp`
- `student.cpp`
- `swap_list.cpp`
- `wait_list.cpp`

Notes:

- You may submit your file multiple times, but only the last submission will be graded. **You do NOT get to choose which version we grade.** If you submit after the deadline, a late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect you would get the grading result report not very quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.
- In the grading report, pay attention to various errors reported. For example, **under the "make" section, if you see a red cross, click on the STDERR tab to see the compilation errors.** You must fix those before seeing any program output for the test cases below.
- Make sure you submit the correct file yourself. You can download your file back from ZINC to verify. Again, **we only grade what you uploaded last to ZINC**.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts you cannot finish, just put in dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have done. Empty implementations can be like:

```cpp
int SomeClass::SomeFunctionICannotFinishRightNow()
{
    return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded(
{
}
```

Late submission policy:

We will normalize the assignment score to 100 by this formula:

```
( YOURSCORE / 55 ) * 100
```

There will be a penalty of -1 point (out of a maximum of 100 points) for every minute you are late. For instance, since the assignment deadline is 23:59:00 on Oct 16th, submitting your solution at 1:00:00 on Oct 17th will be a penalty of -61 points for your assignment. If you submit at 1:39:00 on Oct 17th, you will receive 0 marks for this assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to a late penalty (and any other penalties) will be reset to zero.

Change Log

Since the description of this assignment is long, and there will be frequent updates for the FAQ section, you can rely on this section to check what has been added/changed before the deadline. The following first entry is an example.

- E.g. 2022-10-02 00:00: Assignment Released.
- 2022-10-02 15:30: Skeleton code revised. All accessors of a class are now `const` member functions. Add `const-ness` for class member functions' parameters and helper functions' parameters.
- 2022-10-02 15:30: Test case 17 is revised.

  ```
  -------------------
  Student Name: Cindy
  Student ID: 1002         GPA: 4.3
  Current Credit: 0 out of Max Credit: 18
  ```

  should be changed to:

  ```
  -------------------
  Student Name: Cindy
  Student ID: 1002         GPA: 4.3
  Current Credit: 3 out of Max Credit: 18
  ```

- 2022-10-02 16:13: Skeleton code revised. In test case 20,

  ```
  system->add_course("SOSC1960", 3, 2);
  ```

  is changed to

  ```
  system->add_course("SOSC1960", 4, 2);
  ```

- 2022-10-02 16:13: Test case 20 is revised. Its output should follows Test case 19.

FAQ

Frequently Asked Questions

Q: My code doesn't work / there is an error. Here is the code. Can you help me fix it?

A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own, and we should not finish the tasks for you. We might provide some very general hints, but we shall not fix the problem or debug for you.

Q: Can I add extra helper functions?

A: You may do so in the files that you are allowed to modify and submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?

A: No. Everything you need is already included - there is no need for you to add any include statement (under our official environment).

Q: Can I use global or static variable such as "static int x"?

A: No.

Q: Can I use "auto"?

A: No.

Q: Can I use function X or class Y in this assignment?

A: In general, if it is not forbidden in the description and the previous FAQs, and you can use it without including any additional library on ZINC, then you can use it. We suggest quickly testing it on ZINC before committing to using it, as library inclusion requirements may differ in different environments.

Q: My program gives the correct output on my computer, but it gives a different one on ZINC. What may be the cause?

A: Usually inconsistent strange result (on different machines/platforms, or even different runs on the same machine) is due to relying on uninitialized hence garbage values, missing return statements, accessing out-of-bound array elements, improper use of dynamic memory, or relying on library functions that might be implemented differently on different platforms (such as pow() in cmath).
In this particular PA, it is probably related to misuse of dynamic memory. Good luck with bug hunting!