CS2310 Computer Programming

LT07: Pointer I and C++ Inheritance

Computer Science, City University of Hong Kong

Semester A 2025-26

Outlines

- Pointer and its operations
- Call by pointer/reference
- C++ Inheritance

* and & operator

- To declare a pointer variable, place a "*" sign before an identifier:
  - `char *cPtr;  //a character pointer`
  - `int *nPtr;  //a integer pointer`
  - `float *fp;  //a floating point pointer`
- To retrieve the address of a variable, use the "**&**" operator:
  - `int x;`
  - `nPtr=&x;`
- To access the variable a pointer pointing to, use "*" operator (dereference)
  - `*nPtr=10;//x=10`
  - `int y;`
  - `y=*nPtr;  //y=x`

reference vs. **dereference**

& *****

Code: lec07-01-declare.cpp

Summary

- * operator will give the **value** of pointing variable (so that you can ***indirectly*** update/modify the pointing variable)
  - E.g., int x; int*p=&x; then using “*p” is equal to “x”;

- & operator will give the **address** of a variable

Const pointer and pointer to const

- If we add keyword `const` at the right-hand side of the '*' sign, the declared pointer is a **constant pointer**
- A constant pointer must be initialized in declaration
  - We cannot change the address that a constant pointer is pointed to
  - But we can still modify the value of the variable that the constant pointer is pointed to

```
int num1=100;
int * const ptr1 = &num1; //initialization
*ptr1=40; // value of num1 changes to 40
cout << num1 << endl;
```


Code: lec07-02-const.cpp

Const pointer and pointer to const

- If we add keyword `const` at the left-hand side of the '*' sign, this pointer is pointed to a **constant value**
- This pointer can point to other constant values later. However, we cannot change the value that this pointer is pointed to

```
const int num2 = 100;
const int num3 = 150;
int const * ptr2; // or const int * ptr2
ptr2 = &num2;
*ptr2 = 40; // illegal: cannot change the
            // value of const int
ptr2 = &num3;
```

Pointer array

- We can define a pointer array to manage multiple pointers. For example: `int *n[5];`
- We can use the '*' sign to change the value of the variable that each pointer (in the array) is pointed to

```
int *n[5];
int a[5] = {0}; // Initially all 0
for(int i = 0; i<5; i++){
  n[i] = &a[i];
  *n[i] = i;
} // The value of a[] changes to 0,1,2,3,4
```


Code: lec07-03-array.cpp

Outlines

- Pointer and its operations
- Call by pointer/reference
- C++ Inheritance

Applications of pointer

- **Call by Pointer**
- **Fast Array Access**
  - Will be covered in later class
- **Dynamic Memory Allocation**
  - Require additional memory space for storing value.
  - Similar to variable declaration but the variable is stored outside the program.

Call by pointer

- Pass the address of a variable to a function
- **call by value** cannot be used to update arguments to function
- Consider the following function

```
void f (char c){
   c='B'; // c=66
}

void main(){
   char c='A'; // c =65
   f(c);
   cout << c << endl;
}
```

Call by Pointer

- Add the '*' sign to the function parameters that store the variable call by pointer

```
void cal(int *x, int y){
    //x is call by pointer
    //y is call by value
    ……
    ……
}
```

- Add the '&' sign to the variable when it needs to be call by pointer

```
cal(&x, y);
```

Call by Pointer

```
void f (char *cPtr){
   *cPtr='B';
}

void main(){
   char c='A'; // c =65
   f(&c);
   cout << c << endl;
}
```

When f() is called, the following operation is performed

cPtr = **&c**;

Call by Pointer

```
void f (char *cPtr){
   *cPtr='B';
}

void main(){
   char c='A'; // c =65
   f(&c);
   cout << c << endl; //print 'B'
}
```

points to

Variable	Variable type	Memory location	Content
c	char	**3A8E**	65
cPtr	char pointer	4000	

Assign location 3A8E to cPtr

Location of c (location 3A8E) is assigned to cPtr1

cPtr = **&c**;

Call by Pointer

```
void f (char *cPtr){
   *cPtr='B';
}

void main(){
   char c='A'; // c =65
   f(&c);
   cout << c << endl; //print 'B'
}
```

update

cPtr points to location 3A8E (that is the variable c). *cPtr refers to the variable pointed by cPtr, i.e. the variable stored at 3A8E

Variable	Variable type	Memory location	Content
c	char	3A8E	66
cPtr	char pointer	4000	3A8E

```
cPtr='B'; //error
```

Reason: cPtr1 stores a **location** so it cannot store a **char** (or the ASCII code of a char)

Note the different meaning of *

The type of cPtr is char* (pointer to star)

dereference a pointer:
c='B'

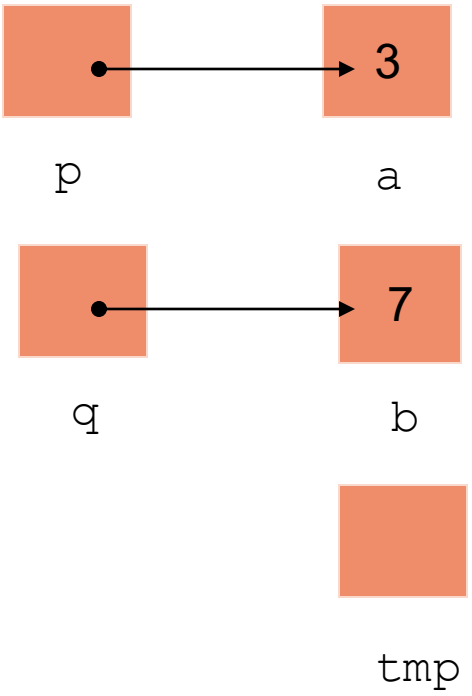
```
void f (char *cPtr){
   *cPtr='B';
}

void main(){
   char c='A'; // c=65
   f(&c);
   cout << c << endl; //print 'B'
}
```

Call by value and call by pointer

- In **call by value**, only a single value can be returned using a return statement
- In **call by pointer**, the argument(s) can be a pointer which may reference or points to the variable(s) in the caller function
  - More than one variables can be updated, achieving the effect of returning multiple values

Example: swapping values

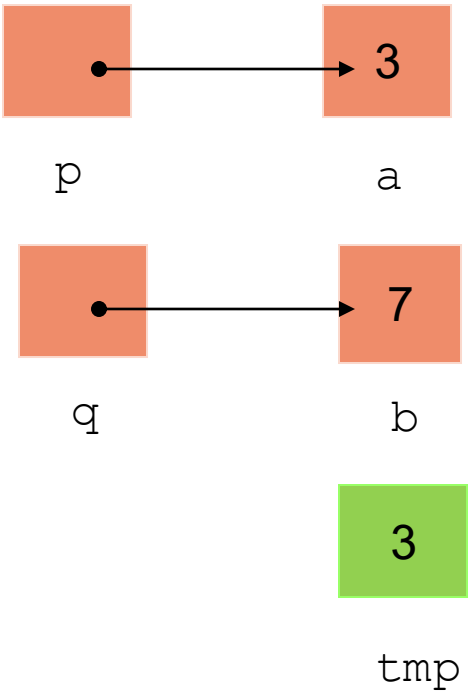
```
#include <iostream>
using namespace std;

void swap(int *p, int *q) {
    int   tmp;

    tmp = *p;              /* tmp = 3 */
    *p = *q;               /* *p = 7 */
    *q = tmp;              /* *q = 3 */
}

int main(void)
{
    int   a = 3, b = 7;
    swap(&a, &b);
    cout << a <<" "<< b <<endl;
         /* 7  3 is printed */
    return 0;
}
```



Code: lec07-04-swap.cpp

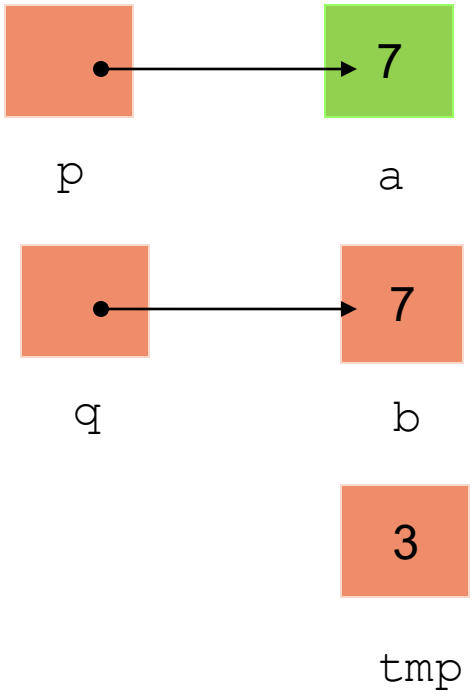
Example: swapping values

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

void swap(int *p, int *q) {
   int   tmp;

   tmp = *p;            /* tmp = 3 */
   *p = *q;             /* *p = 7 */
   *q = tmp;            /* *q = 3 */
}

int main(void)
{
   int   a = 3, b = 7;
   swap(&a, &b);
   cout << a <<" "<< b <<endl;
        /* 7  3 is printed */
   return 0;
}
```



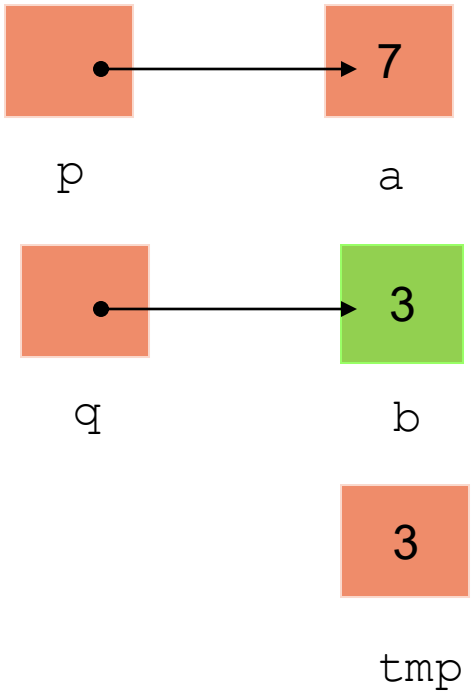
Example: swapping values

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

void swap(int *p, int *q) {
    int   tmp;

    tmp = *p;              /* tmp = 3 */
    *p = *q;               /* *p = 7 */
    *q = tmp;              /* *q = 3 */
}

int main(void)
{
    int   a = 3, b = 7;
    swap(&a, &b);
    cout << a <<" "<< b <<endl;
         /* 7  3 is printed */
    return 0;
}
```



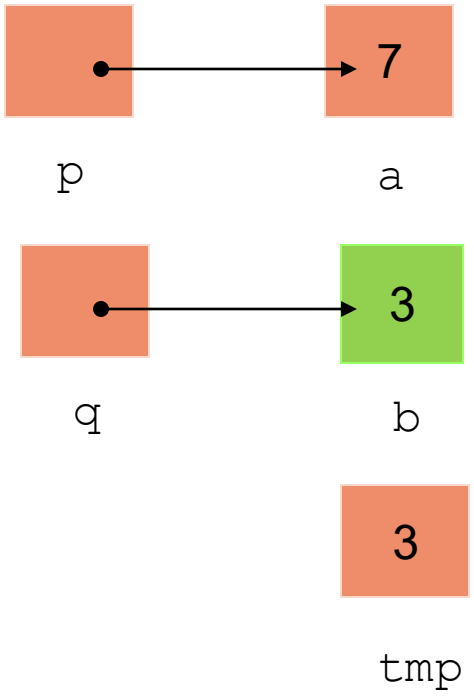
Example: swapping values

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

void swap(int *p, int *q) {
    int   tmp;

    tmp = *p;              /* tmp = 3 */
    *p = *q;               /* *p = 7 */
    *q = tmp;              /* *q = 3 */
}

int main(void)
{
    int   a = 3, b = 7;
    swap(&a, &b);
    cout << a <<" "<< b <<endl;
         /* 7  3 is printed */
    return 0;
}
```



Example: swapping values **(in C++)**

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

void swap(int &p, int &q) {
    int   tmp;

    tmp = p;              /* tmp = 3 */
    p = q;                /* *p = 7 */
    q = tmp;              /* *q = 3 */
}

int main(void)
{
    int   a = 3, b = 7;
    swap(a, b);
    cout << a <<" "<< b <<endl;
         /* 7  3 is printed */
    return 0;
}
```



Call by Reference

- Reference is another “name” of a variable

Call by Reference

- Reference is another "name" of a variable
- Syntax: `dataType &ref = variable;`
  - `ref` must be **initialized** in the declaration
  - `ref` cannot be pointed to another new variable after declaration

```
int n = 100;
int &ref = n;
```


Code: lec07-05-ref.cpp

Call by Reference

- Reference is a constant pointer
  - that is why it must be initialized during declaration and cannot be pointed to another new variable

```
int n = 100;
int &ref = n; // int * const p = &n;

ref = 200; // *p = 200;
```

Call by Reference

- Constant reference
  - The value, pointed to by a constant reference, **cannot** be changed by this reference. But the value can still be changed by its original variable name

```
const int &ref1 = 100;          // ok
int &ref2 = 100;                // error

const int n = 150;
const int &ref3 = n;

int m = 200;
const int &ref4 = m;    // ok
m = 300;
ref4 = 400;             // error
```

- Used in function parameters

```
void f(const int &ref) {
        ref += 100; // error
}
```


Code: lec07-06-constant.cpp

main() function with parameters

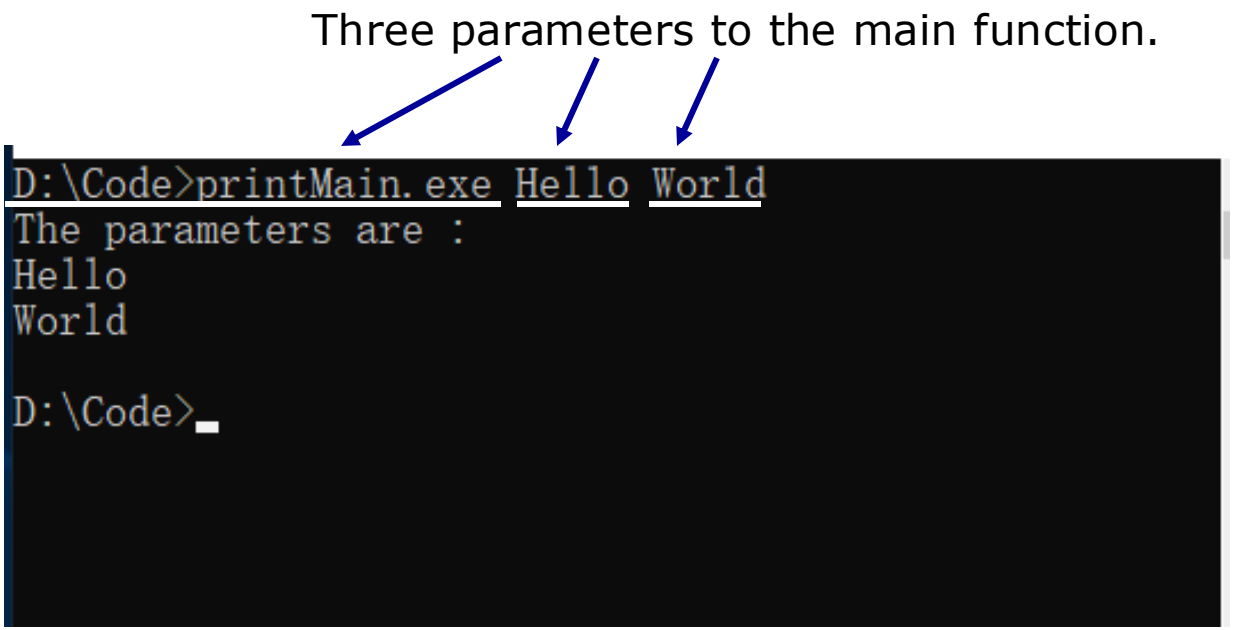
- We can provide parameters to the main() function using cmd in Windows to run the .exe file generated by the complier

```
int main(int argc, char *argv[]){
  cout<<"The parameters are: "<<endl;
  for(int i = 0; i<argc; i++){
    cout<<*(++argv)<<endl;
  }
  return 0;
}
```


Code: lec07-07-main.cpp

main() function with parameter

- *argc* stores the number of parameters, including the absolute path of the .exe file, i.e., argc = 1 when there is no extra parameter
- *argv[]* stores all the parameters in the form of char*



Pointer as a return value

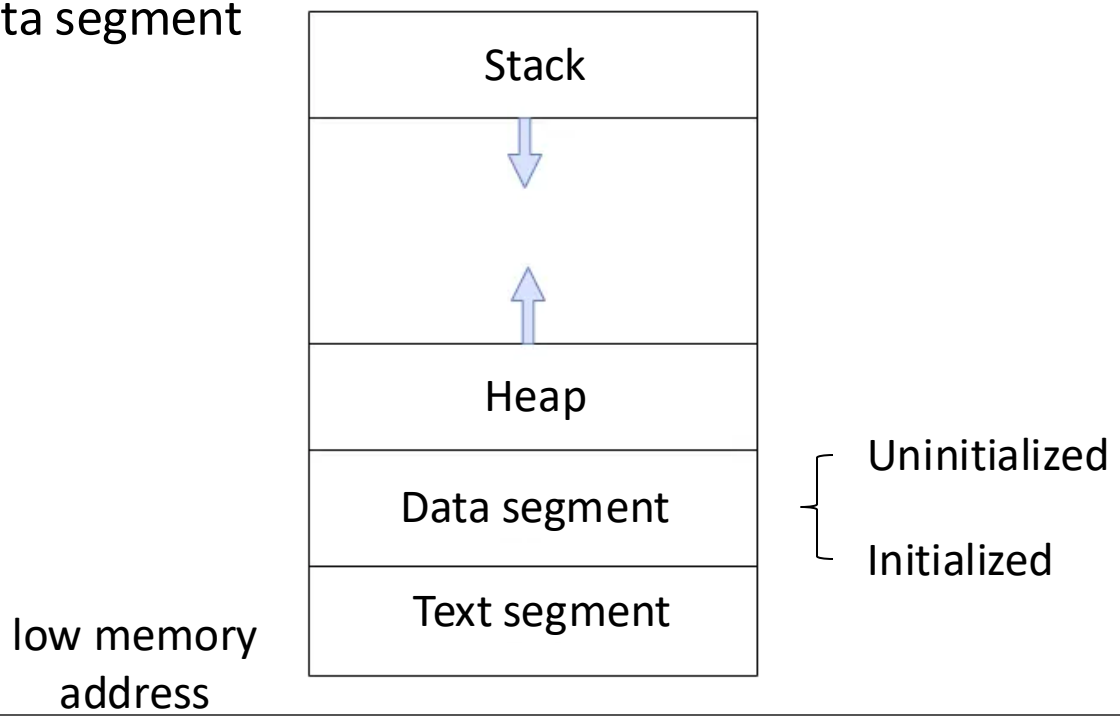
- A function can return a pointer
  - The address of a variable will be returned

```
int* f(int n){
  int *ptr1 = new int(n);
  return ptr1;
}
int main(){
  int *ptr = f(10);
  cout << *ptr << endl;
  cout << *ptr << endl;

  return 0;
}
```

Code: lec07-08-return.cpp

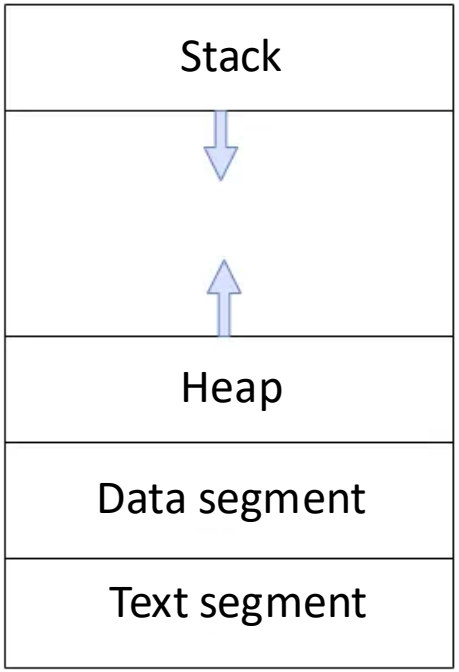
Memory Layout

- Four **main** segments
  - Text segment
  - Data segment
    - Uninitialized data segment (a.k.a. bss)
    - Initialized data segment
  - Stack
  - Heap



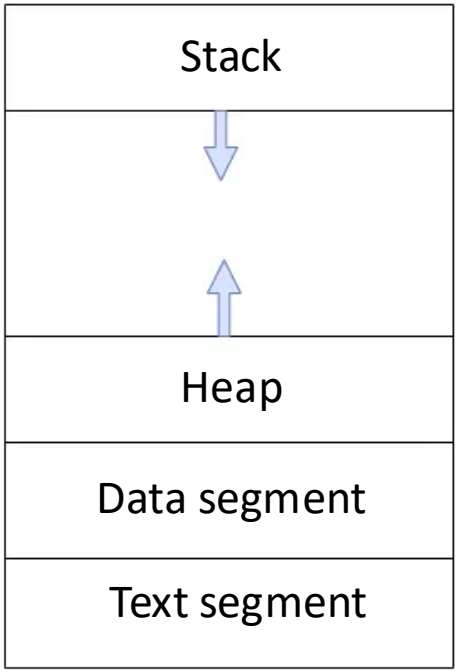
Memory Layout

- Text segment (or code segment)
  - Executable instructions
- Properties
  - Shared area (single copy)
  - Read-only



Memory Layout

- **Data segment**
  - global variables and static variables
  - string constant
  - global constant
- **Stack**
  - local variables
  - function parameters, etc.
- **Heap**
  - dynamic memory allocation



Code: lec07-09-layout.cpp

Outlines

- Pointer and its operations
- Call by pointer/reference
- C++ Inheritance

Object-Oriented Programming (OOP)

- Three important features of OOP
  - **Data encapsulation**
    - Wrap data and functions into a unit called, a class
    - One related concept – **Data abstraction**
      - Displaying only essential information and hiding the details, e.g., using class is a typical way of data abstraction
  - **Inheritance**
    - Get (derive) properties and characteristics from another class
  - **Polymorphism**
    - Have multiple functions with the same name, with different operations

Inheritance in C++

- Why is inheritance needed?

```
class Phone {
public:
  void unlock() {}
  void playSound() {}
  void update() {}

  void call() {}
};
```

```
class Tablet {
public:
  void unlock() {}
  void playSound() {}
  void update() {}

  void splitView() {}
};
```

```
class Watch {
public:
  void unlock() {}
  void playSound() {}
  void update() {}

  void changeWatchFace() {}
};
```

Code: lec07-10-why.cpp

Inheritance in C++

- A new class is created (or inherited) from an existing class
  - Child / derived class: the new class created
  - Parent / base class: the existing class
- Syntax

```
class <derived_class> : <access-specifier> <base_class>
{
        //body
};
```

Child class

```
class Tablet : public Device {
public:
  void splitView() {}
};

class Watch : public Device {
public:
  void changeWatchFace() {}
};
```

Parent class

```
class Device {
public:
  void unlock() {}
  void playSound() {}
  void update() {}
private:
  int power;
};
```

Code: lec07-11-inheritance.cpp

Modes of Inheritance

- Access specifiers
  - **public** mode
    - Public members of the **base class** will be public members in the **derived class**
    - Protected members of the **base class** will become protected members in the **derived class**
  - **protected** mode
    - Both public and protected members of the **base class** will be protected in the **derived class**
  - **private** mode
    - Both public and protected members of the **base class** become private in the **derived class**

Modes of Inheritance

- Access specifiers

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

How are x, y and z inherited in a child class?

public

protected

private

```
class A :public Base{
public:
  int x;
protected:
  int y;
cannot access:
  int z;
};
```

```
class B :protected Base{
protected:
  int x;
  int y;

cannot access:
  int z;
};
```

```
class C :private Base{
private:
  int x;
  int y;

cannot access:
  int z;
};
```


Code: lec07-12-modes.cpp

Modes of Inheritance

- Try to access each member variable from Base

```
class A :public Base{
public:
  int x;
protected:
  int y;
cannot access:
  int z;
public:
  void func();
};
```

```
void A::func() {
  x = 10; // OK
  y = 20; // OK
  z = 30; // Error
};

void main() {
  A a;
  a.x = 1; // OK
  a.y = 2; // Error
  a.z = 3; // Error
};
```

```
class B :protected Base{
protected:
  int x;
  int y;

cannot access:
  int z;
public:
  void func();
};
```

```
void B::func() {
  x = 10; // OK
  y = 20; // OK
  z = 30; // Error
};

void main() {
  B b;
  b.x = 1; // Error
  b.y = 2; // Error
  b.z = 3; // Error
};
```

```
class C :private Base{
private:
  int x;
  int y;

cannot access:
  int z;
public:
  void func();
};
```

```
void C::func() {
  x = 10; // OK
  y = 20; // OK
  z = 30; // Error
};

void main() {
  C c;
  c.x = 1; // Error
  c.y = 2; // Error
  c.z = 3; // Error
};
```

Modes of Inheritance

- Private members in the base class
  - They are still inherited, but hidden in the **child class** (not accessible in the child class)

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

```
class A :public Base{
public:
  int w;
};
```

```
void f () {
  A a;
  cout << sizeof(a) << endl;
};
```


Code: lec07-13-sizeof.cpp

Order of Constructor / Destructor

- Both child and parent classes have constructors and destructors. What is the order in which they are executed?
  - parent-con → child-con → child-des → parent-des

```
class Base {
public:
  Base()  { cout << "base-constructor" << endl;}
  ~Base() { cout << "base-destructor" << endl;}
};

class Child :public Base {
public:
  Child()  { cout << "child-constructor" << endl;}
  ~Child() { cout << "child-destructor" << endl;}
};
```

```
void f () {
  Child c;
};
```


Code: lec07-14-order.cpp

Same Name in Child & Parent Class

- The name in the child class will hide it in the parent class
  - To access this name in parent class, use the scope operator ::

```
class Base {
public:
  Base()         { x=1; }
  void f()       { cout << "base-1" << endl;}
  void f(int z)  { cout << "base-2" << endl;}

  int x;
};
```

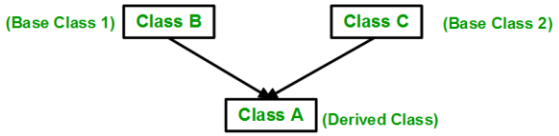
```
class A :public Base{
public:
  A()            { x=2; }
  void f()       { cout << "child-1" << endl;}

  int x;
};
```

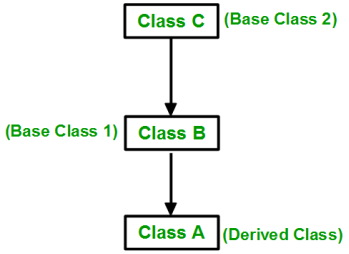
```
void f () {
  A a;
  cout << a.x << endl;          // 2
  cout << a.Base::x << endl;    // 1

  a.f();                        // child-1
  a.Base::f();                  // base-1
  a.Base::f(10);                // base-2
};
```


Code: lec07-15-sameName.cpp

Types of Inheritance

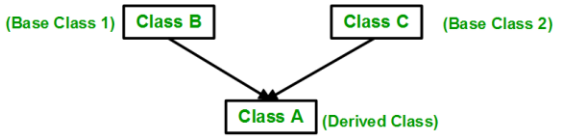
- Types
  - **Single** inheritance
    - a child class inherits from only one base class
  - **Multiple** inheritance
    - a class can inherit from more than one class
  - **Multilevel** inheritance
    - a child (derived) class is created from another derived class
  - …





Types of Inheritance

- Multiple inheritance
- Syntax




```
class child : mode base1, mode base2, …
{
        //body
};
```

```
class Base1 {
public:
  Base1() { x=1; }
  int x;
};


class Base2 {
public:
  Base2() { x=2; }
  int x;
};
```

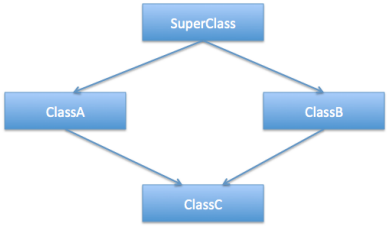
```
class A :public Base1, public Base2 {
public:
  int y;
};
```

```
void f () {
  A a;
  cout << sizeof(a) << endl;
  cout << a.Base1::x << endl;
  cout << a.Base2::x << endl;
};
```


Code: lec07-16-multiple.cpp

Diamond Problem

- When a class inherits from two or more classes that have a common base class, the members of that common base class may cause ambiguity in the program




```
class Person {
public:
  int age;
};
```

```
class Mother :public Person {
public:
  Mother() { age = 50; }
};
```

```
class Father :public Person {
public:
  Father() { age = 51; }
};
```

```
class Child :public Mother, public Father {
public:
  Child() { age = 20; } // Error
};
```


Code: lec07-17-diamond.cpp

Diamond Problem

- Solution 1: use :: to specify

```
class Child :public Mother, public Father {
public:
  Child() { Mother::age = 20; }
};
```

```
void f () {
  Child c;
  cout << c.Mother::age << endl;
};
```

```
class Mother    size(4):
        +---
0       | +--- (base class Person)
0       | | age
        | +---
        +---
```

```
class Person {
public:
  int age;
};
```

```
class Mother :public Person {
public:
  Mother() { age = 50; }
};
```

```
class Child     size(8):
        +---
0       | +--- (base class Mother)
0       | | +--- (base class Person)
0       | | | age
        | | +---
        | +---
4       | +--- (base class Father)
4       | | +--- (base class Person)
4       | | | age
        | | +---
        | +---
        +---
```

Diamond Problem

- Solution 2: virtual inheritance

```
class Mother :virtual public Person {
public:
  Mother() { age = 50; }
};
```

```
class Father :virtual public Person {
public:
  Father() { age = 51; }
};
```

```
class Child :public Mother, public Father {
public:
  Child() { age = 20; } // OK
};
```

```
void f () {
  Child c;
  cout << c.age << endl;
  cout << c.Mother::age << endl;
  cout << c.Father::age << endl;
};
```


Code: lec07-18-virtual.cpp

Diamond Problem [Optional]

- Solution 2: virtual inheritance
  - We **only** have one variable of `age`
  - `vbptr`: virtual base pointer
  - `vbtable`: virtual base table

```
class Person    size(4):
        +---
 0      | age
        +---
```

```
class Child     size(12):
        +---
 0      | +--- (base class Mother)
 0      | | {vbptr}
        | +---
 4      | +--- (base class Father)
 4      | | {vbptr}
        | +---
        +---
        +--- (virtual base Person)
 8      | age
        +---

Child::$vbtable@Mother@:
 0      | 0
 1      | 8 (Childd(Mother+0)Person)

Child::$vbtable@Father@:
 0      | 0
 1      | 4 (Childd(Father+0)Person)
```

```
class Mother    size(8):
        +---
 0      | {vbptr}
        +---
        +--- (virtual base Person)
 4      | age
        +---

Mother::$vbtable@:
 0      | 0
 1      | 4 (Motherd(Mother+0)Person)
```

```
class Mother :virtual public Person {
public:
  Mother() { age = 50; }
};
```