Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Introduction

Grade distribution

- Homework assignments: 10%
- Attendance: 10%
- Midterm exam:20%
- Final exam: 60%

C++: A federation of languages

- C
    - C is the base of C++
- Object-Oriented C++
    - C with classes: **classes（类）**, **encapsulation（封装）**,**inheritance（继承）**, **polymorphism（多态）**, **virtual functions（虚函数）** et al.
- Template（模板） C++
    - **Generic programming（泛型编程）**
- STL
    - **Standard Template Library（标准模版库）**

Chapter 2 Beginning with C++

- C++ has **two conventions** for comment:

```
//

/* */ C-syntax
```

- A line beginning with a **pound sign(#)** is called a **preprocessor directive**

```
#include <iostream>
```

    - Header files contain constant, variable, data type,classes and function **declarations** needed by a program (**no definitions**)

Type of header file	Rule	Examples
C style	With suffix .h	string.h, iostream.h
C++ style	Without suffix .h	string, iostream
Converted c style	With affix c and without suffix .h	cstring, cmath

- C++ provides **namespaces** to prevent name conflicts.

- Namespace defines a scope for the identifiers that are used in a program.
- Namespace is the mechanism for supporting module programming paradigm (模块编程范式).
- **std** is the namespace where identifiers in ANSI C++ **standard libraries** are declared.

```
using namespace std;
```

- An identifier declared within a namespace block **can be accessed directly** only by statements within that block.
- To access an identifier that is **"hidden" inside a namespace**, the programmer has several options. We describe two options here.
    - The first option is to **use a qualified name for the identifier**. A qualified name consists of the name of the namespace, then the **:: operator** (**the scope resolution operator**), and the desired identifier:

```
std::cout
```

    - The second option is use a statement called a using directive:

```
using namespace std;
```

- **Output & Input**
  - For standard output, use **cout**
    - **cout** is a predefined object of the standard output stream

```
cout << S1 << ' ' << S2 << ' ' << S3 << endl;
```

    - The reserved word, **endl**, ensures that the next cout command prints its stuff on a **new line**.
    - The identifier **endl** is a special C++ feature called a **manipulator**.
    - New lines can also be added using '**\n**'.
  - For standard input, use **cin**
    - **cin** is a predefined object of the standard input stream

```
cin >> V1 >> V2 >> V3;
```

    - Split the input elements by **space** or **return**.

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 3 Basic Concepts

- **Tokens（标记）**：the smallest individual units in a program:
  - Keywords （关键字）
  - Identifiers （标识符）
  - Constants （常量）
  - Reference（引用）
  - Strings （字符串）
  - Operators （运算符）

- **Keywords（关键字）**: A reserved word that has special meaning in C++ and can not be used as a programmer-defined identifier.

	C++	keywords	
asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explict	namespace	typeid	

*Note: The ANSI C keywords are shown in **bold** face.*

- **Identifiers（标识符）**: naming program element
    - Identifiers are made up of:
        - letters (A-Z, a-z)
        - digits(0-9)
        - the underscore character(_)
        - and **must begin with a letter or underscore**
    - Example:
        - Sum_of_squares $\rightarrow \surd$
        - J9 $\rightarrow \surd$
        - 40Hours $\rightarrow \times$
        - _22A $\rightarrow \surd$
        - G et Data $\rightarrow \times$
        - Box-1 $\rightarrow \times$
        - Cost_in_$ $\rightarrow \surd$
        - namespace $\rightarrow \times$
- **Constants（常量）**: fix values that do not change during the execution of a program
    - **Literal constant（字面常量）**： Any constant value written in a program, like `10`, `3.14`, `'a'`.

- **Symbolic constant（命名常量）**：A location in memory, referenced by an identifier, that contains a data value that cannot be changed:
  - Using the qualifier **const**
  - Defining a set of integer constants using **enum** keyword
- **Qualifier const（const 说明符）** allow us to create typed constants
  - Called as "**declared constant**" or "**named constant**"
  - A const **must be initialized** when declared
  - A const is **local to the file where it is declared**
    - Use **extern** to make it can be referenced from another file
- **Const with pointers**
  - **Pointer to a constant（指向常量的指针）**：

    ```
    const int * pi;
    ```

    - Pointer `pi` points to a constant
    - Cannot change **the contents of what it points to** through the pointer
    - The **value of the pointer** itself **can be changed**
    - Example:

    ```
    int a = 1, b = 2;
    const int c = 3;
    const int *pi;
    pi = &a; // OK
    *pi = 10; // Err
    a = 10; // OK
    pi = &c; // OK
    *pi = 100; // Err
    c = 100; // Err
    ```

  - **Constant pointer（常指针）**：

    ```
    int a;
    int * const pi = &a;
    ```

    - The pointer itself is a constant，**must be initilaized**
    - Contents of **what it points to can be changed**
    - Example:

    ```
    int a = 1, b = 2;
    const int c = 3;
    int * const pi = &a;
    *pi = 10; // OK
    pi = &b; // Err
    int * const pint_c = &c; // Err
    *pi = 20; // OK
    ```

  - **Constant pointer points to a constant（指向常量的常指针）**：

    ```
    const int * const pi = &a;
    ```

    - The value of the pointer can not be changed
    - Cannot change the contents of what it points to through the pointer
    - Example:

```
int a = 1, b = 2;
const int c = 3;
const int *const pi = &a;
*pi = 10; // Err
pi = &b; // Err
```

- **Enumerated data type （枚举）**
  - Enumerated data type example:

```
enum shape{circle, square, triangle};
shape myshape;
myshape = circle;
```

  - By default, the **enumerators（枚举常量）** are assigned integer values **starting with 0**
  - An enumerated value can be used in place of an **int value**:

```
int c = circle; // equals to int c = 0;
```

  - We can over-ride the default by explicitly assigning integer values:

```
enum shape{circle, square = 4, triangle = 8};
enum shape{circle = 6, square, triangle};
```

- **The type void（空类型）**
  - Specify the return type of a function

```
void func();
```

  - Indicate an empty argument list to a function

```
void func(void);
```

  - Generic pointer（通用指针）/ `void*`

```
void * gp;
int i;
int * ip = &i;
gp = ip; // OK
ip = gp; // Err
ip = (int *)gp; // OK
```

- **Reference Variables（引用）**
  - A reference variable provides an **alias(alternative name)** for a previously defined variable.
  - A reference variable is created as following:

```
data_type reference_name = variable_name
```

    - A reference must be **initialized at the time of declaration**
    - Reference to pointer is legal:

```
int * a;
int * &P = a; // Pointer's reference
int b;
P = &b; // Let P become the reference of b's pointer(address)
int &m = *a; // Let m become the reference of a
```

        - Reference to array is **illegal**:

          ```
          int n[10];
          int & x[10] = n; // Err
          int & x=n[9];
          ```
    - Pass arguments by **references**:

      ```
      void f(int &x){
        x = x + 10
      }

      int main(){
        int m = 10;
        f(m);
        cout << m << endl;
        return 0;
      }
      ```

    - Return by **reference**
        - For a function returns a reference, what returned can be:
            - **independent reference**
            - **array element**
            - **static variable**
            - **variable pointed by a pointer**
        - A returned reference must be valid in the calling program

          ```
          int & get_reference(int x){
            return x; // Err
          }
          ```
- **Operators in C++**
    - All C operators are valid in C++
    - Additional operators include:
        - `::`
        - `::*`
        - `::->`
        - `.->`
        - `delete`
        - `new`
        - `endl`
        - `setw`
    - Scope resolution operator(`::`)
        - The global version of a variable can be accessed from within the inner block by using the scope resolution operator `::` as:

          ```
          :: variable_name
          ```

    - `new` & `delete` operators
        - Malloc & free
            - Function prototype:

              ```
              void * malloc(size_t size);
              void free(void *p);
              ```

        - Shortcomes:
            - User needs to **calculate the size** of the data object
            - The data type of the return value is `void *`.User **need to use a type cast**
            - After free the memory space, the function does **not set the pointer to** `NULL`
    - Operator `new` dynamically allocates memory to hold a data object of type data_type and returns the address of the object:

      ```
      pointer_variable = new data_type;
      pointer_variable = new data_type(value);
      pointer_variable = new data_type[size];
      ```

        - when creating multi-dimensional arrays with `new`, **all the array sizes must be supplied**. The **first dimension may be a variable** whose value is supplied at runtime:

          ```
          int *p = new int[3][4][5];
          int *p = new int[m][4][5]; // Both available
          ```

        - If sufficient memory is not available for allocation, `new` returns a **null pointer** (the value of a **null pointer is 0**)
        - It automatically **computes the size** of the data object.
        - It automatically returns the **correct pointer type**.
        - It is possible to **initialize the object** while creating the memory space.
    - Operator `delete` releases the memory space allocated by `new` for reuse:

      ```
      delete pointer_variable;
      delete []pointer_variable;
      ```

        - The operand must be a pointer returned by `new`
        - `delete` assign **NULL** to the pointer after release the memory pointed by the pointer
    - `new` and `delete` can be **overloaded**
- **Manipulators（控制符）**
    - Many manipulators, user can **define his own manipulators**
        - `endl`: cause a linefeed to be inserted
        - `setw`: specify a common field width for display
    - Example:

      ```
      #include <iomanip>
      #include <iostream>

      using namespace std;

      int main(){
        int Basic = 950, Allowance = 95, Total = 1045;
        cout << setw(10) << "Basic" << setw(10) << Basic << endl
            << setw(10) << "Allowance" << setw(10) << Allowance << endl;
        return 0;
      }
      ```

      Output is **right aligned**:

      ```
           Basic       950
       Allowance        95
      ```

- **Type conversions（类型转换）**
  - Use `typedef` to create an identifier of a type

    ```
    typedef type_name alias

    // Example
    typedef int INT;
    typedef double (*DblArrPtr)[10];

    INT anInt; // Type: INT (aka int)
    DblArrPtr dblArr; // Type: DblArrPtr (aka double (*)[10])
    ```

  - **Implicit conversions（隐式转换）**
    - Expressions with operands of different data type： **"smaller" type converts to the "wider" type**
    - Float point value to an int variable: **discard the fractional part**
    - Integer value to a float variable, **save in exponential form without changing the value**
    - `int`、 `short`、 `long` value to `char` variable, only send **the value saved in the lower 8 bit**
    - `char`、 `short` ⟶ `int` ⟶ `unsigned` ⟶ `long` ⟶ `double` ⟵ `float`
    - It will happen when **actual arguments（实参）** pass value to **formal arguments（形参）**
  - **Explicit type conversion（显式转换）**
    - Explicit type conversion using type cast operator
      - C notation: `(type_name) expression` or `(type_name) (expression)`
      - C++ notation: `type-name (expression)`
    - The functional-call notation can not be used if the type is compound:

      ```
      void *q;
      int *p;

      p = int *(q); // Err
      p = (int *)q; // OK
      ```

    - New **cast operators** added in C++
      - **const_cast**: `const_cast<type_name>(expression)`
        - removing the const_ness of an object
      - **static_cast**:
        - Conversion between build-in data types
        - Convertion from derived class object pointer or reference to base class object pointer or reference
        - Cannot do conversion between **independent types**
          - Example:

            ```
            float fnum = 5.8;
            int inum = static_cast<int>(fnum); // OK

            float *pf = &fnum;
            int *n = static_cast<int*>(&fnum); // Err, conversion betweeen independent types

            void *p = &fnum;
            float *dp = static_cast<float*>(p); // OK, find back the value stored in void* pointer
            ```

      - **dynamic_cast**: 安全的父类指针或引用和子类指针或引用之间的转换（will be mentioned in OOP lecture）
      - **reinterpret_cast**: 将一个指针类型转换成其他指针类型，将一个整数类型转换成指针类型或将指针类型转换成整数类型。常用于在函数指针类型之间进行转换。

        - Example:

```
int num;
int *pnum = &num;

char *pc = reinterpret_cast<char *>(pnum);
```

- **Expressions（表达式）**
    - **Arithmetic expressions（算术表达式）**

Annotation	Description
++	Unary plus
--	Unary minus
+	Addition
-	Subtraction
*	Multiplication
/	Floating-point division (floating-point result) Integer division (No fraction part, gives only the integer quotient（商）
%	Modulus (Used only with integers)

    - **Assignment expressions（赋值表达式）**
        - Assignment expression has a value, the value is saved in a temporary variable and can be assigned to another variable.
        - Example:

```
int y = 10;
int x = (y = 10); // OK
x = y = 100;
float a = b = 12.3; // Err, b must be defined previously
x = (y = 50) + 10;
```

    - **Bool Expressions（布尔表达式）**
        - Relational expressions: <, >, <=, >=, ==, !=
        - Logical expressions: &&, ||, !
    - **Bitwise expressions（位运算表达式）**
    - **Precedence（优先级）**
        - arithmetic > relational > logical > assignment
        - The **unary operations** assume higher precedence
- **Control structures（控制结构）**
    - if-else
    - switch-case
    - while
    - do-while
    - for
    - break and continue
    - go to
    - return
    - try-catch

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 4 Functions in C++

- **Const arguments（常量参数）**：
  - When passing arguments by pointers or reference, to avoid changing the value of real argument by the function, declare an argument as **const**

    ```cpp
    int length(const int &a);
    ```

- **Function overloading（函数重载）**：
  - Use the same function name to create functions that performs a variety of different tasks

    ```cpp
    int volume(int);

    double volume(double, int);

    long volume(long, int, int);
    ```

  - The correct function to be invoked is determined by **checking the number and type of the arguments**, but **not on return type**:
    - First try to **find an exact match** in which the types of arguments are the same
    - If an exact match is not found, try the **integral promotions（整型提升）** such as:
      - `char` to `int`
      - `float` to `double` // 这个似乎不是**整型**提升，但被标注在ppt中
    - If both fails, try the **implicit assignment conversion**. If the conversion is possible to have **multiple matches** the compiler will generate an **error** massage.
    - Example:

    ```cpp
    double abs(double num) {
      return ((num < 0) ? -num : num);
    }

    long abs(long num) {
       return ((num < 0) ? -num : num);
    }

    int main(){
      abs(10); // Passed as num (converted to long)
    }
    ```

- **Default arguments（默认/缺省参数）**
  - Allow to call a function without specifying all the arguments. Normally a function can contain much more arguments than commonly needed.
  - Default values are specified when the function is **declared**:

```
void func(int num=10);
```

        - `num` : default argument
        - `0` : default value
        - two ways in calling the function : `fun();` and `fun(23);`
    - We must add default **from right to left**:

```
void f(int i, int j=2, int k=5); // OK

void f(int i=2, int j, int k); // Err

void f(int i=0, int j, int k=2); // Err
```

    - When overloading a function with default arguments, pay attention to the **ambiguity problem （二义性）**
- **Inline functions（内联函数）**
    - An inline function is a function that is **expanded in line** when it is invoked.
    - Eliminate the cost of calls to small functions.
    - The inline functions are defined as follows:

```
inline function_header{
  function_body
}
```

    - Difference between inline function and predefined macro:
        - inline:

```
#include <iostream>

using namespace std;

inline double cube(double a){
  return (a*a*a);
}

int main(){
  cout << cube(3.0) << '\n'
      << cube(1.5+2.5) << endl;
  return 0;
}
```

          The output is:

```
27
64
```

        - macro:

```
#include <iostream>

using namespace std;

#define cube(a) (a*a*a)

int main(){
  cout << cube(3.0) << '\n'
      << cube(1.5+2.5) << endl;
  return 0;
}
```

The output is:

```
27
11.5
```

However, although we add branket to macro like `#define cube(a) ((a)*(a)*(a))`, it **still can be cracked** if we use the code `cube(i++)`

- If an inline function is **too long or too complicated**, the complier may compile the function **as a normal function**.
  - a loop, a `switch` or a `goto` exists
  - contain `static` variables
  - recursive function
- `inline` is a suggestion to compiler

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 5 Classes and Objects

- **C Structures（C结构体）**
  - It is a **user defined data** type with a **template**

```
struct student{
    char name[20];
    int roll_number;
    float total_mark;
}

struct student stuA;

strcpy(stuA.name, "John");
stuA.roll_number = 999;
```

  - In C, a **struct** models what a thing has/is (i.e., the **data**, also called the **characteristics**), but not what it does (its **behavior**, represented by **functions**)

        - The functions are **outside** and separated from structs
- **How struct Becomes in C++（C++中的结构体）**
    - **First Step**: Put the functions **inside**

      ```
      struct stack{
          int data[100];
          int top;
          void push(int a); // implement outside
          int pop(void); // implement outside
          bool isEmpty(void); // implement outside
      }
      ```

        - In C++, the **characteristics** and **behavior** are **integrated into a single structure**, called a **class**
            - Indeed, C++ has a new reserved word, **class**
        - Any variable of the type defined by **struct** or **class** is called an **object** or **instance** of that class
        - The packaging of the data and the functions into a class type is called **data encapsulation（数据封装）**
        - In C++, the declared variables and functions inside structs/classes are called **members**:
            - member variables
            - member functions (also called **methods**)
    - **Second Step**: data hiding
        - C++, and other **object-oriented programming** languages, allow the programmer to designate certain members of a class as **private**, and other members as **public**.
            - **Private** members cannot be accessed from outside the class, while **public** members can
            - Private members are hidden (thus the term **data hiding**)
- **Specifying a Class（指定一个类）**
    - **Class declaration**:

      ```
      class class_name{
          private:
              member declarations (data + functions);
          public:
              member declarations (data + functions);
      }; // End with a semicolon
      ```

    - **struct**, **union**, **class** all can be used to define a class:
        - **struct**: by default, all members are **public**
        - **union**: all members are **public** and **can not change the visibility**
        - **class**: by default all memmbers are **private**
    - **Defining member functions**:
        - Outside the class definition:

          ```
          return_type class_name::function_name(parameters){
            function body
          }
          ```

        - Inside the class definition:

```
class Item{
  int number;
  float cost;
public:
  void getdata(int a, float b){
      number = a;
      cost = b;
  }
  void putdata(void){
      cout << "number=" << number << ' ' << "cost=" << cost << endl;
  }
};
```

      - When a function is **defined inside a class**, it is treated as an **inline function**
- **Accessing class members（访问类成员）**
  - Inside the class, access directly
  - Outside of the class, only **public** members can be accessed
- **Characteristics of member functions（成员函数特性）**
  - Different classes can use the **same function name**
  - Member functions can access the **private** data of the class
  - A member function can call other member functions directly
    - A **private member function** can only be called by an other member function of the same class
- **Memory allocation for objects（对象内存分配）**
  - Memory of methods created when function defined
    - All objects share one
  - Memory of data created when objects defined
    - Every object has its own data
- `this` **Pointer**
  - For every non-static method in class:

    ```
    class t{
      private:
        int x, y;
      public:
        void set(int a, int b){
            x = a;
            y = b;
        }
    };
    ```

    is equivalent to:

    ```
    class t{
      private:
        int x, y;
      public:
        void set(int a, int b, t* const this){
            this->x = a;
            this->y = b;
        }
    };
    ```

  - `this` pointer points to the object by which a member function is called

- The pointer `this` acts as an **implicit** argument to all the non-static member functions
- When an object of a class is created this pointer is initialized to point to the object
- `this` pointer is a **const** pointer, the value of it cannot be altered:

  ```
  t * const this;
  ```

- `this` pointer can be used explicitly:

  ```
  void set(int a,int b) {
      this->x = a;
      this->y = b;
  }
  ```

- `this` pointer also can be return **（返回this指向的值的引用，实现成员函数的链式调用）**：

  ```
  # include <iostream>

  using namespace std;

  class t{
    public:
     t& set(int a, int b){
      x = a;
      y = b;
      return *this;
     }

     t& print(){
      cout << x << ', ' << y << endl;
      return *this;
     }

    private:
      int x, y;
  };

  void main(){
      t t1;
      t1.set.(10, 20).print().set(30, 40);
  }
  ```

- **static data members（静态数据成员）**
  - Using keyword `static` to declare a data member as **static**
    - static data member is **shared by all the objects** of that class, no matter how many objects are created
    - A static data member should be **initialized outside the declaration of a class**
    - It is **visible only within the class**, but its **lifetime is the entire program**
  - Static data members belong to the **class** instead of **objects**
  - only public static data members can be accessed from outside of the class as:

    ```
    class_name::public static data memeber;
    ```

    - Notice: Use `class_name` to access static data member
  - From inside the class, all the static data members can be accessed directly
  - static data members should be initialized outside the class:

```
type class_name::static_data_name = initial_value;
```

- **static member functions（静态成员函数）**
  - Static member functions are used to **access static member variables**
  - Static member functions have **NO** `this` pointer, it cannot access object's non-static variable directly
  - Static member functions can be called in following form:

    ```
    class_name::static_function_name(arguments_list);
    object_name::static_function_name(arguments_list);
    ```

- **friendly functions（友元函数）**
  - To make an outside function **"friendly"** to a class:

    ```
    class X{
        int i;
        friend void func(X*, int); // friendly function
      public:
        void memeber_func(int);
    };
    ```

    - `func()` is **NOT** the member function of `class X`
    - `func()` can be defined elsewhere in the program like a normal C++ function
    - The definition of the `func()` does not use either the **keyword** `friend` or the **scope operator** `::`
    - `func()` can access **private** members of the `class X`
    - `func()` **cannot** access member names directly:

      ```
      void func(X* xptr, int a){
        xptr-> i = a;
      }
      ```

    - Member functions of one class can be **friend functions of another class**:

      ```
      class X{
        ...
        public:
          void func();
        ...
      };

      class Y{
        ...
        public:
          friend void X::func();
        ...
      }
      ```

    - Declare the `class Y` to be a **friend class（友元类）** of the `class X`, then **all the member functions** of class Y are friend functions of the `class X`
      - Friendly functions are **one-way（单向）**

```
class A{
    friend class B;
    int x;
  public:
    void display(){
      cout << x << endl;
    }
};

class B{
  public:
    void set(int i){
      a.x = i;
    }
    void display(){
      a.display();
    }
  private:
    A a;
}
```

- **Pointers to Members（成员指针）**
  - It is possible to take the address of a **non-static** member of a class and assign it to a pointer:

```
class circle{
  public:
    int radius;
    void setradius(int);
};

int circle::*pint;
pint = &circle::radius;

circle c;
c.radius = 10; // OK
c.*pint = 10; // OK

circle *pc = &c;
pc->radius = 20; // OK
pc->*pint = 20; // OK

pint = &c.radius; // Err
int* ip = &c.radius; // OK
```

  - **Pointers to Member Functions（指向成员函数的指针）**：
    - Syntex: `data_type (class_name::*variable_name)(arglist);`
    - eg(using above circle class):

```
void (circle::*pmf)(int) = &circle::setradius;

c.setradius(10);
(c.*pmf)(10);

pc->setradius(10);
(pc->*pmf)(10);
```

        - Why use pointers to member functions: **Polymorphism（多态）**

```
class screen{
  public:
    screen& home();
    screen& forward();
    screen& back();
    screen& up();
    screen& down();
};

screen& move(screen &obj, screen &(screen::*pmf)()){
  (obj.*pmf)();
}

screen obj;
screen &(screen::*pmf)();

pmf = &screen::home;
move(obj, pmf);
pmf = &screen::forward;
move(obj, pmf);
```

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 6 Constructors and Destructors

- **Constructors（构造函数）**
    - Enable an object to initialize itself when it is created
    - A special member function to initialize the objects of its class
        - **No return value（无返回值）**
        - The function **name is the same as the class（函数名与类名相同）**
        - Should be declared in the **public section（在公有段定义）**：
            - Defined in the **private section**, you can implement **static classes** that cannot create instances.
        - Invoked automatically when the objects are created
        - Allocate memory space for the non-static data members of an object
        - Initialize part or all data members of an object
    - **Multiple constructors（多个构造函数）** in a class
        - When we do **not implement the constructor ourselves**, the compiler will automatically implement a **parameterless constructor** for us
            - If we implement any constructor, this parameterless constructor will **not automatically generated（自动生成的无参构造函数只会在开发者没有自己实现构造函数时生成）**
        - **Overloaded constructors（重载构造函数）**：argument list must be different

```
class X{
  public:
    X();
    X(int);
    X(int, char);
    X(float, char);
  private:
    int m;
    char c;
};

void f(){
  X a;  // invoke construtor X()
  X b(1); // invoke construtor X(int)
  X c(1, 'c');  // invoke construtor X(int, char)
  X d(2.3, 'd');  // invoke construtor X(float, char)
}
```

- **Default argument constructor** and **default constructor**. If a class has a constructor with all arguments have default values, it is **illegal** to overload a default constructor for the class

```
class X{
  public:
    X();
    X(int i = 0);
};

int main(){
  X x; // ambiguity, call X::X() or X::X(int i = 0) ?
}
```

- **Initializer List（初始化列表）**
  - Some special data members (e.g.**constant members**, **reference members** etc.) **cannot** be initialized with assignment statements in the constructor function body:

```
class X{
    const int a;
    const int &r;
  public:
    X(){
      a = 9; // Err
      r = a; // Err
    }
};
```

  - To achieve the above requirement, we need initializer list:

`constructor_name(arglist): member_name(expression), ...{ function_body }`

```
class X{
    const int a;
    const int &r;
  public:
    X(): a(9), r(a){}
};
```

  - Usage of initializer list:

        - Initialize normal data members
        - Initialize **object data members（初始化对象数据成员）**

          ```
          class studentID {
            public:
              studentID(int d) {
                value = d;
                cout << "Assigning student id " << value << endl;
              }

            protected:
              int value;
          };

          class student {
            public:
              student(char *pname = "no name", int ssID = 0){
                // Constructor for 'student' must explicitly initialize
                // the member 'id' which does not have a default constructor
                id = studentID(ssID); // Err
                cout << "Constructing student" << pname << endl;
                name = pname;
              }

            protected:
              char *name;
              studentID id;
          };
          ```

        - Initialize **constant data members（常量初始化）** and **reference data members（引用初始化）**
  - **Default constructor（默认构造函数）**
    - If **no constructor is defined**, the complier supplies a default constructor
    - Default constructor:
      - **Object member（对象成员）**: initialize with the default constructor of its own class（仅使用这些对象成员的默认构造函数初始化）
      - **Data members of build-in or compound data type（内建数据类型或复合数据类型/结构体）**: only initialize global object
    - Classes with **constant member** and **reference member** cannot use the default constructor provided by the complier（具有常量或引用数据成员的类，不能使用默认构造函数）
  - **Arrays of Objects（对象数组）**
    - Arrays of variables that are of the type **class**
    - Arrays of objects **cannot be initialized** unless the class have:
      - no constructor
      - a default constructor
      - a constructor with all parameters have default value
  - **Dynamic Initialization of Objects（对象动态初始化）**
    - The initial value of an object may be provided during run time

```
class teacher{
  private:
    int *id;
  public:
    teacher(int input_id){
      *id = input_id; // Segmentation Fault, the variable *id wasn't allocated memory space.
      id = new int(input_id); // OK
    }
};
```

- **Destructors（析构函数）**
  - Destroy the objects when they are **out of scope（离开作用域）** to clean up storage
  - The destructor is a member function whose name is `~class_name`, it take no argument nor does it return any value
  - When objects are out of scope, compiler **invokes destructor automatically**
  - If no destructor is defined, the **complier** supplies a **default destructor**
  - Constructors and destructors are automatically called in **reverse order**:

```
class X{
  ...
};

void func(int a){
  X aa; // Call constructor for aa
  X bb; // Call constructor for bb
  if (a > 0){
    X cc; // Call constructor for cc
    ...
  } // Destroy cc
  X dd; // Call constructor for dd

  // Destroy dd
  // Destroy bb
  // Destroy aa
}
```

  - Whenever `new` is used to allocate memory in the constructors, we should use `delete` to free that memory:

```
class teacher{
  private:
    int *id;
  public:
    teacher(int input_id){
      id = new int(input_id);
    }

    ~teacher(){
      delete id;
    }
    void show();
};
```

- **Copy constructor（拷贝构造函数）**
  - Copy constructor is a constructor

- The argument of the copy constructor is a **reference to its own class（引用作为参数，直接读取被拷贝对象的内容，一般还加 `const` 修饰，使值仅可读）**

  ```cpp
  class point{
    private:
      int x, y;
    public:
      point(int intx = 0, int inty = 0){
        x = intx;
        y = inty;
      }
      point(const point& pt){
        x = pt.x;
        y = pt.y;
      }
  };
  ```

- If not copy constructor is defined, **compiler** supplies a **default copy constructor**
- **Default copy constructor** assigns the values of one object to another object **member by member**
- When the copy constructor will be used: **（3种拷贝构造函数被调用的情况）**
  - Declare and initialize an object from another object **（初始化对象）**
  - Pass value to an object argument while calling a function **（函数传参——值传递）**
  - When the return value of a function is an object, calling copy constructor to copy the returned object to a temporary object **（函数返回类对象时会创建一个对象拷贝）**

    ```cpp
    class location{
      private:
        int X, Y;
      public:
        location(int xx = 0, int yy = 0){
          X = xx;
          Y = yy;
          cout << "Obj constructed" << endl;
        }
        location(const location& p){
          X = p.X;
          Y = p.Y;
          cout << "Copy constructor called" << endl;
        }
        ~location(){
          cout << X << "," << Y << "Obj destroyed" << endl;
        }
    };

    location g(){
      location A(1, 2);
      return A;
    }

    int main(){
      location B;
      B = g();
      return 0;
    }
    ```

The output of this program is as follows:

```
Obj constructed            // Create object B
Obj constructed            // Create local object A
Copy constructor called    // Initialize temporary object for return value
1 , 2 Obj destroyed        // Delete temporary object
1 , 2 Obj destroyed        // Delete local object A
1 , 2 Obj destroyed        // Delete object B
```

- For some object, we **cannot pass values by simply assigning values member by member** (i.e. cannot use the default copy constructor provided by the complier).

```
class c{
  private:
    int *p;
  public:
    c(){
      p = new int(0);
    }
    c(const c& obj){
      *p = obj.*p;
    }
};
```

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 7 Operator Overloading

- **Operator Overloading（运算符重载）**
  - Operator overloading provides concise notation:

```
c = c1.Add(c2);
c = c1 + c2;
```

    - It is for the code involving your class **easier to write** and especially **easier to read**.
    - All the operators used in expressions that contain only **built-in data types cannot be changed（内建数据类型不能被重载运算符）**. Only an expression **containing a user-defined type** can have an **overloaded operator**.
- **Syntax（语法）**
  - Operator functions must be either:
    - member functions
    - friend functions
  - Overloading operators using member function, the syntax is:

```
type X::operator op(arglist){
    ...
}
```

- Example:

```
class complex{
    double re, im;
  public:
    complex(double r = 0, double i = 0) : re(r), im(i){}
    complex operator+ (complex);
};

complex complex::operator+(complex cobj){
    complex temp;
    temp.re = re + cobj.re;
    temp.im = im + cobj.im;
    return temp;
}

int main(){
    complex obj1(2, 3), obj2(3, 4);
    complex obj3;

    obj3 = obj1.operator+(obj2); // Invoking the function
    obj3 = obj1 + obj2; // Another way to invoking the function
}
```

- Overloading operators using friends:

```
friend type operator op(arglist){
    ...
}
```

- Example:

```
class complex{
    double re, im;
  public:
    complex(double r = 0, double i = 0) : re(r), im(i){}
    friend complex operator+(complex, complex);
};

complex operator+(complex a, complex b){
  return complex((a.re + b.re), (a.im + b.im));
}
```

- **Restrictions on Operator Overloading（运算符重载的限制）**
  - Cannot change
    - **How operators act on built-in data types（内建数据类型的操作符操作）**, e.g. cannot change integer addition
    - **Precedence（运算符优先级）** of operator (order of evaluation)
    - **Associativity（运算符结合性）** (left-to-right or right-to-left)
    - **Number of operands（操作数的个数）**, e.g. `&` is unary, only acts on one operand
  - Cannot create new operators
  - Operators must be overloaded explicitly, e.g. overloading `+` does not overload `+=`

- **Operators not Allowing Overloaded（不允许运算符重载的运算符）**
  - `.` member selection **（成员选择运算符）**
  - `.*` member selection by a pointer **（通过指针取值的成员选择运算符）**
  - `::` scope resolution **（作用域解析运算符）**
  - `?:` ternary conditional expression **（三元条件运算符）**
  - `sizeof`
- **Operators only overloaded with member functions（只能通过成员函数进行重载的运算符）**
  - `=` Assignment operator **（赋值运算符）**
  - `()` Function call operator **（函数调用运算符）**
  - `[]` Subscripting operator **（下标操作运算符）**
  - `->` Class member access operator **（类成员访问运算符）**
- **Operator Functions As Class Members Vs. As Friend Functions（比较采用成员函数或友元函数进行运算符重载）**
  - Member functions:
    - Use `this` pointer to implicitly get **left operand（左值/左操作数）** for binary operators (like `+`)
    - Leftmost object must be of same class as operator **（最左边的对象必须与运算符的类保持一致）**
  - Friend functions:
    - Need parameters for **both operands（左右值都必须为函数参数）**
    - Can have objects of **different classes** as **operands （操作数可以是不同类）**
    - Must be a `friend` to access `private` or `protected` data
- **Overloading `++` and `--`**
  - There are two ways in using `++` and `--`:
    - Prefix(`++aa` and `--aa`):
      - Overload using **member function**: `aa.operator++();`
      - Overload using **friend function**: `operator++(X &aa);`
    - Postfix(`aa++` and `aa--`):
      - Overload using **member function**: `aa.operator++(int);`
      - Overload using **friend function**: `operator++(X &aa, int);`
  - The `int` argument is used to indicate that the function is to be invoked for **postfix** application of `++` or `--`. This `int` is **never used**; the argument is simply a **dummy** used to **distinguish between prefix and postfix**.
    - For example, `i++;` is equivalent to `i++ = 0;`
  - Example:

```cpp
class X{
  private:
    unsigned int value;
  public:
    X(){
      value = 0;
    }
    X & operator++(); // prefix
    X operator++(int); // postfix
};

X & X::operator++(){ // prefix
  value++;
  return *this;
}

X X::operator++(int i){ // postfix
  X temp;
  X.value = value++;
  return temp; // return the unchanged value
}

void f(X a){
  ++a;
  a++;
  a.operator++(); // Explicit call, ++a
  a.operator++(0); // Explicit call, a++
}
```

- Overload using friend function:

```cpp
class Y{
  private:
    unsigned int value;
  public:
    Y(){
      value = 0;
    }
    friend Y & operator++(Y&); // prefix -> ++a;
    friend Y operator++(Y&, int); // postfix -> a++(0);
};

Y & operator++(Y &a){ // prefix
  a.value++;
  return a;
}

Y operator++(Y &a, int i){ // postfix
  Y temp(a);
  a.value++;
  return temp; // return the unchanged value
}
```

- **Overload assignment operator（赋值运算符重载）**
  - C++ will give every class a **default assignment**
  - Returns a **reference** and can be **called in a chain** **（返回引用以实现链式调用）**

```
X & X::operator=(const X &from){
  ...
}
```

- When shall we need define an assignment? **Pointers are data members** of a class

```
class pointer{
  private:
    int *p;
  public:
    pointer(int x){
      p = new int(x);
    }
    ~pointer(){
      delete p;
    }
    pointer & operator=(const pointer & obj){
      if (*this != &obj){ // Judge the condition of "p = p"
        *p = *obj.p;
      }
      return *this;
    }
};

int main(){
  pointer p1(10), p2(20);
  p2 = p1; // Error if we didn't overload = operator
  return 0;
}
```

- **Istream `>>` and Ostream `<<`**
  - Class **istream** overloaded the operator `>>` as **input operator**, `cin` is a reference of class istream
    - `inline istream& istream::operator>>(unsigned char & _c)`
    - Implicit calling: `cin >> a;`
    - Explicit calling: `cin.operator >> (a);`
  - Class **ostream** overloaded the operator `<<` as **output operator**, `cout` is a reference of ostream
    - `inline ostream& ostream::operator<<(unsigned char & _c)`
    - Implicit calling: `cout << a;`
    - Explicit calling: `cout.operator << (a);`
  - To use `<<` and `>>` to output or input **user defined data** types directly, the **operators must be overloaded （用户定义的数据类型想使用上述运算符来输入输出，必须自己重载运算符）**
    - `<<` and `>>` can only be overloaded as **friend** functions **（必须以友元函数形式重载）**

      ```
      friend ostream & operator<<(ostream & os, const Class_name & obj);

      friend istream & operator>>(istream & is, Class_name & obj);
      ```

    - Example:

```
class complex{
    double re, im;
  public:
    complex(double r, double i = 0) : re(r), im(i){}
    complex(){
      re = 0;
      im = 0;
    }
    friend ostream & operator<<(ostream &os, const complex &c);

    // No constant because input will change the value of c
    friend istream & operator>>(istream &is, complex &c);
};

ostream & operator<<(ostream &os, const complex & c){
  os << c.re;
  if (c.im > 0)
    os << "+" << c.im << "i" << endl;
  else
    os << c.im << "i" << endl;
  return os;
}

istream & operator>>(istream &is, complex &c){
  is >> c.re >> c.im;
  return is;
}
```

- **Type Conversions（类型转换）**
  - Conversion from **basic type to class type（基本类型转换为类对象）**
    - **Conversion Constructor（转换构造器）** perform a type conversion from the argument's type to the constructor's class type
    - For a class `X`:

      ```
      X::X(V, V1 = E1, V2 = E2, ...);
      ```

      It can be used for conversion from type of argument `V` to `X` type. A more detailed example is as follows:

      ```
      class X{
        public:
          X(int);
          X(const char*, int = 0);
      };

      void f(X arg);

      int main(){
        X a = 1;         // X a(1);
        X b = "Jessie"; // X b("Jessie");
        a = 2;           // a = X(2);
        f(3);            // f(X(3));
        f(1.5);          // Error
        return 0;
      }
      ```

- Conversion from **class type to basic type（类对象转换为基本类型）**
  - Conversion function **must be member function（必须是成员函数，不能是友元函数）**：

    ```
    X::operator typename(){
      ...
      return typename variable;
    }
    ```

  - `typename` can be a build in data type or user defined data type
  - The function has **no argument**, **no return type**, but **must contain a return statement**
  - example:

    ```
    class complex{
      private:
        double re;
        double im;
      public:
        complex(){
          re = 0;
          im = 0;
        }
        complex(double re, double im){
          this->re = re;
          this->im = im;
        }
        operator double(){
          return re;
        }
    };

    int main(){
      complex obj(3.0, 3.0);
      double x, y;
      x = obj; // x = (double)obj;
      return 0;
    }
    ```

- Conversion from **class type to class type（类对象之间转换，注意实现为单向）**
  - Conversions between objects of different classes can be carried out by:
    - **Conversion constructor（转换构造函数）**：`X::X(Y)`
    - **Conversion function（转换函数）**：`X::operator Y()`
  - Be cautious when doing conversions between objects of different classes:

```
struct Y;

struct X {
  int i;
  X(int);
  X operator+(Y);
};

struct Y {
  int I;
  Y(X);
  Y operator+(X);
  operator int();
  friend X operator*(X, Y);
};

int main() {
  X x = 1;
  Y y = x;
  int i = 2;
  int ret = i + 10;

  // With operand types 'Y' and 'int'
  ret = y + 10;      // Use of overloaded operator '+' is ambiguous

  // With operand types 'int' and 'Y'
  ret = y + 10 * y; // Use of overloaded operator '*' is ambiguous

  // 'X' and 'int'
  ret = x + y + i;  // Invalid operands to binary expression

  ret = x * x;      // Invalid operands to binary expression
  return 0;
}
```

- There are certain situations where we would like to **use friend function rather than a member function** to overload operators:

```cpp
class complex {
  int Real;
  int Imag;

public:
  complex(int a) {
    Real = a;
    Imag = 0;
  }
  complex(int a, int b) {
    Real = a;
    Imag = b;
  }
  complex operator+(complex);
};

int f() {
  complex z(2, 3), k(3, 4);
  z = z + 27;
  z = 27 + z;
}
```

- The expression `z + 27` can be explained as `z.operator+(27)`, since the argument of the overloading function is of the type of class `complex`, the real argument `27` will be converted to the type of class `complex` implicitly by using the constructor `complex(int a)`
- However, the expression `27 + z` can be explained as `27.operator+(z)`, which is **meaningless**, since `27` is not an object of the `complex` class, it cannot invoke the member function of class `complex`.
- Finally we use friend function to fix this error:

```cpp
friend complex operator+(complex, complex);
```

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 8 Inheritance: Extending Classes

- **Inheritance（继承） a.k.a. derivation（派生）**
  - Mechanism of creating a new class from an old one. The new class **inherits all of the traits** from the existing one.
  - The old class: **base class（基类）** or **parent class（父类）**
  - The new one: **derived class（派生类）** or **subclass （子类）**
  - Inheritance is able to define new classes of objects using existing classes as a base
    - The new class inherits **the attributes（属性，即数据成员）** and **behaviors（行为，即方法/函数）** of the parent classes
    - New class is a specialized version of the parent class

- **Reusability（可重用性）**
  - Save time and money
  - Reduce frustration
  - Increase reliability
- **Class Derivation（派生类）**
  - Syntax:

    ```
    class derived_class_name: visibility_mode base_class_name{
        ...
    };
    ```

    - **visibility_mode（可见模式）**: specifies access to the base class members, `public`, `protected`, `private`, and `private` **by default**
  - For `public` derivation `class C: public A`:
    - The inherited **public** members of `A` appear **as public** members of `C`
    - The inherited **protect** members of `A` appear **as protect** members of `C`
    - The inherited **private and unaccessible** members of `A` appear **as unaccessible** to `C`
  - For `private` derivation `class C: private B`:
    - The inherited **public** members of `B` appear **as private** members of `C`
    - The inherited **protect** members of `B` appear **as private** members of `C`
    - The inherited **private and unaccessible** members of `B` appear **as unaccessible** to `C`
  - For `protected` derivation `class C: protected B`:
    - The inherited **public** members of `B` appear **as protect** members of `C`
    - The inherited **protect** members of `B` appear **as protect** members of `C`
    - The inherited **private and unaccessible** members of `B` appear **as unaccessible** to `C`
  - **Private vs. Protected Members（比较继承的私有和保护成员）**
    - Inherited **private** members **CANNOT** be **accessed directly by name** in derived class
    - Inherited **protected** members **can** be **accessed directly by name** in derived class
- **Not "All" Members Inherited（并非所有成员都会被继承）**
  - **Base class members（基本类成员）** not inherited in derived class:
    - Constructors
    - Destructor
    - Copy constructor
    - Assignment operator
  - Sometimes need to be invoked in derived class **except for destructors（除了析构函数外有时需要在派生类内调用父类的基本类成员）**
    - Destructors are always automatically invoked
  - **Base class constructors（基类构造函数）** are **NOT inherited** in derived classes! But they can be invoked within derived class constructor
  - Base class constructor initialize base class member variables
    - **Default constructor** of direct base class is called automatically by derived class constructor
    - Parameter constructors of direct base class are called by including them in the **initializer list（在初始化列表中直接调用基类的构造函数）**
    - **"First" thing derived class constructor** does
    - Derived class constructor can **not** call **indirect base class constructor（不能调用非直接继承类的构造函数）**
  - Example:

```cpp
#include <iostream>

using namespace std;

class A {
  int a;

public:
  A() { cout << "A" << endl; }
  A(int sa) {
    a = sa;
    cout << "A" << endl;
  }
  ~A() { cout << "~A" << endl; }
};

class B : public A {
  int b;

public:
  B(int sa, int sb) : A(sa) {
    b = sb;
    cout << "B" << endl;
  }
  ~B() { cout << "~B" << endl; }
};

class C : public B {
  int c;

public:
  C(int sa, int sb, int sc) : B(sa, sb) {
    c = sc;
    cout << "C" << endl;
  }

  // Error, cannot invoke the indirect base class
  // C(int sa, int sb, int sc) : A(sa), B(sb) {
  //   c = sc;
  //   cout << "C" << endl;
  // }

  ~C() { cout << "~C" << endl; }
};

int main() {
  C objc(1, 2, 3);
  return 0;
}
```

Output:

```
A
B
C
~C
~B
~A
```

    - When calling multiple constructors in initializer list, the calling order **depends on the member declare order（初始化取决于类成员定义的顺序而不是初始化列表的顺序）** instead of initializer list order.
- **Constructor: No Base Class Call（不调用基类构造函数）**
    - Derived class constructor should **always invoke one of the base class's constructors（总是需要在派生类的构造函数中调用基类的构造函数）**
    - If you do not, then the **default base class constructor automatically called（如果不手动调用基类的构造函数，基类的默认构造函数会被调用）**
- **The "Big Three"（三大件）**
    - **Destructors（析构函数）**
        - Only needed when pointers and **dynamic memory allocation** are used
    - **Copy constructors（拷贝构造函数）**
        - NOT inherited, but can be used in derived class definitions, similar to how derived class constructor invokes base class constructor **（不被直接继承，但可以在派生类的定义中被使用，参见在派生类中调用基类构造函数的方式）**
        - Example:

        ```
        Derived::Derived(const Derived& Object): Base(Object), ... {
          ...
        }
        ```

    - **Assignment operators（赋值运算符）**
        - NOT inherited, same as copy constructor
        - Example:

        ```
        Derived& Derived::operator=(const Derived & rightSide) {
          if (this != &rightSide) //avoid self assignment
            Base::operator=(rightSide); // 先调用基类的赋值运算符
          ... // 接下来再编写派生类的赋值部分
        }
        ```

- **Redefinition（重定义）**
    - A derived class can redefine members defined in its parent class. With redefining:
        - the method in the child class **has the identical signature（拥有相同的函数签名：函数的名称、参数类型和参数个数的组合形式，不包括函数返回类型）** to the method in the base class
        - a child class **implements its own version** of a base class method
    - Example:

```cpp
#include <iostream>

using namespace std;

class Point {
  int x, y;

public:
  void set(int a, int b) {
    x = a;
    y = b;
  }
  void print(){
    cout << "Point: x = " << x << ", y = " << y << endl;
  }
};

class Circle : public Point {
  double r;

public:
  void set(int a, int b, double c) { // Overload
    Point::set(a, b);
    r = c;
  }
  void print(){ // Redefinition
    cout << "Circle: radius = " << r << endl;
  }
};
```

- **Redefining vs. Overloading（比较重定义与重载）**
  - **Redefining（重定义）** in derived class:
    - **SAME** parameter list **（同样的参数列表）**
    - Essentially **"re-writes" same function（重写了相同的函数）**
  - **Overloading（重载）**：
    - Different parameter list **（不同参数列表）**
    - Defined "new" function that takes different parameters
    - Overloaded functions must **have different signatures**

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 9 Polymorphism and Virtual Functions

- **Polymorphism（多态性）**
  - One name, multiple forms
  - In C++，it mean **different functions** with the **same function name**.

    - In OOP, it means **objects belonging to different classes（不同类的对象）** are able to **respond to the same message（回应相同的信息）**, but in different forms.
- **Function overloading（函数重载）** is a kind of polymorphism (**early binding（早绑定）** or **static binding（静态绑定，编译阶段确定）**).
    - Overload member functions in one class:

      ```
      show(int, char);
      show(char*, float);
      ```

    - Overload member functions of a base class in a derived class:
        - By matching arguments
        - Using `::`
- C++ supports a more flexible mechanism **virtual function（虚函数）** to achieve **run time polymorphism（运行时多态）**: i.e. select the appropriate member function while the program is running. The process is termed **late binding（迟绑定）** or **dynamic binding（动态绑定，运行时确定）**.
- Three questions, for public inheritance，can we:
  - [x] Assign a derived class object to a base class object?
  - [x] Use a base class object reference to refer to a derived class object?
  - [x] Use a base class object pointer to point to a derived class object?
- `class A` ← `class B`, suppose we have the following code part:

  ```
  A * p ; // pointer refer to class A

  A A_obj ;
  B B_obj ;

  p = & B_obj ; // p refer to object of class B
  ```

    - Using `p`, the `public` **members** of `B_obj` which are inherited **from** `class A` **can be accessed（只有基类`A`的成员可以被访问）**, but **NOT** the members defined by `class B` (unless explicitly cast `p` to `A` type)
    - Using pointer `p` to base class, no matter `p` refer to base class object or derived class, `object.p->func()` **always executes the function defined in the base class**.
    - To execute different version of the functions, we need to use objects explicitly:

      ```
      first_obj.func();
      second_obj.func();
      ```

- **Achieving Polymorphism（实现多态性）**
    - Run time polymorphism is achieved only when a **virtual function** is accessed through **a pointer to the base class**. **（指向基类对象的指针，其调用了基类的虚函数时会唤起多态性）**
    - The **prototypes** of the base class version of a virtual function and all the derived class versions must be **identical**. **（基类的虚函数和派生类的函数必须有相同的函数原型，这样才能启用多态性）**
    - if a virtual function is defined in the base class, it **need not be necessarily redefined in the derived class**. In such cases, calls will **invoke the base function**. **（基类的虚函数不一定需要在派生类中被实现，此时根据多态性，调用的会是基类的函数）**
    - Example:

```
class base {
public:
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  void f();
};

class derived : public base {
public:
  void vf1();     // virtual function
  void vf2(int); // function overloading
  char vf3();     // error
  void f();       // function overloading
};
```

- **Virtual Destructor（虚析构函数）**
  - Declare the destructor of a base class as virtual function, the destructors of **all the classes derived from the base class become virtual functions（注意：派生类继承的函数也全都变成虚函数，但不用加`virtual`关键字）**，although the names of the destructors are different from that of the base class!
  - Cases when the destructor must be virtual: in a class system derived from a base class, if **dynamic create object** is needed, the **destructor must be virtual**, to achieve polymorphism while deleting objects. **（需要动态创建对象时，若该对象所属的类是派生类，它的基类析构函数必须是虚函数，以此实现多态析构）**
  - **Normally the destructor of a base class is declared as virtual（通常析构函数被定义为虚函数，无论是否需要自析构，但这可以确保其派生类完成析构）**. Even when the base class do not need self defined destructor, define an empty virtual destructor, to make sure the derived object will be destroyed properly.
- **Pure Virtual Function（纯虚函数）**
  - Definition: A virtual function declared in a base class that **has no definition relative to the base class（没有定义函数体）**. Such functions are called "do–nothing" functions.
  - Pure virtual functions provide **public interfaces（公有接口）** for derived classes.
  - Syntax of declaring pure virtual function:

    ```
    class class_name{
        ...
        virtual type function_name(arglist) = 0;
    };
    ```

    there is **no function body**, but **not empty function body**. **（没有函数体的定义，但是函数体并不为空？有点绕，总之看上面的例子）**
    Assigning `0` to function name, is **equivalent to assign null to the pointer refers to the function body（与赋值`NULL`到指向函数体的函数指针是等价的）**. The function can not be invoked before it is redefined in the derived classes.
  - **Abstract class（抽象类）**: class which contains **at least one pure virtual functions**. **（类内包含至少一个纯虚函数）**
    - An abstract class is not used to create objects, it can only be used as base class. **（只用于创建基类，不用于创建对象）**
    - Abstract class can be used to declare pointers and references. **（用于声明指针和引用）**
    - Example:

```cpp
class point {
  /*...*/
};

class shape { // 抽象类
  point center;

public:
  point where() { return center; }
  void move(point p) {
    center = p;
    draw();
  }
  virtual void rotate(int) = 0; // 纯虚函数
  virtual void draw() = 0;      // 纯虚函数
};

class abs_circle: public shape {
  int radius;

public:
  void rotate(int) {};
  // abs_circle::draw() is still a pure virtual function if it is undefined in class abs_circle
  // therefore, class abs_circle is still an abstract class
}

int main() {
  shape x;           // error，抽象类不能建立对象
  shape *p;          // ok，可以声明抽象类的指针
  shape f();         // error，抽象类不能作为返回类型
  void g(shape);     // error，抽象类不能作为参数类型
  shape &h(shape &); // ok，可以声明抽象类的引用
  return 0;
}
```

- **Polymorphism Instance（多态实例）**

```cpp
#include <iostream>

using namespace std;

class Number {
public:
  Number(int i) {
    val = i ;
  }
  virtual void Show() = 0;

protected:
  int val;
};

class Hextype: public Number {
public:
  Hextype(int i): Number(i) {}
  void Show () {
    cout << hex << val << endl; // hex和下面的dec是后面涉及到的I/O库定义的函数
  }
};

class Dectype: public Number{
public:
  Dectype(int i): Number(i) {}
  void Show() {
    cout << dec << val << endl;
  }
};

void fun(Number &n) {
  n.Show();
}

int main() {
  Dectype d(50);
  fun(d); // d.Show();
  Hextype h(16);
  fun(h); // h.Show();
  return 0;
}
```

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 10 Console I/O

- C++ provides an alternative with the new **stream** input/output features for two reasons:
  - I/O methods in C++ **support the concept of oop（支持面向对象）**
  - I/O methods in C **cannot handle the user-defined data types（无法处理用户自定数据类型）**
- **C++ Streams（C++流）**
  - A transfer of information in the form of **a sequence of bytes（以字节序列传输数据）**
  - C++ Stream Classes:

$$\begin{tikzcd}
\text{iostream} \arrow[r] & \text{iostream\_withassign} & \text{iostream} \arrow[l] \\
\text{istream} \arrow[u] \arrow[d] & \text{ios} \arrow[l] \arrow[r] \arrow[d] & \text{ostream} \arrow[u] \arrow[d] \\
\text{istream\_withassign} & \text{streambuffer} & \text{ostream\_withassign}
\end{tikzcd}$$

  - **Pre-defined Streams（预定义流）**：

Object Name	Class	Device
`cin`	`istream_withassign`	standard input device(**keyboard**, can be redirected)
`cout`	`ostream_withassign`	standard output device(**screen**, can be redirected)
`cerr`	`ostream_withassign`	standard error output device(**screen**, **can not** be redirected)
`clog`	`ostream_withassign`	standard error output device(**screen**, **can not** be redirected)

    - `cin` 是 `istream` 的**派生类** `istream_withassign` 的对象
    - `cout` 是 `ostream` 的**派生类** `ostream_withassign` 的对象
  - Using the **header file `iostream`**:
    - Include `<iostream>` instead of `<stdio.h>`
    - Standard iostream objects:
      - `cout` - object providing a connection to the **monitor**
      - `cin` - object providing a connection to the **keyboard**
      - `cerr` - object providing a connection to **error stream**
  - The **Insertion** Operator (`<<`):
    - To send output to the **screen**
    - Format: `cout << Expression;`
    - The compiler figures out the type of the object and prints it out appropriately
  - The **Extraction** Operator (`>>`):
    - To get input from the **keyboard**
    - Format: `cin >> Variable;`
    - The compiler figures out the type of the variable and reads in the appropriate type
    - `cin` **ignores whitespaces（无视空白字符）** (spaces, tabs, newlines)
    - Returns **zero (`false`) when `EOF` is encountered（如果遇到 `EOF` 则返回 `false` ）**, otherwise **returns reference to the object from which it was invoked（否则返回自身的引用，实现连续调用）** (i.e. `cin`)
  - `get` and `getline` Member Functions
    - `cin.get(array, size, delimiter)`
      - Accepts 3 arguments: array of characters, the size limit (**character count（字符数）**), and a **delimiter（分隔符）** (default of `'\n'`)

- Input a sequence of characters from stream till the delimiter or `EOF` is encountered or `size-1` characters are read.
- Uses the array as a **buffer（缓冲区）**
- When the delimiter is encountered, it remains in the input stream, unless delimiter flushed from stream, it will stay there **（遇到分隔符时，分隔符会被保留在输入流中）**

```
#include <iostream>

using namespace std;

int main() {
  char *str1 = new char[100];
  char *str2 = new char[100];
  cin.get(str1, 10, ' ');
  cout << str1 << endl;
  // cin.get();
  cin.get(str2, 10, ' ');
  cout << str2 << endl;
  return 0;
}
```

Input & Output:

```
h3art hello // input
h3art       // output
            // output
```

- **Null character (** `'\0'` **)** is inserted into the array **at the end of the characters**

- `cin.getline(array, size)`
  - Operates like `cin.get(buffer, size)` but it **discards the delimiter from the stream（读取一行直到碰到分隔符或行结束，但此时会将分隔符丢弃，不再保留在输入流中）** and does not store it in array
  - Null character inserted into array

- `put` and `write` Member Functions
  - `cout.put(char)`
    - Outputs one character to specified stream
    - **Returns a reference to the object** that called it, so may be **cascaded（连续调用）**：

```
cout.put('A').put('\n');
```

    - May be called with an **ASCII-valued** expression:

```
cout.put(65); // = cout.put('A');
```

  - `cout.write(line, size)`
    - Outputs the entire line till size characters are displayed
    - the functions **will not terminate** at a **newline** character
    - the functions **will not terminate** at a **null** character **（输出流里存在** `'\n'` **或** `'\0'` **都不会终止其输出）**

- **Formatted Console I/O Operations（格式化控制台输出输出操作）**
  - `ios` class constains a large number of **member functions** to format the output:

Function	**Description**
`width()`	Specify the required field size

Function	Description
`precision()`	Specify the number of digits to be displayed after the decimal point
`fill()`	Specify a character that is used to fill the unused portion of field
`setf()`	Specify format flags
`unsetf()`	Clear the flags specified

- Setting the **Width**:
  - Use the `width(int)` function to set the width for printing a value, but it **only works for the next output command（只对下一个输出有效，默认为右对齐）**
  - Example:

    ```
    int x = 42;
    cout.width(5);
    cout << x << '\n';
    cout << x << '\n';
    ```

    Output:

    ```
       42
    42
    ```

- Setting **Precision**:
  - By default, the floating numbers are printed with **six** digits **（默认精度为6）**
  - Use the `precision(int)` function to specify the number of digits to be displayed
  - The setting **stays in effect** until it is **reset （持续有效）**
- Setting the **Fill** Character:
  - Use the `fill(char)` function to set the fill character.
  - The character **remains** as the fill character until set again. **（持续有效）**
  - Example:

    ```
    int x = 42;
    cout.width(5);
    cout.fill('*');
    cout << x << '\n';
    ```

    Output:

    ```
    ***42
    ```

- **Flags（标记位）**：
  - `ios` defined a **word (16 bits)** to control I/O format
  - Each bit represent one format:

Constant	Value	Meaning	I/O	Default	Bit-field
`ios::skipws`	0x0001	Skip white spaces	I		No
`ios::left`	0x0002	Left adjustfied	O	Not set	`ios::adjustfield`
`ios::right`	0x0004	Right adjustfied	O	Set	`ios::adjustfield`
`ios::internal`	0x0008	Internal adjustfied	O		`ios::adjustfield`

Constant	Value	Meaning	I/O	Default	Bit-field
`ios::dec`	0x0010	Decimal base	I/O	Set	`ios::basefield`
`ios::oct`	0x0020	Octal base	I/O	Not set	`ios::basefield`
`ios::hex`	0x0040	Hexadecimal base	I/O	Not set	`ios::basefield`
`ios::showbase`	0x0080	Show the base of an output number	O	Not set	No
`ios::showpoint`	0x0100	Show point	O	Not set	No
`ios::uppercase`	0x0200	Uppercase	O	Not set	No
`ios::showpos`	0x0400	Show positive	O	Not set	No
`ios::scientific`	0x0800	Scientific	O	Not set	`ios::floatfield`
`ios::fixed`	0x1000	Fixed	O	Not set	`ios::floatfield`
`ios::unitbuf`	0x2000	Flush stream after output	O		No
`ios::stdio`	0x4000	Flush stdout and stderr after output	O		No

- To set a flag(s) we use the `setf` function:

```
cout.setf(0x0001);
cout.setf(ios::skipws);
```

- Set more than one bits simutaniously:

```
cout.setf(0x0001|0x0002);
cout.setf(ios::skipws|ios::left);
```

- To unset other flags, we use the `unsetf` function:

```
cout.unsetf(flags);
```

- C++ also provides a short-hand to combine both operations:

```
cout.setf(on_flags, off_flags);
```

  - First turns off the flags `off_flags`
  - Then turns on the flags `on_flags`
- Integer Base Example:

```
int x = 42;

cout.setf(ios::oct, ios::basefield);
cout << x << '\n'; // Outputs 52\n
cout.setf(ios::hex, ios::basefield);
cout << x << '\n'; // Outputs 2a\n
cout.setf(ios::dec, ios::basefield);
cout << x << '\n'; // Outputs 42\n
```

- Floating Point Format:

```
cout.setf(ios::scientific, ios::floatfield);
cout << 123.45 << '\n'; // Outputs 1.2345e+02
cout.setf(ios::fixed,ios::floatfield);
cout << 5.67E1 << '\n'; // Outputs 56.7
```

- Use function `precision(int)` to set the number of significant digits printed (may convert from fixed to scientific to print)
- Effect of precision **depends on format**
    - scientific **(total significant digits总有效位数)**

    ```
    float y = 23.1415;
    cout.precision(1);
    cout << y << '\n';
    // Outputs 2e+01
    cout.precision(2);
    cout << y << '\n';
    // Outputs 23
    cout.precision(3);
    cout << y << '\n';
    // Outputs 23.1
    ```

    - fixed **(how many digits after decimal point小数点后位数)**

    ```
    cout.setf(ios::fixed,ios::floatfield);
    cout.precision(1);
    cout << y << '\n';
    // Outputs 23.1
    cout.precision(2);
    cout << y << '\n';
    // Outputs 23.14
    cout.precision(3);
    cout << y << '\n';
    // Outputs 23.142
    ```

- Showing the Base:
    - The flag `ios::showbase` can be set (its **default is off**), it results in integers being printed in a way that demonstrates their base:
        - decimal - no change
        - **octal（八进制）** - leading `0`
        - **hexadecimal（十六进制）** - leading `0x`
- Showing the Plus Sign:
    - The flag `ios::showpos` can be set (its **default is off**) to print a `+` sign when a **positive integer or floating point** value is printed
- Showing Upper Case Hex Ints:
    - The flag `ios::uppercase` (**default off**) can be used to indicate that the letters making up **hexadecimal numbers should be shown as upper case（十六进制数字的字母会被大写展示）**
- Decimal Points in Floats:
    - Set flag `ios::showpoint` to make sure decimal point appears in output **(C++ only shows significant digits in default默认只展示有效的位)**
    - Example:

```
float y = 3.0;
cout << y << '\n'; // Outputs 3
cout.setf(ios::showpoint);
cout << y << '\n'; // Outputs 3.00000
```

- Displaying bools:
  - Variables of type bool print out as `0` (`false`) or `1` (`true`), to print out words (`false` , `true`) use flag `ios::boolalpha` **（使用该标志符输出字符串形式下的布尔字面量）**

- **`ios` member functions （`ios` 成员函数）**
  - `width()`
  - `fill()`
  - `precision()`
  - `setf()`
  - `unsetf()`
- **Manipulators（操纵符）**
  - A manipulator is a simple function that can be included in an insertion or extraction chain:

    ```
    cout << manip1 << manip2 << manip3 << item;
    ```

  - C++ manipulators:
    - must **include** `<iomanip>` to use
    - several are provided to do useful things
    - you can also **create your own manipulators**
  - Manipulators **without arguments**:

Name	**Description**
`endl`	Outputs a newline character, **flushes** output
`dec`	Sets the base of int output to decimal
`hex`	Sets the base of int output to hexadecimal
`oct`	Sets the base of int output to octal

  - Manipulators taking 1 argument:

Name	**Description**	**Corresponding Member Function**
`setw(int)`	Sets the width to `int` value	`width(int)`
`setfill(char)`	Sets fill char to `char` value	`fill(char)`
`setprecision(int)`	Sets precision to `int` value	`precision(int)`
`setbase(int)`	Sets int output to hex if `int` is `16` , oct if `int` is `8` , dec if `int` is `0` or `10`	
`setiosflags(flags)`	Set flags on	`setf(flags)`
`resetiosflag(flags)`	Set flags off	`unsetf(flags)`

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 11 Working with Files

- **File（文件）**
  - Files in C++ are interpreted as **a sequence of bytes** stored on some storage media.
  - Each file ends with an **end-of-file (EOF)** marker.
- **Using Input/Output Files（文件输入输出）**：
  - File $\longrightarrow$ Program (Input stream) - **reads**
  - Program $\longrightarrow$ File (Output stream) – **write**
  - File streams act as an interface between files and programs, **stream classes for file operations** are declared in the header file `fstream`

$$\begin{tikzcd}
\text{iostream} \arrow[r] & \text{fstream} & \text{iostream} \arrow[l] \\
\text{istream} \arrow[u] \arrow[d] & \text{ios} \arrow[l] \arrow[r] \arrow[d] & \text{ostream} \arrow[u] \arrow[d] \\
\text{ifstream} & \text{streambuffer} \arrow[d] & \text{ofstream} \\
 & \text{filebuf} &
\end{tikzcd}$$

cont.

$$\begin{tikzcd}
 & \text{ios} \arrow[d] & \\
\text{ifstream} & \text{fstreambase} \arrow[l] \arrow[r] \arrow[d] & \text{ofstream} \\
 & \text{fstream} &
\end{tikzcd}$$

- **The Process of Using a File**
  - Three-step process:
    - The file must be **opened**. If the file does not yet exists, opening it means **creating** it.
    - Information is then **saved** to the file, **read** from the file, or **both**.
    - When the program is **finished using the file**, the file must be **closed**.
- **How to open a file in C++**:
  - The following classes can be used:
    - `ofstream` – writing to a file
    - `ifstream` – reading from a file
    - `fstream` – reading or writing
  - A file can be opened in two ways:
    - Using the **constructor function** of the file stream class:

```
ofstream out_file("client.dat", ios::out);
```

- Using the **member function** `open()` of the file stream class:

```
ofstream out_file;
out_file.open("client.dat", ios::out);
```

- **File Open Modes（文件打开模式）**

Mode	**Description**
`ios::in`	Open a file for **input**
`ios::out`	Open a file for **output**, **discard** the contents **if it exists**
`ios::trunc`	**Discard** the contents **if it exists**
`ios::app`	Write all output to the end of file **(append)**
`ios::out|ios::app`	Open a file for **output**， **keep the contents** and write to the end of the file
`ios::ate`	**File pointer** is positioned **at the end of the file**
`ios::out|ios::ate`	Open a file for **output**, discard the contents if it exists
`ios::in|ios::ate`	Open a file for **input**, **file pointer** is positioned **at the end of the file**
`ios::binary`	Read/write data in **binary format**
`ios::nocreate`	If the file does **NOT exists**, the open operation **fails**
`ios::noreplace`	If the file **exists**, the open operation **fails**

File Type	**Default Open Mode**
`ofstream`	The file is opened for **output only**. If the file does **not exist**, it is **created**. If the file **already exists**, its contents are **deleted**.
`ifstream`	The file is opened for **input only**. The file's contents will be **read from its beginning**. If the file does **not exist**, the open function **fails**.

- **Testing for Open Errors（文件打开错误检测）**：

```
#include <fstream>
#include <iostream>

using namespace std;

int main(void) {
  fstream data_file("names.dat", ios::in | ios::out);
  // Stream object returns a value of 0 if any error occurs in the file operations
  if (!data_file) {
    cout << "Error opening file.\n";
  }

  // Another way to test for open errors
  if (data_file.fail()) {
    cout << "Error opening file.\n";
  }
  return 0;
}
```

- **Closing a File（关闭文件）**
  - A file should be closed **when a program is finished using it**:

    ```
    stream_object.close();
    ```

- **Using `<<` to Write Information to a File**
  - The stream **insertion operator（插入运算符）** `<<` can be used to write information to a file.
  - File output can be **formatted the same way as screen output**.
- **Using `>>` to Read Information from a File**
  - The stream **extraction operator（提取运算符）** `>>` may be used to read information from a file.
- **Detecting the End of a File（检测文件结束）**
  - The `eof()` **member function** reports when the end of a file has been encountered:

    ```
    if (in_file.eof())
      in_file.close();
    ```

  - `while(in_file.eof())` is Equivalent to `while(in_file)`
- **More Detailed Error Testing（更多关于错误检测的细节）**：
  - A file which we are attempt to open for reading does not exist. **（文件不存在）**
  - We may attempt an invalide operation such as reading past the end-of-file. **（执行无效操作，如读取EOF后的内容）**
  - There may not be any space in the disk for storing more data. **（磁盘空间不足）**
  - We may use an invalid file name. **（使用非法文件名）**
  - We may attempt to perform an operation when the file is not opened for that purpose **（尝试执行与文件打开模式不对应的操作）**
- **Error State Bits（错误状态位）**
  - All stream objects have error state bits that **indicate the condition of the stream**:

Bit	Description
`ios::eofbit`	Set when the **end** of an **input stream** is encountered
`ios::failbit`	Set when an attempted **operation has failed**
`ios::hardfail`	Set when an **unrecoverable error** has occurred

Bit	Description
`ios::badbit`	Set when an **invalid operation** has been attempted
`ios::goodbit`	Set when **all the flags above are not set**. Indicates the stream is in good condition

- The class `ios` supports several member functions to read the status recorded in a file stream:

Function	Description
`eof()`	Returns **true (non-zero)** if the `eofbit` flag is set, otherwise returns **false**
`fail()`	Returns **true (non-zero)** if the `failbit` or hardfail flags are set, otherwise returns **false**
`bad()`	Returns **true (non-zero)** if the `badbit` flag is set, otherwise returns **false**
`good()`	Returns **true (non-zero)** if the `goodbit` flag is set, otherwise returns **false**
`clear()`	When called with no arguments, **clears all the flags** listed above. Can also be **called with a specific flag** as an **argument**

- **Member Functions for Reading and Writing Files（读写文件的成员函数）**
  - Read and write to a file in **text form（文本形式）**：

    ```
    getline()
    get()
    put()
    ```

  - Read and write to a file in **binary form（二进制形式）**：

    ```
    write()
    read()
    ```

  - The `getline` Member Function:

    ```
    getline(str, 81, '\n');
    ```

    - `str`: This is the **name of a character array（数组名）**，or a **pointer to a section of memory（指向内存节的指针）**. The information read from the file will be stored here.
    - `81`: This number is **one greater than the maximum number（最后有一个 `'\0'`，所以需要比输入的最大80个字符多1个字符）** of characters to be read. In this example, a maximum of 80 characters will be read.
    - `'\n'`: This is a **delimiter（分隔符）** character of your choice. **If this delimiter is encountered, it will cause the function to stop reading（遇到就停止读取了）** before it has read the maximum number of characters. (This argument is **optional（可选参数）**. If it's left, `'\n'` is the default.)
  - The `put` and `get` Member Functions:
    - `get()` is used to **read an alphanumeric character（读取一个字母数字字符？似乎也不太准确，可以读到非操作字符）** from a file.
    - `put()` is used to write a character to a specified file or a specified output stream
- **Binary Files（二进制文件）**：
  - Binary files contain data that is **unformatted（未格式化的）**，and not stored as ASCII text.
  - Example:
    - Text file of `1297` (expressed in ASCII):

49	50	57	55	<EOF>

    - Binary file of `1297` (An integer):

00000101	00010001

  - To open a binary file, use:

    ```
    file.open("stuff.dat", ios::out|ios::binary);
    ```

    - Commonly use `.dat` as binary file extension name.
  - Binary **input** function:

    ```
    read(unsigned char* buffer, int n);
    ```

    - The `<istream>` function `read` **inputs a specified number of bytes** from the current position of the specified stream into an object.
    - Usage:

      ```
      inflie.read((char*) & V, sizeof(V));
      ```

  - Binary **output** function:

    ```
    write(const unsigned char* buffer, int n);
    ```

    - The `<ostream>` member function `write` **outputs a fixed number of bytes** beginning at a specific location in memory to the specific stream.
    - Usage:

      ```
      outfile.write((char*) & V, sizeof (V));
      ```

- **Reading and writing a class object（类对象读写）**
  - **Structures（结构体）** may be used to store **fixed-length** records to a file:

    ```
    struct info {
      char name[51];
      int age;
      char address[51];
      char phone[51];
    };
    ```

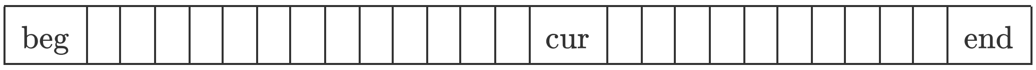
  - Since structures can **contain a mixture of data types（包含混合的数据类型）**, you should **always use the `ios::binary` mode（总是需要以二进制模式读写）** when opening a file to store objects.
- **Random Access Files（随机文件访问）**
  - It means non-sequentially accessing informaiton in a file
  - Every file maintains **two internal pointers（两个内部指针）** known as **file pointers**:
    - **get_pointer**
    - **put_pointer**
  - They enable to **attain the random access（实现随机访问）** in file otherwise which is **sequential in nature（本质上是顺序访问）**.
  - In C++ randomness is achieved by manipulating certain functions.
  - **Default Actions（默认行为）**:
    - When we open a file in **read-only mode（只读）**, the **input pointer(get_pointer)** is automatically set at the **beginning（输入指针设置在文件开头位置）**.
    - When we open a file in **write-only mode（只写）**, the **output pointer(put_pointer)** is automatically set at the **beginning** and **the contents are deleted（原有内容被删除，输出指针也设置在文件开头位置）**
    - When we open a file in **'append' mode(`ios::app`)**, the **output pointer(put_pointer)** is moved to the **end** of the file

- **Moving within the File（在文件内移动）**：
    - `seekg()` / `seekp()` – moving the **get** / **put pointer** to specified situation

      ```
      seekg(offset, refposition);
      seekp(offset, refposition);
      ```

        - Need two parameters: **offset（偏移量）（长整型参数）** and **refposition（参照位置）**
            - Parameter **offset** represents **the number of bytes** the file pointer to be moved from the location specified by the parameter **refposition**.
    - `tellg()` / `tellp()` – getting the position of the **get** / **put pointer**
        - `tellp` returns a **long integer** that is the **current byte number** of the file's **write** position.
        - `tellg` returns a **long integer** that is the **current byte number** of the file's **read** position.
    -

Mode Flag	Description
`ios::beg`	The offset is calculated from the **beginning** of the file.
`ios::end`	The offset is calculated from the **end** of the file.
`ios::cur`	The offset is calculated from the **current** position.



```
fstream file("d:\\file1", ios::in);
// get_pointer move 10 bytes backward from the current position
// Move to Left
file.seekg(-10L, ios::cur);
// get_pointer move 10 bytes forward from the beginning of the stream
// Move to Right
file.seekg(10L, ios::beg); // = file.seekg(10L);
```

    - Example:

```cpp
#include <fstream>
#include <iostream>

using namespace std;

int main(void) {
  fstream file("letters.txt", ios::out | ios::in);

  if (!file) {
    cout << "the file is not opened" << endl;
    abort();
  }

  file << "abcdefghijklmnopqrstuvwxyz";

  char ch;

  file.seekg(5L, ios::beg);
  file.get(ch);
  cout << "Byte 5 from beginning: " << ch << endl;

  file.seekg(-10L, ios::end);
  file.get(ch);
  cout << "Byte 10 from end: " << ch << endl;

  file.seekg(3L, ios::cur);
  file.get(ch);
  cout << "Byte 3 from current: " << ch << endl;

  file.close();
  return 0;
}
```

Output:

```
Byte 5 from beginning: f
Byte 10 from end: q
Byte 3 from current: u
```

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 12 Templates

- C++ **template（模板）** is based upon **the concept of type variable**, i.e. a variable that takes a **type** as its value **（以类型作为变量）**
  - General form:

```
template <class/typename T1, class/typename T2, int i, ...>
return_type function_name(T1 value1, T2 value2, int i, ...){
    ...
}
```

        - Can also use normal function-style parameters (like `int i`) to specify nontype parameters.
    - Example:

```
template <class T> class vector {
  T *v;
  int size;

public:
  vector(int m) {
    v = new T[size = m];
    for (int i = 0; i < size; i++)
      v[i] = 0;
  }
  T operator*(vector &y) {
    T sum = 0;
    for (int i = 0; i < size; i++)
      sum += this->v[i] * y.v[i];
    return sum;
  }
};

int main(){
  vector<int> v1(10);
  vector<double> v2(5);
  return 0;
}
```

- Kinds of Templates
    - **Class template（类模板）**: Permit the development of **generic objects（通用对象）**
    - **Function template（函数模板）**: Permit the development of **generic algorithms（通用算法）**
    - Variable template: C++ new standard
- **Class Template（类模板）**
    - Consists of a template header followed by a normal class definition
    - Format:

```
template <class T>
class Class_name{
    ...
}
```

    - Need not use "`T`", any identifier will work **（这个`T`不唯一，想把它换成任何不是关键字作为类型变量名都可以）**
    - To create an object of the class, type:

```
Class_name<type> my_object;
```

    - Class **template member functions（模板成员函数）**
        - If the member functions are defined **inside the class**, the member functions is defined normally
        - If the member functions are defined **outside the class**, the member functions must be defined by the **function templates（如果成员函数定义在类外，那么必须以函数模板的形式进行定义）**:

```
template <class T>
return_type Class_name<T>::function_name(arglist) {
  ...
}
```

- Example:

```
template <class T>
class Vector{
  T *v;
  int size;
public:
  Vector(int m);

  T operator*(Vector &y){ // inside the class
      T sum = 0;
      for (int i = 0; i < size; i++){
        sum += this->v[i]*y.v[i];
      }
      return sum;
  }
};

template <class T> // outside the class
Vector<T>::Vector(int m){
  v = new T[size=m];
  for (int i = 0; i < size; i++){
      v[i] = 0;
  }
}
```

- **Class Template Instantiation（类模板实例化）**
  - A **template class（模板类）** is a class built from a **class template（类模板）**
  - The process of creating a template class from a class template is called a **instantiation（实例化）**
  - The **compiler** will perform the error analysis **only when an instantiation takes place（编译器仅在实例化时进行错误分析）**
  - Can use **non-type parameters（非类型参数）** in templates
    - Default argument
    - Treated as `const`
    - Example:

```
template <class T, int size>
class array {
  T a[size];
  ...
};

array<int, 10> a1;
```

Creates array **at compiling time（在编译期间就已经创建了数组）**, rather than dynamic allocation at execution time

- **Function Template（函数模板）**
  - General form:

```
template <class T1, class T2, ...>
return_type function_name(T1 value1, T2 value2, ...) {
  ...
}
```

- Example:

```
template <typename T>
T min(const T &a, const T &b) {
  // operator `<` needs to be defined for the actual template parameter type
  // if not, compile-time error occurs
  if (a < b)
    return a;
  else
    return b;
}
```

- All matches for **formal parameters involving type parameters** must be **consistent（在模板函数调用时，包括类型参数在内的形参匹配必须一致）**
  - Only **trivial promotions（只有琐碎地匹配开销是被允许的）** to produce a match are allowed, for example, `int&` to `const int&` is allowed
  - Formal parameters **not involving type parameters must also be matched（非类型参数也需要被匹配）** without nontrivial conversion/promotion.
  - Example:

```
#include <iostream>

using namespace std;

template <class T>
T min(const T &a, const T &b) {
  if (a < b)
    return a;
  else
    return b;
}

int main(){
  int value1 = 100;
  char value2 = 'a';

  cout << min(value1, 97) << endl;
  cout << min(value1, value2) << endl; // error
}
```

- **Overloading** of template functions:

```cpp
#include <iostream>

using namespace std;

template <class T> void display(T x) {
  cout << "template display:" << x << endl;
}

void display(int x) { cout << "Explicit display:" << x << endl; }

int main() {
  display(100); // Explicit display:100
  display(12.34); // template display:12.34
  display('c'); // template display:c
  return 0;
}
```

- **Function call resolution（函数调用解析）**
  - To resolve a function call, compiler follows **3 steps**:
    - Examine **all non-template** versions of the function, if any, for an exact match. **（先找非模板函数）**
      - error if there are more than one exact match.
    - Examine **all template** functions, if any, for an exact match. **（再找模板函数）**
      - error if there are more than one exact match.
    - If steps 1 and 2 do not resolve the call or produce an error, then re-examine all non-template versions of the function using call-resolution rules for regular **overloaded functions**. **（最后以重载函数的常规解析规则来匹配）**
  - Example:

    ```cpp
    #include <iostream>

    using namespace std;

    template <class T> T max(T a, T b) {
      cout << "Call the template function" << endl;
      return (a > b) ? a : b;
    }

    int max(int a, int b) {
      cout << "Call the non-template function" << endl;
      return (a > b) ? a : b;
    }

    void f(int num, char ch) {
      max(num, num); // step 1 match
      max(num, ch); // step 3 match
    }

    int main() {
      f(65, 'a');
      return 0;
    }
    ```

    Output:

```
Call the non-template function
Call the non-template function
```

- **Non-Type Template Arguments（非类型模板参数）**：
  - Treated as `const`:

    ```
    template <typename T, int n>
    T max1(T arr[n]) {
      T ans = arr[0];
      for (int i = 1; i < n; i++) {
        ans = (ans > arr[i]) ? ans : arr[i];
      }
      return ans;
    }

    int main() {
      int k = 5;
      int a[] = {1, 2, 3, 4, 5, 6, 7};
      cout << max1<int, 5>(a) << endl; // ok
      cout << max1<int, k>(a) << endl; // error
      return 0;
    }
    ```

  - Difference between **non-type template arguments** and **common arguments**:

    ```
    template <typename T>
    T max2(T arr[], int n) {
      T ans = arr[0];
      for (int i = 0; i < n; i++) {
        ans = (ans > arr[i]) ? ans : arr[i];
      }
      return ans;
    }

    int main() {
      int k = 5;
      int a[] = {1, 2, 3, 4, 5, 6, 7};
      cout << max2(a, 5) << endl; // ok
      cout << max2(a, k) << endl; // ok
      return 0;
    }
    ```

- **Templates and friends（模板与友元）**
  - Friendships allowed between a **class template** and
    - **Global function**
    - **Member function of another class**
    - **Entire another class**
  - `friend` functions
    - Inside definition of class template `X`:

      -
        ```
        friend void f1();
        ```
        - `f1()` is `friend` of all template class

      -
        ```
        friend void f2(X<T> &);
        ```

            - `f2(X<int> &)` is a `friend` of `X<int>` only. The same applies for `float`, `double`, etc.

        -
          ```
          friend void A::f3();
          ```
            - Member function `f3` of class `A` is a `friend` of all template classes

        -
          ```
          friend void C<T>::f4(X<T> &);
          ```
            - `C<float>::f4(X<float> & )` is a `friend` of class `X<float>` only
    - `friend` classes
        - `friend` class `Y`, declared in template class `X`:
            - Class `Y` is a `friend` of every template class made from `X`.
            - Every member function of `Y` is a `friend` of every template class made from `X`.
        - `friend` class `Z<T>`
            - Class `Z<float>` is a `friend` of class `X<float>`, etc.
- **Templates and static Members（模板与静态成员）**
    - Non-template class
        - `static` data members **shared between all objects**
    - Template classes
        - Each class (`int`, `float`, etc.) **has its own copy** of `static` data members
        - `static` variables **initialized at file scope（在文件中已初始化）**
        - Each template class gets its **own copy of** `static` **member functions（各有各的静态成员函数）**

Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 14 Exception Handling

- **Exception（异常）**: An abnormal condition that occurs during program operation:
    - `new` operator cannot obtain the required memory **（内存分配时可用内存不足）**
    - data subscript is out of bounds **（下标越界）**
    - divisor `0` **（除以 `0`）**
    - invalid parameters **（无效参数）**
    - open non-existent files **（打开不存在的文件）**
- **Exception Handling（异常处理）**: to provide means to **detect** and **report** an "exceptional（异常的） situation" so that **appropriate action** can be taken.
- **Traditional exception handling methods（传统异常处理方式）**: Continuously test the necessary conditions for the program to continue running, and to handle the test results.
    - Example:

```
int x,y,arr[10];

cin >> x >> y;
if (y == 0) cerr << "error";
  else cout << "x/y=" << x/y << endl;

cin >> x;
if (x > 9) cerr << "error";
  else cout << "arr[" << x << "]=" << arr[x] << endl;
```

  - Traditional method makes **error handling code distributed in various parts of the whole program**, which makes the program "polluted" by the error handling code and becomes **diffcult to read（传统异常处理方法，大量异常处理代码四处分布，程序难以阅读）**.
- **Exception Handling Mechanism（异常处理机制）**
  - C++ Exception Handling: the basic idea is to **put the exception occurrence and exception handling in different functions（将异常发起和异常处理置于不同的函数中）**.
  - Three **keywords（关键字）**: `try`, `throw` and `catch`:
    - `try` block: Detect and throws an exception
    - `catch` block: Catch and handles the exception
  - Example:

```
try{ //try block
  ...
  if err_1 throw xx_1
  ...
  if err_2 throw xx_2
  ...
  if err_n throw xx_n
}
catch(type1 arg){...}
catch(type2 arg){...}
catch(typem arg){...}
```

  - Exceptions can be **thrown by functions**:

```
void divided(int x, int y, int z) {
  if((x-y) != 0)
      cout << "Result=" << z / (x - y) << endl;
  else
      throw(x - y);
}

int main() {
  try {
    cout << "we are inside the try block" << endl;
    divide(10, 20, 30);
    divide(10, 10, 20);
  } catch (int i) {
    cout << "Caught the exception" << endl;
  }
  return 0;
}
```

- **The execution logic of exception handling（异常处理的执行逻辑）**

  - When a `try` block is encountered during program execution, it will **enter the `try` block and execute the statements** in it in the **normal program logic order**.
  - If all statements in the `try` block are executed **normally without any exception**, no exception will be thrown in the `try` block. In this case, the program will **ignore all `catch` blocks and continue to execute program** statements other than `catch` blocks in sequence.
  - If an **error occurs** in a statement during the execution of a `try` block and an exception is thrown, the program **control flow will transfer to the `catch` block（控制流进入`catch`块）**, and **all statements after the throw statement in the `try` block will not be executed（`try`块中触发异常的语句之后的所有语句不会再运行）**.
  - C++ will compare the data type of the exception with the data type specified in each `catch` parameter table in the order in which the `catch` blocks appear. As long as **one `catch` block catches the exception, the remaining `catch` blocks will be ignored（`catch`块会根据抛出的异常类型自动捕获异常，当异常被其中一个`catch`块捕获，其他的`catch`块不会再生效）**.
    - Example:

```
#include <iostream>

using namespace std;

int main() {
  cout << "1--befroe try block..." << endl;
  try {
      cout << "2--Inside try block..." << endl;
      throw 10;
      cout << "3--After throw ...." << endl;
  } catch (int i) {
      cout << "4--In catch block1 ... exception ... errcode is ... " << i << endl;
  } catch (char *s) {
      cout << "5--In catch block2 ... exception ... errcode is ... " << s << endl;
  }
  cout << "6--After Catch..." << endl;

  return 0;
}
```

Output:

```
1--befroe try block...
2--Inside try block...
4--In catch block1 ... exception ... errcode is ... 10
6--After Catch...
```

  - If no `catch` can match the exception, C++ will call the system's **default exception handler** to handle the exception, and its usual practice is to **directly terminate the program（系统默认的异常处理是终止程序）**.
- **Handling exceptions in a function（在函数中处理异常）**
  - Exception handling can be **localized to a function**, that is, the `try` - `throw` - `catch` structure for handling exceptions is placed in the function, and **each time the function is called, the exception will be reset（每次函数调用时，异常会被重置）**.
  - Example:

```
#include <iostream>

using namespace std;

void test(int x) {
  try {
    if (x == 1)
      throw x;
    else if (x == 0)
      throw 'x';
    else if (x == -1)
      throw 1.0;
    cout << "End of try block" << endl;
  } catch (char c) {
    cout << "Caught a character" << endl;
  } catch (int m) {
    cout << "Caught an integer" << endl;
  } catch (double d) {
    cout << "Caught a double" << endl;
  }
  cout << "End of try-catch system\n" << endl;
}

int main() {
  test(0);
  test(-1);
  test(2);
  return 0;
}
```

Output:

```
Caught a character
End of try-catch system

Caught a double
End of try-catch system

End of try block
End of try-catch system
```

- Putting the program code that **generates exceptions in one function（产生异常的代码在一个函数中）** and the function code that **detects and handles exceptions in another function（检测和处理异常的代码在另一个函数中）** can make exception handling **more flexible and practical（更灵活更实用地处理异常）**.
  - Example:

```
#include <iostream>

using namespace std;

void temperature(int t) {
    if (t == 100)
      throw "沸点！";
    else if (t == 0)
      throw "冰点！";
    else
      cout << "ok" << endl;
}

int main() {
  try {
    temperature(0);
    temperature(10);
    temperature(100);
  } catch (char const *s) {
    cout << s << endl;
  }
  return 0;
}
```

中间新插入了异常的匹配细节，包括：只有const转换、基类匹配派生类、指针匹配数组名这三种方式可用于异常匹配，以及默认异常处理会将程序终止

- **Catch all exceptions（捕获所有异常）**
  - In most cases, catch is only used to catch a specific type of exception, but it also has the ability to catch all exceptions.
  - **Its form is as `catch(...){}`.**
  - Example:

    ```
    #include <iostream>

    using namespace std;

    void Errhandler(int n) {
      try {
        if(n==1) throw n;
        if(n==2) throw "dx";
        if(n==3) throw 1.1;
      } catch(...) {
        cout<<"catch an exception..."<<endl;
      }
    }

    int main() {
      Errhandler(1);
      Errhandler(2);
      Errhandler(3);
      return 0;
    }
    ```

- **Rethrowing an exception（异常重抛出）**

- If the catch block cannot handle the captured exception, or can only handle part of the exception, the rest of the exception needs to be processed by its outer calling function. It can **use the throw statement without any parameters to throw the exception again**.
- Example:

```cpp
#include <iostream>

using namespace std;

void Errhandler(int n) {
  try {
    if(n==1) throw n;
    cout << "all is ok..." << endl;
  } catch(int n) {
    cout << "catch an int exception inside..." << n << endl;
    throw; //再次抛出本catch捕获的异常
  }
}

int main() {
  try {
    Errhandler(1);
  } catch(int x) {
    cout << "catch int an exception in main..." << x << endl;
  }
  cout << "...End..." << endl;
}
```