CS2311 Computer Programming

LT11 Object Oriented Programming-I

Computer Science, City University of Hong Kong

Semester B 2022-23

Review: Pointer II

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Review: Pointer Arithmetic

- You can perform arithmetic operations on a pointer with four operators
  - ++, --, +, and –
- When you do arithmetic with a pointer *p*, you consider *p* points to an array, and you perform arithmetic as it's an array index
- e.g.

```
int a[4] = {0, 1, 2, 3};
int *p = &a[3];
p -= 2; // now p points to a[1]
cout << *p << endl;
p++;    // now p points to a[2]
cout << *p << endl;
```

Review: Pointer Arithmetic: common errors

- Multiplication and division of pointers are not allowed in C++

```
int *ptr1, *ptr2, *ptr3;

ptr3 = ptr1 * ptr2; // Error: Multiplication of pointers

ptr3 = ptr1 / ptr2; // Error: Division of pointers

int a = 1, b = 2, c = 3;

*ptr1 = &a; *ptr2 = &b; *ptr3 = &c;

*ptr3 = *ptr1 * *ptr2; // No error: c = a * b

*ptr3 = *ptr1 / *ptr2; // No error: c = a / b
```

Review: Pointer Array

- A pointer array's elements are all pointers.
- For example,

```
int a[6] = {0,1,2,3,4,5};
int *m[2] = {&a[0], &a[3]};
for (int row=0; row<2; row++) {
      for (int col=0; col<3; col++)
            cout << m[row][col] << " ";
      cout << "\n";
}
```

```
0 1 2
3 4 5
```

Review: Array Pointer

- Pointer to a one-dimensional array can be declared as:

```
int arr[] = {1,2,3,4,5};
int *p; p = arr;
```

- Similarly, pointer to a two-dimensional array can be declared as:

```
int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
int *p[3] = arr;      // cannot declare as an array of two pointers
int (*p)[3] = arr;    // a pointer to an array of three integers
cout << *(*(p+1)+2) << endl; // p[1][2]
cout << *(p[2]+1)   << endl; // p[2][1]
```

Review: Quick Summary

- Array of pointer

```
int *a[2];
```

- Pointer of array

```
int a[4][2] = {{0,1}, {2,3}, {4,5}, {6,7}}; int (*p)[2] = a;
cout << p[2][1] << " " << *(*(p+2)+1) << " " << *(p[2]+1);
```

- Pointer of pointer

```
int a=4; int *p=&a; int **pp=&p; cout << **pp;
```

- Pointer reference

```
void func(char* &p);
```

Review: Dynamic Memory Allocation

- Dynamic memory: memory that can be *allocated*, *resized*, and *freed* during program runtime.
- When do we need dynamic memory?
    1. when you need a very large array
    2. when we do not know how much amount of memory would be needed for the program beforehand.
    3. when you want to use your memory space more efficiently.
        - e.g., if you have allocated memory space for a 1D array as array[20] and you end up using only 10 memory

Review: Dynamic Memory Allocation

- Keywords: new & delete

```
//  Declaration
int *p0 = new int(10); //  init an integer 10 in memory, make p0 point to it
char *p1 = new char('a'); //  init a char 'a' in memory, make p1 point to it

// Free memory is your duty. Otherwise, the memory space cannot be reused
delete p0; //  free the memory pointed by p0
delete p1; //  free the memory pointed by p1

//  Will be illegal after deletion
*p0 = 10;
```

Review: Dynamic Memory Allocation

- Syntax on array: new [] and delete []

```
//  Declaration
int n; cin >> n;
int *p0 = new int[n]; //  allocate memory for an int array of n elements
char *p1 = new char[n]; //  allocate memory for a char array of n elements

// Free memory is your duty. Otherwise, the memory space cannot be reused
delete[] p0; //  free the memory pointed by p0
delete[] p1; //  free the memory pointed by p1
```

Review: The NULL pointer

- A special value that can be assigned to **any** type of pointer variable
  - e.g., int *a = NULL; double *b = NULL;
- A **symbolic constant** defined in standard library headers, e.g. <iostream>
- When assigned to a pointer variable, that variable points to nothing
- Initialization after declaration

```
int *ptr1 = NULL;
```

- Check null pointer before using the pointer:

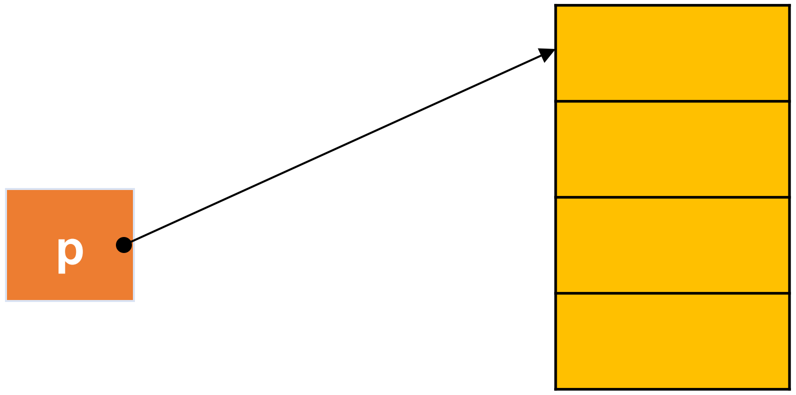
```
if (ptr)
if (!ptr)
```

Review: Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```

Review: Dynamic Memory Walkthrough

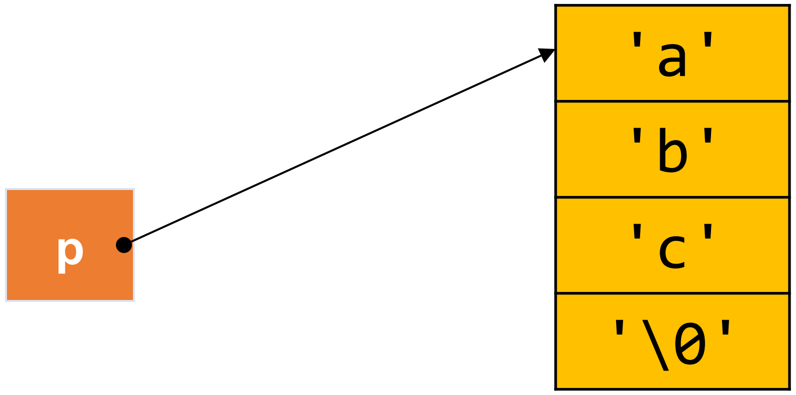
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



new dynamically allocates 4 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

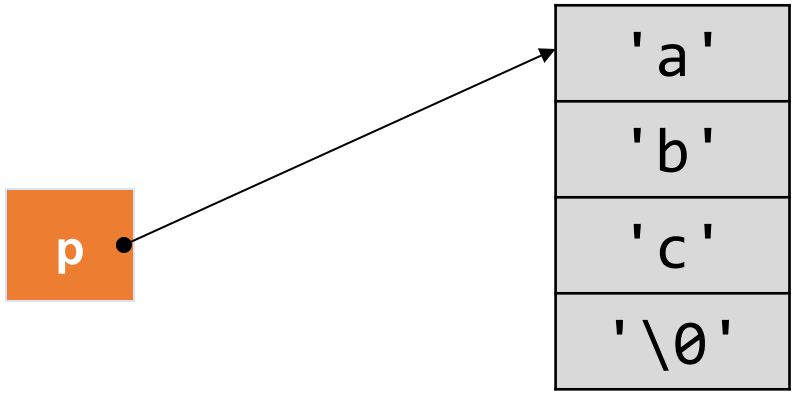
Review: Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



Review: Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```

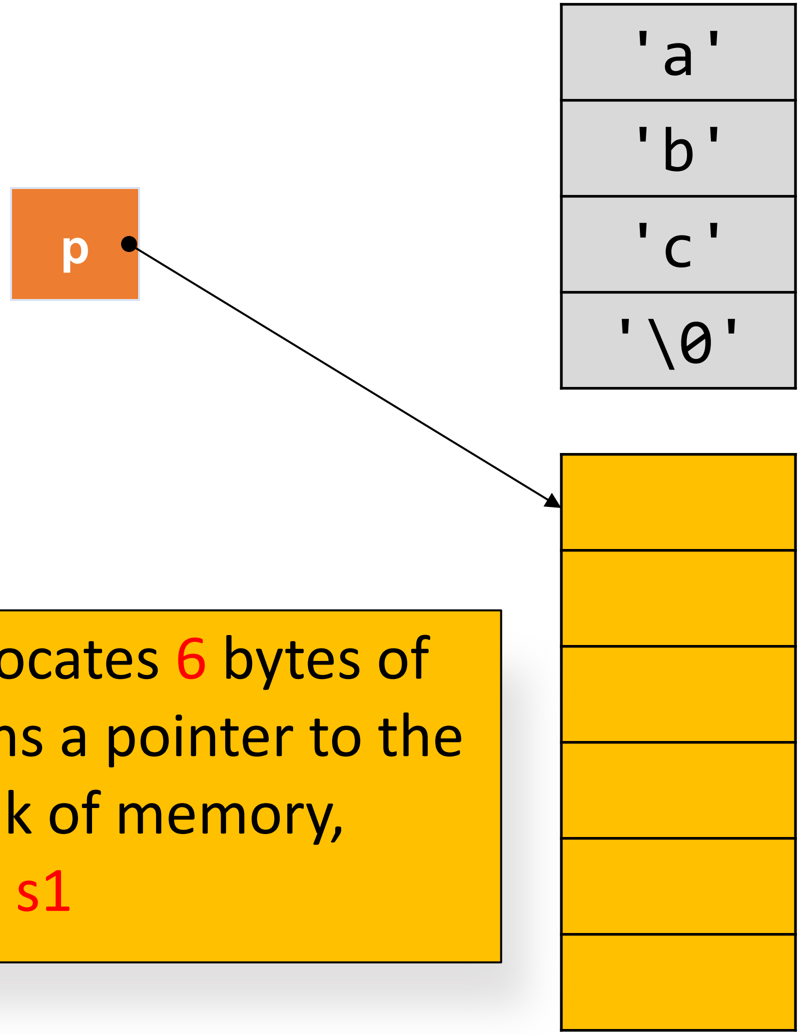


Grey memory means the block of memory is free and can be used to store other data.

p may or may not be pointing to the same address, and you can still print it, but that memory no longer belongs to p.

Review: Dynamic Memory Walkthrough

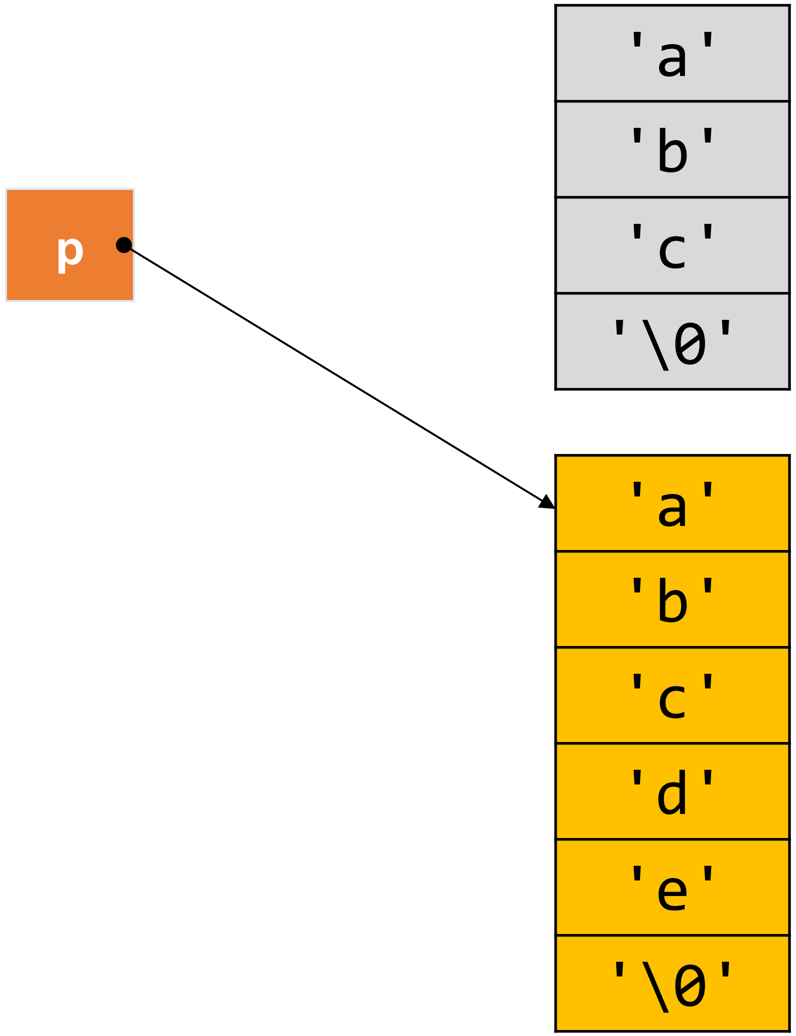
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



new dynamically allocates 6 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

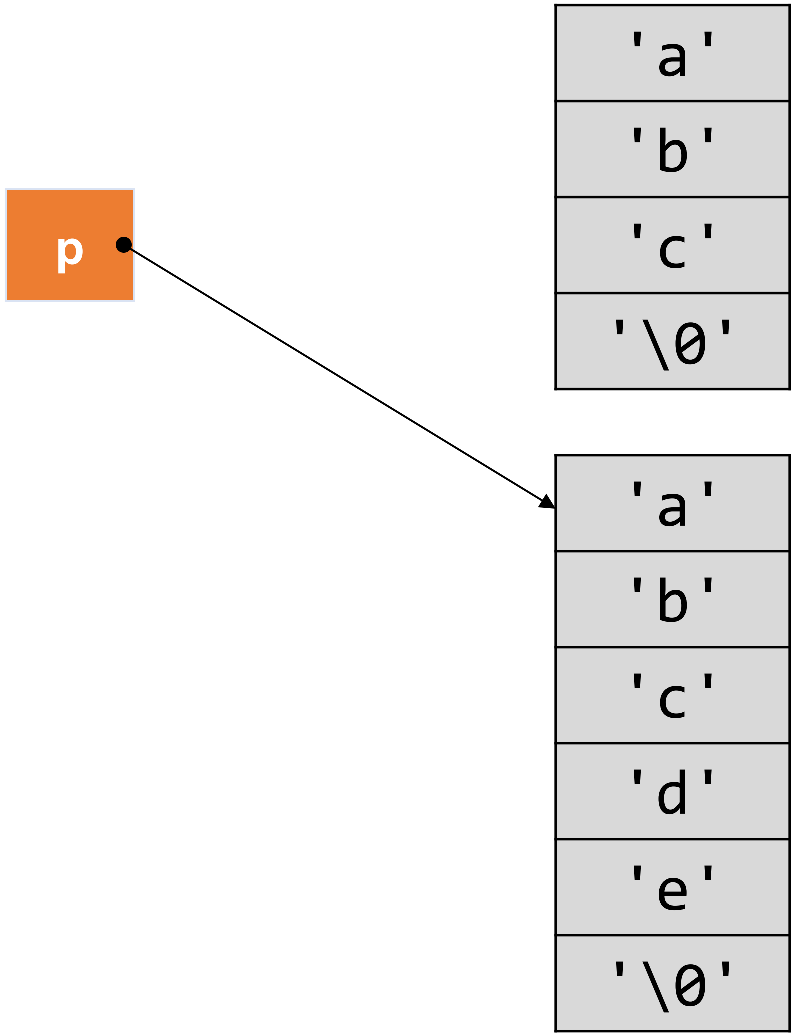
Review: Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL;
```



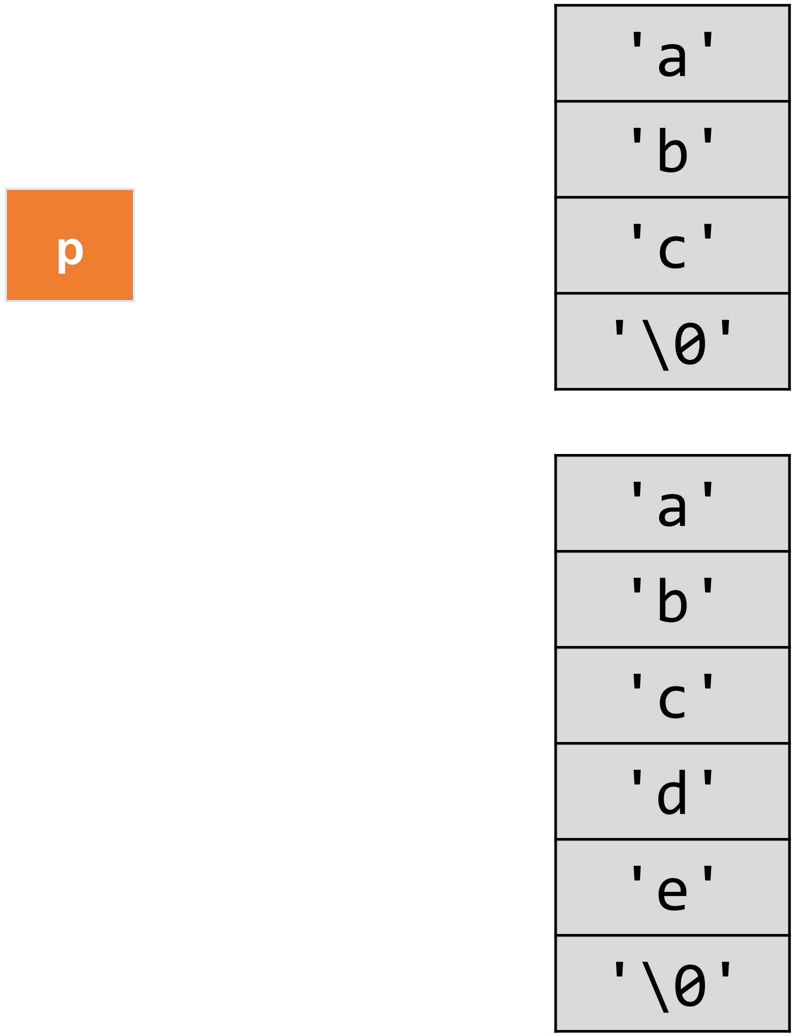
Review: Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL;
```



Review: Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL; // optional
```



Exercise

- Write a function *readInput*() that can read all the integer inputs from the user and print out inputs in a reverse order.
- Assume the first input is n, indicating how many integers we will get from the user.

```
void readInput() {
      ...
}
```

```
Expected Input/Output
3
1
2
3

3 2 1
```

Outline

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

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 "."

```
peter.final += 10;
```

- Structure types can have the same member name without conflict

```
struct CS2311Student {
        char    sid[9];
        float   asg[3];
        float   lab[10];
        float   midterm;
        float   final;
};
```

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

```
CS2311Student peter;
cin >> peter.final;
CS6789Student danny;
cin >> danny.final;
```

Example

```
struct CS2311Student {
    int     sid;
    float   quiz;
    float   asg1;
    float   asg2;
};
```

```
int main() {
  CS2311Student sr;
  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 CS2311Student {
    int     sid;
    float   quiz;
    float   asg1;
    float   asg2;
};
```

Pass/Return Structure to/from Function

- A function can have parameters of structure type:

```
double overall(CS2311Student s) {
       return (s.quiz + s.asg1 + s.asg2)/3;
}
```

- A function can return a value of structure type:

```
CS2311Student newStudent(int sid) {
       CS2311 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
0xa1e	0xa12	dPtr

Structure Pointer: Arrow Syntax

- Arrow syntax `->`: access structure members using pointer
- Example

```
Date d; d.year=2022; d.month=11; d.day=7;
Date *dPtr = &d;
cout << dPtr->year << " " << dPtr->month << " " << dPtr->day << endl;

dPtr->day++; dPtr->month-=2;
cout << dPtr->day << endl;
```

Function Overload

- *Overloading*: two or more functions with the *same name* but *different implementations*

- Two or more functions are said to be overloaded if they differ in
  - the number of arguments, OR
  - the type of arguments, OR
  - the order of arguments

- When an overloaded function is called, the compiler determines the most appropriate call by comparing function argument types

Overload: Example-I

```
void printData(double x) {
   cout << "Print double: " << x << endl;
}
void printData(float x) {
   cout << "Print float: " << x << endl;
}
int main() {
   double a = 0;
   float b = 0;
   printData(a);
   printData(b);
   return 0;
}
```

Overload: Example-II

```
double sum(double x, double y) {
   return x+y;
}
double sum(double x, double y, double z) {
   return x+y+z;
}
int main() {
   double a, b, c;
   cin >> a >> b >> c;
   cout << sum(a, b) << endl;
   cout << sum(a, b, c) << endl;
   return 0;
}
```

Overload: Example-III

```cpp
char *concatenate(char c, char *str) {
    int n = strlen(str);
    char *result = new char[n+2];
    result[0] = c;
    strncpy(result+1, str, n);
    result[n+1] = '\0';
    return result;
}

char *concatenate(char *str, char c) {
    int n = strlen(str);
    char *result = new char[n+2];
    strncpy(result, str, n);
    result[n] = c;
    result[n+1] = '\0';
    return result;
}
```

```cpp
int main() {
    char c = '!';
    char str[] = "Hello";

    char *s0 = concatenate(c, str);
    cout << s0 << endl;
    delete s0;

    char *s1 = concatenate(str, c);
    cout << s1 << endl;
    delete s1;

    return 0;
}
```

Overload: Common Errors

```
int sum(int x, int y) {
    return x+y;
}
int sum(int a, int b) {
    return x+y;
}
int main() {
    int a, b;
    cin >> a >> b;
    cout << sum(a, b);
    return 0;
}
```

```
int sum(int x, int y) {
    return x+y;
}
char sum(int x, int y) {
    return '0'+(char)(x+y);
}
int main() {
    int a, b;
    cin >> a >> b;
    char s = sum(a, b);
    cout << s;
    return 0;
}
```

Overload: Common Errors

```cpp
int sum(int x, int y) {
    return x+y;
}
int sum(int a, int b) {
    return x+y;
}
int main() {
    int a, b;
    cin >> a >> b;
    cout << sum(a, b);
    return 0;
}
```

```cpp
int sum(int x, int y) {
    return x+y;
}
char sum(int x, int y) {
    return '0'+(char)(x+y);
}
int main() {
    int a, b;
    cin >> a >> b;
    char s = sum(a, b);
    cout << s;
    return 0;
}
```

Overload: Ambiguous Call

- Ambiguous call: when the compiler is unable to choose between two correctly overloaded functions

- Automatic type conversions are the main cause of ambiguity

```
void printData(double x) {
    cout << "Print double: " << x << endl;
}
void printData(float x) {
    cout << "Print float: " << x << endl;
}
int main() {
    char a = '0';
    printData(a);   printData((double)a);
    return 0;
}
```

Exercise 1

Given the following structure and structure variable declaration,

```
struct CDAccountV2 {
        double balance;
        double interestRate;
        int term;
        char initial1;
        char initial2;
};
CDAccountV2 account;
```

what is the type of each of the following? Mark any that are not correct.

a. `account.balance`
b. `account.interestRate`
c. `CDAccountV2.term`
d. `account.initial2`
e. `account`

Outline

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

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();
};
```

```
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;
}
```

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
- Usage: resolve shadowing, access currently executing object

```cpp
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";
}
```

Outline

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

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
- <u>Rule I</u>: a constructor must have the same name as the class
- <u>Rule II</u>: 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();
   delete a;

   Circle b;  //  Circle() will be called

   return 0;
}
```

Constructor: Example-II

```
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(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-III

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

```cpp
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;
   }
};
```

```cpp
int main() {
   Circle *a = new Circle();
   delete a;

   Circle b(0, 0, 1);

   Circle c; //  Circle() will be called
   // 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 no parameters 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) {}

   // while 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

```
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; }
};
```

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

    myClass c(10, myint);

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

    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 destructed only when you delete them

Destructor: Example

```
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;
    }
};
```

```
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;
}
```

Outline

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

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 (later)
- By default, member variables and functions are private if no access specifiers are provided

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)

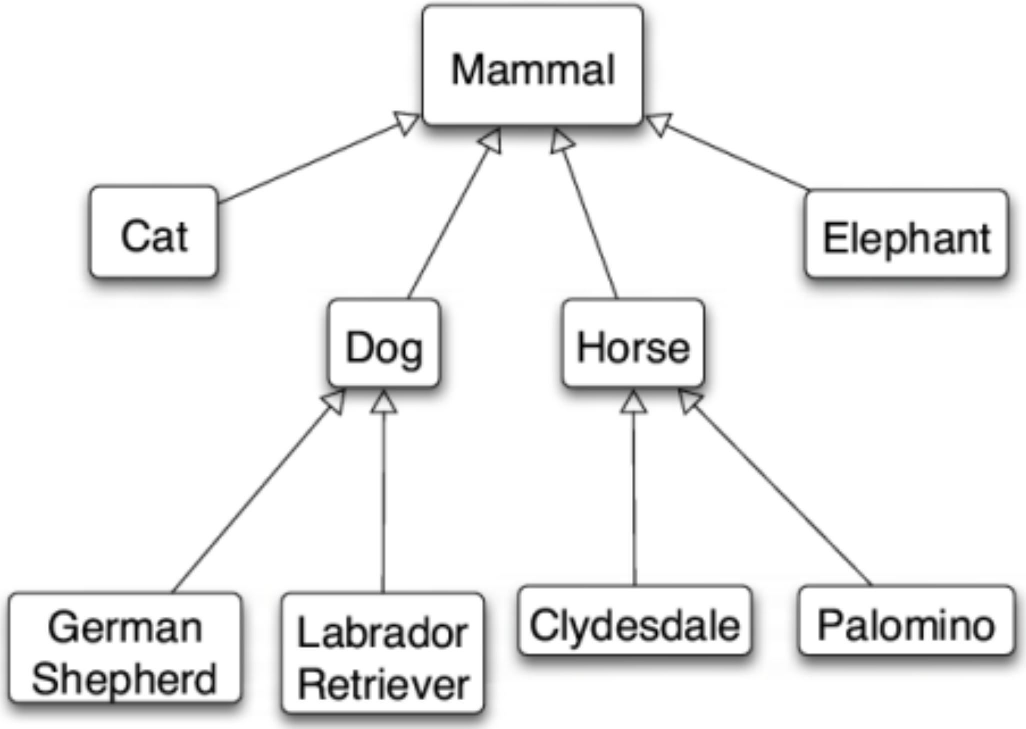
- A common design of OOP is data encapsulation, which is to
  - define all member variables as private
  - provide enough get and set functions to read and write member variables
  - only functions that need to interact with the outside can be made public
  - supporting functions used by the member functions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

Outline

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

What is Inheritance

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
  - every rectangle *is a* shape
  - every lion *is an* animal
  - every lawyer *is an* employee
- **class hierarchy**: A set of data types connected by is-a relationships that **can share common code**.
  - **Re-use**



Basic Concepts

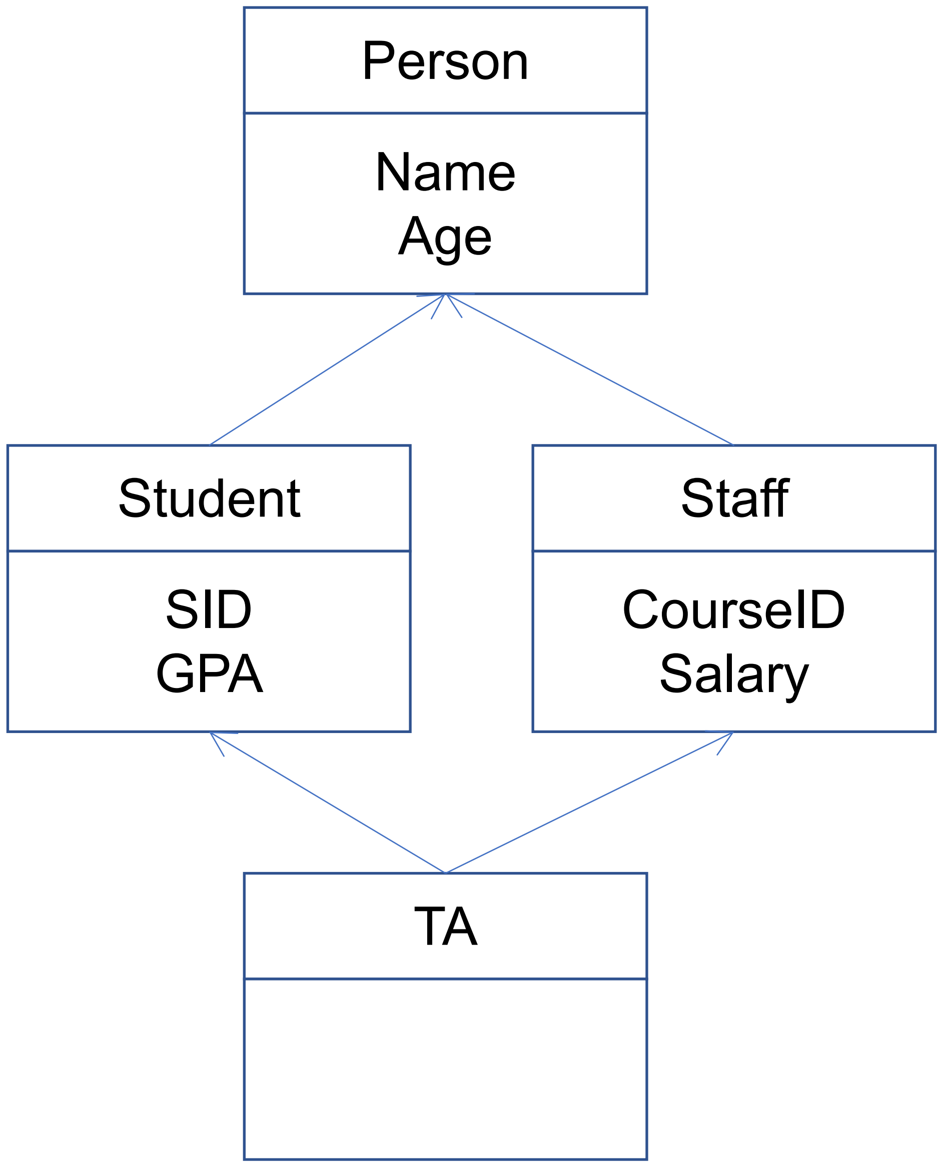
- **Inheritance**: A way to create new classes by extending existing classes

- **Base class**: Parent class that is being extended

- **Derived class**: Child class that inherits from base class(es)
  - A derived class gets a copy of every fields and methods from base class(es).
    - Note: gets a copy does NOT mean can access (details later)
  - A derived class can add its own behavior, and/or change inherited behavior

Basic Concepts

- Multiple inheritance: When one derived class has multiple base classes
- Forbidden in many object-oriented languages (e.g. Java) but allowed in C++.
- Convenient because it allows code sharing from multiple sources.
- Can be confusing or buggy, e.g. when both base classes define a member with the same name.



Syntax

```
class Parent { … };
class Child : AccessSpecifier Parent { … };


class ParentA { … };
class ParentB { … };
class Child : AccessSpecifier ParentA, AccessSpecifier ParentB { … };
```

For example:

```
class TA : public Student, public Staff { … };
```

Inheritance and Access

How inherited base class members appear in derived class

Base class members

```
class Parent {
  private:   x;
  protected: y;
  public:    z;
};
```

class Child : public Parent {…}

```
class Child {
  // x is inaccessible
  protected: y;
  public:    z;
};
```

class Child : protected Parent {…}

```
class Child {
  // x is inaccessible
  protected: y;
  protected: z;
};
```

class Child : private Parent {…}

```
class Child {
  // x is inaccessible
  private:   y;
  private:   z;
};
```

Public Inheritance: Example

<pre>class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};</pre>
	<pre>class B : public A {
	public:
	    void print() {
	        cout << z; // allowed
	        y = 0; // allowed
	        cout << x; // NOT allowed
	    }
	};
	int main() {
	    B obj;
	    obj.y = 0; // NOT allowed, y is protected in B
	    obj.z = 0; // allowed, z is public in B
	    obj.print(); // allowed, print is public in B
	    return 0;
	}</pre>

Protected Inheritance: Example

```cpp
class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};
```

```cpp
class B : protected A {
public:
    void print() {
        cout << z;  // allowed
        y = 0;      // allowed
        cout << x;  // NOT allowed
    }
};
int main() {
    B obj;
    obj.y = 0;     // NOT allowed, y is protected in B
    obj.z = 0;     // NOT allowed, z is protected in B
    obj.print();   // allowed, print is public in B
    return 0;
}
```

Private Inheritance: Example

<pre>class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};</pre>
	<pre>class B : private A {
	public:
	    void print() {
	        cout << z; // allowed
	        y = 0; // allowed
	        cout << x; // NOT allowed
	    }
	};
	int main() {
	    B obj;
	    obj.y = 0; // NOT allowed, y is private in B
	    obj.z = 0; // NOT allowed, z is private in B
	    obj.print(); // allowed, print is public in B
	    return 0;
	}</pre>

Constructors in Inheritance

- Derived classes can have their own constructors

- When an object of a derived class is created, the base class's *default constructor* is executed first *at the beginning* of derived class's constructor, followed by executing the body of the derived class's constructor

```
class A {
public:
   A() { cout << "A's default constructor\n"; }
};
class B : public A {
public:
   B() {
      cout << "B's constructor\n";
   }
};
int main() {
   B b;
}
```

Constructors in Inheritance

- Derived classes can have their own constructors
- When an object of a derived class is created, the base class's *default constructor* is executed first *at the beginning* of derived class's constructor, followed by executing the body of the derived class's constructor

```
class A {
public:
   A() { cout << "A's default constructor\n"; }
   A(int a) {
      cout << "A's non-default constructor\n";
   }
};
class B : public A {
public:
   B() {
      cout << "calling A(2311) in B()\n"; A(2311);
      cout << "calling A() in B()\n";     A();
      cout << "B's constructor\n";
   }
};
int main() {
   B b;
}
```

Passing Arguments to Constructors

```cpp
class Student {
protected:
    int sid;
public:
    Student(int sid=0) : sid(sid) {}
    int getSid() { return sid; }
};
class TA: public Student {
protected:
    int courseid;
public:
    TA(int courseid =0) : courseid(courseid) {}
    int getCourseid() { return courseid; }
};
```

```cpp
#include <iostream>
using namespace std;

int main() {
    Student alice(12345);
    cout << alice.getSid() << endl;

    TA bob(2311);
    cout << bob.getSid() << ": ";
    cout << bob.getCourseid() << endl;

    return 0;
}
```

How to pass parameters to ***base*** *constructor?*

Passing Arguments to Constructors

- To pass arguments from child constructor to parent constructor
  - augment the parameter list of child constructor to include parent constructor parameters, and
  - call parent constructor in initial list

```
class B: public A {
public:
  B(B constructor parameters + A constructor parameters) : A(A constructor's args), … {
    …
  }
};
```

Passing Arguments to Constructors

```
class Student {
protected: int sid;
public:    Student(int sid=0) : sid(sid) {}
           int getSid() { return sid; }
};
class TA: public Student {
protected: int courseid;
public:    TA(int sid=0, int courseid=0) : Student(sid), courseid(courseid) {}
           int getCourseid() { return courseid; }
};
int main() {
    int sid=12345, courseid=2311;
    TA bob(sid, courseid);
    cout << bob.getSid() << ": " << bob.getCourseid() << endl;
    return 0;
}
```

Destructors in Inheritance

- Derived classes can have their own destructors

- When an object of a derived class is destroyed, the derived class's destructor is executed first, followed by the base class's destructor

```
class A {
public:
    ~A() { cout << "A's destructor\n"; }
};
class B : public A {
public:
    ~B() { cout << "B's destructor\n"; }
};
int main() {
    B b = new B();
    delete b;
    return 0;
}
```