CS2310 Computer Programming

LT11 Object Oriented Programming-I

Computer Science, City University of Hong Kong

Semester A 2023-24

Outline

- C-like struct
- Class and objects: basic concepts and syntax
- Constructors and destructors
- Access specifier: public, protect, and private

Struct: Definition

- A *composite data type* that groups a list of variables (possibly different types) under one name
- Variables are stored in a continuous memory areas
- Syntax and example:

```
struct typename {
        type1  member_var1;
        type2  member_var2;
        ...
};
```

```
struct StudentRecord {
        char   name[51];
        char   sid[9];
        float  GPA;
};
```

Initialization

- No memory is allocated when you *define* a struct
- When you declare a variable of a given struct type, enough memory is allocated for storing all struct members *contiguously*
- Example

```
StudentRecord danny = {"Danny", "50123456", 80};
```

Accessing Individual Members

- A member variable can be accessed with the use of the dot operator "." :

```
danny.quiz += 10;
```

- Two structure types can have the same member name:

```
CS2363Student peter;

cin >> peter.quiz;
```

Accessing Individual Members (cont'd)

- A member variable can be accessed with the use of the dot operator "."
  - `danny.gpa += 10;`
- Structure types can have the same member name without confliction

```
struct CS2310Student {
        char    sid[9];
        float   asg[3];
        float   lab[10];
        float   midterm;
        float   final;
};
```

```
struct CS6789Student {
        char    sid[9];
        float   asg[5];
        float   final;
};
```

```
CS2310Student peter;
cin >> peter.final;
CS6789Student danny;
cin >> danny.final;
```

Example

```
struct CS2310Student {
    int     sid;
    float   quiz;
    float   asg1;
    float   asg2;
};
```

```
int main() {
  CS2310Student student;
  cout << "Please enter your id, quiz, a1, and a2 marks\n";
  cin >> sr.id;
  cin >> sr.quiz;
  cin >> sr.asg1;
  cin >> sr.asg2;
  cout << sr.id << " cw:" << (sr.quiz+sr.asg1+sr.asg2)/3 << endl;
  return 0;
}
```

Struct Assignment

- You can assign structure values to a structure variable:

```
danny = kitty;
```

which is equivalent to:

```
danny.sid   = kitty.sid;
danny.quiz  = kitty.quiz;
danny.asg1  = kitty.asg1;
danny.asg2  = kitty.asg2;
```

```
struct CS2310Student {
    int     sid;
    float   quiz;
    float   asg1;
    float   asg2;
};
```

Pass/Return Structure to/from Function

- A function can have parameters of structure type:

```
double overall(CS2310Student s) {
        return (s.quiz + s.asg1 + s.asg2)/3;
}
```

- A function can return a value of structure type:

```
CS2310Student newStudent(int sid) {
        CS2310 stu; stu.sid=sid;
        return stu;
}
```

Hierarchical structures

- A member of a structure can be another structure:

```
struct Date {
    int month, day, year;
};

struct PersonInfo {
    double height, weight;
    Date   birthday;
};

PersonInfo peter;
peter.birthday.year=2001;
```

Struct Pointer

- Struct pointer stores the memory address of the first byte of a struct variable

```
Date d;
d.year = 2022;
d.month = 11;
d.day = 7;
Date *dPtr = &d;
```

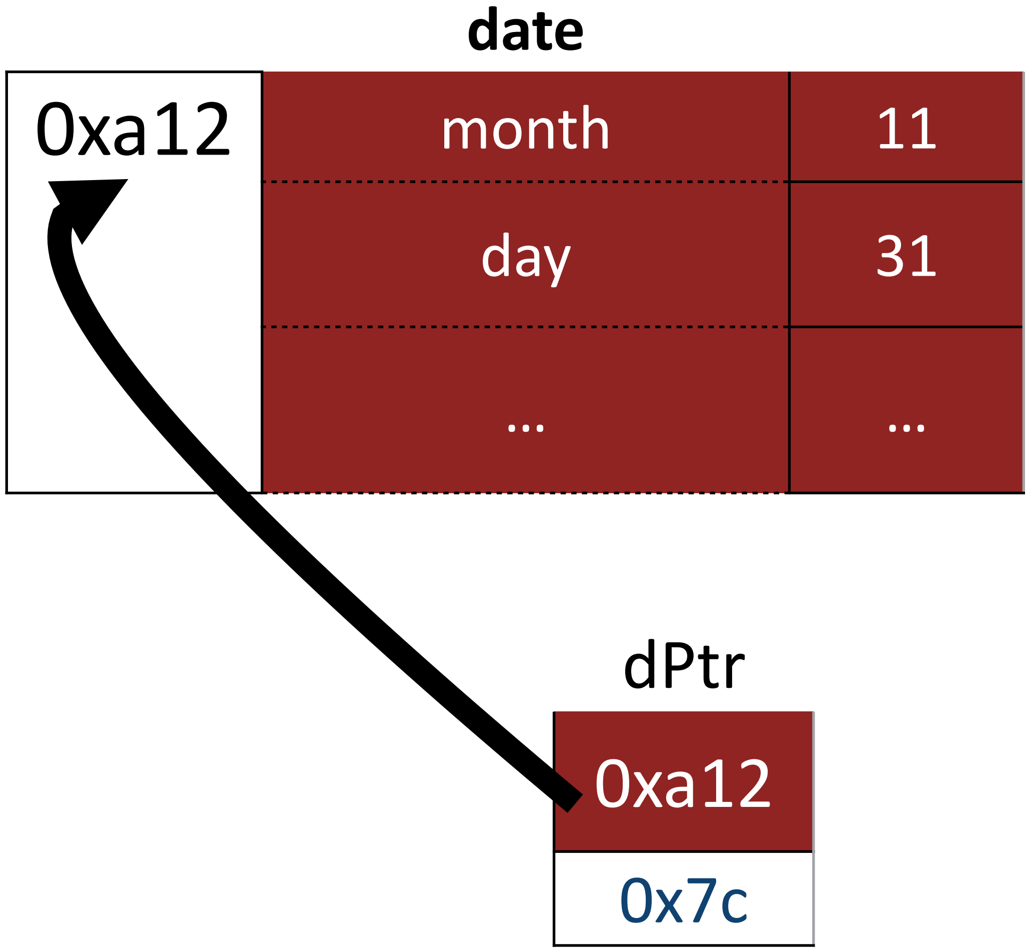
Address	Value	
0xa12	2022	d.year
0xa16	11	d.month
0xa1a	7	d.day
0x7c	0xa12	dPtr

Pointer and Arrow Syntax

- Pointers can point to a struct
- One way to do this would be to dereference and then use dot notation:

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << (*dPtr).month << endl;
```

- But, this notation is cumbersome, and the parenthesis are necessary because the "dot" has a higher precedence than the *.

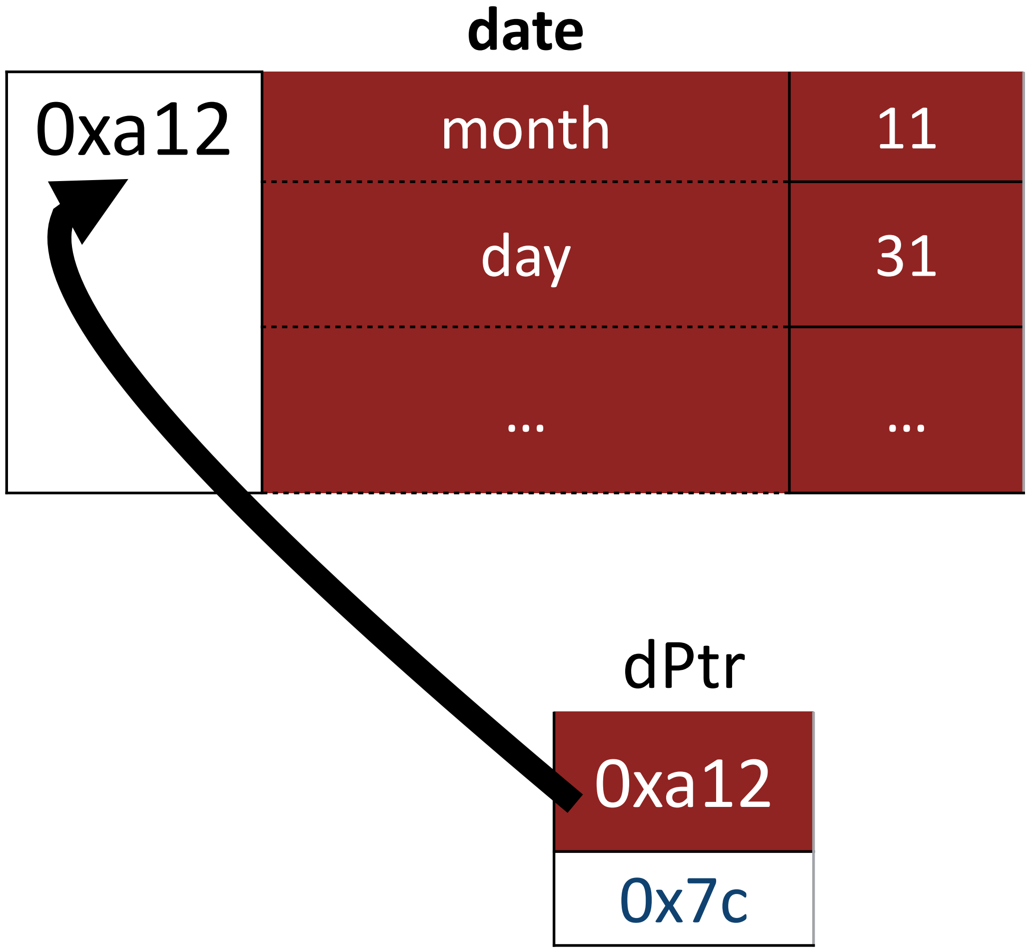


Pointer and Arrow Syntax

- So, we have a different, and more intuitive syntax, called the "arrow" syntax, -> :

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month << endl;
```

- The arrow syntax can be used to set a value as well



Dynamic Memory for Struct: new

```
// allocate an int, default initializer (do nothing)
int * p1 = new int;
// allocate an int, initialized to 0
int * p2 = new int();
// allocate an int, initialized to 5
int * p3 = new int(5);
// allocate an int, initialized to 0
int * p4 = new int{};  // C++11
// allocate an int, initialized to 5
int * p5 = new int {5};  // C++11

// allocate a Date struct variable, default initializer
Date * pd1 = new Date;
// allocate a Date struct variable, initialize the members
Date * pd2 = new Date {2023, 11, 30}; // C++11
```

Dynamic Memory for Struct: new[]

```
// allocate 16 int, default initializer (do nothing)
int * pa1 = new int[16];
// allocate 16 int, zero initialized
int * pa2 = new int[16]();
// allocate 16 int, zero initialized
int * pa3 = new int[16]{}; // C++11
// allocate 16 int, the first 3 element are initialized to 1,2,3, the rest 0
int * pa4 = new int[16]{1,2,3}; // C++11

// allocate memory for 16 Date objects, default initializer
Date * pda1 = new Date [16];
// allocate memory for 16 Date objects, the first two are explicitly initialized
Date * pda2 = new Date [16]{{2023, 11, 30}, {2023, 12, 1}}; // C++11
```

Dynamic Memory for Struct: delete/delete[]

```
// deallocate memory
delete p1;
// deallocate memory
delete pd1;

// deallocate the memory of initialized array
delete []pa1;

// deallocate the memory of uninitialized array, and call the destructors
of all the elements
delete []pda1;
```

We'll explore destructors when we study classes and objects.

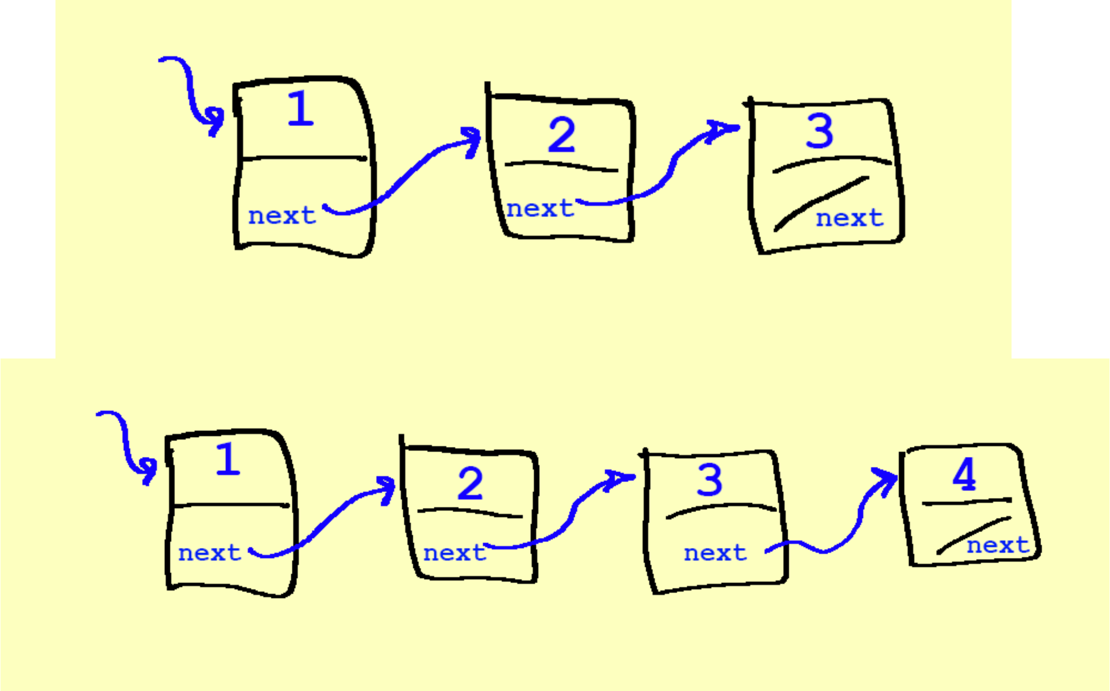
Function Parameter Passing

```
void printDateByValue(Date d) {
    cout << "By Value: " << d.year << "-" <<
d.month << "-" << d.day << endl;
}

void modifyDateByReference(Date &d) {
    d.year += 1; // Modify the year
    cout << "By Reference: " << d.year << "-"
<< d.month << "-" << d.day << endl;
}

void resetDateByPointer(Date *d) {
    d->year = 2000; // Reset the year
    d->month = 1;  // Reset the month
    d->day = 1;    // Reset the day
    cout << "By Pointer: " << d->year << "-" <<
d->month << "-" << d->day << endl;
}
```

```
int main() {
    Date d;
    d.year = 2023; d.month = 11; d.day = 7;
    // Pass by value
    printDateByValue(d);
    // Pass by reference
    modifyDateByReference(d);
    // Print modified date to show the change
    cout << "Modified Date: " << d.year << "-
" << d.month << "-" << d.day << endl;
    // Pass by pointer
    resetDateByPointer(&d);
    // Print reset date to show the change
    cout << "Reset Date: " << d.year << "-"
<< d.month << "-" << d.day << endl;
    return 0;
}
```

Struct and Pointer: Linked List

- A linked list is a chain of *nodes*
- Each node contains two pieces of information:
  - Some piece of data that is stored in the sequence
  - A *link* to the next node in the list
- We can *traverse* the list by starting at the first node and repeatedly following its link
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence



Struct and Pointer: Linked List

```cpp
struct Node {
    int data;
    Node* next;
};
```

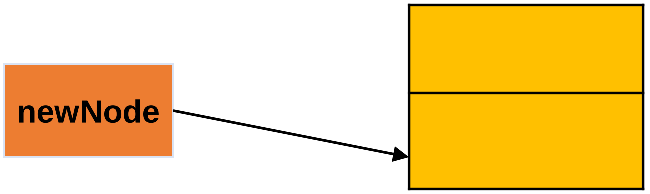
```cpp
// Function to print the linked list
void printList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " -> ";
        head = head->next;
    }
    cout << "NULL\n";
}
```

```cpp
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}
```

```cpp
void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

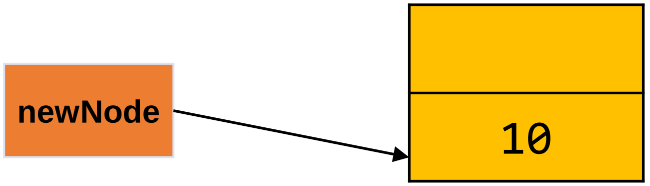
```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

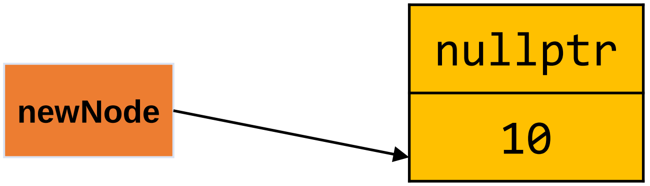
```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

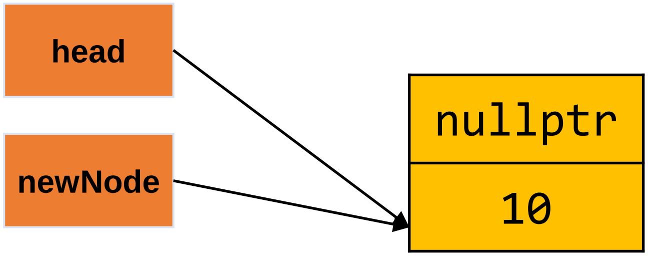
```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

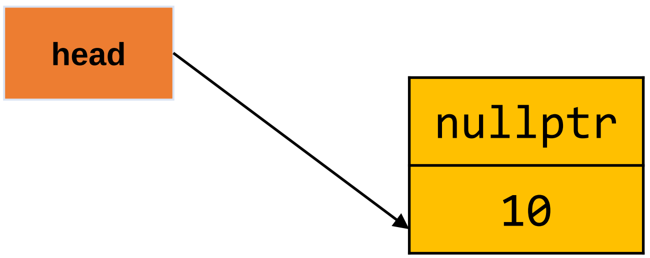
```cpp
struct Node {
    int data;
    Node* next;
};
```




```cpp
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

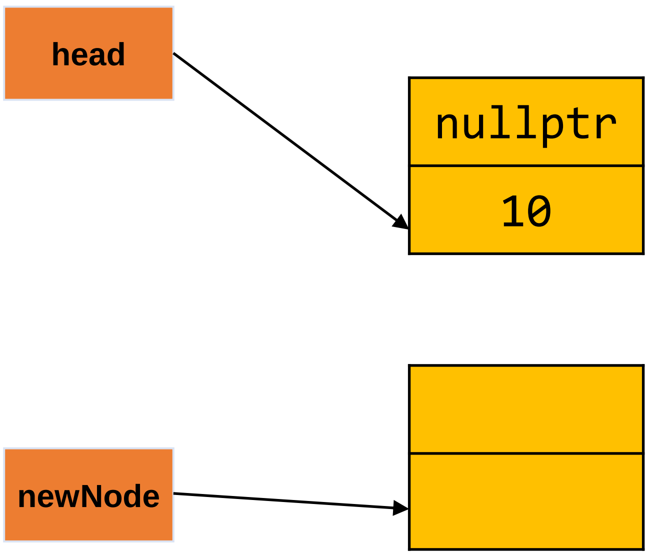
```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

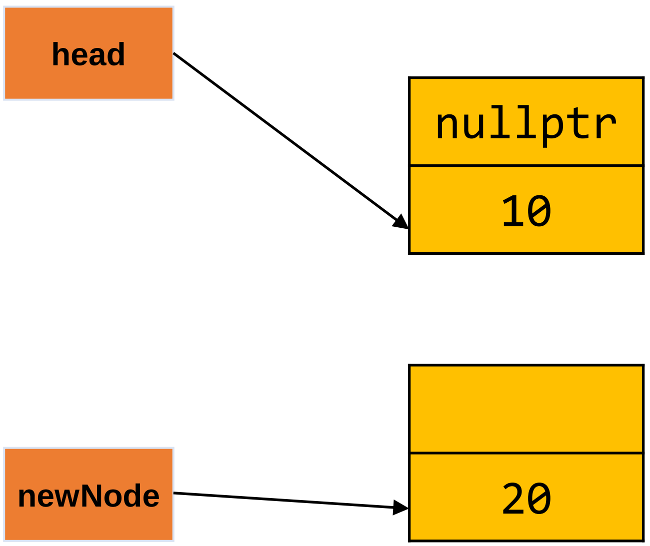
```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

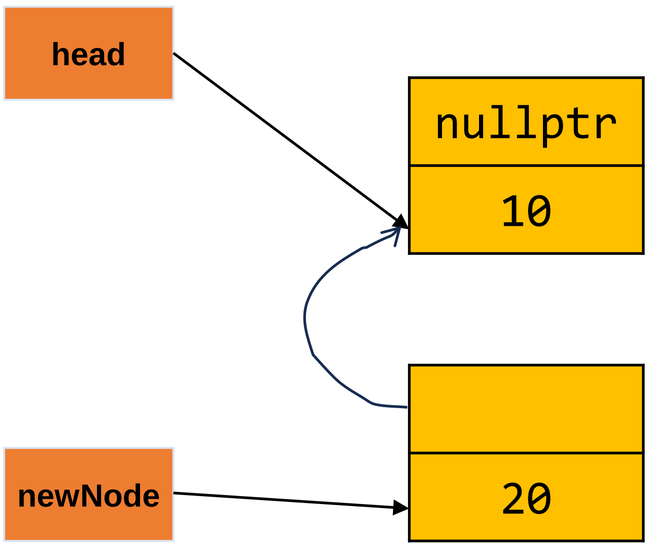
```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

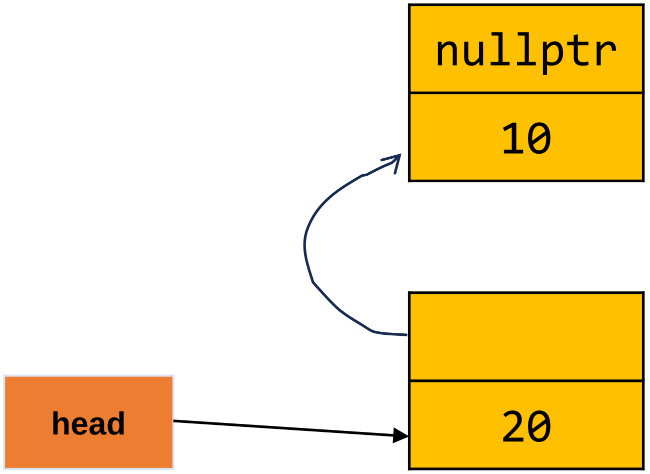
```cpp
struct Node {
    int data;
    Node* next;
};
```




```cpp
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Struct and Pointer: Linked List

```
struct Node {
    int data;
    Node* next;
};
```




```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

void main() {
    Node* head = nullptr; // Start with an empty list
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    cout << "Linked List after insertion: ";
    printList(head);
    // Free the remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Outline

- C-like struct
- Class and objects: basic concepts and syntax
- Constructors and destructors
- Access specifier: public, protect, and private

Class and Objects

- A *class* is a user-defined data type used as a template for creating *objects*
- For example
  - class: Politician objects: Trump, Biden, Obama
  - class: Country objects: China, India …
- A class typically contains:
  - *data fields*: member variables that describe object state (i.e., object attributes or properties)
  - *methods*: member functions that operate on the object (e.g., alter or access object state)

Example

```
char      *body_color;
char      *eye_color;
float      pos_x, pos_y;
float      orient;
float      powerLevel;
Camera     eye;
Speaker    mouth;
Mic        ear;
```

```
void  start();
void  shutdown();
void  moveForward(int step);
void  turnLeft(int degree);
void  turnRight(int degree);
void  listen(Audio *audio);
Audio speak(char *str);
```

class: Robot

Member variables

Member functions

```
Robot eve, wall_e;
eve.body_color ="White";
eve.eye_color = "Blue";
wall_e.body_color = "Yellow";
wall_e.eye_color = "Black";
…
```

Object-Oriented Programming (OOP)

- Conventional procedural programming:
  - A program is divided into small parts called functions
  - Focus on solving a problem step by step

- Object-oriented programming
  - A program is divided into objects, each contains data and functions that describe properties, attributes, and behaviours of the object
  - Focus on modelling object interactions in real-world
  - Code reuse, modularity and flexibility, efficient for large projects
  - However, it's not universally applicable to all problems

Define Classes

```
class Circle {
public: //  access specifier, introduced later
    float x, y, r;

    void setCenter() {
        cout << "Input center:\n";
        cin >> x >> y;
    }
    void setRadius() {
        cout << "Input radius:\n";
        cin >> r;
    }

    bool isWithin(float x0, float y0);
    float perimeter();
    float area();
};
```

"Circle.h"

```
bool Circle::isWithin(float x0, float y0) {
    return (x0-x)*(x0-x)+(y0-y)*(y0-y) < r*r;
}

float Circle::perimeter() {
    return 2*M_PI*r;
}

float Circle::area() {
    return M_PI*r*r;
}
```

"Circle.cpp"

Create and Access Objects

```
int main() {
   Circle a;
   a.setCenter(); a.setRadius();
   cout << "The perimeter of circle a is " << a.perimeter() << endl;

   Circle *b = new Circle();
   b->setCenter(); b->setRadius();
   cout << "The area of circle b is " << b->area() << endl;
   delete b;

   return 0;
};
```

`this` Pointer

- `this` keyword in C++ is *an implicit pointer that points to the object of which the member function is called*
- Every object has its own `this` pointer. Every object can reference itself by `this` pointer

```
class Circle {
public: // access specifier, introduced later
    float x, y, r;

    void setCenter(float x, float y) {
        this->x = x;
        this->y = y;
    }
    void setRadius(float r) {
        this->r = r;
    }
};
```

Pass Class Objects to Functions

- Pass-by-value: class state won't be modified after function call

```
class Student {
public:
    float avg_grade=0; int n_course=0;
    void updateCourse(int n) { this->n_course += n; }
    void updateAvgGrade(float avg_grade) {
        this->avg_grade = avg_grade;
    }
};
int main() {
    Student alice; int grade[3] = {90, 85, 95};
    inputCourseGrade(alice, grade, 3);
    cout << alice.n_course << " ";
    cout << alice.avg_grade << "\n";
    delete alice;
    return 0;
}
```

```
void inputCourseGrades(Student  stu, float grade[], int n) {
    float total =  stu.avg_grade*stu.n_course;
    for (int i = 0; i < n; i++)
        total += grade[i];
    float new_avg = total  / (stu.n_course+n);

    stu.updateAvgGrade(new_avg);
    stu.updateCourse(n);

    cout << stu.n_course;
    cout << " ";
    cout << stu.avg_grade;
    cout << "\n";
}
```

Pass Class Objects to Functions

- Pass-by-pointer

```
class Student {
public:
    float avg_grade=0; int n_course=0;
    void updateCourse(int n) { this->n_course += n; }
    void updateAvgGrade(float avg_grade) {
        this->avg_grade = avg_grade;
    }
};
int main() {
    Student alice; int grade[3] = {90, 85, 95};
    inputCourseGrade(&alice, grade, 3);
    cout << alice.n_course << " ";
    cout << alice.avg_grade << "\n";
    delete alice;
    return 0;
}
```

```
void inputCourseGrades(Student *stu, float grade[], int n) {
    float total =  stu->avg_grade*stu.n_course;
    for (int i = 0; i < n; i++)
        total += grade[i];
    float new_avg = total  / (stu->n_course+n);

    stu->updateAvgGrade(new_avg);
    stu->updateCourse(n);

    cout << stu->n_course;
    cout << " ";
    cout << stu->avg_grade;
    cout << "\n";
}
```

Pass Class Objects to Functions

- Pass-by-reference

```
class Student {
public:
    float avg_grade=0; int n_course=0;
    void updateCourse(int n) { this->n_course += n; }
    void updateAvgGrade(float avg_grade) {
        this->avg_grade = avg_grade;
    }
};
int main() {
    Student alice; int grade[3] = {90, 85, 95};
    inputCourseGrade(alice, grade, 3);
    cout << alice.n_course << " ";
    cout << alice.avg_grade << "\n";
    delete alice;
    return 0;
}
```

```
void inputCourseGrades(Student &stu, float grade[], int n) {
    float total =  stu.avg_grade*stu.n_course;
    for (int i = 0; i < n; i++)
        total += grade[i];
    float new_avg = total  / (stu.n_course+n);

    stu.updateAvgGrade(new_avg);
    stu.updateCourse(n);

    cout << stu.n_course;
    cout << " ";
    cout << stu.avg_grade;
    cout << "\n";
}
```

const Members

```
class Circle {
public: // access specifier, introduced later
    float x, y, r;
    const double PI = 3.1416;
    void setCenter() {
        cout << "Input center:\n";
        cin >> x >> y;
    }
    void setRadius() {
        cout << "Input radius:\n";
        cin >> r;
    }
    bool isWithin(float x0, float y0);
    float perimeter();
    float area();
};
```

```
float Circle::perimeter() const
{
    return 2*M_PI*r;
}

float Circle::area() const {
    return M_PI*r*r;
}
```

- `const` member variables behavior similar with normal const variables
- `const` member functions promise not to modify member variables.

`static` Members

```
class Date {
public:
    int year; int month; int day;
    // Keep track of number of Date objects created.
    static int objectCount;
    void createDate(int y, int m, int d) {
        year = y; month = m; day = d;
        objectCount++; // Increment the static variable each time this method is called
    }
    // Static member function to access static variable
    static int getObjectCount() {
        return objectCount; // Note: Static member functions can only access static variables
    }
};
// Static variable definition and initialization
int Date::objectCount = 0;
int main() {
    Date d1(2022, 11, 7); // Creates a Date object
    Date d2(2023, 1, 1);  // Creates another Date object
    // Accessing the static variable through the class name
    cout << "Total Date objects created: " << Date::getObjectCount() << endl;
    return 0;
}
```

`static` is associated with the class rather than with any object/instance of the class

Outline

- C-like struct
- Class and objects: basic concepts and syntax
- Constructors and destructors
- Access specifier: public, protect, and private

Constructor

- A constructor is a special member function that initializes member variables
- A constructor is automatically called when an object of that class is declared
- Rule I: a constructor must have the same name as the class
- Rule II: a constructor definition cannot return a value

Constructor: Example-I

```cpp
class Circle {
public: //  access specifier, introduced later
   float x, y, r;

   Circle() {
      cout << "Input center:\n";
      cin >> x >> y;
      cout << "Input radius:\n";
      cin >> r;
   }

};
```

```cpp
int main() {
   Circle *a = new Circle(); //Circle() will be called
   delete a;

   Circle b; //  Circle() will be called

   return 0;
}
```

Constructor: Example-II

```cpp
class Circle {
public: //  access specifier, introduced later
   float x, y, r;

   Circle(float x0, float y0, float r0) {
      x = x0; y = y0; r = r0;
   }

};
```

```cpp
int main() {
   Circle a(0, 0, 1);

   Circle *b = new Circle(1, 1, 2);
   delete b;

   //  Note: A constructor cannot be called in the same
   //  way as an ordinary member function is called
   a.Circle(1, 1, 1); //  illegal

   return 0;
}
```

Constructor: Example-II with this Pointer

```
class Circle {
public: //  access specifier, introduced later
   float x, y, r;

   Circle(float x, float y, float r);
};

Circle::Circle(float x, float y, float r) {
    this->x = x;
    this->y = y;
    this->r = r;
}
```

- How does a member function know which `x`?
- All methods in a function have a `this` pointer.
- It is set to the address of the object that invokes the method

Constructor: Example-III

- Constructor is typically overloaded, which allows objects to be initialized in multiple ways

```
class Circle {
public: //  access specifier, introduced later
   float x, y, r;
   Circle() {
      cout << "Input center and radius:\n";
      cin >> x >> y >> r;
   }
   Circle(float x0, float y0, float r0) {
      x = x0; y = y0; r = r0;
   }
};
```

```
int main() {
   Circle *a = new Circle();
   delete a;
   Circle b(0, 0, 1);
   //  Circle() will be called
   Circle c;
   // A constructor behaves like a function that
   // returns an object of its class type
   c = Circle(1, 1, 2);
   return 0;
}
```

Default Constructor

- The constructor with zero arguments is the default constructor
- A default constructor will be generated by compiler automatically if NO constructor is defined

```cpp
class Circle {
public: //  access specifier, introduced later
    float x, y, r;

    void setCenter() {
        cout << "Input center:\n";
        cin >> x >> y;
    }
    void setRadius() {
        cout << "Input radius:\n";
        cin >> r;
    }
};
```

```cpp
int main() {
    Circle a; //  although no constructor is defined,
              //  the compiler will add an empty Circle()
              //  automatically, and call it when a
              //  Circle object is allocated

    a.setCenter();
    a.setRadius();

    return 0;
}
```

Default Constructor (cont'd)

- However, if any non-default constructor is defined, the compiler will not add the default constructor anymore, and call the default constructor will cause compilation error
- In practice, it is almost always right to provide a default constructor if other constructors are being defined

```
class Circle {
public: // access specifier, introduced later
    float x, y, r;

    Circle(float x0, float y0, float r0) {
        x = x0; y = y0; r = r0;
    }

};
```

```
int main() {
    Circle a; // illegal

    Circle *b = new Circle(); // illegal
    delete b;

    return 0;
}
```

Initializer List

- The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon.

```
class Circle {
public: // access specifier, introduced later
   float x, y, r;

   Circle(int x, int y, int r):x(x), y(y), r(r) {}

   // the above initializer list is equivalent to
   // Circle(int x0, int y0, int r0) {
   //    x = x0; y = y0; r = r0;
   // }
};
```

Initializer List

- const and reference member variables MUST be initialized using initializer list

```cpp
class myClass {
public: //  access specifier, introduced later
    const int t1;
    int& t2;

    //  Initializer list must be used
    myClass(int t1, int& t2):t1(t1), t2(t2) {}

    int getT1() { return t1; }
    int getT2() { return t2; }
};
```

```cpp
int main() {
    int myint = 34;

    myClass c(10, myint);

    cout << c.getT1() << endl;
    cout << c.getT2() << endl;

    return 0;
}
```

Copy Constructor

The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

```
class Circle{
private:
    int radius;
public:
    Circle(int r);
    Circle(const Circle& c);
    double getArea();
};
Circle::Circle(int r){
    radius=r;
}
Circle::Circle(const Circle& c){
    radius=c.radius;
}
```

```
double Circle::getArea(){
    return 3.1415*radius;
}


int main(){
    Circle circle(6);
    Circle circle2(circle);
    circle2.getArea();
    return 0;
}
```

Destructor

- A destructor is a special member function which is invoked automatically whenever an object is going to be destroyed
- Rule-I: a destructor has the same name as their class name preceded by a tiled (~) symbol
- Rule-II: a destructor has no return values and parameters
  - destructor overload is NOT allowed
- Statically allocated objects are destructed when the object is out-of-scope
- Dynamically allocated objects are to manually destructed only when you delete them

Destructor: Example

```cpp
class Robot {
public: // access specifier, introduced later
    char *name = NULL;
    Robot(char *name) {
        int n = strlen(name);
        this->name = new char[n+1];
        strncpy(this->name, name, n);
        this->name[n] = '\0';
        cout << "Constructing " << name << endl;
    }
    ~Robot() {
        cout << "Destructing " << name << endl;
        // it's a good practice to free memories allocated
        // for member variables in destructor
        delete[] name;
    }
};
```

```cpp
void func() {
    Robot eve("Eve");
    cout << "func is about to return\n";
    // Automatically calls the destructor when a
    // statically allocated object is out of the
    // scope
}

int main() {
    Robot *wall_e = new Robot("Wall-e");
    func();
    // A dynamically allocated object is destructed
    // only when you explicitly delete it
    delete wall_e;
    cout << "main is about to return\n";
    return 0;
}
```

Class version of Linked List

```cpp
class Node {
public:
    int data;
    Node* next;
    Node(int newData) : data(newData),
next(nullptr) { }
    //  delete `next` is the list's job
    ~Node() { }
};
class LinkedList {
private:
    Node* head;
public:
    LinkedList() : head(nullptr) { }
    ~LinkedList() {
        Node* current = head;
        while (current != nullptr) {
            Node* nextNode = current->next;
            delete current;
            current = nextNode;
        }
    }
    void insertAtBeginning(int newData) {
        Node* newNode = new Node(newData);
        newNode->next = head;
        head = newNode;
    }
    void printList() const {
        Node* current = head;
        while (current != nullptr) {
            cout << current->data << " -> ";
            current = current->next;
        }
        cout << "NULL\n";
    }
};
```

Class version of Linked List

```
int main() {
    LinkedList list;
    list.insertAtBeginning(10);
    list.insertAtBeginning(20);
    cout << "Linked List after
      insertion: ";
    list.printList();
    // LinkedList destructor is
      automatically called here
    return 0;
}
```

```
int main() {
    LinkedList* list = new LinkedList();
    list->insertAtBeginning(10);
    list->insertAtBeginning(20);
    cout << "Linked List after insertion: ";
    list->printList();
    // manually call LinkedList destructor
    delete list;
        return 0;
}
```

Outline

- C-like struct
- Class and objects: basic concepts and syntax
- Constructors and destructors
- Access specifier: public, protect, and private

Access Specifier

- An access specifier defines how the members (data fields and methods) of a class can be accessed
- public: members are accessible from outside the class
- private: members cannot be accessed from outside the class
- protected: members cannot be accessed from outside the class.

  However, they can be accessed in inherited classes (next lecture)
- By default, member variables and functions of class are private if no access specifiers are provided
  - member variables and functions of struct are public by default

Access Specifier: Example

```
class Actress {
private:
  int age;

public:
  char name[255];
  Actress(char *name, int age):age(age) {
    strcpy(this->name, name);
  }
};
```

```
int main() {
  Actress actress("Alice", 25);

  cout << actress.name << endl; // allowed
  cout << actress.age << endl;  // NOT allowed




  return 0;
}
```

Access Specifier: Example

```
class Actress {
private:
  int age;

public:
  char name[255];
  Actress(char *name, int age):age(age) {
    strcpy(this->name, name);
  }
};
```

```
int main() {
  Actress actress("Alice", 25);

  cout << actress.name << endl; // allowed
  cout << actress.age << endl;  // NOT allowed




  strcpy(actress.name, "Eve");  // allowed

  return 0;
}
```

Access Specifier: Example

```
class Actress {
private:
  int age;

public:
  char name[255];
  Actress(char *name, int age):age(age) {
    strcpy(this->name, name);
  }
};
```

```
int main() {
  Actress actress("Alice", 25);

  cout << actress.name << endl; //  allowed
  cout << actress.age << endl;  //  NOT allowed

  //  this is legal but ill-logical
  //  the name of an actress object should NOT
  //  be modified from outside
  strcpy(actress.name, "Eve");  //  allowed

  return 0;
}
```

Access Specifier (cont'd)

```
class Actress {
private:

  int age;

public:
  char name[255];
  Actress(char *name, int age):age(age) {
    strcpy(this->name, name);
  }



};
```

- We want actress name to be *read-only from outside*

Access Specifier (cont'd)

```
class Actress {
private:
  char name[255];
  int age;

public:
  char name[255];
  Actress(char *name, int age):age(age) {
    strcpy(this->name, name);
  }
  char *getName() {
    return name;
  }
};
```

- We want actress name to be *read-only from outside*
- Declare name as private, and then define a public function to read it from outside

Access Specifier (cont'd)

- A common design of OOP is data encapsulation, which is to
  - define all member variables as private
  - provide enough get and set functions to read and write member variables
  - only functions that need to interact with the outside can be made public
  - supporting functions used by the member functions should also be made private