UFUG 2601
C++ Programming

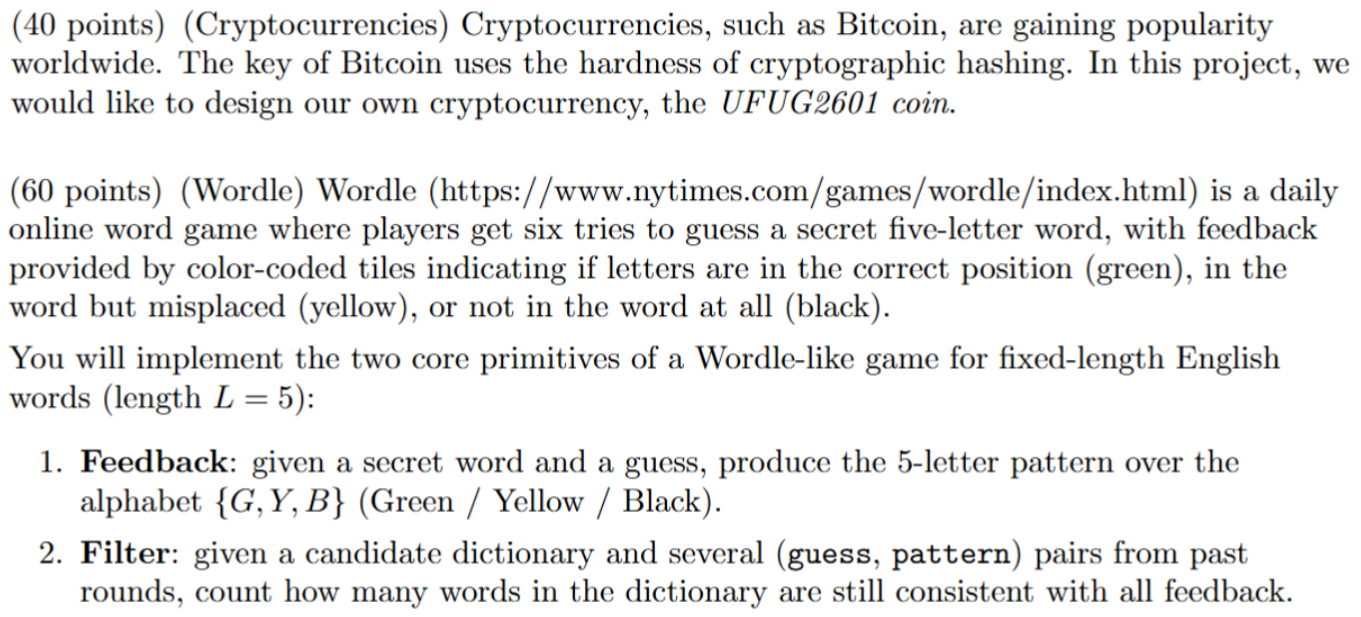
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Midterm Exam

- Date: October 26 (Sun.) 15:00 – 17:00
- Location: Lecture Hall A
- Exam Coverage: All topics up to and including Struct and Class
  - We mainly assess the understanding of C++ code and will not cover other topics (e.g., the history of programming language)
- Sample questions have been provided in canvas

Programming Assignment

- Deadline: Nov. 2nd, 23:59
  - The detailed document and OJ contest will open this week



(40 points) (Cryptocurrencies) Cryptocurrencies, such as Bitcoin, are gaining popularity worldwide. The key of Bitcoin uses the hardness of cryptographic hashing. In this project, we would like to design our own cryptocurrency, the *UFUG2601 coin*.

(60 points) (Wordle) Wordle (https://www.nytimes.com/games/wordle/index.html) is a daily online word game where players get six tries to guess a secret five-letter word, with feedback provided by color-coded tiles indicating if letters are in the correct position (green), in the word but misplaced (yellow), or not in the word at all (black).

You will implement the two core primitives of a Wordle-like game for fixed-length English words (length $L = 5$):

1. **Feedback**: given a secret word and a guess, produce the 5-letter pattern over the alphabet $\{G, Y, B\}$ (Green / Yellow / Black).
2. **Filter**: given a candidate dictionary and several (`guess`, `pattern`) pairs from past rounds, count how many words in the dictionary are still consistent with all feedback.

Feedback

Recursion这节课太难了😢，完全跟不上，好多语句之前没见过，比如for(auto val : v) cin.get cin.peek之类的，加上本身英文不太好，整节课竭尽全力无法跟上😢

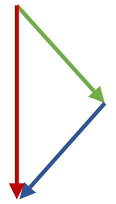
Pace: The current pace of the course feels quite rapid. Slowing down slightly would greatly improve our ability to solidify understanding before moving to new, complex topics.

Unfamiliar Code: A significant challenge is that code examples and assignments often include concepts or syntax that haven't been covered in the lectures. This creates a gap where we are expected to know things that haven't been taught, leading to unnecessary confusion and extra time spent on self-learning.

Suggestion: It would be very helpful if all key concepts and code elements used in assignments were introduced first. A brief overview or reference to these "assumed knowledge" items would make the learning process much smoother

Recap

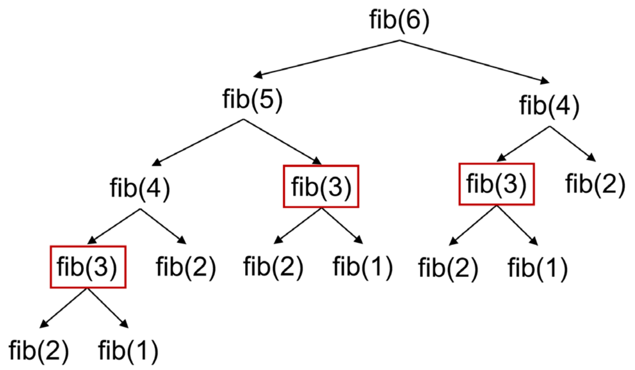
- How to ensure the correctness of a recursion
  - Simplify a complex problem into a simpler problem
  - Handle the simplest computations directly (base cases)
- How to write a recursive function
  - Consider various ways for simplifying inputs
  - Combine solutions with simpler inputs to a solution of the original problem
  - Find solutions to the simplest inputs
  - Implement the solution by combining the simple cases and the reduction step
- How to debug a recursive function
  - Print message when entering/leaving the recursive function
  - Use debugger: Step into / Step out / Step over / Call stack



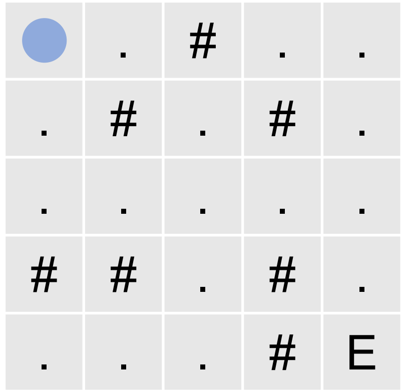
Recap

- How to analyze and improve the efficiency
  - Top-down memoization
  - Bottom-up tabulation
- **Backtracking** in recursion

```
void solve(int step) {
    if (step == n) { // Found a valid solution
        processSolution();
        return;
    }
    for (auto candidate : candidates) {
        if (isValid(candidate)) {
            makeMove(candidate);
            solve(step + 1);
            undoMove(candidate); // Backtrack
        }
    }
}
```



Maze Shortest Path - Recursion



●	.	#	.	.
.	#	.	#	.
.	.	.	.	.
#	#	.	#	.
.	.	.	#	E

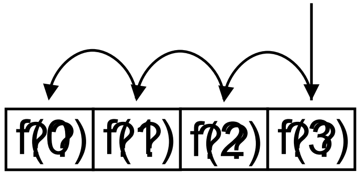
```
int dist(int i, int j)
// return the min distance from (i,j) to 'E'
```

What is the answer? `dist(s_i, s_j)`

What is the base cases?

If it is E, return 0

If it is wall or out of range, return Inf



How to combine solutions with simpler inputs to a solution of the original problem?

```
dist(i, j) = min(dist(i+1,j),dist(i-1,j),dist(i,j+1),dist(i,j-1)) + 1
```

Maze Shortest Path - Recursion

```cpp
int dist(int i, int j) { // return the min distance from (i,j) to 'E'
    if (i < 0 || j < 0 || i >= N || j >= M) return 99999;
    if (map[i][j] == 'E') return 0; // found a path
    if (map[i][j] == '#') return 99999;
    int down = dist(i + 1, j);
    int up = dist(i - 1, j);
    int right = dist(i, j + 1);
    int left = dist(i, j - 1);
    int min_path = std::min({down, up, right, left});
    if (min_path == 99999) return 99999; // no valid path found
    return min_path + 1; // add current step
}
```

Can this code work as expected?

Do we need to consider `dist(0,0)`?

How to avoid this?

S	.	E

$$\begin{tikzcd}
 & & & & \text{dist(0,0)} \arrow[r, dash] & \text{......} \\
 & & \text{dist(0,0)} \arrow[r, dash] & \text{dist(1,0)} \arrow[ur, dash] \arrow[dr, dash] & & \\
\text{dist(0,0)} \arrow[r, dash] & \text{dist(1,0)} \arrow[ur, dash] \arrow[dr, dash] & & & \text{dist(2,0)} & \\
 & & \text{dist(2,0)} \arrow[r, dash] & 0 & &
\end{tikzcd}$$

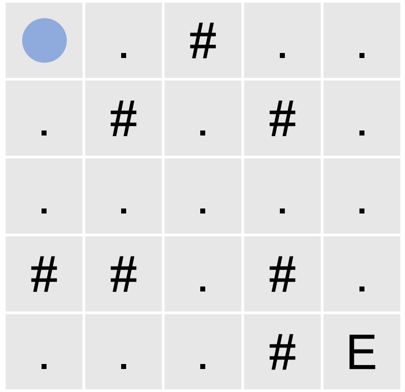
Infinite recursion!

Maze Shortest Path - Recursion

Use `visited[i][j]` to avoid infinite recursion

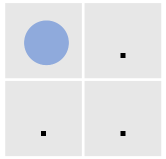
```
int dist(int i, int j) { // return the min distance from (i,j) to 'E'
    if (i < 0 || j < 0 || i >= N || j >= M) return 99999;
    if (map[i][j] == 'E') return 0; // found a path
    if (map[i][j] == '#' || visited[i][j]) return 99999;
    visited[i][j] = true;
    int down = dist(i + 1, j);
    int up = dist(i - 1, j);
    int right = dist(i, j + 1);
    int left = dist(i, j - 1);
    visited[i][j] = false;
    int min_path = std::min({down, up, right, left});
    if (min_path == 99999) return 99999; // no valid path found
    return min_path + 1; // add current step
}
```

Maze Shortest Path-DFS



Assume we try four directions (right, down, left, up) one by one

```
void dfs(int i, int j) { // from (i, j)
    if (map[i][j] == 'E') // finished
    // try four directions
    if (can_move_to(i + 1, j)) dfs(i + 1, j);
    if (can_move_to(i - 1, j)) dfs(i - 1, j);
    if (can_move_to(i, j + 1)) dfs(i, j + 1);
    if (can_move_to(i, j - 1)) dfs(i, j - 1);
}
```



- How to implement `can_move_to`?
- How to record the length of the path?
  - `int ans = 99999;int len = 0;`
  - Update `ans` when find a path, update `len` when enter and exit
- How to avoid loop?
  - Only consider three directions, don’t go back?
  - Don’t visit previous tiles!

Maze Shortest Path

```
bool can_move_to(int i, int j) {
    if (i < 0 || j < 0 || i >= N || j >= M) return false;
    if (map[i][j] == '#' || visited[i][j]) return false;
    return true;
}
void dfs(int i, int j) { // stand on (i, j)
    if (map[i][j] == 'E') { // found a path
        if (len < ans) ans = len; // found a shorter path
        return;
    }
    if (map[i][j] == '#' || visited[i][j]) return;
    visited[i][j] = true;
    if (can_move_to(i + 1, j)) { len++; dfs(i + 1, j); len--; }
    if (can_move_to(i - 1, j)) { len++; dfs(i - 1, j); len--; }
    if (can_move_to(i, j + 1)) { len++; dfs(i, j + 1); len--; }
    if (can_move_to(i, j - 1)) { len++; dfs(i, j - 1); len--; }
    visited[i][j] = false;
}
```

```
void dfs(int i, int j) {
    if (map[i][j] == 'E') {
        if (len < ans) ans = len;
        return;
    }
    if (map[i][j] == '#' ||
visited[i][j]) return;
    visited[i][j] = true;
    if (can_move_to(i + 1, j))
{ len++; dfs(i + 1, j); len--; }
    if (can_move_to(i - 1, j))
{ len++; dfs(i - 1, j); len--; }
    if (can_move_to(i, j + 1))
{ len++; dfs(i, j + 1); len--; }
    if (can_move_to(i, j - 1))
{ len++; dfs(i, j - 1); len--; }
    visited[i][j] = false;
}
```

```
void solve(int step) {
    if (step == n) { // Found a
valid solution
        processSolution();
        return;
    }
    for (auto candidate :
candidates) {
        if (isValid(candidate)) {
            makeMove(candidate);
            solve(step + 1);
            undoMove(candidate); //
Backtrack
        }
    }
}
```

Maze Shortest Path

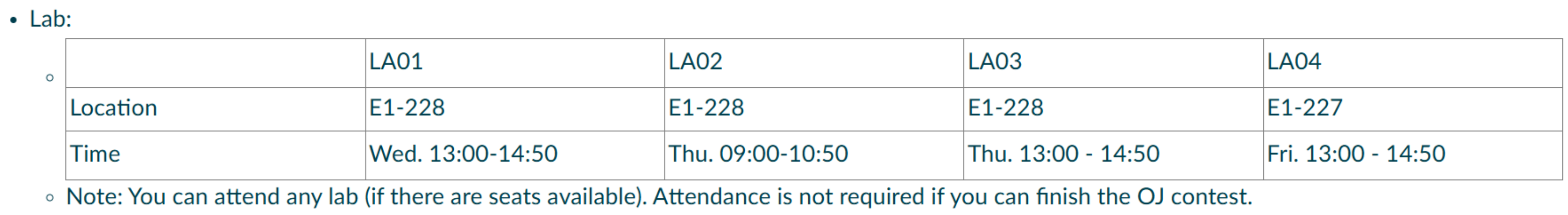
Move then test

●	.	#	.	.
.	#	.	#	.
.	.	.	.	.
#	#	.	#	.
.	.	.	#	E

```
void dfs(int i, int j) { // try to stand on (i, j)
    if (i < 0 || j < 0 || i >= N || j >= M) return;
    if (map[i][j] == 'E') { // found a path
        if (len < ans) ans = len; // found a shorter path
        return;
    }
    if (map[i][j] == '#' || visited[i][j]) return;
    visited[i][j] = true;
    len++;
    dfs(i + 1, j);
    dfs(i - 1, j);
    dfs(i, j + 1);
    dfs(i, j - 1);
    len--;
    visited[i][j] = false;
}
```

Maze Shortest Path

- All the questions we provided won’t require concepts you havn’t learnt (e.g., BFS, DFS)
- If you still have any question, please feel free to reach us during office hour.



- Lab:
  -

	LA01	LA02	LA03	LA04
Location	E1-228	E1-228	E1-228	E1-227
Time	Wed. 13:00-14:50	Thu. 09:00-10:50	Thu. 13:00 - 14:50	Fri. 13:00 - 14:50

  - Note: You can attend any lab (if there are seats available). Attendance is not required if you can finish the OJ contest.

Struct & Class

Goal of This Lecture

- To be able to implement your own classes
- To master the separation of interface and implementation
- To understand the concept of encapsulation
- To understand object construction
- To design and implement accessor and mutator member functions

Struct

- In C++, `struct` and `class` is almost functionally equivalent
- The only difference is that the default accessibility of `struct` is public, whereas in `class` the default is private
- We only need to learn `class`

Motivation

- We may have a number of related variables that all refer to the same concept
  - For an array, we (usually) need an accompany variable to store how many elements are actually stored
  - For each student, we have its name and id
- How can we sort students based on id?

Motivation

```
void bubble_sort(vector<string>& names, vector<int>& ids) {
    int n = names.size();
    for (int i = 0; i < n - 1; i++) { // n-1 passes
        for (int j = 0; j < n-i-1; j++) { // last i elements are sorted
            if (ids[j] > ids[j+1]) { // swap if out of order
                std::swap(names[j], names[j + 1]);
                std::swap(ids[j], ids[j + 1]);
            }
        }
    }
}
```

- What is the potential issues of this code?
  - What if we have more attributes of students
  - What if the `names` and `ids` are mismatched?

Motivation

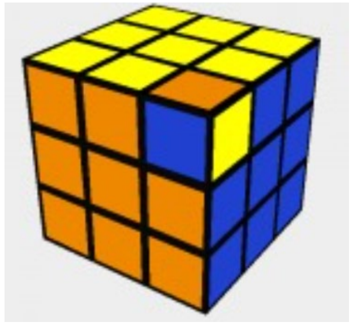
Each `Student` has its own name and id

```
void bubble_sort(vector<Student>& stu) {
    int n = names.size();
    for (int i = 0; i < n - 1; i++) { // n-1 passes
        for (int j = 0; j < n-i-1; j++) { // last i elements are sorted
            if (stu[j].id > stu[j+1].id) { // swap if out of order
                std::swap(stu[j], stu[j+1]);
            }
        }
    }
}
```

Group related variables together

Motivation

- Sometimes we don’t want to directly modify the value from outside



- You should not re-color a cubic, or just rotate a corner piece
- You can only do a set of feasible modifications, which ensures that the cubic is always in a valid state
  - Rotating upper/down/left/right/front/back side

enforcement of data constraints and invariants through controlled access and validation

Class

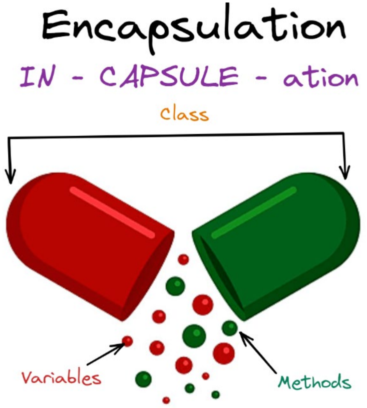
- A user-defined datatype which groups together related pieces of information

```
class Point2D {
    double x;
    double y;
};
```

```
class Product {
public:
    string name;
    double price;
    int score;
};
```

- It allows a user to **encapsulate** data and functionality using member variables and member functions

```
Product product_1, product_2;
```



Objects (Instances)

- An object is an occurrence of a class
- Different objects can have their own set of values in their fields
- If you want to represent 2 different students who have different names and IDs, you use 2 objects of `Student`
- Access the members using dot

```
Student stu_1, stu_2;
stu_1.name = "Alice";
stu_2.name = "Bob";
stu_1.print();
```

Interface

- The interface consists of all member functions we want to apply to objects of that type
- The interface is specified in the class definition
- Example: Product class member functions:
  - Make a new product
  - Read in a product object
  - Compare two products and find out which one is better
  - Print a product

Interface

- Classes are broken into public and private sections
- Public: members are accessible from outside the class
- Private: members are not accessible from outside the class
- Protected: introduced in later lectures

```
class Product {
public:
    Product();
    void read();
    bool is_better_than(const
Product &b) const;
    void print() const;
private:
    string name;
    double price;
    int score;
};
```

Default accessibility of `struct` is public, whereas in `class` the default is private

Constructor

- The first member function `Product()` is a constructor: the function that is used to initialize new objects
- We have used some of them in previous lectures

```
vector<int> firstV(4); // four ints with default value 0
vector<int> secondV(4, 10); // four ints with 10
vector<int> thirdV(secondV); // copy of secondV
```

- We can implement our constructor in class definition or later

```
class Product {
public:
    Product() {
        name = "Unknown";
        price = 0;
        score = 0;
    }
    // ...
};
```

```
class Product {
public:
    Product();
    // ...
};
Product::Product() {
    name = "Unknown";
    price = 0;
    score = 0;
}
```

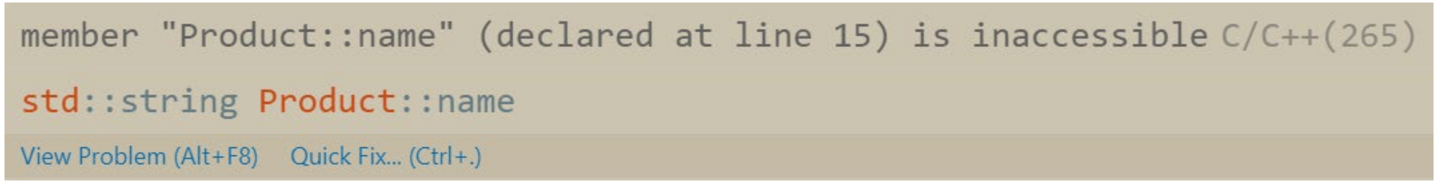
Other Member Functions

- The member function `void read();` is a mutator function, i.e., an operation that modifies the object
- The member functions `bool is_better_than(const Product &b) const;` and `void print() const;` are accessor functions, i.e., functions that query the object for some information without changing it
  - The keyword `const` indicates that the member function will not modify the object

Encapsulation

- The data fields are usually defined in the private sections of the class definition
- Only member functions can access and modify private data
- Hiding data (and unnecessary details) is called encapsulation

```
int main() {
    // Error! Use print() instead
    cout << bestProduct.name;
}
```



```
member "Product::name" (declared at line 15) is inaccessible C/C++(265)

std::string Product::name
```

Encapsulation

- Encapsulation forces indirect access to the data, guaranteeing that it cannot accidentally be put in an incorrect state

```
class Time {
public:
    Time();
    Time(int hrs, int min, int sec);
    void add_seconds(int s);
    int get_seconds() const;
    int get_minutes() const;
    int get_hours() const;
    int seconds_from() const;
private:
    int time_in_secs;
};
```

Encapsulation

- Bad example

```
Time liftoff(19, 30, 0);
// liftoff is delayed by six hours
// won't compile, but suppose it did
liftoff.hours = liftoff.hours + 6;
```

- Does this mean liftoff is at 25:30:00?
- In `add_seconds`, function knows about the length of a day and always does a correct calculation and produces a valid result

```
liftoff.add_seconds(6 * 60 * 60);
```

Member Functions Implementations

- Each member function declared in the class interface must be implemented in class definition or later

```
class Product {
public:
    Product();
    void read();
// ……
private:
    string name;
    double price;
    int score;
};
```

```
void Product::read() {
    cout << "Please enter the model name: ";
    getline(cin, name);
    cout << "Please enter the price: ";
    cin >> price;
    cout << "Please enter the score: ";
    cin >> score;
}
```

- A member function is called using dot `product_1.read();`
- The member function operates on the specific object that calls it

Const Member Functions

- When the keyword `const` is used in the declaration, it must also be placed in the member function definition

```
class Product {
public:
    // ……
    void print() const;
private:
    // ……
};
```

```
void Product::print() const {
    cout << name
    << " Price: " << price
    << " Score: " << score;
}
```

- The keyword `const` indicates that the member function will not modify the object that called it

"This" Pointer

```
void Product::print() const {
    cout << name
    << " Price: " << price
    << " Score: " << score;
}
```

equivalent

```
void Product::print() const {
    cout << this->name
    << " Price: " << this->price
    << " Score: " << this->score;
}
```

- `ptr->mem` is equivalent to `(*ptr).mem`
- `this` is a special pointer to the object that called it
- It might be useful if you have local variable of the same name, or return a reference to the current object `return *this;`

```
Point2D::Point2D(int x, int y) {
    this->x = x;
    this->y = y;
}
```

Default Member Functions

- Compiler will provide following member functions by default
  - Default Constructor
  - Copy Constructor
  - Assignment Operator
  - Move Constructor
  - Move Assignment Operator
  - Destructor

Constructors

- Constructor is used to construct an object
- Constructors are different from other normal member functions
  - It has no return type
  - The constructor is invoked only when an object is first created
- You cannot call the constructor to reset an object

```
Point2D p(2, 3);
p.Point2D(3, 5); // Error
```

- Just construct a new object and overwrite it

```
p = Point2D(3, 5) // Ok
```

Constructors

- Constructors is usually used to initialize member variables when an object is created, which has two styles

Member assignment

```
Point2D::Point2D(int x, int y) {
    this->x = x;
    this->y = y;
}
```

The member variables are first default-constructed (without arguments), and then assigned new values inside the constructor body.

Similar to

```
int x;
x = 3;
```

More recommended: Member initialization

```
Point2D::Point2D(int x, int y) :
x(x), y(y) {}
```

Directly initialize member variables before the constructor body runs.

Similar to

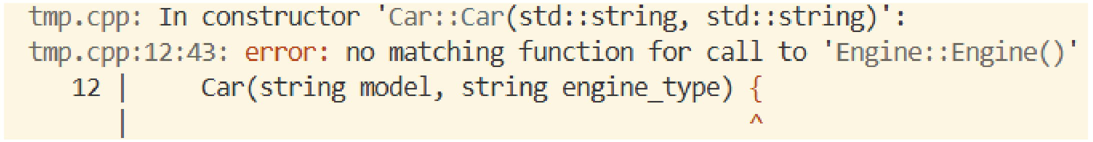
```
int x = 3;
```

But be careful that the order is not specified

```
Point2D::Point2D(int val) : x(val), y(x) {} // Don't
Point2D::Point2D(int val) : x(val), y(val) {} // Do this
```

Constructor

Can this code compile?



```
tmp.cpp: In constructor 'Car::Car(std::string, std::string)':
tmp.cpp:12:43: error: no matching function for call to 'Engine::Engine()'
   12 |     Car(string model, string engine_type) {
      |                                           ^
```

```cpp
class Engine {
public:
    Engine(string type) {
        this->type = type;
    }
    string type;
};
class Car {
public:
    Car(string model, string engine_type) {
        this->model = model;
        this->engine = Engine(engine_type);
    }
    string model;
    Engine engine;
};
```

```cpp
string model;
Engine engine;
model = model;
engine = Engine(engine_type);
```

```cpp
Engine(string type) : type(type) {};
Car(string model, string engine_type) :
model(model), engine(engine_type) {};
```

Default Constructor

- Default constructor is a constructor which can be called with no arguments
- If you do not provide any constructors, C++ will (try to) provide a trivial default constructor, which performs no action
  - If the member variables of the class are built-in types or classes with default constructors, all these member variables are default-constructed
  - If some members cannot be default-constructed, the default constructor cannot be provided (e.g., `Car`)

```cpp
class Point2D {
public:
    int x, y;
}
```

```cpp
int main() {
    Point2D p; // call trivial default constructor
    std::cout << p.x << ", " << p.y; // garbage value
}
```

Default Constructor

- If you have provided any constructor, no trivial default constructor will be generated

```
Point2D::Point2D(int x, int y) {
    this->x = x;
    this->y = y;
}
int main() {
    Point2D p; // Error: no default constructor
    std::cout << p.x << ", " << p.y;
}
```

Default Constructor

- You can manually provide a default constructor

```
Point2D(int x=0, int y=0);
```

or

```
Point2D() : x(0), y(0) {}
```

default argument is provided in the interface

```
Point2D::Point2D(int x, int y):
x(x), y(y) {}
```

Cannot provide default argument in the implementation

- But of course, you cannot use provide both of them. Why?
- If you still want compiler to generate a trivial one, use

```
Point2D() = default;
```

- If you want compiler NOT to generate a trivial one, use

```
Point2D() = delete;
```

Default Constructor

- What happen if we compile these two codes?

```
class Point2D{
public:
    Point2D(int x, int y);
    Point2D() = default;
    int x;
    int y;
};
Point2D::Point2D(int x, int y) {
    this->x = x;
    this->y = y;
}
int main() {
    Point2D p;
}
```

call the trivial one

```
class Point2D{
public:
    Point2D(int x = 0, int y = 0);
    Point2D() = default;
    int x;
    int y;
};
Point2D::Point2D(int x, int y) {
    this->x = x;
    this->y = y;
}
int main() {
    Point2D p;
}
```

call of overloaded ‘Point2D()’ is ambiguous

Copy Constructor & Assignment Operator

```
Point2D(const Point2D& rhs):
x(rhs.x), y(rhs.y) {
    cout << "Copy Constructor\n";
};
```

```
Point2D& operator=(const Point2D& rhs) {
    cout << "Copy assignment\n";
    if (this == &rhs) return *this;
    // self-assignment check
    x = rhs.x;
    y = rhs.y;
    return *this;
}
void foo(Point2D p);
void bar(Point2D& p)
```

- Which it will be called?
  - `Point2D p1(1, 2);`
  - `Point2D p2 = p1;`
  - `p2 = Point2D(2, 3);`
  - `foo(p1);`
  - `bar(p1);`

Constructor
Copy Constructor
Constructor & Copy Assignment
Copy Constructor
Nothing happen

Delegating Constructor

- A delegating constructor uses another constructor to do the initialization

```
class Point2D{
public:
    Point2D(int x, int y) :x(x), y(y) {}
    Point2D(): Point2D(0, 0){}
    Point2D(const Point2D& p): Point2D(p.x, p.y) {}
    int x;
    int y;
};
```

Destructor

- A function that will be called when the object is destroyed
  - Local variable is destroyed when a function exists
- `~Point2D() {}`
- It performs no action in most cases
  - You can add `cout` to help you understand when the object is destroyed
  - If you acquire some resources (e.g., memory, open file) when creating an object, you should release them in destructor (e.g, release memory, close file)
  - Resource acquisition is initialization - Wikipedia

List Initialization (Uniform initialization)

- Again, we can use list initialization to initialize a class

```
Point2D p(3, 5);
Point2D p{3, 5}; // Equivlent to the code above
vector<Point2D> vec {{3, 5}, {5, 7}};
```

- Recall why we say it is uniform

```
int a{3};
int arr[]{3, 5};
vector<int> vec{3, 5};
```

- Why this is not equivalent with `vector<int> vec(3, 5)`

List-initialization prefers constructors with a std::initializer_list argument

```
class A {
public:
    A() {cout << "A's default\n";}
    A(int x) : x(x) {cout << "A's non-default\n";}
    int x;
};
class B {
public:
    B() {cout << "B's default\n";}
    B(int x):a(x) {cout << "B's non-default\n";}
    A a;因为先执行记忆（A）
};
class C {
public:
    C() {cout << "C's default\n";}
    C(int x) {this->a = x; cout << "C's non-default\n";}
    A a;同理
```

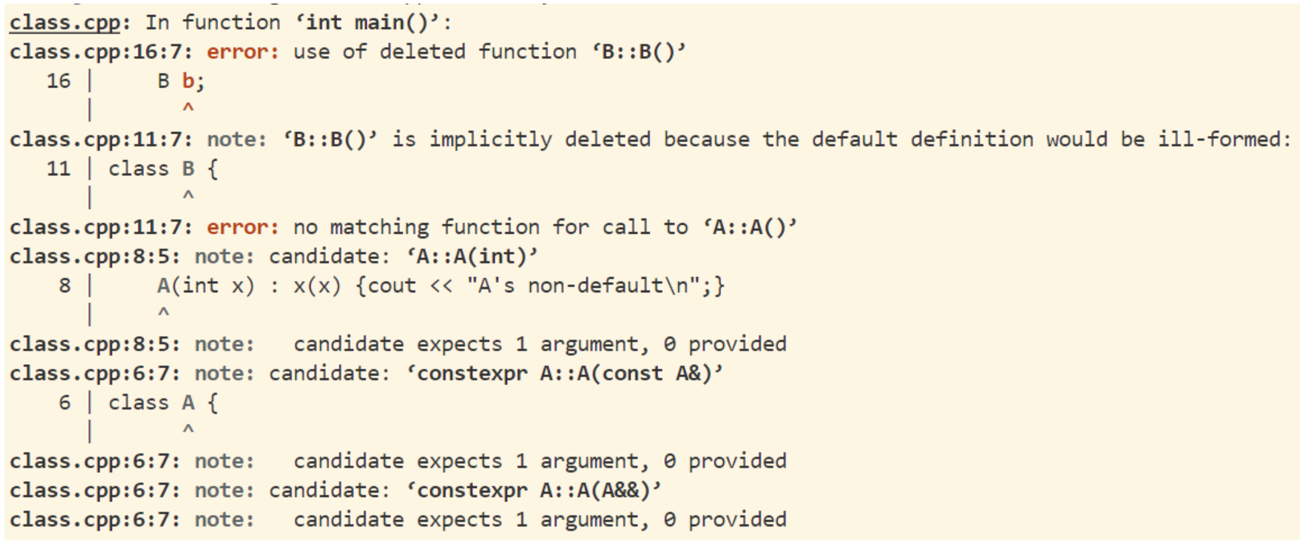
```
int main() {
    B b1;
    C c1;
    B b2(3);
    C c2(3);
}
```

```
A's default
B's default

A's default
C's default

A's non-default
B's non-default

A's default
A's non-default
C's non-default
```

```
class Point2D {
public:
    Point2D() : Point2D(0, 0) { cout << "Default constructor called\n"; }
    Point2D(int x, int y) : x(x), y(y) { cout << "Parameterized constructor
called\n"; }
    Point2D(const Point2D& other) : x(other.x), y(other.y) { cout << "Copy
constructor called\n"; }
    Point2D& operator=(const Point2D& other) {
        cout << "Copy assignment operator called\n";
        if (this == &other) return *this; // self-assignment check
        x = other.x;
        y = other.y;
        return *this;
    }
    ~Point2D() { cout << "Destructor called\n"; }
    void print() const {
        cout << "Point2D(" << x << ", " << y << ")\n";
    }
private:
    int x, y;
};
```

Try to play with this code to understand these functions

```cpp
class A {
public:
    A(int x) : x(x) {cout << "A's non-default\n";}
    int x;
};
class B {
public:
    A a;
};
int main() {
    B b;
}
```



```
class.cpp: In function ‘int main()’:
class.cpp:16:7: error: use of deleted function ‘B::B()’
   16 |     B b;
      |       ^
class.cpp:11:7: note: ‘B::B()’ is implicitly deleted because the default definition would be ill-formed:
   11 | class B {
      |       ^
class.cpp:11:7: error: no matching function for call to ‘A::A()’
class.cpp:8:5: note: candidate: ‘A::A(int)’
    8 |     A(int x) : x(x) {cout << "A's non-default\n";}
      |     ^
class.cpp:8:5: note:   candidate expects 1 argument, 0 provided
class.cpp:6:7: note: candidate: ‘constexpr A::A(const A&)’
    6 | class A {
      |       ^
class.cpp:6:7: note:   candidate expects 1 argument, 0 provided
class.cpp:6:7: note: candidate: ‘constexpr A::A(A&&)’
class.cpp:6:7: note:   candidate expects 1 argument, 0 provided
```

香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Converting Constructor

- If a constructor only accepts an argument, it provides an implicit conversion and thus called converting constructor

```
class Student {
public:
    Student(const string& name, int id=0):
name(name), id(id){}
    string name;
    int id;
};   string name = "Alice";
     Student s1 = name; // ok, string -> Student
     Student s2 = "Alice"; // error, two-step conversion:
 const char[6] -> string -> Student
     Student s3 = string("Alice"); // ok
```

Converting Constructor

- Sometimes the implicit conversion can be annoying
  - `vector<int>` can take an integer as parameter to construct a vector

```
vector<int> vec(3); // consturct a vector of size 3 with default values (0)
vector<int> vec2 = 3; // but you don’t want this conversion in this case
void foo(vector<int> vec); foo(3); // and this case
```

In certain situations, it is preferable to have an error generated rather than allowing an implicit conversion to occur

Converting Constructor

- Use `explicit` to prevent accidental implicit conversion

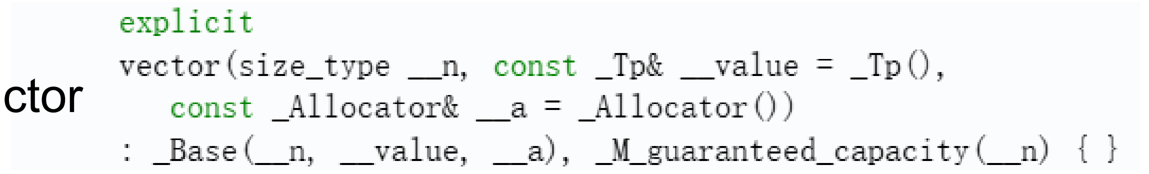
```
class Student {
public:
    explicit Student(const string&
name, int id=0): name(name), id(id){}
    string name;
    int id;
};
```

This prevents an implicit conversion from `string` to `Student`

```
Student s3 = string("Alice"); // error now
Student s4 = Student(string("Alice")); // work
Student s5 = Student("Alice"); // work or not?
```

✓

`vector<int>` uses explicit constructor



```
explicit
vector(size_type __n, const _Tp& __value = _Tp(),
    const _Allocator& __a = _Allocator())
: _Base(__n, __value, __a), _M_guaranteed_capacity(__n) { }
```

Accessing Data Fields

- Only member functions of a class are allowed to access the private data fields of objects of that class

```
void raise_salary(Employee &e, double percent) {
    e.salary = e.salary * (1 + percent / 100);
    // error: 'double Employer::salary' is private within this context
}
```

- Data fields must be accessed by accessor and mutator functions

```
void raise_salary(Employee &e, double percent) {
    double new_salary = e.get_salary() * (1 + percent / 100);
    e.set_salary(new_salary);
}
```

Accessing Data Fields

- Consider the previous nonmember function `raise_salary`

```
void raise_salary(Emplyee& e, double percent) {
    double new_salary = e.get_salary() * (1 + percent / 100);
    e.set_salary(new_salary);
}
```

It is kind of C-style

- versus the member function

```
void Employee::raise_salary(double percent) {
    salary = salary * (1 + percent / 100);
}
```

Without needing getters/setters
The function logically belongs inside the Employee class

Accessing Data Fields

- A nonmember function is called with two explicit parameters
  - `raise_salary(harry, 7);`
- A member function is called using the dot notation, with one explicit parameter
  - `harry.raise_salary(7);`
- A member function can invoke another member function on the implicit parameter without using the dot notation

```
void Employee::print() const {
    cout << "Name: " << get_name()
    << "Salary: " << get_salary();
}
```

We will see more difference between them after learning inheritance

Accessing Data Fields

- Not every data member needs accessor functions (the `Product` class did not have a `get_score()` function)
- Not every get function needs a matching set (the `Time` class can `get_minutes()` but not `set_minutes()`)
- Remember that implementation is supposed to be hidden – just because a class has member functions named get or set does not necessarily explain how the class is designed

```
Time::Time(int hour, int min, int sec) {
    time_in_secs = 60 * 60 * hour + 60 * min + sec;
}
int Time::get_minutes() const {
    return (time_in_secs / 60) % 60;
}
```

Static Member

- A static member belongs to the class, not an individual object

```
class Account {
public:
    Account(const string &name): name(name) {++account_cnt;}
    static int get_count() {return account_cnt;}
private:
    static int account_cnt;
    string name;
};
int Account::accountCnt = 0;
int main() {
    Account a("a"), b("b");
    cout << a.get_count() << b.get_count() << Account::get_count();
}
```

Operator Overloading

- We have overloaded `=`

```
Point2D& operator=(const Point2D& p) {
    this->x = p.x;
    this->y = p.y;
    cout << "Assignment\n";
    return *this;
}
```

- We can overload many operators
  - +,-,*,/,%
  - +=,-=,*=,/=,%=
  - ++,--
  - <,>
  - ….

Operator Overloading

`return-type operator op (arg)`

- To overload unary operator
  - `arg` is empty
  - Exception: ++,--
    - `Point2D& operator++();      // Prefix increment operator`
    - `Point2D operator++(int);    // Postfix increment operator`
- To overload binary operator
  - `arg` is the second operand (right-hand side, rhs)
  - e.g., `Point2D operator+(const Point2D& rhs);`

```
auto p3 = p1 + p2;
auto p4 = p1.operator+(p2); // equivalent
```

Operator Overloading

- What if the first operator is not the class we define?
  - e.g. we want to do something like `std::cout << p`
- Overload as non-member function

```
class Point2D {
public:
    Point2D(double x, double y):x(x),y(y) {}
    double x;
    double y;
};
ostream& operator<<(ostream& os, const Point2D& p) {
        os << '(' << p.x << ", " << p.y << ')';
        return os;
}
```

Friend Function / Friend Declaration

- To allow a function access private data

```
class Point2D {
public:
    Point2D(double x, double y):x(x),y(y) {}
    friend ostream& operator<<(ostream& os, const Point2D& p);
private:
    double x;
    double y;
};
ostream& operator<<(ostream& os, const Point2D& p) {
        os << '(' << p.x << ", " << p.y << ')';
        return os;
}
```

Exercise: polynomial_multiplication

https://onlinejudge.hkust-gz.edu.cn/problem/HW1-01

- The polynomial $x - 1$ is represented as degree 1 and the list of coefficients [1, -1].
- The polynomial $2x^2 + 1$ is represented as degree 2 and the list of coefficients [2, 0, 1].
- The polynomial $3x^4 - x + 1$ is represented as degree 4, and the list of coefficients [3, 0, 0, -1, 1].

Sample Input 1

```
2 2 0 1
4 3 0 0 0 1
```

Sample Output 1

```
6x^6+3x^4+2x^2+1
```

Sample Input 2

```
1 1 1
2 1 0 1
```

Sample Output 2

```
x^3+x^2+x+1
```

Sample Input 3

```
2 1 -2 0
3 1 0 1 1
```

Sample Output 3

```
x^5-2x^4+x^3-x^2-2x
```

Exercise

- Implement a class of `Polynomial` (with integer coefficients) so that it is closed under addition/multiplication

```
class Polynomial {
public:
    // what should I provide here?
private:
    // what should I store here?
};
```

Next Week

- Class and Structure (cont')

If you have any question or feedback…

Anonymous questionnaire

https://www.wjx.cn/vm/OPiwiXj.aspx

Thanks