1. 1. C++ BASICS

1.1 1.1 Advantage of Structured Programming

Although it is a trend to use object-oriented programming, structured programming also has its advantage. Internal structure of objects are often best built with structured programming techniques. Also, the logic of manipulating objects is best expressed with structured programming..

1.2 1.2 Interpreter and Compiler Program

Interpreter programs were developed that can directly execute high-level language programs without the need of compiling these programs into machine language. It is used on cases that programs are frequently updated. Once the program is developed, a compiled version will be produced to run quickly..

1.3 1.3 Escape Sequence

"\" is escape character. It together with the following character forms an escape sequence used in output statement. They are used to control the printer and are not printed out.

"\n" change to a new line. It is a good common practice to add it in the end of all printing statements.

"\t" horizontal tab.

"\r" carriage return – return the cursor to the beginning of current line.

"\a" alert. Sound the system bell.

'\\" used to print out a "\".

"\"" used to print out a double quote.

1.4 1.4 Namespace

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

int main( )
{
   cout << "Welcome! \n";
   std::cout << "Welcome! \n";
   return 0;
}
```

A namespace is a group of software components with the same name prefix – e.g., std. Each header file in the C++ draft standard uses a namespace called "std" to guarantee that every feature provided by the C++ standard library is unique from software components developed by other programmers. The statement above simply says that we are using software components from the C++ standard library, such as cout. If we use the namespace in front of each software component, then we needn't the statement at the top. If we want to define our own class library, we should use our own namespace.

1.5 1.5 Advantages of OO Programming

OO programming has a lot of advantages over procedure programming, many of them are achieved by data encapsulation.

Procedure programming can also call existing functions to achieve code reusing, but functions and data are separate - in other words, functions are stateless - they don't remember their own state across different calls. Whenever a function is called, all data to be manipulated have to be passed to and returned by the function. Data are therefore exposed to programmers, who may manipulate the data wierdly or carelessly. Besides, a single piece of data may be passed around and handled by a lot of functions within a large project. When an error finally emerges, it may be very difficult to find out where the error happens.

In comparison, OO programming with data encapsulation can avoid these problems. Because data are encapsulated with functions to form an object, functions can remember their own state and their results. There is no longer need to pass this part of data to these functions every time they are called. Sensitive or private data can be therefore isolated from outside world. If anyone wants to do anything on the data, they have to do it through the object's functions, which can be well defined and error-proof.

This simulates real world objects better. Take a man as an object. This object can have a data members such as body temprature. Other people can not directly change his body temprature without going through his own cooperation and physical reaction. So body temprature is a private data members isolated from the public. This isolation eliminates the possibility that other programs may accidentally change an object's own sensitive data inproperly.

A well-written class is cohesive and self-contained. It contains all necessary data to similate the real-world problem, all necessary functions to manipulate these data and provide services to outside world. It exists and functions quite independently, reducing the coupling with outside world to the lowest extent, and therefore may be reused in different occasions.

An object "knows" how to behave itself properly – how to construct and initialize itself, how to response to different kinds of external messages properly, how to destroy itself, etc.. Once a class is properly written and fully tested, other programs can simply make use of it to perform a certain task, and will have little chance to encounter any error. This greatly reduces the complexity of programming and possibility of errors.

So generally speaking, OO programming and data encapsulation has the following advantages:

1. 1. Reduces data exposure: because private data are encapsulated within objects, when their functions are called, less data need to be tranmitted. These reduces the exposure of data and the chance of errors.
2. 2. Clearer responsibility for data: because each object is responsible for his own private data and others can not wrongly change it, when a piece of data is wrong, we easily know who is doing the wrong thing.
3. 3. Less error-prone: if a project uses only those well-written and fully tested classes, it is much easier to organize, modify, debug and maintain than those written in procedural languages. The bigger the size, the more obvious the advantage.
4. 4. More stable for clients: when a class's implementation is changed, as long as its function signatures are not changed, a program which uses this class will not be affected.
5. 5. Enhances software reuse: by composition (including other object as data member) and inheritance (build a new class based on a base class), software reuse is greatly strengthened.

1.6 1.6 *Library math.h*

Consists of all mathematical functions:

ceil (x)	ceil (9.2) = 10
cos (x)	
exp (x)	e^x
fabs(x)	absolute value of x

floor (x)	floor (9.9) = 9
fmod (x, y)	reminder of x/y as a float. fmod (9.85, 3.2) = 0.25
log (x)	log $_e^x$
log10 (x)	log $_{10}^x$
pow (x, y)	x ^{y}
sin (x)	
sqrt (x)	sqrt (9) = 3
tan (x)	

1.7 1.7 Function Prototype

The function prototype is placed outside any function including "main", right after the "#include" preprocessor directive. That's why function prototype has file scope.

Function Prototype is something FORTRAN and earlier versions of C do not have. It is designed to enable the compiler to know what kind of a function it is before a function is called. That's why when the function body is before the calling statement the function prototype is not necessary. It also enables the compiler to find out whether a function is properly called i. g. with wrong numbers of arguments, and convert the parameter type if the arguments supplied is not correct.

You can include the names of the parameters into the function prototype to make it look more clear, but the compiler will ignore the parameters.

1.8 1.8 Block

The declarations and statements enclosed in braces " { }" is called a block. A block is just a compound statement which includes declarations. A block instead of a function is the smallest combination of statements in C++. You can create a block anywhere you like – just enclose at least one declaration and one statement with braces.

The variables declared in a block has a "block scope" – they are hidden outside this block. Variables declared outside this block in the same function will still be "visible" inside this block (so are global variables, of course). So the character of a block is different from a function: a function is of two-way information hiding, while a block is of one-way only.

1.9 1.9 time(0)

This function returns the current calendar time in seconds. Its header file is “**time.h**”.

1.10 1.10 Random Number Generation

```
unsigned seed;
cin >> seed;
srand(seed); // or srand (time (0) )
y = a + rand( ) % b;
```

♦ ☐ **Pseudo-random sequence**

Function “**rand**” returns a pseudo-random sequence of integer between 0 and RAND_MAX (a symbolic constant defined in the “**stdlib.h**”). The function prototype of “rand” is in this header file “**stdlib.h**”. The numbers generated by “rand” is actually a fixed sequence. Each time you run this program, you get the same sequence. To change this sequence, you have to use another function “**srand**” to “seed” the “rand” function with different unsigned integer. Its header file is also “**stdlib.h**”. For each integer a unique and fixed sequence is generated by “rand”.

♦ ☐ **Scaling**

The function to generate a random sequence of number within a certain range is called “scaling”:

```
n = a + rand () % b
```

the random numbers are from a to a + b.

1.11 1.11 Enumeration

An enum is an alias of an integer. It setups up a corresponding relationship between a specific integer and a user-chosen identifier. Instead of having to remember what an integer represents, the programmer can easily remember the self-explained alias. For example, a set of integers from 0 to 7 represents color black, blue, green, cyan, red, magenta, brown and white. After we define

```
  enum Color {Black, Blue, Green, Cyan, Red, Magenta, Brown,
White};
```

we can always use “Black” instead of 0, but the compiler will just treat “Black” as 0.

Now “Color” is a user-defined type. You can use this type just like int, float to declare variables, but these variables can only have the values enclosed in { }:

```
Color c1, c2;
c1 = Green;
c2 = Brown;
if(c1 == Red)...;
```

You can also join the definition of the enum type and declaration of its variables together:

```
  enum {Black, Blue, Green, Cyan, Red, Magenta, Brown, White}
c1, c2;
```

You can assign enums to both variables of their own types, or simply integer variables:

```
  enum {Black, Blue, Green, Cyan, Red, Magenta, Brown,
White};
  int c1, c2;
  c1 = Green;
  c2 = Brown;
  if(c1 == Red)...;
```

By default, the first enum enclosed in { } has the value of 0, the next 1,... , unless specifically defined:

```
  enum {Black, Blue, Green, Cyan = 23, Red, Magenta, Brown,
White};
```

Then Black = 0, Blue = 1, Green = 2, Cyan = 23, Red = 24, Magenta = 25,...

1.12 1.12 Global Variables

Global variables are variables declared outside any block including **main** function. They are visible in all blocks in all files in the same process. In files other than the one where the global variable is defined, you have to use keyword **extern** to tell the compiler: "The following global variable whose name is xxx and whose type is xxx is defined elsewhere, and you have to find out where yourself."

Global variables can be defined in either a header file or a source file. But if you define a global variable in a header file, then when more than one files include that header file, multiple copies of the same global variable will be instantiated, and you will have link errors. So you should normally put the definition of global variables in a source file.

```
//************ globals.cpp *****************
// You don't need a globals.h!
#include "stdafx.h"

int array[3] = {1, 2, 3}; // array is the global object


//*************** A.cpp ******************
#include "stdafx.h"
#include "A.h"
#include <iostream.h>

extern int array[];

A::A()
{
    cout << array[0] << " " << array[1] << " " << array[2] <<
endl;
}


//************** B.cpp ********************
#include "stdafx.h"
#include "B.h"
#include <iostream.h>

extern int array[];

B::B()
{
    cout << array[0] << " " << array[1] << " " << array[2] <<
endl;
}


//*************** main ****************
#include "stdafx.h"
#include "a.h"
#include "b.h"

int main(int argc, char* argv[])
{
        A a;
        B b;
        return 0;
    }
```

Output will be:

```
    1 2 3
    1 2 3
```

But the above approach is not what people normally do. Because this approach requires each file which uses those global variables to declare all of them which keyword extern one by one, causing redundant code and is error-prone. The normal approach is to create a separate header file and put all **extern** declarations of the global variables in it. Logically this header file should have the same name as the source file, but it can be different (as shown in the following example). Then all files which accesses the global variables can simply include this header file:

```
    //************ globals.cpp *****************
    #include "stdafx.h"

    int array[3] = {1, 2, 3}; // array is the global object
    int ANY = 4;


    //************** Any.h *****************
    #ifndef _ANY_H
    #define _ANY_H

    extern int array[];
    extern int ANY;

    #endif


    //*************** A.cpp *******************
    #include "stdafx.h"
    #include "A.h"
    #include <iostream.h>
    #include "Any.h"

    A::A()
    {
       cout << array[0] <<" "<< array[1] <<" "<< array[2] <<"
  "<< ANY << endl;
```

```
}


//************** B.cpp ********************
#include "stdafx.h"
#include "B.h"
#include <iostream.h>
#include "Any.h"

B::B()
{
    cout << array[0] <<" "<< array[1] <<" "<< array[2] <<"
  "<< ANY << endl;
}


//*************** main ***************
#include "stdafx.h"
#include "a.h"
#include "b.h"

int main(int argc, char* argv[])
{
    A a;
    B b;
    return 0;
}
```

Output will be:

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

1.13 1.13 Constant Global Variables

In a sense, the reason you want to use global variable is to pass data between several objects which do not have much coupling between them. Therefore global variables should normally not be constant. When you use a constant global variable, you actually want a symbol or an alias for a constant. Therefore it is clearer to use a **#define** to define this constant. For this reason, C++ compiler would not support syntax "**extern const ...**". If you want to have a constant global variable, you should put it in the header file (Any.h). Compiler will treat it in the same way as a #define.

1.14 1.14 Creation and Deletion of Global Variables

The global object's constructors are called before any program is execuated, and their destructors are called after all program ends.

```
//************** A.h **************
class A
{
public:
	void Say();
	A(char * name);
	virtual ~A();
private:
	char * m_name;
};

//************* A.cpp ***************
#include "stdafx.h"
#include "A.h"

A::A(char * name) : m_name(name)
{
	char buf[80];
	sprintf(buf, "Constructor of %s\n", m_name);
	printf(buf);
}

A::~A()
{
	char buf[80];
	sprintf(buf, "Destructor of %s\n", m_name);
	printf(buf);
}

void A::Say()
{
	char buf[80];
	sprintf(buf, "My name is %s\n", m_name);
	printf(buf);
}

**************** B.h *********************
class B
{
public:
        B();
        virtual ~B();
};

**************** B.cpp ********************
#include "stdafx.h"
#include "B.h"
#include "A.h"

A g_b("GLOBAL_IN_B");

B::B()
{
        char buf[80];
        sprintf(buf, "Constructor of class B\n");
        printf(buf);
}

B::~B()
{
        char buf[80];
        sprintf(buf, "Constructor of class B\n");
        printf(buf);
}

*************** Test.cpp *****************
#include "stdafx.h"
#include "a.h"

A g_main("GLOBAL_IN_MAIN");
extern A g_b;

int main(int argc, char* argv[])
{
        printf("\nBeginning of main!\n");
        g_b.Say();
        printf("End of main!\n\n");
        return 0;
}
```

The output will be:

```
        Constructor of GLOBAL_IN_MAIN
        Constructor of GLOBAL_IN_B
```

```
Beginning of main!
My name is GLOBAL_IN_B
End of main!

Destructor of GLOBAL_IN_B
Destructor of GLOBAL_IN_MAIN
```

Because global objects are created before any code is executed, they must not be any resource that can not be initialized "on the spot" and has to be initialized by the program. If you have to have a global resource like that, create a global reference (pointer) to the resource, which can be initialized to NULL on the spot, then create and initialize the resource in the program.

1.15 1.15 Name Conflict Between Local and Global Variable

If in a block a local variable of the same name as the global variable is declared, the local variable will suppress the global one from the declaration line onwards. To access the global variable from this block, use unary scope resolution operator "::" in front of the identifier.

```
int x = 10;        // Global variable

int main ( )
{
  int x = 4, y;
  y = x/ ::x;      // Value should be 0.4
}
```

1.16 1.16 Storage Class

Each variable or object has its attributes including storage class, scope and linkage.

♦ ☐ **auto**

Variables of automatic/local storage class are created when the block in which they are defined is entered, and destroyed when the block is exited. **Local variables are by default of automatic storage class.** So the “auto” specifier is rarely used.

- ☐ **register**

  When “register” specifier is placed before an automatic/local variable, it is to suggest the compiler to keep this variable in one of the computer’s high-speed registers in stead of memory. The loading and saving time is shorter than memory. By placing a heavily used variable into the register, the total run time can be reduced. However, the compiler may ignore this suggestion, if there is no register available. On the other hand, today’s optimizing compiler are capable of recognizing frequently used variables can decide to put them in register without the need for a register declaration.

  Not often used.

- ☐ **static**

  If a local variable is declared “static”, it is still only known in the function in which they are defined, but when the function is exited, that variable is not destroyed, instead it is retained for next use.

  Before the first time of use, if not specially initialized, all numeric variables are initialized to zero.

  If there is an initialization in the function such as "static int x = 1;" it only works first time the function is called. Otherwise the variable can not keep the value of last time and therefore has no difference with normal local variables.

  This kind of scope is mainly used by procedural languages. For OO it is rarely needed.

- ☐ **extern**

  Global variables (and function definitions) are created by placing variable declarations outside any function definition. **Global variables default to storage class specifier “extern”.** They keep their values throughout the execution of the program, and can be referred by any functions that follows their declarations or definitions in the file. According to PLP, the use of global variables should be avoided unless there is a special need.

1.17 1.17 Scope

- ☐ **File scope**

  A physical file is of file class. Therefore, global variables, function prototypes and function definitions, which are out of any function body, are of file scope.

- ☐ **Function scope**

  Because a function is not the smallest unit in C++, only labels are with function scope. Labels are identifiers followed by a colon, i. g., the case label "case 3.5: " in "switch" structure. Labels are used in "switch" structure or "goto" statement to mark the location of a statement, so that other statement knows where to go. They are implementation details that functions hide from one another.

- ☐ **Block scope**

  Because block is the smallest unit in C++, most variables/identifiers are of block scope. They are local variables declared in a block.

- ☐ **Function-prototype scope**

  Identifiers in the function-prototype declaration have function-prototype scope. The identifiers can be reused elsewhere without ambiguity. This is the lowest scope in C++.

1.18 1.18 Recursion

A recursive function is a function that calls itself either directly or through other function. There are such kind of problems that the direct solution can not be easily expressed, but the relationship between the problem and its simpler version is explicit. Thus the problem can be simplified repeatedly reduced to a smaller version, until it reaches the simplest case called "base case", and finally becomes known. In short, recursion keeps producing simpler version of the original problem until the base case is revealed.

From logical point of view, recursion is invalid or impractical: you can not use an unknown solution to solve a problem. But in C++ recursion only means to make another copy of the function and call it again. So recursion in C++ is not real recursion. Therefore it is simple.

A smart use of recursion on other issues:

```
int main()
{
  int c;

  if( ( c = cin.get() ) != EOF)
  {
    main( );
    cout << c;
  }
```

```
    return 0;
  }
```

♦ ☐ **Recursion & iteration**

All recursive solutions can be substituted by iterations. Iteration solutions are better than recursion in respect to performance, because recursion produces a series of function calls and copies of variables, thus takes more overhead, which iteration can avoid. Recursion is only good when it can mirror the real-world problem more naturally, thus the program is easier to understand and debug.

♦ ☐ **Exponential complexity caused by recursive calls**

If in a recursive function there are three recursive calls, and number of recursion layers is n, then the total of recursive calls will be 3^n. This is called "exponent explosion". Try to avoid this situation.

1.19 1.19 Recursion Exercise 1 – Binary Search of an Array

```
  #include <iostream>
  #include <math>
  #include <stdlib>

  void bisearch (const int number, int & location, const int
array[],
        int from, int to);

  int main ()
  {
    const int asize = 13;
    int location, temp;
    int student [asize] = {0,1,2,3,4,5,6,7,8,9,10,111,222};
    bisearch (7, location, student, 0, asize-1);

    if(location < 0)
      cout << "The number " << number <<
           " is not in this array." << endl;
    else
      cout << "The "<< location << "th element contains your
number "
         << number << endl << endl;
```

```
     return 0;
  }


  void bisearch (int number, int & location, int array [], int from,
int to)
  { // ********** Base Case ************
     int middle = (from + to) / 2;

     if(array[middle] == number)
     {
        location = middle;
        return;
     }

     if( ( number > array [to]) || ( number < array [from] ) )
     {
        location = -1;
        return;
     }

     if(middle == from)
        if(array[to] == number)
        {
           location = to;
           return;
        }
     else
     {
        location = -1;
        return;
     }

     // ********** Recursion *************
     if( number > array [middle])
        bisearch (number, location, array, middle, to);
     else
        bisearch (number, location, array, from, middle);

     return;
  }
```

1.20 1.20 Recursion Exercise 2 – Towers of Hanoi

```
#include <iostream>

void hanoi (char, char, char, int);

int main ( )
{
  int n = 4;
  hanoi ('a', 'b', 'c', n);
  cin >> n;
  return 0;
}

void hanoi (char from, char via, char to, int n)
{
  if(n==1)
  {
    cout << from << " => " << to << endl;
    return;
  }

  n--;
  hanoi (from, to, via, n);
  cout << from << " => " << to << endl;
  hanoi (via, from, to, n);
  return;
}
```

1.21 1.21 Order of Evaluation on Operands

```
a = function1 (a, b) + function2 (c, d);
```

In C++ the order of evaluation of the two operands beside some operators such as "+" is indefinite. If the result of the calculation depends on the order of evaluation of the two operands i. e. the calling order of the two functions - it is a lousy design indeed - then the result will be indefinite.

1.22 1.22 Constant Variable

To put "const" qualifier in front of a parameter in both the **function prototype** and **function definition** is to tell the compiler that the function doesn't modify the variable. A constant variable or pointer should be initialized when declared.

1.23 1.23 *Principle of Least Privilege (PLP)*

The principle of least privilege is to always assign least data accessing privilege to the program. In most cases it is achieved by using of qualifier "const". "const" is used to pass variables and arrays to functions in which they should not be modified. It is also used to define a local variable that shouldn't be changed. Any attempt to modify the constant variable will be checked out by compiler before the program is run. Using this principle to properly design software can greatly reduce debugging time and improper side effects, and can make a program easier to modify and maintain.

1.24 1.24 *Inline Functions*

Some functions are of quite small size, but quite frequently called. Compared with the function call overhead, the program size reduced by not repeatedly include the block of statements may be trivial. In these cases we put "**inline**" qualifier in front of the function definition to tell compiler to insert the body of the called function into the calling function to avoid a call.

When the **inline** function is changed, all functions that call it should be re-compiled.

Keyword **inline** is specially useful for wrapper classes.

1.25 1.25 *Reference*

There are two ways to pass arguments to called functions in many languages: "call-by-value" and "call-by-reference".

♦ ☐ **call-by-value**

When an argument is passed by value, only a copy of the argument is passed to the called function. The copy constructor of the passed object is called to make the copy, requiring extra overhead. The original values are untouched. This is good for data security, but bad for performance.

♦ ☐ **call-by-reference**

A reference is declared by putting "&" in front of the identifier in the function header. A reference is not a variable. It is just an alias of a variable.

When we talk about a variable, it is actually the address of one memory cell used to hold different values. In machine language we would directly use the address to represent the variable, but in high-level languages identifiers are used to present addresses. So when we write

```
   int a = 33;
```

we actually created an alias to represent the address of a memory cell holding the value of 33. When compiler sees “a”, it just converts it to the corresponding address.

Therefore, when we write

```
   int & b = a;
```

we just created another alias for that address. Now for the same address we have two alias: “a” and “b”. Although "b" is created "out of" a, but once created they are equal.

Therefore, when a object is passed by reference, a new alias is created in the called function to refer to the address of that object. With this alias the called object can do anything directly to the object itself.

```
   void double(int & x)  // function definition indicating the
  reference
   {  x * = 2; }

   int main ( )
   {
     int a = 3;
     double(a);
   }
```

♦ ☐ **Comparison between call-by-value, call-by-reference and call-by reference with pointer**

	Call-by-value	Call-by-reference	Call-by-reference with pointer
call	sum(a, b);	sum(a, b);	sum(&a, &b);
prototype	int sum(int a, int b)	int sum(int &a, int &b)	int sum(int * ptr1, int * Ptr2)

From the above form you can see that the most explicit expression is call-by-reference with pointer. The calling statement of call-by-reference is the same as call-by-value, therefore the programmer may forget that he is calling by reference.

1.26 1.26 Default Arguments

When the arguments you pass to the called function are most probably of some definite values, you can specify these values in the function prototype. When the values of the parameters are the default ones, you can omit the parameters.

```
void count(int x = 1, int y = 1, int z = 1)
{...}

int main ( )
{
   count();
   count (2, 3);
}
```

Only the rightmost arguments can be omitted -- you can not provide the 1st and 3rd argument and omit the middle one.

1.27 1.27 Overloading Functions

Functions with the same name but different signature (i.e. argument list) are called overloaded functions. Overloaded functions are recognized by the compiler through the argument list in the call:

```
#include <iostream>

void print(int i)
{
   cout << "int i = " << i << endl;
}

void print(char c)
{
   cout << "char c = " << c << endl;
}

int main()
{
   int i = 1234;
   char c = 'C';
   print(i);
   print(c);
   cin >> i;
}
```

1.28 1.28 EOF

An integer constant defined in the "**iostream.h**" header file. It is a system-dependent keystroke combination. For example, in MS-DOS it is “Ctrl-Z”, while in UNIX it is “Ctrl-D”. In other system it may be “EOF” or even “Stop here!”. The value of EOF in ANSI standard is a negative integer value, normally −1. In Borland C++ it is also −1.

1.29 1.29 New Line

A new line is regarded as a character in C++: “\n”. You can use such statement to detect a Return:

```
if(c = = ‘\n’)…
```

2. 2. PRIMITIVE TYPES AND OPERATORS

2.1 2.1 *Modulus Operator %*

It yields the remainder after integer division. This calculation has the same priority as manipulation and division.

2.2 2.2 *Conversion Between Different Types in Calculation*

In the calculation of different types of variables, C++ promotes the lower-level variable to higher-level variable, and the result is higher-level. For example, the result of int with float is float.

2.3 2.3 *Do Not Rely On the Precision of Float Numbers*

Floating-point numbers are represented only approximately in most computers. So do not compare floating-point values for equality. Instead, test whether the absolute value of the difference is less than a specified small value.

2.4 2.4 *Assignment Operators*

c += 7	c = c + 7
c -= 7	c = c – 7
c *= 7	c = c * 7
c /= 7	c = c / 7
c %= 7	c = c % 7
a = ++b	b = b + 1, then a = b
a = b++	a = b, then b = b + 1
a = --b	b = b – 1, then a = b
a = b--	a = b, then b = b – 1

"++b" or "b++" can either be used as a operand in a expression, or as a independent statement. Using assignment operator can save a bit of compiling and running time. It is useful when the statement is in a loop which will be run many times.

2.5 2.5 *Value of Assignment Expression*

Because "=" operator operates **from right to left**, such expression can be used: a=b=c=d=3. An assignment expression itself also has a value:'

```
if( (a = cin.get( ) ) != "a")…
```

2.6 2.6 The Integer Value of Character

The integer value of a character is its **ASCII code**, for example, 97 for 'a', 98 for 'b', 99 for 'c',... To acquire this value, use variable type converting operator "static_cast < >":

```
  #include <iostream>
  #include <iostream>
  int main ()
  {  char keyboard;
     int a;
     do
     {  cout << "Enter one character, and I will tell you its ASCII:
\n \n";
        cin >> keyboard;
        cout << (a = static_cast <int> (keyboard)) << endl <<"\n
\n \n \n";
     }while (a != 101 && a != 69);
     return 0;
  }
```

2.7 2.7 Logic Operators

&&: AND

||: OR

!: Logic Negation, turning the value of a logic expression from true to false or from false to true.

3. 3. CONTROL STRUCTURES

3.1 3.1 *Control Structures*

C++ has only seven control structures:

Sequence		
Selection	"if"	single-selection structure
Selection	"if/else"	double-selection structure
Selection	"switch"	multi-selection structure
Repetition	"while"	
Repetition	"do/while"	
Repetition	"for"	

3.2 3.2 *if*

```
if( grade >= 60 )
  cout << "Passed! \n";
```

3.3 3.3 *if-else*

```
if( grade >= 60 )
   cout << "Passed! \n";
  else if( grade >= 40 )
   cout << "Failed! \n";
  else
   cout << "Shamed! \n";
```

3.4 3.4 *Conditional Operator*

Conditional operator "?" and ":" are used to form an operand or statement, the value of which is chosen from two options, depending on the value of the condition expression:

```
condition expression ? option 1 : option 2;
```

Example:

```
cout << ( grade >= 60 ? "Passed! \n" : "Failed! \n");
grade >= 60 ? cout << "Passed! \n" : cout << "Failed! \n";
```

3.5 3.5 *while*

```
while ( condition expression ) statement;
```

If the condition expression is true the statement will be performed, and the condition expression checked again. If there is no action in the statement to cause the condition expression to become false eventually, it will cause a "infinite loop".

"while" structure actually is an "if" structure with a "go to" statement at the end to go back to "if".

3.6 3.6 for Loop

```
for(expression 1; expression 2; expression 3)
  statement
```

expression 1: initialize the loop's control variable;

expression 2: loop-continuation condition;

expression 3: increment the control variable.

As a complete loop, first the condition is checked, if satisfied, the statement is executed, then the control variable is incremented.

Example:

```
for( int n = 1, m = 1, counter = 1; counter <=10; counter =
counter +2)
  cout << counter << endl;
```

Notice that after the loop the variable "counter" will have a value of 12.

Expression 1 and 3 can be lists of comma-separated expressions. The comma used here are "comma operators". The value of the list is the value of the last expression. It is usable when you need more than one local variable in the loop. **By initializing these variables inside the loop instead of outside the loop, it makes the program clearer, and also conforms with the Principle of Least Privilege (PLP)**.

If expression 1 is missing from the "for" header but it has been initialized before the loop, the value will be used. If expression 2 is missing, the continuation condition will be by default true and the loop will run infinitely. If the control variable is incremented in the loop body, then expression 3 can be saved. Anyway the colon can not be saved.

Many programmers prefer expression 3 to be "counter ++", because increment occurs only after the loop body is executed.

3.7 3.7 *Avoid Modifying the Control Variable in the Loop Body*

It is not a error but it can produce unexpected results.

3.8 3.8 *switch*

```
switch (controlling expression)
{
   case a:
   case b:
   statements;
   break;
   case c:
   statements;
   break;
   default:
   statements;
}
```

a, b, c are called "case labels". They can only be a constant, or a character represented by 'a' or 'A'. When "switch" statement is executed, the case labels are one by one compared with the controlling expression. When one is equal to the expression, all the statements after that case label will be executed, until meeting one "break" statement. So putting different labels together simply means "OR".

If a "default:" label is put, when no case label is matched, the statements after the "default:" label is executed. It is not a must but a good practice to always put a "default" label even if you are absolutely sure your program is free of bugs.

A "break" statement is not required after the "default" case if it is at the last.

3.9 3.9 *do-while*

```
do
{  statement1;
   statement2;
}while (continuation statement);
```

The only difference between this and "while" statement is that the continuation condition is checked after the body had be executed.

```
#include <iostream>
```

```
   int main ()
   {  char keyboard;
      int a;
      do
      {
         cout << "Enter one character, and I will tell you its ASCII:
 \n \n";
         cin >> keyboard;
         cout << (a = static_cast <int> (keyboard)) << endl <<"\n
 \n \n \n";
      }while (a != 101 && a != 69);
      return 0; }
```

If we use "while" we have to add one statement before the loop: "a=0;".

3.10 3.10 *"break" and "continue"*

In the body of "**while**", "**for**", "**do/while**" or "**switch**", "**break**" statement causes immediate exit from the statement, while "**continue**" skips the statements after it until the end of loop, and begins as normal the next loop.

Notice that break can only exit one layer of loop. If there are more than one layers of nested control structure, break can not exit all of them:

```
   int main()
   {
      fstream file1("test.txt", ios::out);
      int i, j;

      for (i = 0; i < 10; i++)
      {
         file1 << "Outer loop: i = " << i << endl;

         for (j = 0; j < 10; j++)
         {
            file << "Inner loop: j = " << j << endl;
            if(i == 3 && j == 3) break;
         }

         file1 << endl;
      }
```

```
        file1.close();
        cin >> i;
    }
```

In this case, a unstructured programming technique **goto** can be used.

4. 4. ARRAYS

4.1 4.1 *Declare and Initialize Array*

To initialize the array when declaring:

```
int x, y, student[5] = {0, 1, 4, 9, 16};
```

The numbers in {} are called **initializers**. If the number of initializers are less than the number of the array elements, the remaining elements are automatically initialized to zero. There must be at least one initializer in the {}. **But such kind of expression can only be used in declaration. You can't use " {0, 1,..} " in assignment.**

4.2 4.2 *Array Size Must Be Constant*

Instead of directly placing a figure such as "21" in the braces of the array declaration, it is better to place a constant variable. In such way when you need the change the array size you only need to change one place.

```
const int size = 21;
```

The other reason is to avoid "**magic number**": if number 21 frequently appears in the program, and an other irrelevant "21" happens to appear, it will mislead the reader that this "21" has something to do with the former one.

Only constant variable can be used as array size. Therefore you can not make the array size dynamic by inputting an integer from keyboard in run time and use it as array size.

4.3 4.3 *Array Size*

In Java, an array is an object with an array with fixed-size, plus data member ("length") indicating the array size, and compiling-time boundary checking. But in C++ an array is just an address of the first array element. Declaring the size of the array can only help compiler to allocate memory for the array, but the compiler never checks whether the array bound or size is exceeded:

```
int main ()
{
  int b, a[3] = {0,1,2};
  a[3]=3;
  cout << "a[3] = " << a[3] <<endl;
```

```
    a[4]=4;
    cout << "a[4] = " << a[4] <<endl;
    cin >> b;
  }
```

The problem that will happen if you exceed the array bound is: because the compiler was told that the array was only of 3 elements, so it may have put other variables in the succeeding memory locations. Therefore by declaring an array of 3 elements then putting a value into the 4th element, you may have overwritten another variables and produced very serious logic errors which is very difficult to find out.

Therefore, the size of an array should be carefully observed.

4.4 4.4 Array and Pointer

There are two widely used formats to represent a block of data: a pointer and an array:

```
  char * cPtr;
  char buf[80];
```

The name of an array is a constant pointer to the first element of the array. Therefore, the name of a charcter array is equal to **const char ***. You can not assign anther address to the array name like

```
  buf = cPtr;
```

The other difference between a pointer and array is: a pointer such as **char * cPtr** can point to anywhere, most probably somewhere in the OS's territory that you can't access. An array such as **char buf[80]** however, points to a block of memory allocated by the compiler.

If a block of characters ends with **NULL** i.e. 0, it can be treated as a string, which is recognized in most applications.

♦ ☐ Double-quoted constant string

A double-quote enclosed string represents a **const char const *** pointing to some compiler-allocated space holding the string, with the last character being **NULL**. Therefore, when you say

```
  char * cPtr;
  cPtr = "Hello world!";
```

Compiler will allocate a continuous 13 bytes (last byte to hold

NULL or 0) some where, set them to be "Hello world!", and assign its address to pointer **cPtr**. Because the bytes are constant, you can never amend the content of the string.

Therefore, if you want to amend the content of "Hello world!", you can not directly assign its address to a pointer. You have to copy the constant bytes into a character array:

```
char str[100];
strcpy(str, "Hello world!");
char * substr = strstr(str, "world");
memcpy(substr, "W", 1);
msg(str);
```

The output will be

```
Hello World!
```

A special case is when you initialize a character array with the constant string:

```
char buf[80] = "Hello world!";
char buf1[] = "Hello Frank!";
```

Compiler will create a character array of the specified length or the length of the constant string if not specified, than fill it with the content of the constant string. You can then amend the content of the array later.

However, you can not assign a constant string to an array name after it is already created:

```
char buf[80];
buf = "Hello world!"; // Not allowed!
```

Because as said before, the array name is a constant pointer which can not be assigned.

♦ ☐ Formatting character array

When you create a char array by saying

```
char buf[80];
```

every byte of it is uninitialized. So it is not a NULL-terminated string and can not be used in string manipulation functions such as **strcpy**, **strlen**, **strcat**, etc. To turn it into an empty but legal string:

```
sprintf(buf, "");
```

To write into a char array:

```
sprintf(buf, "%s, %.6d, %c, %.3f, 0x%.8x",
        "Hello World!", 1234, 'A', 123.4, 0xaabbbb);
```

The output will be:

```
Hello World!, 001234, A, 123.400, 0x00aabbbb
```

For a list of all format specifications, search MSDN with title "Format Specification Fields: printf and wprintf Functions".

4.5 4.5 Pass Array to Function

```
int calculate (int member[], int size)
{
   int i, average = 0;

   for( i = 0; i < size; i++)
      average += member [i];

   average /= size;
   return average;
}

int main ( )
{
   int average, student [5] = {67, 93, 88, 89, 74};
   average = calculate (student, 5);
   cout << "The average score is " << average << endl;
   cin >> average;
   return 0;
}
```

To indicate to the compiler that a function is expecting an array, there must be “[ ]” following the array name in both the function prototype and function definition.

C++ automatically passes arrays by reference. Dereferencing operator “&” is not used, because the name of the array is the address of the first array element - the name of array "student" is equal to "&student[0]".

4.6 4.6 Searching Array

There are two ways to search for a figure in an array:

Linear search: compare each array element one by one with the searched figure until it is found or reaching the end of the array. Average comparison time: half of the array size.

Binary search: can only be applied on sorted array. By checking the middle element you will find out which half of the array contains the element. Maximum comparison time: $\log_n{}^{arraysize}$. If an array needs to be searched frequently, then it is worthwhile to spend a great time to sort it first. Refer to the program in "3.11 Exercise – Binary Search of an Array".

4.7 4.7 Multiple-Subscripted Array

```
#include <iostream>

void print (int [] [3], int, int);

int main ( )
{
  int x;
  int array1 [2] [3] = { {1, 2, 3}, {4, 5, 6}};
  int array2 [2] [3] = {1, 2, 3, 4};
  print (array1, 2, 3);
  print (array2, 2, 3);
  cin >> x;
  return 0;
}

void print (int a [] [3], int first, int second)
{
  for(int i = 0; i < first; i++)
  {
    for(int j = 0; j < second; j++)
          cout << a [i] [j];

    cout << endl;
  }

  return;
}
```

♦ ☐ **Declaration of the dimensions**

When passing the array to a called function, each dimension of the array should be declared in both the function prototype and function definition. The first dimension does not need a number, just like single-dimension arrays, but the subsequent dimensions does.

An n x 3 array is located in the memory in such a sequence: (0, 0) (0, 1) (0, 2) (1, 0) (1, 1) (1, 2)... If the compiler knows the number of columns which is 3, it will put the fist element (0, 0) of first row on the first memory location, the first element on second row on the fourth memory location, the first element on third row on the 7th location,...Without the number of columns the compiler is not able to organize memory for the array.

♦ ☐ Initializers

The initializers of an array can work in two ways:

initializers for each row are enclosed in second class of "{ }";

all initializers are enclosed in only one "{ }", and apply to elements first by row then by column.

4.8 4.8 String Handling Functions

Their function prototypes are in "**string.h**" header file. Data type "size_t" used in these functions is an unsigned integer type.

char * **strcpy**(char * s1, const char * s2) copy s2 into s1

char * **strncpy**(char * s1, const char * s2, size_t n) copy n of s2 into s1

char * **strcat**(char * s1, const char * s2) append s2 to s1

char * **strncat**(char * s1, const char * s2, size_t n) append n of s2 to s1

int **strcmp**(const char * s1, const char * s2) compare s1 with s2. Return positive, 0 or negative

int **strncmp**(const char * s1, const char * s2, size_t n) compare n of s1 with s2

char * **strtok**(char * s1, const char * s2) break s1 into tokens separated by s2

size_t **strlen**(const char * s) length of s, **not counting NULL**

5. 5. POINTERS AND STRINGS

5.1 5.1 Pointer Declaration and Initialization

```
int x, y, qty, *xPtr1 = 0, *yPtr1 = NULL, *qtyPtr1 = &qty;
float z, *zPtr1
zPtr1 = &z;
```

Pointer variable xPtr1, yPtr1, zPtr1 and qtyPtr1 are variables containing the addresses of another normal variables. " ***** " indicates that the following variable is a pointer. " **int *** " indicates that the following variable is a pointer pointing to an integer. When used in type declaration, function prototype or function definition, "*" is not a dereferencing operator.

"NULL" is defined as 0 in <**iostream**> and several standard library header files. A pointer with a value of 0 points to nowhere.

5.2 5.2 Casting Between Numeric Address and Pointer

You can acquire the numeric address contained in a pointer and vice versa through casting between the integer type and the pointer type:

```
  typedef unsigned long      DWORD; // each unsign char takes
four byte
  typedef unsigned char      BYTE;  // each unsign char takes one
byte

  struct Any {
      BYTE m_ba1[100];
      BYTE m_ba2[100];
      BYTE m_ba3[100];
  };

  void main()
  {
      Any * pAny = new Any;
      DWORD dwBase = (DWORD)pAny;  // casting pointer to
DWORD address
      cout << "Offset of m_ba1 is " << (DWORD)pAny->m_ba1 –
dwBase << endl;
      cout << "Offset of m_ba2 is " << (DWORD)pAny->m_ba2 –
dwBase << endl;
```

```
        cout << "Offset of m_ba3 is " << (DWORD)pAny->m_ba3 –
   dwBase << endl;

        ::memset(pAny1->m_ba3, 123, 100);
        Any * pAny2 = (Any *)(dwBase + 200);  // casting
   DWORD address to pointer
        cout << "pAny1 offset by 200: m_ba1[23] = "
            << (int)(pAny2->m_ba1[23]) << endl;
    }
```

Output will be:

```
   Offset of m_ba1 is 0
   Offset of m_ba2 is 100
   Offset of m_ba3 is 200
   pAny1 offset by 200: m_ba1[23] = 123
```

5.3 5.3 Constant Pointer and Pointer to Constant

A pointer can be pointed to a constant, and the pointer itself can also be a constant. If a pointer is a constant pointer, it should be initialized when declared, and it can not be pointed to any other variable.

Suppose p1 is a pointer pointing to a "Person" object. There are three kinds of use of "const":

print(Person * **const** p1)   pointer is constant but object is not.

print(**const** Person * p1)   object is constant but pointer is not

print(**const** Person * **const** p1)   both the object and the pointer are constant

5.4 5.4 Pass Pointer By Reference

If we want to pass a pointer to a function and modify that pointer in the function, we can not say

```
   Type * ptr = new Type;
   Test(ptr);

   void Test(Type * ptr0)
   {...}
```

Because the pointer is passed by value, and the original pointer "ptr" will keep unchanged even if you change the "ptr0" in the function. You have to pass the pointer by reference:

```
Test(&ptr);

void Test(Type ** ptr0)
{
  (*ptr0) = &x;
  ...
}
```

In this way you can use " **(*ptr0)** " to access the original pointer "**ptr**".

5.5 5.5 *Receive array with pointer*

When passing the name of an array to a called function, the name of array is an address of the first element. Because of this, a pointer can be used in the called function to receive the array. Then by moving the pointer (e. g. pointer ++), all the rest array element can be accessed.

5.6 5.6 *Pointer Expressions and Arithmetic*

There are three kinds of arithmetic operations that can be done to a pointer:

- ☐ **Increment**

```
pointer ++ / --;
pointer += / -= 3;
```

It means moving the pointer to the next or previous 3rd element, not just increase the value of the pointer by 3. If the size of the variable to which the pointer is pointing to is 4, then actually the value of the pointer will be increased by 3 x 4.

- ☐ **Difference**

```
int x = pointer2 – pointer1;
```

If pointer1 is pointing to the 5th element and pointer2 the 8th, then x will be 3, not 3x4 (suppose the type size is 4).

- ☐ **Assignment**

```
pointer1 = pointer2;
```

pointer1 and pointer2 must be of the same type, otherwise a cast operator must be used to convert the type of the pointer. The only exception is when pointer1 is declared to be type "**void**" (i.e., **void ***). Any type of pointer can be assigned to a pointer to void without casting. However, it can not be conversed.

♦ ☐ **Comparison**

The two pointers for comparison must be pointing to the same array. The result may show which one is pointer to a higher-numbered element.

Pointer arithmetic (including increment and difference) is meaningless unless performed on one array, because we are only sure that array elements are located one after another. We can not assume two separate variables are put together in the memory.

The following four expressions is doing the same thing:

```
cout << array[4];
cout << *(array + 4);
cout << arrayptr1[4];
cout << *(arrayPtr1 + 4);
```

5.7 5.7 Pointer Offset and Subscription

The reason pointer concept is created initially is not to point to a single primitive, but to manipulate arrays and strings and custom types.

When a pointer is pointed to an array or a string, it is actually pointed to the **first element** of the array (subscription 0). To refer to the elements in the following array, a pointer **offset** or **subscription** can be used:

```
int b[5];
int * ptr = b;
*(ptr + 3) = 7;
// or ptr[3] = 7;
```

*(ptr+3) refers to the element with subscrip 3 (4th element). This element can also be represented by ptr[3].

If you **point a pointer to the middle of an array**, what will happen? The pointer can be used as the name of a new array, whose first element is the element to which the pointer is pointing to.

```
#include <iostream>
#include "conio.h"

int main()
{
    char * a1 = "0123456789";
    int a2[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    char * ptr1 = &a1[3];
    int * ptr2 = &a2[3];
    cout << "ptr1[0] = " << ptr1[0] << ",  ptr1[6] = " <<
ptr1[6] << endl;
    cout << "ptr2[0] = " << ptr2[0] << ",  ptr2[6] = " <<
ptr2[6] << endl;
    cout << "\n Press any key to quit..." << endl;
    getch();
 }
```

Output result:

```
 ptr1[0] = 3,
ptr1[6] = 9
 ptr2[0] = 3,
ptr2[6] = 9
```

5.8 5.8 "sizeof" Operator

C++ provides unary operator "**sizeof**" to determine the number of bytes occupied by an array, variable, constant or type name such as "int", "char", etc. When applied to type name, "( )" is needed:

```
int a, b, n;
float x, array [3];
a = sizeof x;
b = sizeof array;
n = sizeof array / sizeof (float);
```

The last statement is to find the number of elements of an array.

Notice that the number of bytes for a certain type is different for different systems. C++ is platform-dependent, not like Java.

5.9 5.9 Size of String and Character Array

Although a string “char *” and a character array “char [ ]” can be used in the same way in most cases, when it comes to size issue, they are different. The size of the “char *” is the size of the char pointer, while the size of the char array is the size of the array:

```
char * temp1 = "abcdefg";
char temp2[20] = "abcdefg";
char temp3[30] = "abcdefg";
cout << "Size of a char * is " << sizeof temp1
```

```
        << ", size of char[20] is " << sizeof temp2
        << ", size of char[30] is " << sizeof temp3 << endl;
```

Output will be:

```
Size of a char * is 4, size of char[20] is 20, size of char[30] is 30
```

5.10 5.10 Function Pointer and Call Back Technique

Function pointers are mainly used to achieve call back technique, which will be discussed right after.

Just like an array name is the address of the first array element, a function name is a actually the starting address of the function code. A function pointer holds the address of a function, just like a float pointer holds the address of a float variable. A function pointer can point to either a global function or a class function.

♦ ☐ **Global function pointer**

```
#include "stdafx.h"

typedef void(*CALLBACK_FUNCTION)(int);  // define the function pointer

void Bark(int nRepeat)  // actual function to be passed to the pointer
{
    for(int i = 0; i < nRepeat; i++)
        printf("Bark!\n");
}

void Cry(int nRepeat)  // actual function to be passed to the pointer
{
    for(int i = 0; i < nRepeat; i++)
        printf("Woooo!\n");
}

void Server(CALLBACK_FUNCTION m, int nRepeat)
{
    (*m)(nRepeat);  // Invoking through function pointer
    printf("\n");
}
```

```
int main(int argc, char* argv[])
{
    Server(&Bark, 2);  // passing function pointer
    Server(&Cry, 3);
    return 0;
}
```

```
Bark!
Bark!

Woooo!
Woooo!
Woooo!
```

♦ ☐ Class method pointer

```
#include "stdafx.h"

class Creature {};

typedef char * (Creature::*CALLBACK_METHOD)(int); // define the function pointer

class Cat : public Creature {
public:
    Cat(char * name) : m_name(name) {}

    char * Miao(int nRepeat)  // actual function to be passed to the pointer
    {
        for(int i = 0; i < nRepeat; i++)
            printf("Miao!\n");
        return m_name;
    }

private:
    char * m_name;
};

class Snake : public Creature {
public:
    Snake(char * name) : m_name(name) {}

    char * Ssss(int nRepeat)  // actual function to be passed to the pointer
    {
        for(int i = 0; i < nRepeat; i++)
            printf("Ssss!\n");
        return m_name;
    }

private:
    char * m_name;
};


void Server(Creature * pCreature, CALLBACK_METHOD m,
int nRepeat)
{
    char * name = (pCreature->*m)(nRepeat); // invoking
through function pointer
    char buf[80];
    sprintf(buf, "My name is %s!\n\n", name);
    printf(buf);
}

int main(int argc, char* argv[])
{
    Cat * pCat = new Cat("Jessy");
    Snake * pSnake = new Snake("John");
    Server(pCat, (CALLBACK_METHOD)&Cat::Miao, 2);  //
passing function pointer
    Server(pSnake, (CALLBACK_METHOD)&Snake::Ssss, 3);
    return 0;
}
```

```
Miao!
Miao!
My name is Jessy!

Ssss!
Ssss!
Ssss!
My name is John!
```

♦ ☐ Call Back Technique

Callback technique is an effort to seperate "what" from "how". When you want your program to be applicable to different use cases, you may find that at a certian point in your program, you need to invoke a function which has the same signature but different implementation for different use cases. In other words, for every specific case, the type of information passed to and returned by that function (which represents "what") is the same, but the implementation of the function (which represents "how") is different. Different client (who uses your program) may have different implementations.

In this case, you have to provide a mechanism so that the client may register his own version of that function to your program before the invoking point, so that your program knows which one to call. This technique is called "callback".

There are three ways to achieve call-back.

The OO approach of callback is to let the client class inherit from and implement an interface. In your code you simply hold an interface pointer and call the interface methods through that pointer. The client program will create his implementation object, assign its pointer to the interface pointer in your class before the calling pointer in your class is reached.

However, if the client class is already finished and did not implement the interface you want, you have to use a less OO approach. The client programmer (or your do it for him) should write a separate matching class for the client class, which inherit from the desired interface with the function you will call back, which provides the service you need by manipulating client-class objects. In your code you have a pointer to the interface and invoke the service provided by the separate class.

The least OO approach is to use function pointers like the above example.

5.11 5.11 Array of Function Pointer and Menu-Driven System

One use of function pointers is in menu-driven systems. The user is prompted to select an option from a menu (e.g., 1~5). Each option is severed by a different function. Pointers to different functions is stored in an array of function pointers. The user's choice is used as a subscript of the array.

```
int fun1();
int fun2();
int fun3();
int fun4();
int fun5();
int main()
{
   char (*funPtr[3])(int) = {fun1, fun2, fun3};
   int choice = 0;

   while(choice != -1)
   {
      (*funPtr[choice])();
      chin >> choice;
   }
}
```

The limitation this application is that all the functions should have the same signature and return type.

5.12 5.12 *Power and Flexibility of Pointers*

Pointers in C++ is a very powerful tool. It is extremely flexible and therefore can generate every kind of errors if misused.

For example, you can only cast an object of a sub-class to its super-class. No other casting between user-defined classes is allowed. However, pointers to different classes can be cast to each other without any restrict. What is passed in the casting is only the address. So you can cast a pointer to an integer to a Employee-class pointer. Then the Employee pointer will just assume that starting from the passed address it can find all attributes of Employee class.

```
class Person {
public:
      Person(char * = 0, int = 0, bool = true, char = ' ');
      void print() const;
      void doNothing();
private:
      char * name;
      int age;
      bool genda;
      char blood;
};

Person::Person(char * n, int a, bool g, char b)
    : name(n), age(a), genda(g), blood(b) {}

void Person::print() const
```

```
{
        cout << name << " " << age << " " << genda << " "
          << blood << endl;
}

void Person::doNothing() {}


class Employee {
public:
        Employee(char * = 0, int = 0, bool = true, char * = 0);
        void display();
private:
        char * empName;
        int empAge;
        bool empGenda;
        char * empTitle;
};

Employee::Employee(char * n, int a, bool g, char * t)
    : empName(n), empAge(a), empGenda(g), empTitle(t) {}

void Employee::display()
{
        cout << empName << " " << empAge << " " <<
empGenda
          << " " << empTitle << endl;
}


int main(int argc, char* argv[])
{
        Derived d1(1234, 5678);
        Derived d2(d1);
        d2.print();
        cout << endl;

        Person p1("Frank", 34, true, 'O');
        Employee e1("Frank", 34, true, "Engineer");

        Person * p2 = (Person *)&e1;
        Employee * e2 = (Employee *)&p1;

        cout << "p1 = ";
        p1.print();
        cout << "p2 = (Person *)&e1 = ";
        p2->print();

        cout << "\ne1 = ";
        e1.display();
        cout << "e2 = (Employee *)&p1 = ";
        e2->display();

        return 0;
    }
```

Output will be:

```
    p1 = Frank 34 1
    p2 = (Person *)&e1 = Frank 34 1

    e1 = Frank 34 1 Engineer
    e2 = (Employee *)&p1 = Frank 34 1 -?@
```

6. 6. CLASS

6.1 6.1 *Class*

Up to this stage we've been mainly talking about issues which are common to both procedural (such as C) and OO languages. From now on we will talk more about OO issues.

As a class, it combined the data attributes and the behavior attributes of an object together, and formed an object which can simulate both aspects of a real-world object.

To distinguish independent functions such as those string handling functions in "**string.h**" and functions which belong to a class, class member functions are thereinafter called "methods".

The **"public:"** and **"private:"** labels are called **"member access specifiers"**. All data members and methods declared by the "**public**" specifier are accessible from outside, and all declared by "**private**" are only accessible for methods.

Actually an object contains only the data members. Methods do not belong to any specific object. The belong to the class. All objects share one copy of methods. When you use **"sizeof"** operator on a class or an object you will only get the total size of the data members.

6.2 6.2 *Methods of a Class*

If the method is defined inside the class body, it is automatically **inlined**. It may improve the performance, but it is not good for information hiding, because the client of the object is able to "see" the implementation of its methods. If a method is defined outside the class body, you have to use keyword "**inline**" if you want it to be inlined. Only the simplest methods can be defined inside the class body.

To define the method outside the class body, you have to use **scope resolution operator "::"**, which we have used before to access global variables, when a local variable with the same name had been declared. Use "class name::" in front of the method definition to make it unique, because other classes may have methods of the same name.

Methods which changes the data members are sometimes called "**commands**", and methods which do not change are called "**queries**". Separating the commands and queries leads to simpler, more flexible and easy-understanding interfaces.

♦ ☐ **Call a method**

To call a method, use the object name plus "**.**" plus the method name, or a pointer to that object plus "**->**" plus the method name.

6.3 6.3 Constructor

There is a special method with the same name as the class called **constructor**. There is no return type (even not void) for this special method.

Suppose "Test" is a class, the following line

```
Test t1(35, "Frank");
```

creates a Test object in compile time, assigning its address to "t1".

```
Test * ptr = new Test(35, "Frank");
```

"new" is a special method which creates a Test object dynamically at run time and returns a pointer to that new object. The returned pointer is received by "ptr". The following lines

```
int calculate(Test &);  // Function prototype

calculate( Test(35, "Frank") );
Test * ptr = &Test(35, "Frank");
```

create a Test object in compile time, but do not assign a name, instead point a pointer or reference to it.

Default arguments are recommended for constructors so that even if no arguments are passed to the object the data members can still be initialized to a consistent state. STL containers requires the objects to have default constructors.

Constructor can be overloaded. Normally three kinds of constructors are needed:

1. 1. Default constructor: no arguments;
2. 2. Constructor: has all arguments to construct an unique object;
3. 3. Copy constructor: has an argument its own type, to copy a new object out of it.

The default constructor and normal constructor can be merged into one if the normal constructor uses default arguments.

If no constructor is provided, the compiler automatically insert a default constructor. This default constructor does not perform any operation. So the data members of the object may not be consistent.

Built-in types are not initialized when created.

♦ ☐ User-defined Converters

Suppose a method has an argument of type "Child", which has an one-argument constructor "Child(Parent)". When you call this method, if you pass a "Parent" object instead of "Child", the compiler will implicitly call the one-argument constructor and convert the "Parent" object to "Child".

```
class Base {
public:
   Base(int a) : member(a)
   {
      cout << "Base constructor called with " << a << endl;
   }
private:
   int member;
};

void test(Base obj1)
{
   cout << "Base object's member = " << obj1.member;
}

int main()
{
   test(333);
}
```

The output will be:

```
Base constructor called with 333
Base object's member = 333
```

One-argument constructors are called user-defined converters.

6.4 6.4 Default Construcotr

Default constructor is called implicitly when you create an array of objects. If you want to have an array of objects that do not have a default constructor, the work-around is to have an array of pointers and initialize them using operator **new**.

6.5 6.5 *Copy Constructor*

A copy constructor is not only explicitly called by the programmer to create new objects by copying an existing object, it is also implicitly called by the compiler to make a copy of an object when it is passed by value. If copy constructor is not provided, compiler will provide a default copy constructor, which makes default memberwise copy, which can not deal with objects with pointer members.

There are two rules for the parameter of copy constructor:

1. 1. Copy constructor's argument can not be passed by value. Otherwise the copy constructor call results in infinite recursion, because for call-by-value, a copy of the object passed to the copy constructor must be made, which results in the copy constructor being called recursively.
2. 2. The object argument passed to the copy constructor must be constant. Otherwise it can not be applied on constant object.

6.6 6.6 *Accessing Class Members*

A class's data members and methods have class scope. Independent functions have file scope.

Data members and methods are directly accessible by other methods of the same class. Programs outside a class can only access a class's public members through one of the handles of an object: object name, reference to object, pointer to object.

So two kinds of variables may appear in a method: local variables with block scope which is destroyed after the call, and data members.

Public members of a class is designed to be an interface for its clients. It is recommended to keep all the data members under "private", and provide for clients public methods to set or get their values. This helps to hide implementation details from the clients, reducing bugs and improving program modifiability. It also simplifies the debugging process because problems with data manipulations are localized to either the class's methods or friends.

Private data members can also be changed by "**friends**" of its class. Because of this, the use of "friends" is deemed by some people to be a violation of information hiding.

Both structures and classes have private, public and protected access. Default of classes is private, default for structure is public.

6.7 6.7 Typical Methods

♦ ☐ Constructors

Discussed before.

♦ ☐ Access methods

To allow outside clients to modify private data, the class should provide “set” methods. To allow clients to read the values of private data, the class should provide some “get” methods. These methods are called “access methods”. They can also translate the data format used internally during implementation into the format for clients. For example, time may be most conveniently expressed in seconds (which is the return type of function “time(0)”), but clients may very possibly want the format of “06:30”.

♦ ☐ Service methods

These methods provide services for clients.

♦ ☐ Utility methods

They are only called by other methods, and normally are private.

♦ ☐ Destructors

Automatically called when an object leave scope to release all resources held by the object. The name of a destructor for a class is the tilde (~) character followed by the class name.

Stack memory resources held by the object such as its compiler-created members are released automatically when the object leaves scope or explicitly deleted. A destructor is only needed to release resources which can not be automatically released, such as dynamically allocated memory (using "new") or network or database connection.

6.8 6.8 Avoid Repeating Code

Always try to avoid repeating code if they must be kept identical. Although writing the same statements again can avoid a method call and thus good for performance, it is bad for maintenance, because once the program need to be changed both places should be changed meantime. Extra attention should be paid to always keep them identical. So always use a method call to avoid repeating code.

If you really want to avoid the method call, use “**inline**” qualifier in front of the method definition.

6.9 6.9 When Constructors and Destructors are Called

For global objects, constructors are called before any other methods including main begins execution. Destructors are called when main terminates or "exit" method is called.

For automatic local objects, constructors are called when execution reaches the point where the objects are declared. Destructors are called when the objects leave scope i.e. the block in which they are declared is exited.

For static local objects, constructors are called only first time when the execution reaches the point where the objects are declared. Destructors are called when main terminates or "exit" method is called.

It is the same in Java.

6.10 6.10 Default Memberwise Copy and Default Memberwise Assignment

When a copy of an object needs to be made, if no copy constructor is provided, a default memberwise copy will happen. For objects without dynamic members i.e. pointers, a default memberwise copy can do the job. But for objects containing pointers to other objects, a default memberwise copy will only point the pointers of the two objects to the same other object. This is called shallow copy.

When assignment operator "=" is used to assign one object to another, if no overloaded assignment operator is provided, a **default memberwise assignment** will happen. It is the same as default memberwise copy.

6.11 6.11 Pass-by-value and Copy Constructor

Both in C++ and Java, copy constructors are not designed for cloning objects explicitly, because copy constructor does not support polymorphism.

In C++, objects are by default passed by value, and when it happens, a copy of the argument object is automatically made by the compiler for the called method. Therefore, copy constructor is a must for any class which needs deep copy that default memberwise copy can not achieve. Copy constructor is therefore given big importance and becomes one of the four elements of the OCF: all classes should provide their properly implemented copy constructors.

In Java, because all objects are passed by reference and the language even does not support automatic pass-by-value, there is no need for enforcement of copy constructors. For the purpose of cloning objects, Java provides a more robust and automated facility – the clone mechanism.

6.12 6.12 Copy Constructor vs. Factory Method

A factory method is a method which uses dynamic memory allocation to clone itself and return a pointer to the new copy. Suppose you have a abstract class Shape and a series of derived concrete classes such as Circle, Square, Rectangle, etc. A factory method of Circle looks like

```
Shape * clone()
{
   return new Circle(*this);  // calling copy constructor
}
```

Copy constructor can not be used to clone objects in case of polymorphism. This is true in both C++ and Java, because copy constructor does not support polymorphism.

Suppose you have a method which receives a concrete-class object with a Shape handle and do something on it. Now if you want to make a copy of it in that method, with a factory method you can say

```
public void modifyAndDisplay(Shape * obj)
{
   Shape obj1 = obj. clone();
  ...
}
```

If the passed argument is a Circle, the Shape pointer "obj" will get a Circle, if it's Square, you will get a Square. But if you say

```
   Shape obj1 = new Shape(obj);
```

because copy constructor can only produce and return an object of its own type, you will only get a Shape object. You will lose all information of the derived-class part of data.

6.13 6.13 Various Places to use "const": Data Member, Method, Argument and Return type

♦ ☐ Constant data member

When a data member is declared "constant", it must be initialized meantime. It can not be modified, and only constant methods can access it. Non-constant methods can not access constant members, even if they do not modify the objects.

Declaring an object to be constant not only can prevent it from being modified, it is also good for performance: today's sophisticated optimizing compilers can perform certain optimizations on constants, but can not do it on variables.

♦ ☐ Constant argument of a method

Declaring an argument "**const**" will prevent the method to modify it. If you return this constant argument back, but did not declare the return type constant, the compiler will complain.

♦ ☐ Constant return type of a method

It is meaningless to declare the return type constant if it is return by value. Declaring the return by reference constant is to prevent the client from accessing the private data member through the reference. If a method returns one private data member by reference, the client who calls this method can modify reversibly this member.

For the same reason, if a constant method's return type is a reference to a data member, the return type should also be constant - otherwise the data member

♦ ☐ Constant method

A method is declared constant by putting "const" after its function header. A constant method can not modify any data member. It still can modify received arguments and local variables. Only class methods can be declared "constant", independent functions can not.

When a constant object is created out of a class, all its non-constant methods are forbidden to be called by the compiler. In the following example, compiler will prompt error on method call "t1.print( )":

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

```
        int get() const; // can be called for a constant object
        void print();  // can not be called for a constant object
private:
         int member;
};

TestConstant::TestConstant( int i)
{     member = I;  }

int TestConstant::get() const
{     return member;  }

void TestConstant::print() const
{     cout << "Hello the world!";  }

int main(int argc, char* argv[])
{
        const TestConstant t1(1234);
        cout << "The member is " << t1.get() << endl;
        cout << "The message is ";
        t1.print();
        cout << endl;
        return 0;
}
```

Therefore, always try to declare as many methods constant as you can, especially those modification-free methods, so that when a client creates a constant object, he can still call its modification-free methods.

Declaring modification-free methods constant comes with another benefit: if you inadvertently modify the object in this method, the compiler can always find it out for you. It can help to eliminate many bugs.

If the return type of a constant method is a reference to a data member, the return type must be also be constant, otherwise the client can modify the data member through the reference, which shouldn't happen because the method is constant.

However, there are cases when you hope that if the object is not constant, you want to modify the data member through the return type, while if the object is constant, you still want to read the data member through the return type. If you only provide a non-constant method with non-constant return type, it can not be called for a constant object, while if you only provide a constant method with constant return type, it can not modify the data member. To solve this problem, you can provide a pair of overloaded methods:

```
const int & get() const
{ return a; }

int & get()
{ return a; }
```

6.14 6.14 Member initializer

Assignment statements can not be used in a constructor to initialize constant data members. Member initializers must be used to initialize constant data members. A list of initializers start with a " **:** " after the constructor header, speparated by ",". Each initializer is the name of the data member followed by its value in brackets:

```
  Test::Test(int a, int b, int c): member1(a), member2(b),
member3(c)
  {...}
```

All data members CAN be initialized using member initializer syntax, but the following things MUST be initialized with member initializers:

1. 1. constant data members,
2. 2. references,
3. 3. base class portions of derived classes.

6.15 6.15 Member Objects

A data member of a user-defined type is called a member object. When a parent object is created, the member objects are created first, then they are used to construct the parent object. The order of the creation of member objects is decided by the order they are declared in the class definition, not the order of their member initializers.

Member objects do not have to be initialized explicitly. If member initializers are not provided, the member object’s default constructor will be called implicitly. Not providing a default constructor for the class of a member object when no member initializer is provided for that member object is a syntax error.

Member objects still keep their privacy from their owner. Owner class's methods can not directly access their private data members. They have to access them through their “get” or “set” methods.

A member objects can be automatic - sometimes called “value semantics”, or an reference to another object -sometimes called “reference semantics”.

Member objects are also called servers, and owners called clients.

6.16 6.16 Member Objects Should Be Initialized with Member Initializers

Compiler does not force you to initialize member objects with initializers, but you are strongly recommended to do so.

If a member object is initialized in the constructor with an assignment operator, its default constructor will be called first, then its assignment operator. If it is initialized with initializer, only its constructor will be called. It is not only the matter of saving one method call, but also the matter of safety. For a class without a properly implemented default constructor or assignment operator, using assignment operator to initialize it may cause unexpected logic errors such as shallow copy.

```
class Base {
public:
      Base();
      Base(const int i);
      const Base & operator =(const Base & b1);
private:
      int member;
};

Base::Base()
{   cout << "Base's default constructor!" << endl;  }

Base::Base(const int i): member(i)
{   cout << "Base's constructor!" << endl;  }

const Base & Base::operator =(const Base & b1)
{
      cout << "Base's assignment operator!" << endl;
      member = b1.member;
      return *this;
}
```

```
class User {
public:
	User(const Base & b1);
private:
	Base member;
};

User::User(const Base & b1)
{
	cout << "User's constructor!" << endl;
	member = b1;
}

int main(int argc, char* argv[])
{
	Base b1(1234);
	User u1(b1);
	return 0;
}
```

Output will be:

```
Base's constructor!
Base's default constructor!
User's constructor!
Base's assignment operator!
```

Now if you change the User's constructor to use initializer to initialize Base object:

```
User::User(const Base & b1) : member(b1)
{   cout << "User's constructor!" << endl;  }
```

Output will become:

```
Base's constructor!
User's constructor!
```

6.17 6.17 Friend

An independent function can be granted the privilledge to access a class's private members - if that class declares this function to be a friend of his. A function can not declare itself to be a friend of a class.

To be able to access a class, this independent function should usually receive an argument of that class, so that it can use the passed handle.

```
void showPrivacy(const NeedFriends & n1) const
{
  cout << "Object's private member is " << n1.member << 
endl;
}

class NeedFriends {
  friend void showPrivacy(const NeedFriends & n1) const;

public:
  NeedFriends(int i);

private:
  int member;
};
```

A method can be friends of different classes. Overloaded methods can be friends of a class.

6.18 6.18 this pointer

We already know that a class method is different from an independent function. When we call an independent function such as "test(int i)", we say

```
test1(1234);
```

But when we call a method test( ) of object o1, we have to call through this object's handle:

```
o1.test2(1234);
```

However, internally a method and an independent function are the same for the compiler. When the compiler sees a method call, it implicitly convert it to add one more argument - the object through which the method is called, so that the method knows which object to access. So the above method call is implicitly converted to something like

```
test2(&o1, 1234);
```

Inside the method, the passed handle is represented by pointer "**this**". You can use it to access the object.

For example, if class Test has three methods method1, method2 and method3, their return types are all Test, and they all end with

```
return *this;
```

Then you can write a line of code like

```
o1.method1().method2().method3();
```

This is called cascaded method call.

6.19 6.19 Memory Allocation and Manipulation

There are two ways to allocate memory for an object: statically at compile time and dynamically at run time.

♦ ☐ Static memory allocation

To allocate memory statically at compile time, the compiler must know for sure the size of the object. When you say

```
int a;
int b[100];
float b;
Employee c;
```

The compiler reads the type definition of the object (for object c it is the class definition of class Employee) and knows the size of the object.

But if you say

```
int size;
cin >> size;
float array[size];
```

Compiler will have no way to know how many bytes of memory to allocate for the array. Therefore it will complain.

♦ ☐ Dynamic memory allocation

To allocate memory at run time, there are two ways: C-style memory allocation using **malloc** and **free**, and C++ style allocation using **new** and **delete**.

To use C-style memory allocation for an int array of size 120:

```
int * pInt = (int *)malloc(120 * sizeof(int));
if(pInt == NULL)
{
   cout << "Memory allocation failed!\n";
   return;
}
memset(pInt, 0, 120);
```

There is a significant difference between C and C++ style dynamic memory allocation. **malloc** allocates exactly the amount of memory that you want, and it doesn't care what you are going to put into that block of memory. It is also not responsible for initializing the allocated memory. So usually there is a memset function call after malloc to initialize it explicitly.

In comparison, C++'s operatior new requires a type definition instead of the number of bytes you want to allocate. It reads the type definition and allocate exactly the amout of memory needed to hold the object of the given type, then it calls the constructor of that type to initialize the object.

Therefore, **malloc** is the most flexible way to allocate memory, for it does the least thing for you and leave you with all freedom. But it is also error-prone. It is much safer and simpler to allocate memory for an encapsulated C++ object.

Besides, C-style function memset may breach the encapsulation law. It can directly access private data members of an object.

♦ ☐ Memory de-allocation

To free the memory with C-style code:

```
free(pInt);
```

To free the memory in C++ code:

```
delete pFloat;
```

Again, C++'s delete is more convenient to use. It calls the destructor of the type before freeing the memory.

♦ ☐ Difference between static allocation and dynamic allocation

The overhead of dynamic memory allocation is that it takes computer time to obtain memory from the OS, and it may not always succeed. So for the sake of performance, if you can decide the size of the memory, always allocate memory at compile time using declarations.

Local objects created by declaration is discarded after leaving scope, but objects created by dynamic memory allocation is not destroyed after leaving scope. If not deleted it exists until the end of run.

6.20 6.20 Pointer Status after delete

After an object is freed using operator **delete** on its pointer, the object's memory space is freed, but the pointer itself still exists, because it is a local object. It is still pointing to the same memory location which has now been reclaimed by the OS. Therefore, if you delete it again the OS will shut down your program, because you are trying to delete something in the OS's territory.

However, if you delete a pointer with a value of 0, the **delete** operation doesn't do anything. Therefore, to prevent somebody or even yourself from accidentally deleting a pointer after it has already been deleted, assign 0 to a pointer after deleting it.

6.21 6.21 Memory Leak

If you keep asking from the OS dynamic memory but never remember to release them back to OS with **delete** after you no longer need them, finally the OS will tell your memory that no memory is available. It is called "memory leak", because your program is presently using very little memory but OS told you there is none left - it seems as if the memory resource has leaked away from a crack like water.

6.22 6.22 Who is Responsible to Delete an Object

When we delete a dynamicly created object, the program calls its destructor to delete cascaded dynamic objects pointed by data members of this object. Then the object itself including all its data members are destroyed and memory released to the OS.

Normally a class doesn't contain any code to delete itself - it is only responsible for deleting its own dynamically created members. It is the one who created an instance of this class on the heap who is responsible for deleting this object, not the object itself, because the object can only be deleted when it is created on the heap, and the code in the class implementation has no way to know whether each instance of itself is created on the heap or stack.

However, in some special cases when a class is designed to be created on the heap and it has to delete itself, you can put "**delete this;**" in the class to delete itself. It has the same effect as when a client deletes this object.

Consider the following example.

```
class Employee
```

```
{
public:
    void ChangeAge();
    void DisplayAge();
    void Delete();
    void DisplayName();
    Employee();
    virtual ~Employee();

protected:
    int m_nAge;
    char * m_strName;
};

Employee::Employee() :  m_nAge(34)
{
    m_strName = new char[30];
    strcpy(m_strName, "Frank Liu");
}

Employee::~Employee()
{  delete m_strName;  }

void Employee::DisplayName()
{  cout << m_strName << endl;  }

void Employee::DisplayAge()
{  cout << m_nAge << endl;  }

void Employee::ChangeAge()
{  m_nAge = 35;  }

void Employee::Delete()
{  delete this;  }
```

Suppose we have created two instances of Employee, one statically and one dynamically:

```
Employee e1;
Employee * e2 = new Employee;
```

If we say

```
delete e1;
```

the compiler will complain because “e1” is created statically.

However, we can cheat the compiler by saying

```
e1.Delete();
```

but there will be a run-time error, because inside Delete function we are still deleting a statically-created object.

If we say

```
delete e2;
```

or

```
e2->Delete();
```

They are doing the same thing and both allowed. Then if we say

```
e2->DisplayAge();
```

An undefined value will be displayed. If we say

```
e2->DisplayName();
```

Run-time error will happen.

This proves one thing: after an object is deleted from the heap, the memory space it used to occupy is retrieved by the OS, and you can not access it anymore.

6.23 6.23 Static Data Member

Normally each object has its own copy of all the data members. But sometimes all the objects share one data member. In this case, we declare this data member "static".

A class's static data members exists before any object is created. They must be initialized at file scope in the class source file, using class name and binary scope resolution operator **"::**" (see the following example). Both public and private members can be accessed this way. Even if you will use "set" method in other methods such as "main" to initialize the static data members, you still have to initialize them first in the class source file.

Static array is initialized like

```
int array[] = {1,2,3,...}
```

6.24 6.24 Static Method

A static method is defined by putting keyword "static" in front of the method prototype in the class definition, but DO NOT put "static" in front of the method definition in the class's source file.

A static method can not access any non-static data members. As said before in the discussion about "this" pointer, a normal method receives implicitly the handle of the object so that it knows which objec to access. But a static method is not attached to any object of the class and thus does not receive any object handle. So it has no way to access any object data member. It can only access static data members.

Static data members are also called "class data", and static methods are also called "class methods".

In file "Employee.h":

```
class Employee {
public:
   Employee(char *);
   const char *getName() const;
   static void setTotal(int);
   static int getTotal();
private:
   char * name;
   static int total;
};
```

In file "Employee.cpp":

```
#include "Employee.h"

int Employee::total = 0; // file scope initialization of static data
member

Employee::Employee(char *n)
{
   total ++;  // manipulation of the static data member in
constructor
   name = n;
}

void Employee::setTotal(int t)
{  total = t; }

const char * Employee::getName() const
{  return name; }

int Employee::getTotal()
```

```
   {  return total; }
```

Notice the use of keyword "int" in the initialization of the static data member "total". Because no object is created yet, this statement tells the compiler to allocate a memory space for "total" of the size of an integer.

```
   #include "employ. h"

   int main ()
   {
      Employee::set(33);
      Employee e1("John Smith");
      Employee e2("Frank Liu");
      cout << Employee::getTotal() << e1.getName()<<
  e2.getName() << endl;
      Employee.setTotal(77);
      cout << "There are " << e1.getTotal() <<" employees."
  <<endl;
   }
```

Notice the two ways to access static data member: through the class name with "::" and through the handle of an object. Through class name is logically clearer.

6.25 6.25 assert

The "**assert**" macro tests the value of a condition enclosed in "( )":

```
   assert (continuation condition);
```

If the condition is true, it continues to the next statement. If it is false, it will call method "**abort**", and print out an error message, including the line number, the condition and the file name, and terminate the program. It is a very useful debugging tool.

When you write a complex project, you can put "**assert**" statements after important operations to make sure that the result is right. It helps you to filter out bugs at a early stage before it causes complex confusions.

After the whole program is debugged, you needn't delete those "assert" statements. Just add one line at the beginning of the file:

```
   #define NDEBUG
```

This causes the preprocessor to ignore all assertions.

"assert" is defined in header file "**assert.h**". Method "**abort**" is defined in header file "**stdlib.h**".

6.26 6.26 Proxy/Wrapper Classes

As we discussed before, separating interface from implementation helps hiding the implementation details from the clients. However, the clients can still see the class's private data members. By providing clients with a proxy/wrapper class of the original class, the original class can be totally hidden from the clients.

For example, the original class is:

In "origin.h":

```
class Origin {
public:
   Origin(int);
   void set(int);
   const int get() const;
private:
   int value;
};
```

In "origin.cpp":

```
#include <iostream>
#include "origin. h"

Origin::Origin(int v)
{  value = v; }

void Origin::set(int v)
{  value = v; }

const int Origin::get() const
{  return value; }
```

The wrapper class wrapping around the original class:

In "proxy. h":

```
class Origin;  // forward class declaration

class Proxy {
```

```
public:
   Proxy(int);
   void set(int);
   const int get() const;
private:
   Origin * ptr;
};
```

In "proxy. cpp":

```
#include "Origin. h"
#include "Proxy. h"

Proxy::Proxy(int v) : ptr(new Origin(v))
{ }

void Proxy::set(int v)
{  ptr->set(v); }

const int Proxy::get() const
{  return ptr->get(); }
```

The reason the wrapper class wraps around a pointer instead of a member object is: if a class only has a pointer pointing to another class, the header file of the other class is not required to be included. You can simply declare that class as a data type with a forward class declaration. This is the key factor that makes it possible to hide the private data members of the original class from clients.

Notice that there is not proceeding preprocessor directives

```
#ifndef XXXX_H
#define XXXX_H
...
#endif
```

7. 7. OPERATOR OVERLOADING

7.1 7.1 Fundamentals of Operator Overloading

When operators such as +, -, *, /, = are used for different built-in types such as “int” and “float”, they actually have been overloaded by C++. They can also be overloaded for any user-defined type, to perform the same or similar operation.

For built-in type e.g. int, you can say “ c = a + b ” or “cout << a << b << c”. Now with operator overloading, you can declare a new type e.g. “Test a, b, c”, calculate them with “ c = a + b ”, output them with “cout << a << b << c”. Therefore, the user-defined type “Test” is fully equal to built-in type such as “int”. That’s why we say that C++ is an extensible language.

When overloading (), [], ->, or =, the operator overloading method must be a class member.

7.2 7.2 Overloading binary operators

A binary overloading member method taks one argument:

```
Test & operator+(Test &);
```

When the compiler sees a binary formula such as "a + b", it calls the "operator+" method of the left-hand operand "a", and passes the right-hand operand "b" as its argument. So "c = a + b" is implicitly converted to

```
c = a.operator+(b);
```

A global binary overloading function takes two arguments:

```
Test & operator<<(Test &, Test &);
```

When the compiler sees " a + b", it calls the independent "operator+" method, and passes the two operands "a" and "b" as its arguments:

```
operator<<(a, b);
```

7.3 7.3 Operator << and >> can not be Member Functions

The global binary operator overloading function is specifically useful when the left-hand operand does not belong to the class in question, such as

```
CMyOwnClass a, b;
cout << a;
cin >> b;
```

Because class cout and cin have not overloaded operator << and >> for type CMyOwnClass, you have to overload it yourself. However, you can not put this function in class CMyOwnClass, because a and b are not left-hand operand. So you have to use a global operator overloading function, such as stream-insertion and extraction operators:

```
ostream & operator<<(ostream &, Test &);
istream & operator>>(istream &, Test &);
```

As an independent function, for performance reason, it is better to be a **friend** of the class, so that it can directly access its private data members. Otherwise it has to access them through "get" and "set" method calls. If that's the case, you can make these non-methods **inlined** to reduce the overhead.

7.4 7.4 Overloading Unary Operators

Because the operand of a unary operator is always on the right, the unary operator method can either be a class member or an independent function. But it is preferable to make it a method. The use of friend here violates the encapsulation of a class.

♦ ☐ Method

Suppose "a" is an object of class "Test", and operator "!" is overloaded by a method, then this method has no argument:

```
bool operator!() const;
```

and "!a " is equal to "a.operator!() ".

♦ ☐ Independent function

Suppose "!" is overloaded by a **friend** method, then the method has one argument:

```
bool operator!(const Array &);
```

and "!a " is equal to " operator!(a) ".

7.5 7.5 Operator Cascading

The return type of all operator overloading methods, both methods and independent functions, can be void. The disadvantage of having void return type is, you can’t use cascading operators, such as ‶ cout << c << endl″ or ‶ c = a + b ″, because ‶ cout << c ″ as a method call returns nothing, and apparently ‶ void << endl ″ has no meaning. So is ‶ c = void; ″.

To enable such cascading, normally operator overloading methods has its return type defined as class type, and returns the result object. Return-by reference is widely used here, because it can save the copying process, which is a big overhead for objects of large size.

7.6 7.6 Subscription Operator [ ]

The subscript operator [ ] is not restricted for use only with arrays; it can be used to select elements from any kinds of container classes such as linked lists, strings, dictionaries, and so on. Also, subscripts no longer have to be integers; characters or strings could be used, for example.

7.7 7.7 "operator =" and Default Memberwise Copy

If you do not explicitly overload assignment operator, the compiler will use default memberwise copy to perform object assignment. Normally it will do the job, but for objects with pointers, it will create a new pointer pointing to the same memory location instead of new memory location.

7.8 7.8 Orthodox Canonical Form (OCF)

A constructor, a destructor, an overloaded assignment operator, and a copy constructor are recommended for all classes, even if it does not have dynamic members at this stage. Programs containing these four components are said to be in Orthodox Canonical Form.

7.9 7.9 Check for Self-assignment

The check for self-assignment is only necessary for a assignment operator when **dynamic memory allocation** is involved, in which some deletion job is done. In such a case if we don’t check for self-assignment, the object’s memory space will be deleted first and all the info stored in the dynamic memory will be lost.

7.10 7.10 An Example about Pass-by-reference

```
class Test {
public:
   int & bridge( int & );
};

int & Test::bridge (int & input)
{ return input; }

int main()
{
   Test a;
   int b=333;
   int c = a.bridge(b);
   cout << "c = " << c << endl; // should be 333
   a.bridge(b) = 444;
   cout << "b= " << b << endl; // used to be 333, now should
 be 444
}
```

Because a reference is the address of the original object, this address can be passed back and forth to anywhere, and all parties who have a copy of this address can directly manipulate the original object.

Method "bridge" receives one reference from calling method, and return this reference back to the calling method. Any modification on the returned value will affect the passed argument.

7.11 7.11 lvalue and rvalue

"lvalue" means a variable that is modifiable, such as the left variable in an assignment. "rvalue" means a variable which doesn't need to be modified, such as the variable on the right of an assignment.

7.12 7.12 Overloading ++ and --

- ☐ **Preincrementing**

The prefix versions of ++ and -- (such as ++a and --a) are overloaded exactly as any other prefix unary operator:

```
Test & operator++();
```

Suppose there is an object "a" of class "Test", when the compiler sees the preincrementing expression "++a", it generates the method call

```
a.operator++();
```

When the preincrementing is implemented as an independent function:

```
    Test & operator++(Test &);
```

When the compiler sees the expression "++a", it generates the method call

```
    operator++(a);
```

- ☐ **Postincrementing**

To make the overloaded method of postincrementing operator distinguishable from preincrementing one, the operator method should be:

```
    Test & operator++(int);
```

When the compiler sees the expression "a++", it generates a method call

```
    a.operator++(0);
```

"0" is a "dummy value" to make the parameter list of the overloaded operator method different from preincrementing one.

If the postincrementing is implemented as an independent function:

```
    Test operator++(Test &, int);
```

when the compiler sees the expression "a++", it generates method call

```
    operator(a, 0);
```

The return type for postincrementing can not be a reference, because the returned object should be the object BEFORE increment, not after increment. So we have to create a temporary local object to hold the value of the original object, then increment the original object, and return the temporary one.

7.13 7.13 Example: Date Class

```
    #include <iostream>
    #include <conio.h>

    class Date {
       friend ostream & operator<<(ostream &, const Date &);
    public:
```

```
    Date(int=1, int=1, int=1900);
    void set(int, int, int);
    const Date & operator++();
    const Date operator++(int);
    const Date & operator +=(int);
  private:
    int day;
    int month;
    int year;
    static const int days[13];
    bool leap();
    const int checkDay();
    void increment();
  };

  const int Date::days[] =
{0,31,28,31,30,31,30,31,31,30,31,30,31};

  Date::Date(int d, int m, int y)
  {  set(d, m, y); }

  void Date::set(int d, int m, int y)
  {
    day = d;
    month = m;
    year = y;

    if(checkDay1()<0 || month<=0 || month>12)
    {
      cout << "Wrong Date format!  Date is set to be
1/1/1900.\n";
      day = 1;
      month = 1;
      year = 1900;
    }
  }

  const Date & Date::operator ++()
  {
    increment();
    return *this;
  }

  const Date Date::operator ++(int)
```

```
{
   Date temp = *this;              // default memberwise copy
   increment();
   return temp;
}

const Date & Date::operator +=(int dd1)
{
   for(int i=1; i<=dd; i++)
      increment();
   return *this;
}

bool Date::leap()
{
   if (year % 400 == 0 || (year % 100 != 0 && year % 4 ==
0))
      return true;
   else
      return false;
}

const int Date::checkDay()
{
   if(month == 2 && leap())
      return 29 - day;
   else
      return days[month] - day;
}

void Date::increment()
{
   if( checkDay1()>0 )
      day ++;
   else
   {
      day = 1;
      if (month < 12)
         month ++;
      else
      {
         month = 1;
         year ++;
      }
   }
 }

 ostream & operator<<(ostream & output, const Date & obj)
 {
   output << "Date: " << obj.day << "/" << obj.month << "/"
<< obj.year << endl;
   return output;
 }

 int main()
 {
   Date d1(31,12,1999), d2(31,1,1999), d3(23,2,2000),
   d4(23,2,4444), d5(23,2,1999);
   d4.set(23,2,1212);
   ++d1;
   cout << d1;
   d2++;
   cout << d2;
   d3 += 6;
   cout << d3;
   d4 += 7;
   cout << d4;
   d5 += 6;
   cout << d5 << "Press any key when ready...";
   getch();
 }
```

8. 8. INHERITANCE

8.1 8.1 *Method Overriding*

When a method of the derived class has the same signature as that of the base class, it is said that the method of the derived class overrides the method of the base class. The base-class version of method is only overridden from external point of view – they are still accessible from the derived class internally. The derived-class method often needs to call its overridden base-class method to perform part of the job related to the base-class data members.

In this case, if you forget to use the scope resolution operator in front of the invoked overridden base-class method, it will actually call itself and thus create a infinite recursion until the memory is run out.

One interesting point: normally all overloaded methods can be accessed through different signatures, but if you overload a base-class method in the derived class – with a different signature, the base-class method is actually overridden instead of overloaded:

```
#include <iostream>

class Base {
public:
   int print(int i)
   {
      cout << "Base-class version, int a = " << i << "\n";
      return i;
   }
};

class Derived : public Base {
public:
   char print(char c)
   {
      cout << "Derived-class version, char c = " << c << "\n";
      return c;
   }
};

int main()
   {
      Derived d1;
      char c = 'a';
      int i = 1234;
      d1.print(i);
      d1.print(c);
      cin >> i;
   }
```

Output will be:

```
Derived-class version, char c = π
Derived-class version, char c = a
```

Only the derived-class method will be called. This rule looks a bit wierd. In Java, a base-class method can be overloaded in the derived-class and both methods can be accessed by clients (if they are both public).

8.2 8.2 Initialization of the Base-class Part of the Object

To initialize the base-class part of data members of the derived class, member initializer must be used.

If a base-class constructor is not explicitly invoked, the compiler will implicitly call the base-class default constructor. If no base-class default constructor is provided, the compiler will issue a syntax error.

```
class Derived : public Base {
public:
      Derived(const int = 0, const int = 0);
      Derived(const Derived &);
      const Derived & operator=(const Derived &);
private:
      int member;
};

Derived::Derived(const int i1, const int i2) : Base(i1),
member(i2)
{}

Derived::Derived(const Derived & d) : Base(d),
member(d.member)
{}

const Derived & Derived::operator=(const Derived & rv)
{
      member = rv.member;
      Base::operator=(rv);
      return *this;
}
```

When a derived-class object is created, the base-class constructor will be called first, then the derived-class constructor. When it is to be deleted, the derived-class destructor is called first, then the base-class destructor.

In a multi-level inheritance, the constructor of a certain level is only responsible to call the constructor of the next-level class.

8.3 8.3 Conversion between base class and derived class

Objects of a derived class may be used as an object of the base class. The compiler will make an implicit conversion. But objects of the base class can not be used as objects of derived class. The derived class is more specific and thus contains more info. To cast a less specific class to a more specific class, the extra info needed to construct the later one is missing. It may cause serious run-time errors.

Suppose:

1. 1. "Base" is a base class, "b1" is a base-class object, "basePtr1" is a base-class pointer;
2. 2. "Derived" is a derived class, "d1" is a derived-class object, "derivedPtr1" is a derived-class pointer
3. 3. Both the base and the derived class has a method "print( )"

Using object name:

```
b1.print();      // base-class version "print()" called
d1.print();      // derived-class version "print()" called
```

Using Pointer:

```
basePtr1 = &b1;
derivedPtr1 = &d1;
basePtr1->print();      // Base-class version "print()" called
derivedPtr1->print();   // Derived-class version "print()" called
derivedPtr1 = &b1;       // Try to convert b1 to Derived-class
object. Illegal!
basePtr1 = &d1;          // Implicitly convert d1 to Base-class
object.
```

```
basePtr1->print();      // Still the base-class version "print()"
called
```

However, when you use

```
derivedPtr1 = static_cast<Derived *>(&b1);
```

you are telling the compiler that you know the danger and you deliberately want to take that risk. So compiler will allow that operation. But the derived-class part of data will remain undefined.

8.4 8.4 *"is-a", "has-a", "Use-A" and "Know-A" Relationship*

"is a" relationship is inheritance. In an "is-a" relationship, an object of a derived class can also be treated as an object of the base class, just like a 4WD can be treated as a vehicle.

"has a" relationship is composition. In a "has-a" relationship, an object has one or more objects of other classes as members. Just like a 4WD has an engine.

A person is not a car and do not contain a car, but he uses a car. A method uses an object simply by issuing a method call to a method of that object.

An object can know another object by containing a pointer to it. This is called "know a" relationship. Sometimes it is called "association".

8.5 8.5 *Public, Protected and Private Inheritance*

Public inheritance inherits base class's public and protected members as its own public and protected members, and base class's private members are hidden.

Therefore, if you do not want derived classes to access a member, you should declare it **private**. If you do not allow clients but would allow derived classes to access it, you should declare it **protected**.

Protected data breaks **encapsulation** – a change to protected members of a base class may require modifications of all derived classes. Therefore, always try to declare data members **private**, and use **protected** as a final resort.

♦ ☐ **Public inheritance**

From base-class to derived-class:

public => public

protected => protected

private => hidden

- ☐ **Protected inheritance**

public => protected

protected => protected

private => hidden

- ☐ **Private inheritance**

public => private

protected => private

private => hidden

8.6 8.6 Shrinking Inheritance

Through **private** inheritance, all the members of the base class are made hidden or private in the derived class, thus inaccessible to the clients. This is useful for shrinking inheritance: you only want to inherit part of the methods from a class. You override the base-class methods with a simple call it, and for those unselected methods, they became private members and suppressed to the clients.

For example, suppose "Base" is a base class, and the derived class will be like:

```
template <class Type>

class Derived : private Base {
public:
   Derived(const Type & x) : Base(x) {}
   void setX1(const Type & x)  {  Base::setX1(x); }
   void setY1(const Type & x)  {  Base::setY1(x); }
   const Type & getX1() const  {  return Base::getX1(); }
   const Type & getY1() const  (  return Base::getY1(); )
};
```

8.7 8.7 Methods That Are Not Inherited

The four Orthodox Canonical Form methods – constructors, copy constructors, assignment operators and destructors, are not inherited in C++.

8.8 8.8 Software Engineering with Inheritance

A derived class does not need to access the source code of the base class, but only need the base class's object code. Therefore, independent software vendors (ISV) can develop their own class libraries and provide clients with only object codes.

Modifications to a base class do not require derived classes to change, as long as the public and protected interfaces of the base class remain unchanged. Derived classes may, however, need to be recompiled.

Although in theory users do not have to know the source code of the inherited class, in practice lots of programmers still seem reluctant to use something that they don't know. On the other hand, when performance is a major concern, programmers may want to see source code of classes they are inheriting from, so that they can tune the code to meet their performance requirements.

A base class specifies commonality -- all classes derived from a base class inherit the capabilities and interfaces of that base class. In the object oriented design process, the designer looks for commonality and "factors it out" to form a base class. Derived classes are then customized upon the base class.

In a object oriented system, classes are often closely related. So the best way is to "factor out" common attributes and behaviors and place them in a base class, then use inheritance to form derived classes.

8.9 8.9 Partial Assignment

Partial assignment is: when using base-class pointer or reference to make assignment on derived-class objects, only the base-class part of the data is assigned. The reason for this is: because assignment operator can not be virtual, when using base-class reference to assign derived-class objects, only the base-class assignment operator is called.

```
class Base {
public:
   Base(int a) : _a(a) {}

   const Base & operator=(const Base & obj)
   {
      cout << "Base-class assignment operator called! \n";
      _a = obj. _a;
```

```
        return *this;
    }

    const int get() const { return _a; }

    virtual void print() const = 0;

private:
    int _a;
};

class Derived : public Base {
public:
    Derived(int a, int b) : Base(a), _b(b) {}

    const Derived & operator=(const Derived & obj)
    {
        cout << "Derived-class assignment operator called! \n";
        _b = obj. _b;
        Base::operator=(obj);
        return *this;
    }

    virtual void print() const
    { cout << "_a = " << Base::get() << ", _b = " << _b <<
endl; }

private:
    int _b;
};

int main()
{
    Base * ptr1 = new Derived(11, 111);
    Base * ptr2 = new Derived(22, 222);
    (*ptr1) = (*ptr2);
    cout << "Derived 1: ";
    ptr1->print();
    cout << "Derived 2: ";
    ptr2->print();
}
```

output will be:

```
Base-class assignment operator called!
```

```
Derived 1: _a = 22, _b = 111
Derived 2: _a = 22, _b = 222
```

There is no way to make assignment without partial assignment on derived-class objects with base-class reference or pointer. To forbit doing this, declare the base-class assignment operator **protected**, so that it can not be invoked. Then to conform to OCF this class should be abstract – because an OCF must have a public assignment operator.

Even if the base class is an ABC, if assignment operator is not protected, partial assignment will still happen.

So generally speaking, to avoid partial assignment, always inherit from ABC and make its assignment operator protected.

8.10 8.10 Sequence of Constructor Call in Inheritance

Suppose a class inherits from a base class and contains a member object, when an object is created, the base-class constructor will be called first, then the composition class, then the derived class itself. For example:

```
class Base1 {
public:
  Base1() {  cout << "Base1 Constructor !\n"; }
};

class Base2 {
public:
  Base2() {  cout << "Base2 Constructor! \n"; }
};

class Derived : public Base1 {
private:
  Base2 b1;
  int d;
public:
  Derived(int d1) : d(d1) {  cout << "Derived Constructor! \n";
}
};

int main()
{  Derived d1(33); }
```

Output will be:

```
Base1 Constructor!
Base2 Constructor!
Derived Constructor!
```

8.11 8.11 Default Constructor in Inheritance

When a base class have constructors but does not have a default constructor, and in the derived class there is no explicit call to base-class constructor, compiler will prompt error. But if the base class has no constructors at all, compiler will generate one implicitly. This is the same in Java.

9. 9. POLYMORPHISM

9.1 9.1 *Virtual Methods*

When a method is declared **virtual**, it remains **virtual** all the way down the inheritance hierarchy even if the overridden method in the derived class is not declared virtual. However, it is a good practice to explicitly declare all the overriding methods down in the hierarchy to be **virtual** to promote program clarity.

If a derived class doesn't provide an overriding method, it will simply inherits its base class's virtual method.

9.2 9.2 *Polymorphism*

When different classes inherits from the same base class, we can use a base-class pointer to point to any derived-class objects, and access their virtual methods. Objects of different classes related by inheritance from the same base class can response differently to the same method call.

9.3 9.3 *Dynamic and Static Binding*

♦ ☐ Dynamic Binding

When a pointer is used to refer to a derived-class **virtual** method:

```
basePtr1 = &DeriveObj1;
basePtr1->print();
```

the correct method is chosen at run time dynamically. This is called "dynamic binding".

Because the correct method is not chosen at compile time, a program can be written to receive and make use of an object of a class which has not be developed yet. It can simply put a base-class pointer in the parameter list, and perform all the operations through this pointer.

♦ ☐ Static Binding

If the object name is used to call its method:

```
DerivedObj1.print();
```

the correct method is chosen at compile time. This is called "static binding".

9.4 9.4 *Abstract Base Class (ABC)*

We declare the base-class virtual method "**pure**" by putting "**= 0**" at the end of the method prototype. We needn't but we are allowed to provide method definition for **pure** methods.

A base class with a pure method is called **abstract base class (ABC)**. It is designed purely to be inherited, and is not allowed to have any instance. The only way to make a class abstract is to have a pure method. In Java, you can simply put "abstract" in front of a class header to make it abstract.

If a class is derived from an abstract class and does not provide an overriding method for the pure virtual method, it will inherit the pure virtual method and thus become an ABC too.

A hierarchy does not need to contain any abstract classes, but many good OO systems have class hierarchies headed by one or even several levels of abstract classes. To prevent partial assignment, it is a good practice to always inherit from ABC.

Pure virtual methods do not need to have any implementation, but it can be implemented – although it is very misleading. When all the methods of the base class need to do something – it is always better to put as much as common behaviors of different derived classes into the base class – but still we want this class to be ABC, we make the destructor **pure** virtual.

9.5 9.5 *Virtual Destructor*

Although a constructor can not be virtual, a destructor can be, so that you can delete a derived-class object though a base-class pointer. Otherwise when deleting this object only the base-class destructor will be called.

9.6 9.6 *Hierarchy for Interface and Implementation*

Hierarchies designed for implementation tend to have their methodality high in the hierarchy. Derived classes inherit these implementations.

Hierarchies designed for interface tend to have their methodality lower in the hierarchy. The base class only provide an interface (e.g. with pure virtual methods), and the derived class provide their own implementations. Like Java's interfaces.

When a base class is purely designed for providing an interface, it may only contain pure virtual methods and no data members.

9.7 9.7 Base Class Idiom

Virtual destructor and protected assignment operator is called the base class idiom.

9.8 9.8 Apply Polymorphism on Operator << and >>

As discussed before, stream insertion and extraction operators (>> and <<) can not be member methods. But we can still make them applicable to polymorphism:

```
ostream & operator<<(ostream & os, const Base & obj)
{  return obj.output(os);  }
```

Then we write a virtual output method for all derived classes:

```
virtual ostream & output(ostream & os)
```

10. 10. STREAM IO

10.1 10.1 *Iostream Library Header Files*

1. 1. "iostream.h" contains basic information required for all stream-IO operations.
2. 2. "iomanip.h" contains information useful for performing formatted IO with parameterized stream manipulators.
3. 3. "fstream.h" contains information for user-controlled file processing operations.

10.2 10.2 *Stream IO Classes and Objects*

"iostream.h" contains many classes for handling a wide variety of IO operations. The "**istream**" class and "**ostream**" class are both derived from "**ios**" class. The "**iostream**" class is derived through multiple inheritance from both "**istream**" and "**ostream**" class.

"**cin**" is an object of "**istream**" class and "**cout**" is an object of "**ostream**" class. They are tied to standard input and output device such as keyboard and screen. Left-shift operator "<<" is overloaded in the class as a stream-insertion operator to perform stream output, and right-shift operator ">>" is overloaded as a stream-extraction operator to perform stream input. These overloaded operators made it possible to perform IO with simple statement like

```
cin >> a;
cout << b;
```

"**cerr**" and "**clog**" are objects of the "**ostream**" class and tied to the standard error device. Outputs to "**cerr**" is not buffered and outputs to "**clog**" is buffered.

10.3 10.3 *Output the address of a pointer*

When we output a pointer of a type other than char, the address of the object to which the pointer is pointing to is output. However, if we output a char pointer, the string will be output. To output the address of the string, we have to cast the char * to void *:

```
int main(int argc, char* argv[])
{
      Derived * derivedPtr = new Derived(111, 222);
      cout << "derivedPtr = " << derivedPtr << endl;
```

```
        char * charPtr = "Frank Liu";
        cout << "charPtr = " << charPtr << endl;
        cout << "(void *)charPtr = " << (void *)charPtr <<
"\n\n";

        return 0;
}
```

Output will be:

```
derivedPtr = 0x00301A60
charPtr = Frank Liu
(void *)charPtr = 0x004270E4
```

10.4 10.4 Method put

The **put** method outputs one character:

```
cout.put(‘A’);
```

put can be cascaded as

```
cout.put(‘A’).put(‘B’);
```

10.5 10.5 Stream Input

♦ ☐ Stream Extraction Operator

A stream extraction operator reads from the input stream until a whitespace character such as a blank, tab and newline is encountered. The operator returns a reference to the object through which it is invoked (e.g. cin). But if the EOF is encountered, it will return zero. Therefore you can use a "while" loop to input a series of values:

```
while(cin >> a)
{...}
```

♦ ☐ get( ) and getline( )

Method **get** with no arguments inputs one character from the designated stream (even if it is whitespace) and returns it. It returns EOF if EOF is encountered.

Method **get** with a character argument inputs one character from the input stream (even if it is whitespace) and assign it to the character argument. It returns a reference to the object through which it is invoked (e.g. cin), and returns 0 when EOF is encountered.

Therefore, you can use a while loop to input a series of characters:

```
while(c = cin.get() != EOF)
{...}

while(cin.get(c))
{...}
```

The above two formats do the same job.

The third version of method **get** takes three arguments: a character array, a size limit, and a delimiter with default value '\n':

```
const int size = 35;
char array[size];
cin.get(array, size)       // use default value '\n' as delimiter
cin.get(array, size, '%')  // use '%' as delimiter
```

This version reads characters from the input stream and load them into the character array, until (size - 1) characters had been read, or before that the delimiter is encountered. Finally a NULL character is inserted to the end of the inputted character string in the array.

When the delimiter is encountered, the method does not load it into the character array, and it remains in the input stream. Therefore, the next input method will get this delimitor such as a new line.

The **getline** method is the same as the third version of **get**, except that it reads in the delimiter character from the input stream and discard it.

♦ ☐ **ignore( )**

The **ignore** method reads in and discards a number of characters (default is one) from the input stream, or until encounters a designated delimiter (default is EOF, which causes skipping to the end of the whole file).

♦ ☐ **putback( )**

The **putback** method places the last character obtained by **get** method back to the input stream. It is useful when you check characters one by one with **get** method looking for a field beginning with a specific character. When you find this character you put it back to the input stream, so that other input statements can input it correctly.

- ☐ **peek( )**

It is the combinition of **get** and **putback**. It returns next character in the input stream, but doesn't remove that character from the stream.

10.6 10.6 Unformatted IO

Unformatted IO inputs or outputs a certain number of bytes without any format.

- ☐ **write( )**

It has two arguments: a character array and the number of bytes to be outputted:

```
char * array = "ABCDEFGHIJKLMN";
cout.write(array, 5);     // "ABCDE" will be outputted.
```

- ☐ **read( )**

Its arguments are the same as **write**. If not enough bytes are read, **failbit** will be set.

- ☐ **gcount( )**

It returns the number of bytes read by last input operation.

```
const int size = 80;
char buffer[size];
cin.read(buffer, 20);
cout.write(buffer, cin.gcount());
```

10.7 10.7 Stream manipulators

When a stream manipulator is inserted, the format of the following output are all determined by it, until a different manipulator is inserted again.

- ☐ **Parameterized Stream Manipulator**

Stream Manipulators which take arguments are called parameterized stream manipulators, such as **setprecision**, **setbase**, etc. Their header file is **<iomanip.h>**.

- ☐ **Integral Stream Base**

Integers are by default interpreted as decimal values. To change the base, insert the manipulator "**hex**" for hexadecimal, "**oct**" for octal, and "**dec**" to change back to decimal. Or you can use parameterized stream manipulator **setbase**, whose arguments may be 8, 10 or 16.

```
int a = 11;
cout << a << endl          // "11" will be outputted
     << oct << a << endl   // "14" will be outputted
     << hex << a << endl;  // "B"  will be outputted
```

♦ ☐ Floating-Point Output Precision

The **precision** method or **setprecision** parameterized stream manipulator control the number of digits to the right of the decimal point. The **precision** method with no arguments returns the current precision setting:

```
int a = 3.14159265;
cout << setprecision(2) << a;     // "3.14" will be outputted
cout.precision(4);
cout << a                         // "3.1416" will be outputted
cout << precision() << endl;      // "4" will be outputted
```

It will affect all the following IO operations until specified again.

♦ ☐ Field Width

Method **width** and parameterized stream manipulator **setw** sets the IO field width. Method **width** returns the previous width. If the actual width is smaller than the set width, fill characters are inserted as padding. If it is wider then the set width, the full number will be printed. It only affect one succeeding data.

When inputting characters with the width is set to n, only n-1 characters will be inputted, and the last character will be set to NULL.

Example using **width** method:

```
int w = 2;
char c[10];
cin.width(3);

while(cin >> c)
{  cout.width(w++);
   cout << c << endl;
   cin.width(3);
}
```

When you input “abcdefghijklnmopqrstuvwxyz”, the output will be

```
ab
 cd
```

```
ef
 gh
  ij
   kl
    nm
     op
      qr
       st
        uv
         wx
          yz
```

Example using **setw** stream manipulator:

```
int w = 4;
char c[10];

while(cin >> setw(5) >> c)
   cout << setw(w++) << c << endl;
```

♦ ☐ **Format State Flags**

1. 1. **ios::skipws** Skip whitespace characters on an input stream

2. 2. **ios::left** Left justify

3. 3. **ios::right** Right justified

4. 4. **ios::internal** Number’s sign (+, -) left justified, magnitude right justified

5. 5. **ios::dec** Integer treated as decimal

6. 6. **ios::oct** Integer treated as octal

7. 7. **ios::hex** Integer treated as hexadecimal

8. 8. **ios::showbase** Put 0 in front of octal numbers, 0x or 0X before hexadecimal, to indicate the base

9. 9. **ios::uppercase** Use 0X instead of 0x for hexadecimals, and E instead of e for scientific notion

10. 10. **ios::showpoint** Float numbers should be outputted with a decimal point. Normally used with **ios::fixed** to guarantee a certain number of digits to the right of the decimal point. For floating-point-format output.

11. 11. **ios::fixed** Float numbers should be outputted with a specific number of digits to the right of the decimal point. Specially for floating-point format.

12. 12. **ios::showpos** the positive sign should be shown

13. 13. **ios::scientific** Outputs a number in scientific notion

The static data member **ios::adjustfield** flag includes the bits **left**, **right**, and **internal**. The **ios::basefield** includes the **oct**, **hex** and **dec** bits. The **ios::floatfield** contains the flags **scientific** and **fixed**.

All of these format state flags are defined as enums in class **ios**. They represent different bits of one long format state number, which is the settings of the IO stream.

The **flags**, **setf** and **unsetf** methods and **setiosflags** and **resetiosflags** parameterized stream manipulators set and reset these flags.

The **flags** method sets a number of flags and returns the previous settings. In its parameter list, you can "or" different flags with "|". Any flags not specified in the list is reset. The **unsetf** and the two manipulators works similarly.

The **setf** method sets one flag and the rest remains the same. When two flags separated with "," are listed:

```
cout.setf( ios::left, ios::adjustfield );
```

the first is set, the second is reset.

All of this format state flags affect all the following IO operations until specified again.

```
float a = 7.6;
float b = 333.14159;
cout << setw(20) << setprecision(3)
     << setiosflags(ios::internal | ios::showpos |
ios::scientific) << a << endl;
cout << setw(20) << b << endl;
```

Output will be:

```
+          7.600e+00
+          3.331e+02
```

♦ ☐ Padding (fill, setfill)

The **fill** method and **setfill** stream manipulator set the padding character. Default is space character.

```
float a = 3.3;
cout << setw(8)  << setfill('#') << a;
```

will have an output

```
#####3.3
```

10.8 10.8 Stream Error States

The state of a stream may be tested through bits in **ios** class.

- ☐ **eofbit and eof( ) Method**

The eofbit is automatically set for an **input** stream when EOF is encountered. When EOF is encountered on **cin**, the call **cin.eof** returns **true**. Here "encounter" means that there is no more bytes to read from the input stream. It does not mean that the reading method has already hitted the wall.

- ☐ **failbit and fail( ) Method**

The failbit is set and **fail** method returns true when recoverable format error occurs on the IO stream. When EOF is encountered during input, failbit is set for **cin**.

- ☐ **badbit and bad( ) Method**

The badbit is set and **bad** method returns true when irrecoverable lost-data error occurs on the IO stream.

- ☐ **goodbit and good( ) Method**

The goodbit is set and **good** method returns true when neither of the above errors occurs.

- ☐ **clear( ) Method**

The **clear** method is normally used to restore a stream's state to "good" so that IO may proceed on that stream. Any error state bit listed above which becomes the argument is set. The default argument of **clear** method is **ios::goodbit**, so the statement "**cin.clear**" clears cin and sets goodbit for the statement, and "**cin.clear(ios::failbit)**" actually sets the **failbit**.

11. 11. FILE PROCESSING

11.1 11.1 *Data Hierarchy*

Field a group of characters conveying meaning

Record composed of several fields (in C++ called members), such as structure or class

File a group of related records

Database a group of related files

A collection of programs designed to manage database is called a **Database Management System (DBMS)**.

11.2 11.2 *Primary Key*

To facilitate the retrieval of specific records from a file, at least one field in each record should be chosen as a primary key. A primary key represents something unique of each entry, such as account number of bank customers, so that all entries can be uniquely identified by their record keys.

11.3 11.3 *Files and Streams*

The source of input may be keyboard, files on hard disk or other devices. The output destination may be screen, files on hard disk or other devices.

For standard IO devices – keyboard and screen, C++ provides **iostream** objects **cin, cout, cerr, clog**. For other devices, you have to create objects yourself.

An **ifstream** object can be used to input from a file on disk, an **ofstream object** can be used to output to a file on disk. A **fstream** object can be used both to input from and output to a file. You can use their methods **open** and **close** to open and close a certain file. For example, you can use

```
ofstream file1("Frank.txt", ios::out);
```

or you can use

```
ofstream file1;
file1.open("Frank.txt", ios::out);
```

You can also create a fstream object, which can be used for both input and output:

```
fstream file("Frank.txt", iso::out | ios::in);
```

The file connected to an object of **ifstream**, **ofstream** or **fstream** class will be automatically closed when the object leave scope and is destroyed. However, it is a good practice to explicitly close the file as soon as you do not use it any longer. Reasons are:

1. 1. Reduce resource usage;
2. 2. Improve the program's clarity;
3. 3. Prevent future misuse.

11.4 11.4 File Open Modes

When opening a file, you can pass together with file name the following open modes:

1. 1. **ios::app:** Append output to the end of the file
2. 2. **ios::ate:** Open a file and move the **file position pointer** to the end of the file. Normally used for starting at appending data, but by moving the file position pointer, data can be written anywhere in the file
3. 3. **ios::in:** Open a file for input
4. 4. **ios::out:** Open a file for output -- existing data in the file will be lost. If no such file, will create one
5. 5. **ios::trunc:** Discard the file's existing contents (same as ios::out)
6. 6. **ios::nocreate**: Do not create new files -- if no such file then open operation fails
7. 7. **ios::noreplace**: Do not replace -- if file exists then open operation fails

By default, **ofstream** files are opened for output (**ios::out**), and **ifstream** files are opened for input **(ios::in**).

11.5 11.5 Checking open and input success

Overloaded **ios** operator method "**operator!**" is used to check the success of file opening. It returns nonzero (true) when either **failbit** or **badbit** is set for the stream on **open** operation, either for input or output. Some possible errors are attempting to open a nonexistent file for reading, to open a file for reading without permission, to open a file for writing when no disk space is available. Example:

```
ostream file1("Test.txt");
```

```
if(!file1)
{
  cerr << "Can't open file! \n";
  exit(1);
}
```

Another overloaded **ios** operator method **operator void*** is used to check the success of stream **input**. It converts the IO stream into a pointer so it can be tested as 0 or nonzero. If **failbit** or **badbit** is set for the stream, 0 is returned. When EOF is encountered during input, **failbit** is set for **cin**. Example:

```
while(cin>>a)
while(file>>a)
```

the condition in the while header automatically invokes the **operator void*** method. When EOF is inputted from keyboard or encountered in file1, 0 is returned and the while loop will stop.

11.6 11.6 Method exit

Method **exit** is called to end the program. Its argument is returned to the OS so that it can respond appropriately to the error. Argument 0 indicates that program terminates normally, other nonzero values indicate that program terminates due to error.

11.7 11.7 File Position Pointer

The value of the **file position pointer** is the number of the byte which is to be inputted or outputted. It can also be called "**offset**". To set this pointer for **ifstream** object, use method "**seekg**". For **ofstream** objects, use "**seekp**". The arguments of these two methods is normally a **long** integer.

A second argument can also be passed to the methods as **origin** of the offset:

1. 1. **ios::beg** default -- count from beginning of the file
2. 2. **ios::cur** count from current location
3. 3. **ios::end** count from the end of the file

Example:

```
file1.seekg(0);           // beginning of the file
file1.seekp(n, ios::cur)  // nth byte from current position
file1.seekg(n, ios::end)  // nth byte from the end of file
file1.seekp(0, ios::end)  // last byte in the file
```

To get the current position of the file position pointer, use method "**tellg**" for **ifstream** objects and "**tellp**" for **ofstream**:

```
long location = file1.tellp();
```

If you open a file for both input and output:

```
fstream file("Frank.txt", ios::in | ios::out);
```

you can write into it, relocate the file position pointer, then read from it, vice versa.

11.8 11.8 Sequential Access File

In computer's internal memory, different integers such as 7 or 7777 are stored with the same number of bytes. But in a file they are stored in different sizes. Therefore, when you overwrite an original number with a longer one in a file, it will overwrite the following field.

For this reason, this kind of files which store data in varied length are called "sequential access files". You have no idea on exactly how long each record occupies. Therefore, to find a particular record in the file, you have to read from the first one sequentially until you reach the one you want. You can not jump to a certain record directly. If you want to modify a record in the middle of the file, unless the length of the record is not changed, you have to first copy the records before the one to be updated into a file, then append the updated record to that file, then append the records following the updated one to the file.

11.9 11.9 Random Access File and Object Serialization

Random access file is opened in the same way as normal files, but the way to write data in it is different. Complete objects instead of primitives are read or written with class **iostream's** method **read** and **write**. The object to be read and written can contain member objects, pointers and references to other objects, and other objects can again contain member objects, pointers and references. Method **write** will write everything necessary into the file, including the type information and the whole network of objects, so that later method **read** can recover it. The process of breaking an object into data stream is called "object serialization".

There is one restraint: the object to be serialized and all its network objects should all have fixed size. If the class contains a char * data member, because the length of the string is variable, method **write** can not properly allocate space for each record, and run-time error may happen. In such a case, use char [ ] instead of char *.

Method **read** and **write** have the same functionality as Java's **readObject** and **writeObject** method of class **ObjectInputStream** and **ObjectOutputStream**.

Because you can not decide the format or sequence of each field, this way of IO is also called unformatted IO, whereas conventional way of IO is called formatted IO.

Method **write** takes two arguments. First argument is the address of the object to be serialized, and it should be casted to the type of "const char *" type. The second argument is an integer of **size_t** specifying the number of bytes of the record. Keyword **sizeof** can be used to get this size.

Method **read** has similar arguments as **write**, except that the pointer type is "**char ***".

```
  file.write(reinterpret_cast<const char *>(&recordName),
sizeof(recordName));
  file.read(reinterpret_cast<char *>(&recordName),
sizeof(recordName));
```

The position to start reading or writing is decided by file position pointer, which is set by **seekg** for inputting files and **seekp** for outputting files.

```
#include "iostream.h"

//************* class Base ****************
class Base {
public:
   Base(const int = 0);
   Base(const Base &);
   const int get() const;
protected:
   const Base & operator=(const Base &);
private:
   int member;
};

Base::Base(const int i): member(i) {}

Base::Base(const Base & b): member(b.member) {}
```

```
 const Base & Base::operator=(const Base & rv)
 {
   member = rv.member;
   return *this;
 }

 const int Base::get() const
 {   return member;  }


 //***************** class Derived *******************
 class Derived : public Base {
 public:
   Derived(const int = 0, const int = 0);
   Derived(const Derived &);
   const Derived & operator=(const Derived &);
   void print() const;
 private:
   int member;
 };

 Derived::Derived(const int i1, const int i2) : Base(i1),
member(i2)
 {}

 Derived::Derived(const Derived & d) : Base(d),
member(d.member)
 {}

 const Derived & Derived::operator=(const Derived & rv)
 {
   member = rv.member;
   Base::operator=(rv);
   return *this;
 }

 void Derived::print() const
 {
   cout << "Base member is " << Base::get() << endl;
   cout << "Derived member is " << member << endl;
 }


 //************** class ClientData *******************
```

```
class ClientData {
public:
   ClientData(int = 0, Derived * = 0);
   int getId() const;
   void print() const;
private:
   int id;
   Derived * derived;
};

ClientData::ClientData(int i, Derived * d): id(i), derived(d) {}

int ClientData::getId() const
{   return id;  }

void ClientData::print() const
{
   cout << id << ": ";
   derived->print();
}


//***************** main *****************
int main(int argc,
```