# Programming Design

# Data Structures

Ling-Chieh Kung

Department of Information Management
National Taiwan University

# Outline

- **Basic ideas**
- Lists: **`class JobList`**
- Linked lists: **`JobLinkedList`**
- More data structures

# Data structures

- A **data structure** is a specific way to **store** data.
- Usually it also provides interfaces for people to **access** data.
- Real-life examples: A dictionary.
  - It stores words.
  - It sorts words alphabetically.

# Data structures

- In large-scale software systems, there are a lot of data. We want to create data structures to store and manage them.
- We want our data structures to be **safe**, **effective**, and **efficient**
  - Encapsulation: People can access data only through managed interfaces.
  - We can store and access data correctly.
  - The number of steps required for a task is small; consider a dictionary with words not sorted!

# Data structures

- An **array** is a very simple data structure.
- Is it safe, effective, and efficient?
  - Safety: Only if suitable interfaces are provided.
  - Effectiveness: Only if suitable interfaces are provided.
  - Efficiency: To be discussed later.
- Therefore, our first attempt will be to build a "more complicated" data structure based on an array.

# Outline

- Basic ideas
- **Lists: `class JobList`**
- Linked lists: `JobLinkedList`
- More data structures

# Lists

- A list is a **linear** data structure. It stores items in a line.
  - E.g., a dictionary, a personal schedule, a team of characters, etc.
- As an example, we will implement a job list, which stores jobs.
- The class **`JobList`** will use an **array** to store jobs.
  - Jobs with a smaller index has higher priority.
- More importantly, it will provide **interfaces** to access those jobs.
  - The array will be a **private** or **protected** member variable.
  - The interfaces will be **public** member functions.

# Job

```
class Job
{ // nothing special
private:
  string name;
  int hour;
public:
  Job() { this->name = ""; this->hour = 0; }
  Job(string name, int hour)
  { this->name = name; this->hour = hour; }
  void setHour(int hour) { this->hour = hour; }
  string getName() { return this->name; }
  double getHour() { return this->hour; }
  void print() {
    cout << "(" << this->name
         << ", " << this->hour << ")";
  }
};
```

# JobList

```
const int MAX_JOBS = 100; // a global variable

class JobList
{
private:
  Job jobs[MAX_JOBS]; // where we store the data
  int count; // other attributes
public:
  JobList();
  // interfaces
  int getCount(); // should we have a setter?
  void print();
  bool insert(Job job, int index);
  Job remove(int index);
};
```

```
JobList::JobList() : count(0) {}
int JobList::getCount()
{
  return this->count;
}
void JobList::print()
{
  for(int i = 0; i < this->count; i++)
  {
    cout << "Job " << i + 1 << ": ";
    this->jobs[i].print();
    cout << endl;
  }
}
```

# JobList::insert() and remove()

```
bool JobList::insert(Job job, int index)
{
  if(index < 0 || this->count == MAX_JOBS)
    return false; // fail to insert
  else if(index > this->count)
    // insert at the end
    this->jobs[this->count] = job;
  else // usual insertion
  {
    for(int i = count - 1; i >= index; i--)
      this->jobs[i+1] = this->jobs[i];
    this->jobs[index] = job;
  }
  this->count++;
  return true;
}
```

```
Job JobList::remove(int index)
{
  Job toRemove; // to be removed and returned
  if(index < 0 || this->count == 0)
    return toRemove; // nothing to remove
  else if(index > this->count) // remove the last one
    toRemove = this->jobs[this->count];
  else // usual removal
  {
    toRemove = this->jobs[index];
    for(int i = index; i < this->count - 1; i++)
      this->jobs[i] = this->jobs[i+1];
  }
  this->count--; // the effective action of removal
  return toRemove;
}
```

# Remarks

- Is `JobList` safe, effective, and efficient?
  - Safety: People can access these data **only through** public interfaces.
  - Effectiveness: We have implemented **fail-safe** interfaces.
  - Efficiency: Not so efficient! Insertion and removal may need to move all jobs (i.e., $O(n)$).
- Drawbacks:
  - There is a limit on the total number of jobs.
  - A lot of storage spaces are wasted.
- These drawbacks exist for almost every data structure implemented with arrays, even with dynamic memory allocation.
- We will introduce another "list" that does not use an array.

# Outline

- Basic ideas
- Lists: **`class JobList`**
- **Linked lists: `JobLinkedList`**
- More data structures

# Linked lists

- A **linked list** is a list implemented by using **pointers** so that "each element has a pointer pointing to the next element".
- Advantages:
  - No limit on the number of elements stored.
  - Dynamically allocate memory spaces. Can save spaces.
  - Efficiency may be improved (in some cases).
- Disadvantages:
  - Harder to implement.
  - Efficiency may be worsen (in some cases).

# Job (a new definition)

```
class Job
{
friend class JobLinkedList; // discussed later
private:
  string name;
  int hour;
  Job* next; // pointing to the next job
public:
  // has the next job only if put in a list
  Job() : name(""), hour(0), next(NULL) {}
  Job(string name, int hour)
    : name(name), hour(hour), next(NULL) {}
  void setHour(int hour);
  string getName();
  double getHour();
  void print();
};
```

```
void Job::setHour(int hour)
{
  this->hour = hour;
}
string Job::getName()
{
  return this->name;
}
double Job::getHour()
{
  return this->hour;
}
void Job:: print()
{
    cout << "(" << this->name
         << ", " << this->hour << ")";
}
```

# JobLinkedList

```
class JobLinkedList
{
protected:
  int count;
  Job* head; // pointing to the first Job
public:
  JobLinkedList() : count(0), head(NULL) {}
  ~JobLinkedList();
  // same interfaces
  int getCount() { return this->count; }
  bool insert(Job job, int index);
  Job remove(int index);
  void print();
};
```

```
int JobLinkedList::getCount()
{
  return this->count;
}
void JobLinkedList::print()
{
  Job* temp = this->head;
  for(int i = 0; i < this->count; i++)
  {
    // print out one job
    cout << "Job " << i + 1 << ": ";
    temp->print();
    cout << endl;
    // move to the next job
    temp = temp->next;
  }
}
```

# JobLinkedList::insert()

```
bool JobLinkedList::insert(Job job, int index)
{
  Job* toInsert = new Job(job.name, job.hour);
  if(index < 0) // fail-safe
    return false;
  else if(index == 0) // insert it as the head
  {
    if(this->count > 0)
      toInsert->next = this->head;
    this->head = toInsert;
  }
```

```
  else // insert it somewhere in the list
  {
    if(index > this->count) // fail-safe
      index = this->count;
    Job* temp = this->head; // find the place
    for(int i = 0; i < index - 1; i++)
      temp = temp->next;
    toInsert->next = temp->next; // insertion
    temp->next = toInsert;
  }
  this->count++;
  return true;
}
```

# JobLinkedList::remove()

```
Job JobLinkedList::remove(int index)
{
  Job toRemove;
  if(index < 0 || this->count == 0)
    return toRemove; // return an empty job
  else if(index <= 1)
  {
    toRemove = *(this->head); // return the head
    Job* temp = this->head; // removal
    this->head = temp->next;
    delete temp;
  }
```

```
  else
  {
    Job* temp = head; // find the place
    for(int i = 0; i < index - 2; i++)
      temp = temp->next;
    Job* tempNext = temp->next; // removal
    temp->next = tempNext->next;
    toRemove = *tempNext; // return this one
    delete tempNext;
  }
  this->count--;
  toRemove.next = NULL;
  return toRemove;
}
```

# Remarks

- Common errors:
  - If a `Job` pointer `job` is `NULL`, then accessing `job->next` generates a run-time error. Set `next` to `NULL` to "create" run-time errors.
- In general, a list is a **linear data structure**. It stores multiple "nodes", which is another elementary data structure.
  - In a linked list, each node contains **a pointer for the next node**.
  - Because a job linked list "**has a**" job, we make job linked list as job's friend.
- For our `JobLinkedList`:
  - There is no limit on the number of nodes stored.
  - Spaces are saved by using dynamic memory allocation.
  - Efficiency is roughly the same as `JobList`: Insertion and removal are $O(n)$.

# Encapsulation

- We implemented two lists:
  - **`JobList`**: using an array.
  - **`JobLinkedList`**: using pointers.
- Though the private storages are different, the **public interfaces** are identical!

```
JobLinkedList(); // or JobList();
int getCount();
bool insert(Job job, int index);
Job remove(int index);
void print();
```

  - One **uses** these two classes in the same way.
  - Except for **`JobList`** there is a limit on the number of jobs.

# Encapsulation

- One does not need to (also should not) know how the list is implemented.
- One should just know **how to use it**.
- What if I can see and access the array in **`JobList`**?
  - I may write codes to access the array **directly**: The data structure is not safe.
  - In the future if the implementation of **`JobList`** is modified, I may also need to modify my codes (even if the interfaces all remain the same).

# Destructors

- If **dynamic memory allocation** is implemented, we need to release those dynamically-allocated spaces by the delete statement.
- Consider this main function:

```
int main()
{
  JobLinkedList jll;
  Job j1("j1", 1), j2("j2", 2), j3("j3", 3);
  // memory spaces are allocated statically
  jll.insert(j1);
  jll.insert(j2);
  jll.insert(j3);
  // 3 new statements are executed
  return 0;
} // no delete statement is executed!
  // a destructor is useful in this case
```

# JobLinkedList::~JobLinkedList()

```
JobLinkedList::~JobLinkedList() // version 1
{
  Job* temp = this->head;
  Job* tempNext = NULL;
  // Do not write "Job* tempNext = this->head->next;"
  // If we do so, what happens on an empty list?

  for(int i = 0; i < this->count; i++)
  {
    tempNext = temp->next;
    delete temp; // release memory
    temp = tempNext;
  }
}
```

```
JobLinkedList::~JobLinkedList() // version 2
{
  while(this->count > 0)
    this->remove(0); // release memory
}
```

```
JobLinkedList::~JobLinkedList() // version 3
{
  for(int i = 0; i < this->count; i++)
    this->remove(0);
}
// is this OK?
```

# Good Programming style

- Be very careful when using pointers.
- Write your codes slowly and as clear as possible.
  - Compile and test your program whenever you complete a function!
- When there is a run-time error, check whether you are accessing a **`NULL`** pointer.
- Check whether you need a destructor (or a copy constructor or an assignment operator) when your class has a pointer member.

# Outline

- Basic ideas
- Lists: **`class JobList`**
- Linked lists: **`JobLinkedList`**
- **More data structures**

# Stacks and queues

- A **stack** is a special list. A **queue** is another special list.
- Nodes cannot be inserted/removed at any place we want.
  - Stack: last-in-first-out (**LIFO**). A node can be inserted and removed only at the **top** of the stack.
  - Queue: first-in-first-out (**FIFO**). A node can be inserted only at the **tail** and removed only at the **head**.
- Many real-life situations can be modeled as stacks or queues.
  - The poker game solitaire; the Hanoi tower; function calls in your programs; calculators; graph traversal: depth-first search (DFS).
  - Consumer waiting lines; FIFO job scheduling; topological sorting; graph traversal: Breadth-first search (BFS).

# Creating a job stack by inheritance

- Though not realistic, we will implement a job stack.
  - The implementation of a job queue is left to you.
- This example shows
  - The application of **inheritance**: Once you have a list, it is very easy to create a stack or a queue.
  - The application of **encapsulation**: The idea of interfaces.
  - The application of **protected inheritance**: Not all public members of the parent class should be public for the child class.

# JobStack

```
class JobStack : protected JobLinkedList
// protected: we want to hide insert()
// and remove() inherited from JobLinkedList
{
public:
  JobStack();
  ~JobStack();
  void push(Job job);
  Job pop();
  void print();
};
JobStack::JobStack() : JobLinkedList() {}
JobStack::~JobStack() {}
```

```
// You need print() due to protected inheritance
void JobStack::print()
{
  JobLinkedList::print();
}
// insert at top (end)
void JobStack::push(Job job)
{
  JobLinkedList::insert(job, this->count);
}

// remove the one at top (end)
Job JobStack::pop()
{
  return JobLinkedList::remove(this->count);
}
```

# Remarks

- The class `JobStack` is indeed a stack. It is **safe and effective**.
- However, it is **not very efficient**.
  - Operations are executed through another class.
  - `push()` and `pop()` are both $O(n)$.
  - With `Job* tail` (as a new instance variable), they can be both $O(1)$.
- Be careful that `insert()` and `remove()` should be **hided**.
  - If you use public inheritance, you may override them.
- Inheriting `JobList` also creates a safe and effective job stack.

# Trees

- A list, stack, or queue is a linear (one-dimensional) data structure.
- A **tree** is a **two-dimensional** data structure.
- A **binary tree** is a useful two-dimensional data structure.

```
class BTNode
{
private;
  BTNode* left;
  BTNode* right;
  // …
}
```