Chapter 3. Expressions and Interactivity

cin object and stream extraction operator

```
  cin >> a;
```

This line uses the cin object to read a value from keyboard. >> is a stream extraction operator. It gets the characters from the stream object on its left and stores them in the variable that appreas on the right. In this line, the stream object is cin. cin "streams" characters from keyboard. Thus, the characters from keyboard can be stored at the variable on the left side of the operator.

cin will automatically convert the data read from keyboard to the data type the same sa the variable that is receiving the data (to store the data).

You should include <iostream> header file to enable cin function. cin has to be used in sequence.

Expression

Definition: an expression is a programming statement that has a value.

Chapter 4. Making Decisions

Expression's value

Every expression in C++ has a value.

For example, the value of an assignment expression is the value being assigned to the variable on the left side of the = operator:

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

int main()
{
  double x;
  cout << (x=2) << endl;
  return 0;
}
```

the out put is: 2.

If statement

The if statement has the following syntax:

```
  if (expression) //if the expression in the parenthese is true,
    statement; // the very next statement is excuted.
```

You can use a set of braces to enclose all of the statements you want to make in an if statement.:

```
  if (expression)
  {
    statement;
    statement;
    //Place as many statements here as necessary
  }
```

If/else Statement

The if/else statement will execute one group of statements if the expression is true, or another group of statements if the expression is false.

```
 if (expression)
   statement or block;
 else
   statement or block;
```

If/else if Statement

The general structure is:

```
  if (expression_1)
  {
    statement;    //if expression_1 is true these statements are
    statement;    //executed, and the rest of the structure is
    etc;      //ignored.
  }

  else if (expression_2)
  {
    statement;    //Otherwise, if expression_2 is true, these
    statement;    //statements are executed, and the rest of
    etc;      //the structure is ignored
  }
  //insert as many else if clauses as necessary
  else
  {
    statement;    //These statements are executed if none of the
    statement;    //expressions above are true
    etc;
  }
```

The last else clause, which does not have an if statement following it, is referred to as the trailing else. The trailing else is optional, but in most cases you'll use it. For example, you can use the trailing else as a check for error input (shown in program 4-14).

Nested if Statements

To test more than one condition, an if statement can be nested inside another if statement.

Bool value for infinity (inf)

Infinity will come from, say, division by zero. If infinity is used in if statement, it will consider it true. In another words, the bool value for infinity is 1. The result of the following code is 12345.

```
#include <iostream>

using namespace std;


int main()

{

 double a = 123;

 a /= 0;


 if (a)

  cout << " 12345 " ;

 else

  cout << " 00000 " ;


 return 0;

}
```

Comparing Floating-Point Numbers

Because of the way that floating-point numbers are stored in memory, rounding errors sometimes occur. This is because some fractional numbers cannot be exactly represented using binary. So you should be careful when using the equality operator (= =) to compare floating-point numbers.

To prevent round-off errors from causing this type of problem, you should stick with greater-than and less-than comparisons with floating-point numbers.

I tried the following code but its still not working:

```
//Program 4-4: Comparing floating-point numbers
#include <iostream>
using namespace std;

int main()
{
  double a = 1.5, b = 1.5;
  b += 0.0000000000000001;

  if (a > b)
    cout <<  " a is not equal to b.\n " ;
  else if (a < b)
    cout <<  " a is not equal to b.\n " ;
  else
    cout <<  " a is equal to b.\n " ;
  return 0;
}
```

Block of code

Enclosing a group of statements inside a set of braces creates a *block* of code.

cin.ingore() and getline()

When using `getline()` (and `cin.get()`), cin.get() will collect the new line character (`'\n'`), and getline() will skip the new line character. Make sure that you use `cin.ignore()` to skip the `'\n'`.

(I had trouble when writing code for program 4-12).

Flags

A flag is a Boolean or integer variable that signals when a condition exists in the program. When the flag variable is set to false, it indicates that the condition does not exist. When the flag variable is set to true, it means the condition does exist.

Following block shows whether a sales person has met the sales quota.

```
  //define a Boolean value to be used as flag
```

```
bool sales_quot_met = false;

//if the sales amount is big enough, then set the flag to be true
if (sale >= QUOTA_AMOUNT)
  sales_quot_met = true;
else
  sales_quot_met = false;

//use the flag as criteria later
if (sales_quot_met)
  cout <<  " You have met your sales quota!\n " ;
```

Integer Flags

Integer variables may also used as flags. This is because in C++ the value 0 is considered false, and any nonzero value is considered true. You can use the integer flag as follows:

```
//define a integer variable to be used as flag
int sales_quot_met = 0;

//if the sales amount is big enough, then set the flag to be true
if (sale >= QUOTA_AMOUNT)
  sales_quot_met = 1;
else
  sales_quot_met = 0;

//use the flag as criteria later
if (sales_quot_met)
  cout <<  " You have met your sales quota!\n " ;
```

Logical Operators

Logical operators connect two or more relational expressions into one or reverse the logic of an expression. The following table lists C++'s logical operators.

Operator	Meaning	Precedence

!	NOT	Highest
& &	AND	Medium
| |	OR	Lowest

Examples:

&&:

```
 if (temperature < 20 && minutes>12)
   cout <<  " The temperature is in danger zone. " ;
```

You must provide complete expressions on both sides of the && operator.

Pay attention: if the sub-expression on the left side of an && operator is false, the expression on the right side will not be checked (to save CPU time), this is called short circuit evaluation. This is also true when you use the | | operator.

The Precedence of logical operators is: ! > && > | |. ! has a higher precedence than many of the C++ operators, to avoid an error, you should always enclose its operand in parentheses.

Input validation

Input validation is the process of inspecting data given to a program by the user and determing if it is valid.

Conditional operator and conditional expression

Syntax:

```
 expression1 ? expression2 : expression3;
```

```
 x < 0 ? y = 10 : z = 20;
```

The first expression is the condition to be tested. If the first expression is true, then the second expression will be executed. If the first expression is false, then the third expression will be executed. This is the same as:

```
 if (x < 0)
   y = 10;
 else
   z = 20;
```

The conditional operator is actually (? :), it is a ternary operator, which requires three operands.

Remember, in C++ all expressions have a value, and the conditional expression has a value. The value of a conditional expression depends on the true value of the first expression. If the first expression is true, then the value of the conditional expression is the second expression. If the first expression is false, then the value of the conditional expression is the third expression.

You cannot use cin, cout in the conditional operator (in expression 2 and 3). However, you can achieve similar goal by:

```
cout << (x > 10 ?  " x is larger than 10\n "  :  " x is equal or smaller than 10\n " );
```

The switch Statement

The switch statement lets the value of a variable or expression determine where the program will branch. It tests the value of an integer expression (data type: int, char or enumerated data type (check chapter 11)) and then uses that value to determine which set of statements to branch to. The format of the switch statement is as follows:

```
swtich(Integer_Expression)
{
  case constant_expression:
    //place one or more
    //statements here
  case constant_expression:
    //place one or more
    //statements here

  //case statements may be repeated as many
  //times as necessary

  default:
    //place one or more
    //statements here
}
```

Without break statement, the program “falls through” all of the statements below the one with the matching case expression.

Swtich statement sometimes can be eaiser to use than if/else statement, especially when you need to design a menu.

Pay attention! You shouldn’t forget the break; statement. If there is a nested switch statement, don’t forget the corresponding break; Otherwise your program has malfunction.

Scope of a variable

The scope of a variable is limited to the block in which it is defined (*local scope* or *block scope*). Enclosing a group of statements inside a set of braces creats a *block* of code.

Chapter 5. Loops and Files

Increment and Decrement

To increment a value means to increase it by one, to decrement a value means to decrease it by one. The increment and decrement operator in C++ is:

```
//Postfix mode
num++;
num--;


//Prefix mode
++num;
--num;
```

Both prefix mode and post mode can add 1 or subtract 1 from their operand. However, when an operand is engaged in situation that needs to be used twice or more in different operators, prefix mode and postfix mode is different. Prefix mode will increment or decrement the operand **BEFORE** the operand is used in other operators, while postfix mode will increment or decrement the operand **AFTER** the operand is used in other operators.

Use + + and – – in Mathematical Expressions

The increment or decrement operator can be used in mathematical expressions. For example:

```
int a = 2, b = 5,c;
c = a*b++;
cout << a << endl << b << endl << c << endl;
```

The output is:

```
2
6
10
```

Or, in prefix mode (pay attention to the precedence):

```
int a = 2, b = 5,c;
c = a*++b;
```

```
 cout << a << endl << b << endl << c << endl;
 return 0;
```

The output is:

```
 2
 6
 12
```

However, the increment or decrement operator can only be used on modifiable lvalue (left-value, which means the value can appear on the left side of an assignment operator). Thus, following assignment statement is not working:

```
 c = ++(a*b); //not working because a*b is not a lvalue
```

Generally speaking, anything that can go on the left side of an = operator is legal.

Use + + and – – in Relational Expressions

The idea is the same (prefix mode: change first before being used by other operator; postfix mode: after being used by other operator, then change). For example:

```
 x = 10;
 if (x++ > 10)
   cout <<  " x is greater than 10.\n " ;
```

Here, *x* = 10 will first be used in the relational expression, then *x* = *x* + 1.

Loop

A loop is part of the program that repeats. C++ has three looping control structures: the while loop, the do-while loop and the for loop.

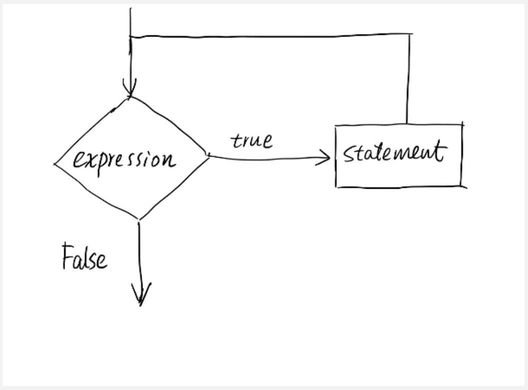
While Loop

The while loop has two important parts:

(1) an expression that is tested for a true or false value.

(2) a statement or block that is repeated as long as the expression is true.

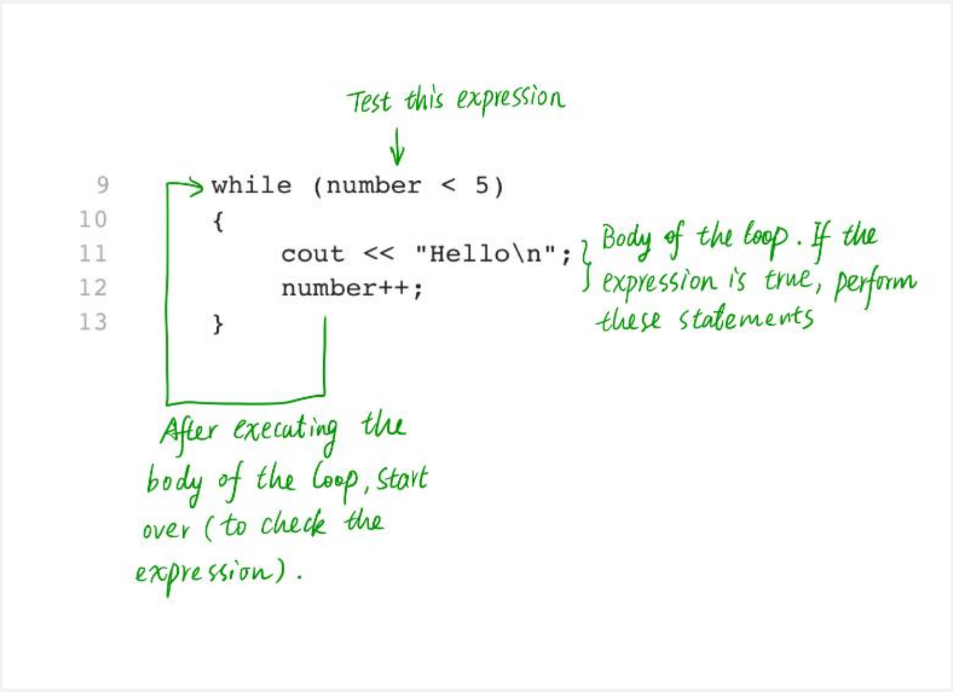
Following figure shows the logic of while loop.



The general format for the while loop is:

```
 //one line of statement
 while (expression)
   statement;


 //block of statements
 while (expression)
 {
   statement;
   statement;
   //place as many statements as
   //necessary here
 }
```

Description of the while loop:



```
while (number < 5)
{
     cout << "Hello\n";
     number++;
}
```

Test this expression
Body of the loop. If the expression is true, perform these statements
After executing the body of the loop, start over (to check the expression).

The `number` is referred to as the loop control variable because it controls the number of times that the loop iterates.

while Loop is a Pretest Loop

The while loop is a pretest loop. It tests its expression before each iteration.

Using the while Loop for Input Validation

The while loop is especially useful for validating input. If the data user input is not the accepted data type, the program can ask the user to re-enter the data. For example:

```
//Input validation
#include <iostream>
using namespace std;
int main()
{
 int a;
 cout <<  " Please input a data that is higher than 0 and lower than 100.\n " ;
 cin >> a;

 while (a <= 0 || a >= 100)
 {
  cout <<  " Your input is invalid, please input again.\n " ;
  cin >> a;
 }

 cout << " You inputed " << a << endl;
 return 0;
}
```

Counters

A counter is a variable that is regularly incremented or decremented each time a loop iterates.

do-while Loop

The do-while loop is a posttest loop, which means its expression is tested after each iteration. The syntax for do-while loop is:

```
 do
   statement;
 while (expression);
```

You can insert multiple statements in the do section:

```
 do
 {
   statement1;
   statement2;
   statement3;
 }
 while (expression);
```

Using do-while with Menus

The do-while loop is a good choice for repeating a menu. You can repeat the menu as long as the “quit” choice is not made.

The for Loop

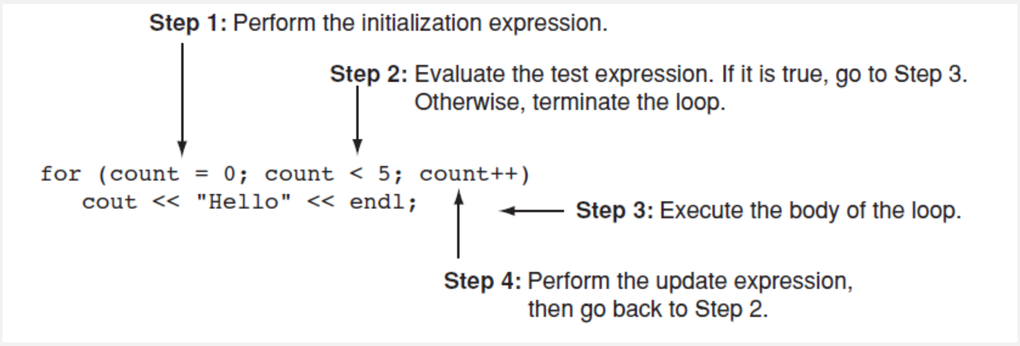
The for loop is ideal for performing a known number of iterations. It is specifically designed to initialize, test and update a counter variable. The syntax of the for loop is:

```
  //for loop: single statement
  for (initialization; test; update)    //pay attention, semicolon! Not comma
```

```
    statement;
  //for loop: a block of statement
  for (initialization; test; update)    //counters should be defined first
  {            //You can define at the initialization
    statement1;
    statement2;
    //add as many statements as necessary
  }
```

The test expression will be tested at the loop header, so for loop is pretest loop.

The orders of the execution of for loop is as follows:


Step 1: Perform the initialization expression.
Step 2: Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop.
Step 3: Execute the body of the loop.
Step 4: Perform the update expression, then go back to Step 2.

```
for (count = 0; count < 5; count++)
   cout << "Hello" << endl;
```

Use Counter Variable inside for Loop

The counter can be used inside the body of for loop. However, you should avoid modifying the counter variable in the body of the for loop.

Times to Use for Loop

The characteristic of using for loop:

1) Requires an initialization
2) Use a false condition to stop
3) requires an update to occur at the end of each loop iteration
4)

Defining a Variable in the for loop's initialization expression

The counter variable can be defined in the initialization expression, the syntax is as follows:

```
for (int count = N_min; count <= N_max; count++)
```

When a variable is defined in the loop header's initialization section, its scope is within the for section (either one line or a block). You can't use the variable outside of the block.

Using Multiple Statements in the Initialization and Update Expressions

You can execute more than one statement in the initialization expression and the updated expression. When using multiple statements in either of these expressions, simply separate the statements with commas.

If you want to combine multiple expressions in the test expression, you must used && or || operators.

Pay attention, the variable in the test section should be initiated before the program runs the test section.

Omitting the for Loop's Expressions

If the initialization has already been performed or no initialization is needed, the initialization section in the for loop can be omitted.

You may also omit the update expression if it is being performed elsewhere in the loop or if none is needed, although this type of code is not recommended. For example, the following for loop works just like a while loop:

```
#include <iostream>

using namespace std;

int main()
{
  int i = 1;
  for (; i <= 5;)
  {
    cout << i << endl;
    i++;
  }
  return 0;
}
```

If all three expressions from the for loop are omitted, the loop becomes an infinite loop.

Running Total

A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator.

Programs that calculate the total of a series of numbers typically use two elements:

1) A loop that reads each number in the series; (includes a counter)
2) A variable that accumulates the total of the numbers as they are read; (includes a accumulator)

Sentinels

A sentinel is a special value that marks the end of a list of values. When the user enters the sentinel, the loop terminates.

Which Loop to Use?

While:

Conditional loop;

Pretest loop;

Examples: validating input, sentinel value.

do-while:

Conditional loop;

Posttest loop;

Examples: repeating a menue (after choosing quit, then quit. This is posttest)

for:

Posttest loop;

Example: situations where the exact number of iterations is known.

Nested Loop

A loop that is inside another loop is called a nested loop.

Files

When a file is used by a program, three steps must be taken:

1) Open the file. This step creates the connection between the file and the program. So the program can either write data onto the file or read data from the file.
2) Process the file. Data is either written to or read from the file.
3) Close the file. Closing the file disconnects the file from the program.

File type

Generally, there are two types of files: text and binary. A text file contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters.

File Access Methods

There are two general ways to access data stored in a file:

1) Sequential access

When you work with a sequential access file, you access data from the beginning of the file to the end of the file.

2) Direct access

When you work with a random access file (also known as a direct access file), you can jump directly to any piece of data in the file without reading the data that comes before it.

Filenames and File Stream Objects

Files on a disk are identified by a filename. Each operating system has its own rules for naming files. Many systems, including Windows, support the use of filename extensions.

In order for a program to work with a file on the computer's disk, the program must create a *file stream object* in memory. A file stream object is an object that is associated with a specific file and procides a way for the program to work with that file. (stream means, the file can be viewed as a stream of data).

File stream objects work very much like the cin and cout objects. A stream of data may be sent to cout, which causes values to be displayed on the screen. A stream of data may be read from the keyboard by cin, and stored in variables. Likewise, streams of data may be sent to a file stream object, which writes the data to a file. When data is read from a file, the data flows from the file stream object that is associated with the file, into variables. The difference between the file stream object and cin, cout object is that, for each file, there is a file stream object corresponds to it. However, there is only one cin, cout object (correspond to keyboard input and console output).

Setting Up a Program for File Input/Output

Just as cin and cout require the iostream file to be included in the program, C++ file access requires another header file. The file <ftream> contains all the declarations necessary for file operations. It is included with the following statement:

```
#include <fstream>
```

The fstream header file defines the following data type (these are the data type you can define after you include the fstream header file):

1) oftream

Output file stream. You create an object of this data type when you want to create a file and write data to it (write only).

2) iftream

Input file stream. You create an object of this data type when you want to open an existing file and read data from it (read only)

3) fstream

File stream. Objects of this data type can be used to open files for reading, writing or both.

Remember, ofstream, ifstream and fstream are the “keywords” to define objects (these keywords defines the data type of the object). They are the same as int, double etc.

Creating a File Object and Opening a File

Before data can be written to or read from a file, the following things must happen:

- Create a file stream object
- Open a file and link the file to the file stream object

The following code shows an example of opening a file for input (reading).

```
ifstream input_file; //defines an ifstream object named input_file
```

```
input_file.open( " Customers.txt " ); //calls the object’s open member function, passing the string “Customers.txt” as an argument
```

After `input_file.open( " Customers.txt " );` the open member function opens the Customers.txt file and links it with the input_file object. After this code executes, you’ll be able to use the input_file object to read data from the Customers.txt file.

The following code shows an example of opening a file for outputting (writing).

```
ofstream output_file;

output_file.open( " Employees.txt " );
```

After the second line, the output_file object is linked with Employees.txt file, then you will be able to use the output_file object to write data to the Employees.txt file.

When you call an ofstream object’s open member function, the specified file will be created. If the specified file already exists, it will be erased, and a new file with the same name will be created. (Not write data into the old file!!).

When opening a file, you will need to specify its path as well as its name, and extension. For example:

```
output_file.open( " C:\\Users\\My\\Desktop\\jiesuoma.txt " );
```

In this statement, the file C:\Users\My\Desktop\jiesuoma.txt is opened and linked to output_file.

Notice the use of two backslashes in the file’s path. Two backslashes are needed to represent one backslash in a string literal (\\, similar with \t, \n).

You can define a file stream object and open a file in one statement, for example:

```
ofstream output_file( " C:\\Users\\My\\Desktop\\jiesuoma.txt " );
```

Open a File at the designated path

When you call the .open member function, you can pass a string variable which contains the path information of the file you’re going to work with to the function (as an argument). For example:

```
 string path;
cout <<  " Please input the path of your data file.\n " ;
 getline(cin, path);
 read.open(path);
```

Closing a File

It is a good programming practice to write statements that close a file, because:

1) Save data in the file buffer to the file. So the data in the file can be used later in this program.
2) To save the system's resources.

You can close the file by calling the file stream object's close member function. For example:

```
 ifstream input_file( " C:\\Users\\My\\Desktop\\jiesuoma.txt " ); //to define a file stream object: input_file and open a file, link the file to input_file.
 input_file.close(); //close the file, disconnect the link between file stream object (input_file) and the file (jiesuoma.txt)
```

Writing Data to a File

Writing data to a file is similar with writing data to the screen. In the latter circumstance, you use the stream insertion operator (<<) with the cout object to write data to the screen. When you write data to a file, you can use the stream insertion operator and the file stream object to direct data stream "flows" into the file. Following is an example:

```
 ofstream output_file( " C:\\Users\\My\\Desktop\\jiesuoma.txt " ),
    output_file2( " C:\\Users\\My\\Desktop\\jiesuoma2.txt " );
 output_file <<  " You are making progress!\n " ;
 output_file2 <<  " You are writing into another file!\n " ;
```

In the above example, I created two file stream object, linked them to different files. Use different file stream object to write data into the corresponding files.

Reading Data from a File

When the >> operator is used with cin object, it directs the data stream from the keyboard input to the variable. You can also use >> operator to direct data stream from the file to the variable. Reading data from a file is very similar with cin >>. Combine the << operator with the file stream object, you can read data from a file. Example:

```
 ifstream read_file;
```

```
read_file.open( " C:\\a.txt " );

read_file >> day;
```

Read Position

When a file has been opened for input, the file stream object internally maintains a special value known as a read position. A file's read position marks the location of the next byte that will be read from the file. When an input file is opened, its read position is initially set to the first byte in the file. As data is read from the file, the read position moves forward, toward the end of the file. When the >> operator extracts data from a file, it expects to read pieces of data that are separated by whitespace characters (spaces, tabs, or newlines).

If you want to extract a line from the text, you can use getline function (which will not be stopped by whitespace characters other than newline)

Detecting the End of the File

When the >> operator reads data from a file, it also returns a true or false value indicating whether the data was successfully read or not. If the operator returns true, then a value was successfully read. If the operator returns false, it means that no value was read from the file. For example:

```
int count = 0;

while (read >> number)

  count++;
```

In the above statements, variable count will store the number of data in the file. Notice that the statement that extracts data from the file is used as the Boolean expression in the while loop. It works in the following way:

1) The expression read >> number executes
2) If an item is successfully read from the file, the item is stored in the number variable, and the expression returns true to indicate that it succeeded. In that case, the statement count++ executes and the loop repeats.
3) If there are no more items to read from the file, the expression returns false, indicating that it did not read a value. In that case, the loop terminates.

Testing for File Open Errors

After you call the open member function, you can test the file stream object as if it were a Boolean expression. For example:

```
cout <<  " Please input the address of your data file.\n " ;

getline(cin, address);

read.open(address);
```

```
while (!read) //or you can use the .fail member function
{
  cout <<  " File openning failed, please input again.\n " ;
  getline(cin, address);
  read.open(address);
}
```

This code asks the user to input a path of the file they want to read. If the opening is unsuccessful, the program will ask user to re-enter the path information, until it can open the file successfully.

Breaking and Continuing a Loop

Break statement

The break statement causes a loop to terminate early. The continue statement causes a loop tp stop its current iteration and begin the next one.

The break statement was first introduced in switch statement. It can also be placed inside a loop. When it is encountered, the loop stops, and the program jumps to the statement immediately following the loop.

In nested loop, the break statement only interrupts the loop that is placed in.

Continue statement

The continue statement causes the current iteration of a loop to end immediately. When continue is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration.

Chapter 6. Functions

Defining and Calling Functions

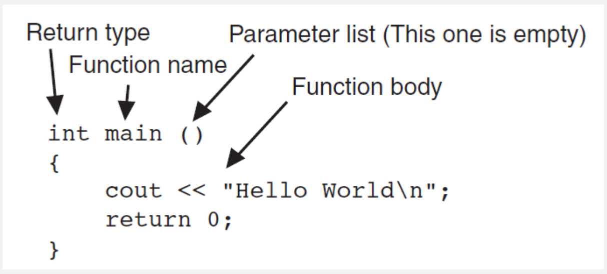
A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function:

```
void function()
{
 statement1;
 statement2;
```

statement3;

}

or:



```
int main ()
{
    cout << "Hello World\n";
    return 0;
}
```

When you define a function, the parameters should be defined separately, you should state the data for each parameters, separate by a comma. The scope of a parameter is limited to the body of the function that uses it.

A function call is a statement.

About function name: you can't use name of an existed function, even if you don't include corresponding header file. For example, you can't name a function "pow" even if you don't include <cmath>.

Functions can be called in hierarchical, or layered fashion (you can call a function inside a function, but everytime you call a function, you should either define it before the call of a function, or put a function prototype before the call of function). A function prototype looks like this:

```
void print_hello();
```

Before the compiler encounters a call to a particular function, it must already know the function's return type, the number of parameters it uses, and the type of each parameter. The statement above tells the compiler that the function print_hello has a void return type (it doesn't return value), and uses no parameters. Another example of function prototype is:

```
double time2(double,int)
```

It tells the compiler that the function time2 has a double return type, uses two parameters, the type of first variable is double, the type of second variable is integer. It is not necessary to list the name of the parameter variable inside the parentheses. Only its data type is required (data type for each parameter, separate by a coma).

It is a common practice to place the main() function at the beginning of a program. Place function prototypes at the beginning for those functions that are defined after main() function.

Sending Data into a Function

Values that are sent to a function are called arguments (arguments are sent to parameters of the function). A parameter is a special variable that holds a value that is being passed into a function. Following example shows the definition of a function that uses a parameter:

```
void display_Value(int num)
{
  std::cout <<  " The value is  "  << num << endl;
}
```

Notice the integer variable definition inside the parentheses (int num). The variable num is a parameter. This enables the function display_Value to accept an integer value as an argument. When you call the function, the values inside the parentheses is copied to the parameter variable (in this case, num). If you pass an argument whose type is not the same as the parameter's type, the argument will be promoted or demoted automatically.

When an argument is passed into a parameter, only a copy of the argument's value is passed. Changes to the parameter do not affect the original argument.

The return Statement

The return statement causes a function to end immediately, and control of the program returns to the statement that called the function.

Even in void function, you can use return to terminate the function and return to the main program.

Returning a Boolean Value

Functions may return true or false values. Just define the return type as bool.

Local and Global Variables

A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope.

Initializing Local Variables with Parameter Values

We can initialize local variable with parameter values, example:

```
int sum(int num1, int num2)
{
  int result = num1 + num2;
  return result;
}
```

Defining Global variable

The global variable can be defined outside all function body (block). Example:

```
int num = 2;            // Global variable
int main()
{
   …
}

void anotherFunction()
{
   …
}
```

Unless you explicitly initialize numeric global variables, they are automatically initialized to zero. Global character variables are initialized to NULL, the first slot in global string variable is initialized to NULL.

Be Mindful When Using Global Variable

Although global variable might make a program easier to write, it usually causes problems later.

Local and Global Variables with the Same Name

When local and global variable (or constant) has the same name, the global variable (or global constant) will be shadowed from view (you can only access the local variable or constant in the local scope).

Global Constants

A global constant is a named constant that is available to every function in a program. It is used to represent unchanging values that are needed through out a program.

Static Local Variables

Normally, local variables in a function is destroyed when the function terminates, and then created again when the function is called again.

However, you can create a variable that can be remembered between every function calls. You can accomplish this by making the variable static, following is how:

```
void show_static()

{

static int num;

}
```

You add “static” in front of the data type of the variable. In this way, the variable exists for the lifetime of the program. **By default, the static local variables are initialized to zero.** If you provide an initialization value for the static variable, the initialization only occurs once.

Default Arguments

Default arguments are passed to parameters automatically if no argument is provided in the function call. When there is no argument is provided when the function is being called, the default argument is passed to the parameter. The default arguments are usually listed in the function prototype:

```
  void show_area(double = 20, double = 10);
```

You don't have to provide the parameter names when defining default argument, since they are optional in function prototypes, but you have to write the ′ =′ .

Also, the default arguments can be listed in the function definition. However, it is better to assign the default argument in the function portotype because it appears first. You don't have to list the default arguments in the function definition if you've already listed in header.

It's possible for a function to have some parameters with default arguments and some without. When calling these functions, arguments must always be specified for the parameters which have no default arguments. When a function uses a mixture of parameters with and without default arguments, **the parameters with default arguments must be defined last.** (because as long as you defined a default parameter, you can′ t define an un-default parameter, the following parameter should all have default argument)

Just define the default value once in the program. Do not define it at the same time in function prototype and function header. Otherwise, you′ ll receive " `redefinition of default argument: parameter` " error when you compile the program.

About the Omission of Arguments when Call the Function

When an argument is left out of a function call, all arguments that come after it must be left out as well.

Using Reference Variables as Parameters

When used as parameters, reference variables allow a function to access the parameter's original argument. Changes to the parameter are also made to the argument. By using a reference variable as a parameter, a function may change a variable that is defined in another function.

Defining: A reference variable is defined like regular variables, except you place an ampersand (&) in front of the name. For example:

```
void double_number(int &ref_var)
{
  ref_var *= 2;
}
```

When prototyping a function with a reference variable, be sure to include the ampersand after the data type. The prototype of the above function is:

```
void double_number(int &);
```

or:

```
void double_number(int &, int &, double &); //multiple reference variables
```

When calling a function with a reference variable, you don't have to input the ampersand (&).

The variable defined in the header of the function with reference variable still bond to the local scope. It can not be used elsewhere. When a program works with a reference variable, it is actually working with the variable it references, or points to. When a reference parameter is used, it is said that the argument is passed by reference.

Be mindful that only variables may be passed by reference.

[the ampersand & is actually the address operator, see more detail in chapter 9]

Overloading Functions

Two or more functions may have the same name, as long as their parameter lists are different (data type or the number of terms). This is because in C++, each function has a signature. The ***function signature*** is the name of the function and the data types of the function's parameters in the proper order. However, the function return value is not part of the signature.

The exit() Function

The exit() function causes a program to terminate, regardless of which function or control mechanism is executing. The head file <cstdlib> is needed for the exit function.

The exit() function takes an integer argument. This argument is the exit code you wish the program to pass back to the computer's operating system. exit(0) commonly indicates a successful exit. Also, you can use two named constants:

exit(EXIT_FAILURE); //unsuccessful termination

exit(EXIT_SUCCESS); //successful termination

These codes are tested outside of the program. Generally, if it is not used, just pass zero, or EXIT_SUCCESS.

Stubs and Drivers

Stub

A stub is a dummy function that is called instead of the actual function it represents. It usually displays a test message acknowledging that it was called, the arguments passed to the parameter, nothing more (display the information in human language). If the stub shows the function is called, you know that the parts of the program that call the function is working properly. Otherwise, it may have some problems to be fixed. Thus, by replacing an actual function with a stub, you can concentrate your testing efforts on the parts of the program that call the function. If the stub represents a function that returns a value, then the stub should return a value. This helps you confirm that the returned value is being handled properly.

Basically, stub can test the part in the program that call the function. If they can call the function successfully, you can then move on to testing and debugging the actual functions. This is where drivers become useful.

(stub is dummy function)

Drivers

A driver is a program that tests a function by simply calling it. If the function accepts arguments, the driver passes test data. If the function returns a value, the driver displays the return value on the screen. This allows you to see how the function performs in isolation.

Basically, driver is a very basic environment (main() function) that can drive the function to run (establish an environment that other functions can run).

Chapter 7. Arrays

Array Characteristic

An array allows you to store and work with multiple values of the same data type. The values are stored together in consecutive memory locations. Definition of array is:

```
int days[4];
```

The name of the array is days. The number inside the brackets is the array's size declaratory, it indicates the number of elements, or values, the array can hold. An array's size declaratory must be a constant integer expression with a value greater than zero (it can be literal or named constants).

The amount of memory used by an array depends on the array's data type and the number of elements. The size of an array can be calculated by multiplying the size of an individual element by the number of elements in the array. You can use `sizeof()` to get the size of an array you defined:

```
int list[10];
std::cout << "The size is: " << sizeof(list) << '\n' ;
```

The result will show: 40.

Accessing Array Elements

The elements may be accessed and used as individual variables. Because the individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements. If an array has N elements, the first element is assigned the subscripts 0, and the Nth element is assigned the subscripts (N-1). Notice that subscripts in C++ always starts at zero, the subscripts of the last element is one less than the array's size declarator. Elements in local array don't have initial value unless you initiate it. If an array is defined globally, all of its elements are initialized to zero by default.

Accessing the array element (Inputting or outputting data) must normally be done one element at a time (you must specify the subscripts).

Array initialization

Arrays may be initialized when they are defined:

```
int days[12] = { 31, 28, 31, 30, 31, 30, 31, 30 };
```

You can also spread the initialization list across multiple lines:

```
int days[12] = { 31,
  30,
  31,
  30,
  31 };
```

Initialize with strings:

```
std::string name[5] = {  " sun " ,  " earth " ,  " pluto " ,  " mars " ,  " moon "  };
```

If an array is partially initialized, the remaining element will be set to:

Array Type	Default Element
int	0
char	NULL
string	Nothing?

C++ does not provide a way to skip elements in the initialization list.

Implicit Array Sizing

It is possible to define an array without specifying its size, as long as you provide an initialization list.

Range-based for Loop

The range-based for loop is a loop that iterates once for each element in an array. The range-based for loop automatically knows the number of elements in an array. Each time the loop iterates, it copies an element from the array to a variable, which is a built-in variable known as the range variable. For example, the first time the loop iterates, the range variable will contain the value of element 0, the second time the loop iterates, the range variable will contain the value of element 1, and so forth.

The format for the range-based for loop is as follows:

```
  for (data_type range_variable : array_name)
  {
    statement_1;
    statement_2;
    ...
    statement_n;
  }
```

You can change the name of the range variable as you like. Data type of the range variable should be same as the array, or can be automatically converted to the array's data type. You can use "auto" to declare the data type of the range variable. In this way, the data type of the range variable will be the same as the array.

Modify Array Elements with Range-Based for Loop

When you define the range variable, you can define it (declare) as a reference variable, so you can access the original memory position of the array's element.

The name of an array

Anytime the name of an array is used without brackets and a subscript, it is seen as the array's beginning memory address. For example, the following code will show the memory address for the array's first element:

```
#include <iostream>
int main()
{
  int num[3];
  std::cout << num << std::endl;
}
```

The result is: 010FF824.

Arrays as Function Arguments

You can write functions that can accept the entire array as arguments. The setup of parameters is slightly different from parameters for single variable:

```
void show_values(int num[], int size)
{
  for (int i = 0; i < size; i++)
    std::cout << num[i] << std::endl;
}
```

The parameter num is followed by **an empty set of brackets**. This indicates that the argument will be an array, not a single value. The num is actually not an array, but a variable that will hold the memory address of the array that is passed to this function.

When an entire array is passed to a function, it is not passed by value, but passed by reference.

You can write the function prototype as:

```
void show_values(int [], int);
```

When you pass the array name to a function, you are actually passing the memory address of the first element in the array to the function. However, sometimes you don't want a function to be able to modify the contents of an array that is passed to it as an argument (just remain in the old way of non-reference, pass-by-value style parameter). You can add "const" before the parameter declaration. In this way, if the function attempts to modify array value, the program won't compile.

Range-based for loop in function

It seems you can't use range-based for loop in function if only the array is passed to the function. For example, following function is not working:

```
void double_array(int num[])
{
  for (auto element:num)
    element *= 2;
}
```

With an error message: this range-based 'for' statement requires a suitable "begin" function and none was found.

Two-Dimensional Arrays

Two dimensional arrays are like matrix. To define a two-dimensional array, two size declarators are required. The first one is the size declaratory for rows, the second one is the size declaratory for columns. For example:

```
double scores[3][4];
```

When processing the data in a two-dimensional array, each element has two subscripts: one is for row and one is for column.

Initialization

When initializing a two-dimensional array, it helps to enclose each row's initialization list in a set of braces. For example:

```
int hours[3][2] = { {8,5},{7,9},{6,3} };
```

The inside braces are optional, which means you can also write like this:

```
int hours[3][2] = {8,5,7,9,6,3};
```

Initialization of elements is proceed row by row.

You can also partially initialize your 2D array:

```
int hours[2][2] = { {1},{2,3} };
```

In this example, hour[0][0]=1, hour[0][1]=0 (automatically initialized to zero), hour[1][0]=2, hour[1][1]=3.

Passing two-dimensional arrays to functions

When a two-dimensional array is passed to a function, the parameter type must contain a size declaratory for the number of columns:

```
const int COLS = 4;
void show_array(double num[][COLS], int rows)
```

Pay attention, size declarator can be only named constants or literals greater than zero. In another word, a function can only accept limited types of two-dimensional array (with the same column number declared in the function header).

When you write the function prototype, the information of column number should also be included:

```
void show_array(double [][COLS], int);
```

Arrays with Three or More Dimensions

C++ does not limit the number of dimensions that an array may have. Below is an example to create a three-dimensional array:

double seats[3][5][8];

This array can be thought of 3 sets of 5 × 8 matrixes.

When writing functions that accept multidimensional arrays as arguments, all but the first dimension must be explicitly stated in the parameter list.

STL Vectors

General

To use vectors in your program, you have to include the vector header file:

```
#include <vector>
```

the definition of vector is as follows:

```
  std::vector<int> num;
```

This statement defines `num` as a vector of ints. The vector expands in size as you add values to it (you cannot do it using the array subscripts operator [ ], you need to call push_back member function of the vector to do this), you don't have to declare a size. You can define a starting size, if you prefer:

```
  std::vector<int> num(10);
```

Not like array, you use parenthesis (not bracket) to specify the starting size of the vector (you still use bracket to access members in a vector). When you specify a starting size of a vector, you may also specify an initialization value. The initialization value is copied to each element. Here is an example:

```
  std::vector<int> num(10, 2);
```

After the above statement, each element in num is initialized to the value of 2.

Also, you can put pointer type into the parenthese, especially when you want to initialize a vector using an array containing certain values, example:

```
  int array[] = { 1,2,3,4,5 };
  std::vector<int> v1(array, array + sizeof(array)/sizeof(array[0]));
```

Code in red is just to calculate the number of element in the array.

You may also initialize a vector with the values in another vector

Initialization vector with a list

You can initialize a vector with a list of variables, as shown below:

```
  std::vector<int> num{ 10,20,30,40,50 };
```

Notice that there is no = operator before the list, which is different from initialization of arrays.

Using the push_back Member Function (to add elements to vector)

Use push_back member function to store a value to a vector that is already full or **doesn't have a starting size**. The push_back member function accepts a value as an argument and store the value after the last element in the vector. For example:

```
num.push_back(25);
```

Data type in the parenthese should be the same as declared data type of the vector. If not, an implicit conversion will occur if possible. If its not possible, an error will occur and the code won't compile.

Using the pop_back Member Function (to remove elements from vector)

Use pop_back member function to remove the last element from the vector:

collection.pop_back();

Using the clear() Member Function (to clear all elements)

Using clear() member function will clear all the elements in vector, you won't be able to use [ ] operator any more if no element is stored in the vector.

Determining the Size of a vector

Using a size member function, vector can report the number of elements they contain. For example:

```
num_of_elements = vector_set.size();
```

After the statement executes, num_of_elements will contain the number of elements in vector_set.

Vectors as Function Arguments

Vectors can be used as function arguments. Following example shows the head of such function:

```
void show_value(std::vector<int> hour)
```

Pay attention, passing vectors as function parameters don't have [ ].

Vectors are passed by value, not like array. You must use an ampersand before the name of vector to pass it by reference.

Additional Member Functions of Vector

The details for vector's member function can be found at:

http://www.cplusplus.com/reference/vector/vector/

In this section, brief introduction will be provided for some other member function for vector.

at()

Parameter: at(n);

Function: returns the value of the element located at position *n* in the vector.

Example: x = num.at(5);

The value of element 5 of num vector has been assigned to x.

what is the difference between num[5]?

back()

Parameter: none

Function: returns a **reference** to the last element in the vector

Example: std::cout << num.back() << " \n " ;

Pay attention that the return type is a reference, so the last element can be accessed by this member function. For example:

```
int main() {
  std::vector<int> list{ 1,2,3,4,5 };

  list.back()++;
  ShowValue(list);

  return 0;
}
```

The result will be 1 2 3 4 6, the last element has been modified through the reference.

begin()

Parameter: none

Function: returns an iterator pointing to the first element of the vector

Example: iter = num.begin();

After the statement, iterator `iter` now pointing to the first element of vector `num`.

capacity()

Parameter: none

Function: returns the maximum number of elements that may be stored in the vector without additional memory being allocated.

Example: x = vect.capacity();

clear()

Parameter: none

Function: clears a vector of all its elements, this member function will not delete the allocated memory space.

Example: vect.clear();

empty()

Parameter: none

Function: returns true of the vector is empty (i.e., vect.size() == 0), otherwise, return false

Example:

```
 if (vect.empty)
   std::cout <<  " The vector is empty. " ;
```

end()

Parameter: none

Function: returns an iterator pointing to the *past-the-end* element of the vector container: the theoretical element after the last element of the vector container.

Example: iter = num.end() - 1;

After the statement, iterator `iter` now points to the last element of vector `num`.

erase()
Overload_1 (for deleting single element)

Parameter: random-access iterator (iterator and const_iterator) that points to elements (pay attention, the parameter is not integeral subscript!)

Function: removes the single element pointed by the argument iterator

Example:

```
std::vector<int> vect;
vect.push_back(1);
iter = vect.begin();
vect.erase(iter);
```

(PS, `iter = vect.begin();` should be placed after `vect.push_back(1);` Because the beginning memory space has not been determined before `vect.push_back(1);`)

Overload_2 (for deleting a range of elements, ([first, last))

Parameter: two random-access iterators that determine a range: [first, last).

Function: removes all the vector elements from iterator first to iterator last.

Example: vect.erase(first, last);

To erase a value from a vector requires all the elements after the removal point to be moved one position toward the vector's beginning (pay attention to the number of moves it may take).

front()
Parameter: none;

Function: returns a reference to the vector's first element

Example: std::cout << num.front " \n " ;

insert()
Parameter: insert(iter, value)

Function: inserts the value into the vector. The value is inserted into the element before the one pointed to by the iter.

(check the detailed explaination for usage)

To insert a value into the middle of a vector requires all the elements after the insertion point to be moved one position toward the vector's end, thus making room for the new value.

resize()
Parameters: resize(n, value)

Function: Resizes a vector so it contains n elements. If n is smaller than the current size, the content is reduced to its first *n* elements, removing those beyond;

If n is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of n. If value is specified, the new elements are initialized as copies of value, otherwise, they are value-initialized.

If n is also greater than the current container capacity, an automatic reallocation of the allocated storage space takes place.

Example:

```
std::vector<int> list{ 1,2,3,4,5 };


list.resize(3, 0); // list becomes { 1, 2, 3 }

list.resize(7, 0); // list becomes { 1, 2, 3, 4, 5, 0, 0 }
```

reverse()
Parameter: none;

Function: reverse the order of the items stored in a vector

Example:

(no reverse()?)

Range based for loop for vector

From C++11, we can use range based for loop to traverse vector. Syntax:

```
for (auto i : your_vector)
```

For each iteration, the variable i has the same type and value as the corresponding entry in the vector (but its not reference, you can't modify whatever in the vector by i).

Example:

```
  std::vector<int> v2{ 1,2,3,4,5 };
  for(auto i : v2) {
    std::cout << i << ' \n' ;
  }
```

Vector of Pairs

Before proceeding, check the STL Pair for a brief introduction on pair container.

What is vector of pairs? A pair is a container which stores two values mapped to each other, and a vector containing multiple number of such pairs is called a vector of pairs.

Declaration

Remember for normal data types (e.g. int, double etc), we declare a vector of such type using following syntax:

```
#include <vector>
int main() {
  std::vector<data_type> v1;
}
```

The same goes for declaring a vector that contains multiple pairs:

```
std::vector<std::pair<data_type_1, data_type_2>> v_pair;
```

An example of declaring a vector with `<int, int>` pair is as follows:

```
  std::vector<std::pair<int, int>> v1;
  std::pair<int, int> p1;

  int array1[] = { 1,2,3,4,5 };
  int array2[] = { 10, 20, 30, 40, 50 };

  for (int i = 0; i < sizeof(array1)/sizeof(array1[0]); i++) {
    p1 = std::make_pair(array1[i], array2[i]);
    v1.push_back(p1);
  }
```

STL Pair

General

Reference: https://www.geeksforgeeks.org/pair-in-cpp-stl/

Few things to point out:

- The pair container is defined in <utility> header
- Pair is a container, just like string or vector, a pair contains two data elements or objects
- The first element is referenced as “first” and the second element as “second”, the order of the two elements is fixed

- The type of two elements can be same or different. Pair provides a way to store two heterogeneous object as a single unit
- Pair can be assigned, copied and compared. The array of objects allocated in a map or hash_map are of type "pair" by default in which all the "first" elements are unique keys associated with their "second" value objects
- To access the two element stored in one pair, we use the name of the pair (the variable name we defined) followed by dot operator, and then followed by the keyword "first" or "second".

Basic Operation

Define

Before using pair container, we have to include the <utility> header file

```
#include <utility>
```

Then, we can define a pair using following syntax:

```
std::pair<data_type_1, data_type_2> name_of_the_pair;
```

For example, if I define a pair named p1, which contains an integer and a character, then:

```
std::pair<int, char> p1;
```

Assign (access individual element)

Use dot operator and two keywords ("first" and "second") to access the two elements stored in pair. Take p1 as an example, now I want to store 55 into the first element, and 'M' to the second element:

```
p1.first = 55;
p1.second = 'M';
```

If I want to print:

```
std::cout << p1.first;
std::cout << p1.second;
```

Initialization

There are different ways to initialize a pair:

1. `std::pair<data_type_1, data_type_2> p1(value_1, value_2);`

This is to assign value when defining a new pair.

Example: `std::pair<int, char> p1(55, 'M');`

2. `std::pair p2(p1);`

Use another pair to initialize the newly defined one.

3. `p1 = std::make_pair(value_1, value_2)`

The `make_pair()` function provides a simple way to assign value to a pre-declared pair (so you don't have to write separate assignment for first and second element in that pair).

To use make_pair, you have to declare the pair first. Then call the `make_pair()` function as shown above and provide values. Pay attention that value provided in the parenthese should correspond to the value type of the pre-declared pair. If not the same, an implicit conversion will be performed. For example:

```
std::pair<int, int> p1;
p1 = std::make_pair(55.5, 'A');
std::cout << p1.first << ", " << p1.second << '\n';
```

Result: `55, 65`

If implicit conversion is not possible, code will not compile and an error will occur, for example:

```
std::pair<int, char> p1;
p1 = std::make_pair(55.5, "ABC");
//this is actually: std::pair<int, char> = std::pair<int, const char *>
```

Operators

Equal (=)

Assigns new object for a pair object. Syntax:

```
pair& operator=(const pair& rhs_pair);
```

This assigns pair on the right hand side (rhs_pair) to the pair on the left hand side. The first element on left hand side is assigned the value of the first element of rhs_pair, and the second element is assigned the value of the second element of rhs_pair.

When performing the assignment, the implicit conversion will still occur if types are different, for example:

```
#include <iostream>
#include <utility>

// function template for printing out the content of a pair
template <class T>
void ShowPairValue(T input_pair) {
  std::cout << input_pair.first << ' \n' ;
  std::cout << input_pair.second << ' \n' ;
}

int main() {

  std::pair<int, char> p1;
  std::pair<char, int> p2;

  p1 = std::make_pair(65, ' B' );
  p2 = p1;
  ShowPairValue(p2);

  return 0;
}
```

Result:

```
A
66
```

Comparison (==)

For given two pairs (p1 and p2), comparison operator compares the first value and second value of those two pairs.

```
if (p1.first == p2.first) && (p1.second == p2.second)
  return true;
else
return false;
```

Pay attention that only pair with the same kind can be compared by = =.

Not equal (!=)

For given two pairs (p1 and p2), compare their first elements. If the first element is equal, then compare the second elements.

```
if (p1.first != p2.first)
  return true;
else
  return (p1.second != p2.second);
```

Logical (>=, <=, >, <)

For given two pairs (p1 and p2), compare the value of their first element. If the first elements are the same, then compare the second element.

Swap

This function swaps the contents of one pair object with the contents of another pair object. The pairs must be of same type (implicit conversion is not available because the function is actually swapping reference, not value! (by my understanding))

You have two ways to call the swap function. For given two pairs of the same type p1 and p2:

1. `p1.swap(p2)` or `p2.swap(p1)`

2. `swap(p1, p2)`

PS: no std:: required.

Chapter 8. Searching and Sorting Arrays

Two Basic Search Algorithms

Search algorithms are used for finding specific items in your data (structured data). In this chapter, two basic search algorithms will be introduced: 1) linear search; 2) binary search.

Linear Search

Linear search is also called sequential search. It uses a loop to sequentially step through an array, starting with the first element, ends at the final element. If it finds the correct value, it will report or record down the position of this element. If no match is found, it will unsuccessfully search to the end of the array.

Binary Search

The binary search is more efficient than linear search. The only requirement is that the values in the array be sorted in order (i.e., from lowest to highest, or from highest to lowest).

Binary search starts at the middle of an array. Suppose we have an array that is in ascending order. We check if the middle value is equal with the searching value. If the middle is not the value we are searching for, check if its greater or smaller than the value.

If its greater, we can cut the rest half of the array because all of them should be greater than the value. We now focus on the first half of the array. We check if the middle of the first half is greater or smaller than the value, then update the search region.

Same idea for the situation that the middle value is less than the search value.

Two Basic Sorting Algorithms

Sorting algorithms can be used to sort the data in an array in some order. A sorting algorithm can scan through an array and rearranging its contents in some specific order. In this chapter, two basic sorting algorithms will be introduced: 1) bubble sort; 2) selection sort.

The Bubble Sort

The bubble sort is an easy way to arrange data in ascending or descending order. The bubble sort starts by comparing the first two elements in the array.

Let's consider to sort the array in an ascending order. If element 0 is greater than element 1, they will be exchanged. Then compare element 1 and element 2, etc. After one run through, the array is not necessarily in the right order. A bubble sorting is complete only if you run through the array and no exchanges are made. You can use a flag to record whether there is an exchange occurred during the run through.

The Selection Sort

Selection sorting algorithm is another method to sort an array.

Considering sort an array in ascending order. The array will be scaned first and the smallest value will be determined. Then this value is exchanged with element 0 (they exchange their position). The next scan starts at element 1 (because element 0 already contains the smallest value). This process continues until the starting element is the final elements).

Chapter 9. Pointers

Address of a Variable

Every variable is allocated a section of memory large enough to hold a value of the variables data type. Each byte of memory has a unique address. A variable's address is the address of the first byte allocated to that variable.

The address operator (&) returns the memory address of a variable. When the address operator is placed in front of a variable name, it returns the address of that variable. The address of the variable is displayed in hexadecimal.

Pointers

Pointer, or pointer variable, is designed to hold memory addresses. Just like int variable is used to hold integer data, and string variable is used to hold string data. With pointer variable you can indirectly manipulate data stored in other variables.

Definition and Initialization of Pointers

The definition of a pointer variable is like follows:

```
int *ptr;
```

The asterisk in front of the variable name indicates that ptr is a pointer variable. The int data type indicates that ptr can be used to hold the address of an integer variable. the definition statement above would read "ptr is a pointer to an int".

If you want to emphasize the pointer nature of the pointer variable (rather than int), you can write the definition of pointer variable in the following way:

```
int* ptr;
```

You should initialize your pointer variable when you define it, because if you inadvertently use an uninitialized pointer variable, you'll be affecting some unknown location in memory. You can initialize your pointer variable with a special value `nullptr`.

The `nullptr` is a keyword in C++ to represent address 0. When a pointer is set to the address 0, it is referred to as a null pointer. Following example shows how you define a pointer variable initialized with `nullptr`.

```
int* ptr = nullptr;
```

Pointers can be defined in the same statement as other variables of the same type:

```
int num, *ptr_int = &num;
```

In the above example, pointer `ptr_int` is defined and initialized with the memory address of `num`. Pointer can only be initialized with the address of an object that has already been defined.

Use Pointer to Indirectly Access and Modify the Variable it points to

The real benefit of a pointer is that it allows you to indirectly access and modify the variable being pointed to. This is done with the **indirection operator** (*). When the indirection operator is placed in front of a pointer name, it dereferences the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to (just like a referenced variable).

The priority of increment and decrement operator (++ and - -) is higher than the indirection operator. So, when you use these two operators, you should use parenthesis): (*ptr)++, otherwise you are ++ or – – the memory address which is stored in the pointer itself. For example, the code `std::cout << (*ptr)++` will first print out value stored in the variable being pointed by `ptr`, then the address stored in ptr will be modified (++). However, the code `std::cout << ++*ptr` will first modify the value of the variable which being pointed by ptr, then print out the value.

The Relationships Between Arrays and pointers

Array names can be used as constant pointers (which means the address this pointer hold can't be changed), and pointers can be used as array names.

We know that array names without brackets and subscript, represent the starting address of the array. This means the array name is a pointer.

Pointer Arithmetic

Some mathematical operations may be performed on pointers.

Allowable arithmetic operations are:

> ++ - -: used to increment or decrement a pointer value;

> add or subtract an integer from a pointer variable (+ -, +=, -=);

> a pointer may be subtracted from another pointer; (how??)

Comparing Pointers

If one address comes before another address in memory, the first address is considered "less than" the second. C++'s relational operators can be used to compare pointer values (> < == != >= <=). The capability of comparing address gives you another way to be sure a pointer does not go beyond the boundaries of an array.

Pay attention that comparing two pointers is not the same as comparing the values of the two pointers point to.

Pointers as Function Parameters

Concept

A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does. However, for a reference parameter, it can do the deferencing and indirection automatically. If you use a pointer as a function parameter, you have to use indirection operator to access the content of the memory address that a pointer points to.

When you define a function, you should declare it uses a pointer parameter:

```
void double_value(int *num)
{
  *num *= 2;
}
```

The "num" is a pointer. When you add a indirection operator (*) in front of the pointer, it is dereferenced, and will access the data stored in the memory address num points to.

When the function is called, a memory address should be used as the parameter (datatype in this memory address should be the same as the type of the parameter defined in function declaration). You can use address operator or pointer to pass the memory address to the function:

```
  int *ptr_i = &a;
  double_value(&a);
  double_value(ptr_i);
```

Look at following term:

```
  sum += *ptr++;
```

Interpretation: the * operator will first dereference ptr, then *ptr will be added to sum, and then the ++ operator will increment the address in ptr.

Additional notes

It should be noted that although passing a pointer enables you to access the original variable which is pointed by the pointer, the pointer itself is passed by value, which means it is only a copy. If you want to modify the pointer itself in the function, you have to declare a reference pointer:

```
void double_value(int*& num)
```

`int*` tells the compiler that the parameter is a pointer that pointing to int type entity.

`& num` tells the compiler that, the parameter is a reference parameter, which means any change done to the parameter in the function body will affect the original entity.

Pointers to Constants

Sometimes it is necessary to pass the address of a const item into a pointer. When this is the case, the pointer must be defined as a pointer to const item.

However, a pointer to constant can also store memory address of a nonconstant item. When this is the case, the value stored in memory address can not be modified by this pointer (although it can still be modified by other pointers, because essentially it is not constant).

The complete applicability table is as follows:

	Pointer to Constant (read value)	Pointer to Constant (change value)	Pointer to non-constant (read value)	Pointer to non-constant (change value)
Function declared having a pointer to constant parameter	Yes	No	Yes	No
Function declared having a normal pointer parameter	No	No	Yes	Yes
Memory address of a constant item	Yes	No	No	No
Memory address of a non-constant item	Yes	No	Yes	Yes

Assigning:

1) You can assign a [pointer] to a [pointer to constant].

```
const char *ptr1;
char str[] = "12345";
ptr1 = str;
```

2) You can’t assign a [pointer to constant] to a [pointer]

```
const int num = 5;
const int *num_ptr = &num;
int *ptr = nullptr;
ptr = num_ptr; //this is invalid
```

(because normal pointer can be used to change the value it points to. This is not allowed if a pointer to constant is pointing to a constant)

Constant Pointers

Unlike pointer to constant, constant pointers' stored memory address will not change (just like the name of the array).

Following is an example of constant pointer definition:

```
int * const ptr3 = num1;
```

The `const` key word is used for defining a constant pointer. Pay attention that "const" is placed between pointer name and the asterisk. A constant pointer has to be initialized when being defined, just as the constant term.

Although the memory address stored in a constant pointer can't be changed, the content stored in the memory address can be changed as long as its not constant. Thus, the following code is legal:

```
  int num[3] = { 1,2,3 };
  int * const ptr = num;
  *num += 1;
```

To define a constant pointer to constant, here is how:

```
 const int * const ptr3 = num1;
```

Dynamic Memory Allocation

Variables may be created and destroyed while a program is running.

To dynamically allocate memory means that a program asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type.

For example, if the program needs to create an integer variable, it will make request to the computer that it allocate enough bytes to store an int. The computer will find and set aside a chunk of unused memory large enough for the variable, then it will give the program the starting address of the chunk of memory. The program needs a pointer to hold the address of this newly allocated memory (by the starting address given by the computer).

C++ program requests dynamically allocated memory through the new operator:

```
  int *ptr_i = nullptr; //define a pointer, initialize with address 0
  ptr_i = new int; //request a new chunk of memory to store an integer data, the address
is stored in ptr_i
```

After the second statement executes, ptr_i will contain the address of the newly allocated memory. A value can be stored in this newly variable by dereferencing the pointer:

```
  *ptr_i = 25; //the newly created chunk of memory (and variable) can only be accessed
by the pointer
```

Or you can directly assign the integer value you want to store in this newly allocated memory:

```
ptr_i = new int {5};
```

A more practical use of the new operator is to dynamically create an array:

```
int *ptr_i = nullptr;
ptr_i = new int[100]; //request new memory space for 100 integer
```

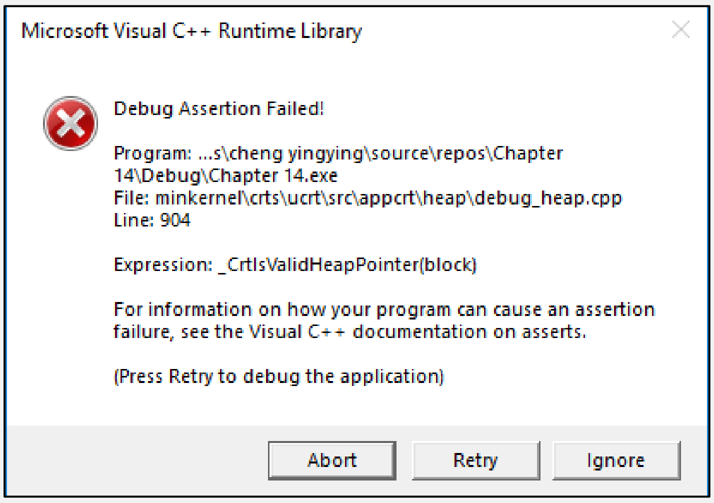
You can also put a variable inside the square bracket, so you can specify the size of the array while the program is running.

When the program is finished using a dynamically allocated chunk of memory, it should release it for future use. The delete operator is used to free memory that was allocated with new. Here is an example:

```
delete ptr_i; //free memory of a single variable
delete [] ptr_i; //free memory of an array
```

After freeing the dynamically allocated memory, you should let pointer points to nullptr, to make the program safer.

Be careful, do not delete a memory space twice. If so, you may get error (assertion failed):



```
Microsoft Visual C++ Runtime Library

Debug Assertion Failed!

Program: ...s\cheng yingying\source\repos\Chapter
14\Debug\Chapter 14.exe
File: minkernel\crts\ucrt\src\appcrt\heap\debug_heap.cpp
Line: 904

Expression: _CrtIsValidHeapPointer(block)

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)
```

Memory Leak

Consider the following example:

```
void grab_memory()
{
  int size = 100;
  int *ptr = new int [size]; //request to allocate space for 100 int elements
}
```

This function request a memory space enough to store 100 int elements. Then the function exited. The allocated memory, however, remains un-deleted. However, The pointer defined in the function is “destroyed” when the function exits, so there is no way to access or delete the allocated memory, it will sit unused as long as the program is running. The right way to do is to free the allocated memory before exiting the program.

```
void grab_memory(int *ptr2)
{
  int size = 100;
  int *ptr = new int [size]; //request to allocate space for 100 int elements

  delete[] ptr;
}
```

Only use pointers with delete that were previously used with new, not pointers containing the memory address of another pre-defined item.

Returning Pointers from Functions

Functions can return pointers, but you must be sure the item the pointer references still exits.

Like any other data type, functions may return pointers, as long as you define the return type of the function as pointer. Following is an example:

```
char *find_null(char *str)
{
  while (*str != ' \0' )
    str++;
  return str;
}
```

The return type of this function is a pointer to char. The parameter of this function is memory address of a char type data.

It should be noticed that, do not return pointers or memory types that no longer containing data. You should return a pointer from a function only if it is:

1) A pointer to an item that was passed into the function as an argument

2) A pointer to a dynamically allocated chunk of memory

For example, in the following code, address of a local array has been returned. When the function exit, the array variable will be destroyed and the memory position of this array will no longer contain data. Thus the pointer points to nothing meaningful.

```
int *AllocateArray() {
  int array[100];
  return array;
}
```

Smart Pointers

Smart pointers are objects work like pointers, but have the ability to automatically delete dynamically allocated memory that is no longer being used. This helps to prevent memory leaks from occurring.

There are three types of smart pointer:

1) unique_ptr

2) shared_ptr

3) weak_ptr

To use any of these smart pointers, you have to include the <memory> header file by:

#include <memory>

This book introduces unique_ptr.

unique_ptr

The definition of an unique pointer is as follows:

```
  std::unique_ptr<int> ptr(new int);
```

`std::unique_ptr<int>` determines the pointer is an unique pointer that can point to `int`. `ptr` is the name of the smart pointer. The expression `new int` that appears inside the parenthese allocates a chunk of memory to hold an int. The address of the chunk of the memory will be assigned to the smart pointer.

Use a unique_ptr to dynamically allocate an array of integers.

```
 std::unique_ptr<int[]> num(new int[100]);
```

It should be noticed that, `int[]` is the declared type.

Chapter 10. Characters, C-Strings, and More About the `string` Class

Character Testing (isalpha, isprint etc)

The C++ library provides several functions for testing characters. To use these functions you must include the <cctype> header file. (Naming is not in std namespace, don't have to add std:: in front of the function name.

Test function table:

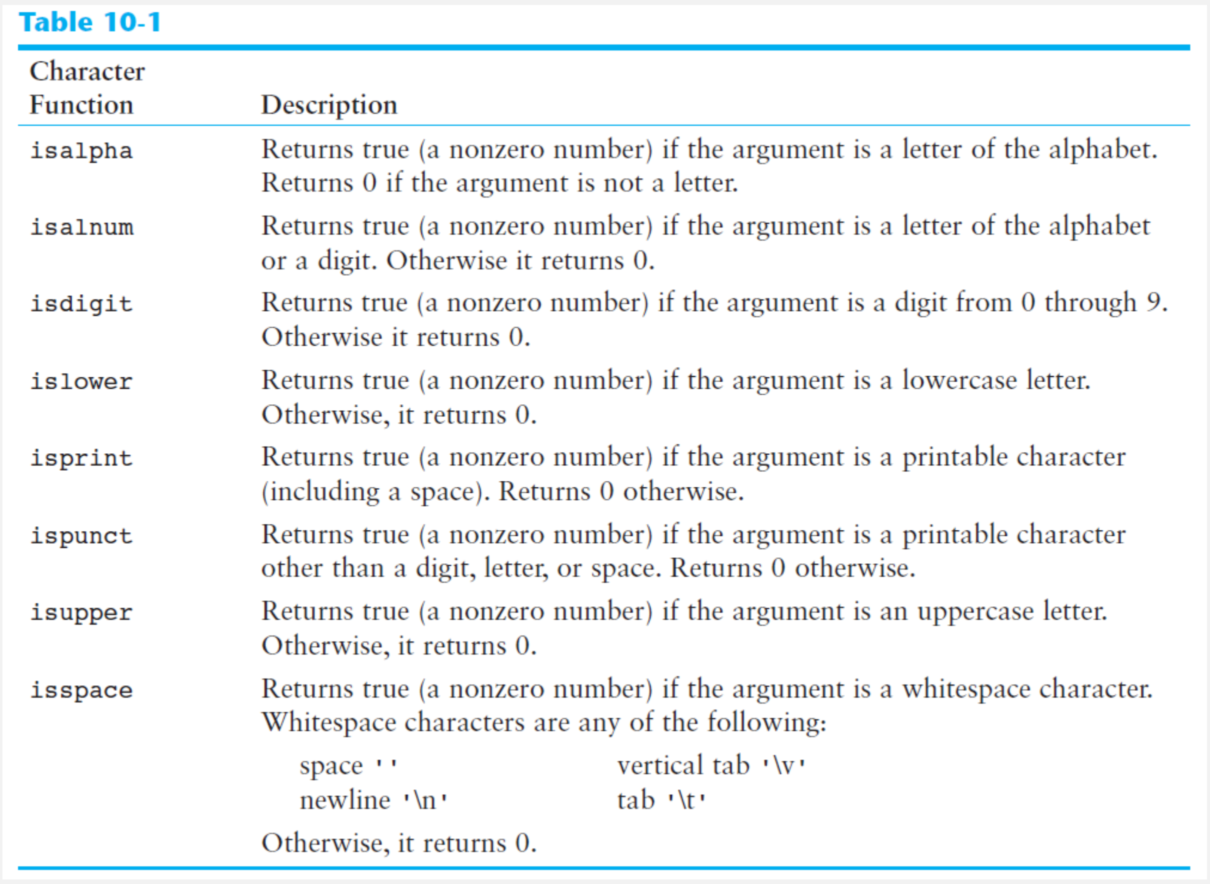


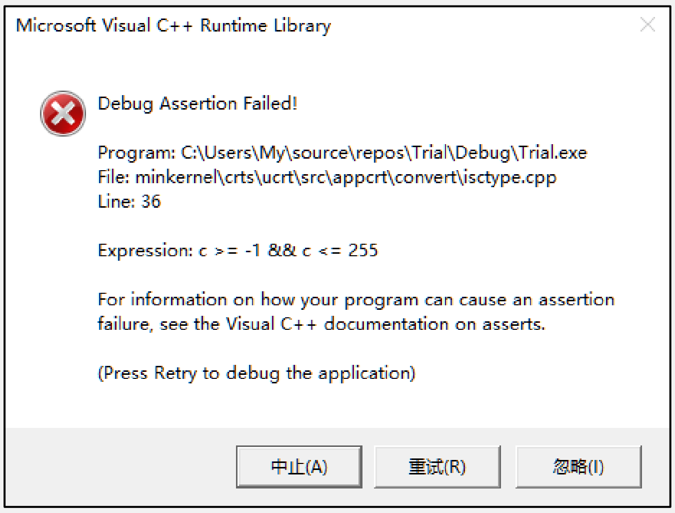
Table 10-1

Character Function	Description
`isalpha`	Returns true (a nonzero number) if the argument is a letter of the alphabet. Returns 0 if the argument is not a letter.
`isalnum`	Returns true (a nonzero number) if the argument is a letter of the alphabet or a digit. Otherwise it returns 0.
`isdigit`	Returns true (a nonzero number) if the argument is a digit from 0 through 9. Otherwise it returns 0.
`islower`	Returns true (a nonzero number) if the argument is a lowercase letter. Otherwise, it returns 0.
`isprint`	Returns true (a nonzero number) if the argument is a printable character (including a space). Returns 0 otherwise.
`ispunct`	Returns true (a nonzero number) if the argument is a printable character other than a digit, letter, or space. Returns 0 otherwise.
`isupper`	Returns true (a nonzero number) if the argument is an uppercase letter. Otherwise, it returns 0.
`isspace`	Returns true (a nonzero number) if the argument is a whitespace character. Whitespace characters are any of the following: space `' '`, vertical tab `'\v'`, newline `'\n'`, tab `'\t'`. Otherwise, it returns 0.

Notice that the range of arguments in these function is -1 ~ 255 (the integer value). If you inputed other values, an error will occur and terminate your program. For example:

```
int num = 256;
std::cout << static_cast<char>(num) << " : " << isalnum(num) << "\n";
return 0;
```

This program will terminate and following error message will show up:



```
Microsoft Visual C++ Runtime Library

Debug Assertion Failed!

Program: C:\Users\My\source\repos\Trial\Debug\Trial.exe
File: minkernel\crts\ucrt\src\appcrt\convert\isctype.cpp
Line: 36

Expression: c >= -1 && c <= 255

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)
```

Character Case Conversion (toupper() and tolower())

The C++ library offers functions for converting a character to upper or lowercase. The functions are prototypes in the header file <cctype>.

```
  toupper(letter); //will output the upper case equivalent of letter
         //letter won′ t be changed.
  tolower(letter); //will output the lower case equivalent of letter
         //letter won′ t be changed.
```

The return type is integer.

C-strings

In C++, a C-string is a sequence of characters stored in consecutive memory locations, terminated by a null character (pay attention, strings and C-string is not the same thing). In C++ language, there are two primary ways that strings are stored in memory: as string objects (you need to include <string> header file to be able to use the string objects), or as C-strings.

In C++, all string literals are stored in memory as C-strings (string literals are things like the following).

```
 " Bailey "
```

The last slot contains the null terminator, which marks the end of the C-string. Without it, the length of a C-string can't be determined.

If you want to store a C-string in memory, you have to define a char array that is large enough to hold the string, plus one extra element for the null character, for example:

```
 const int SIZE = 20;
 char C_string[SIZE] = " Good Day " ;
```

You can initialize the a char array with a string literal. The null character is automatically added as the last element. If the length of your string literal is longer than the character array, the program will not compile.

C-string input can be performed by the cin object. For example, the following code allows the user to enter a string (with no whitespace characters) into the name array:

```
 const int SIZE = 20;
 char name[SIZE];
 std::cin >> name;
```

The array name with no brackets and no subscript is converted into the beginning address of the array. In the above code, cin can't check the length of inputted data. If the length of input is longer than the size of the array (actually is size-1, the last element should be null), the console will pop up an error message.

Pay attention, strings inputed in this way can't contain space!

cin.getline Function

"getline" can also be a member function of cin. The use of .getline member function is as follows:

```
#include <iostream>

int main()
{
  char letters[10];
  std::cin.getline(letters, 6);
  for (int i = 0; i < 5; i++)
    std::cout << letters[i] << std::endl;
  return 0;
}
```

There are two parameters: the first one is the name of a char array (the constant pointer), this is where your input characters will be saved. The second parameter is an int N. The maximum characters to be stored in the array is (N-1), because it includes the null character at the end of the string. If you inputted characters with a length longer than (N-1), it will cut the rest of them and store the characters before (N-1). This is how you control the length of string you input.

When you call this function, you input a string, and each character in your string will be saved separetly in the character array.

C-strings: string editing functions

In order to access the various library functions for working with C-strings, the <cstring> header file must be included.

strlen

The strlen function accepts a pointer to a C-string as its argument. It returns the length of the string, which is the number of characters up to (not including the null terminator):

```
char num[] = " 12345 " ;
strlen(num) = 5; //you have to define an array with 5+1 characters to actually hold this array
```

It works like this. It will receive a memory address, then it will begin to count the number of characters it encounter as it moves forward, until it encounters a null character '\0'. Then it will return an integer, which is the number of characters it counts before the null character.

Its working mechanism determins that the number it returns is not the size of the array, but the size of string it stores (it stops counting once encountered '\0'.)

PS: My compiler can use this function when <cstring> or <iostream> header file is included.

strcat

The strcat function accepts two pointers to C-strings as its arguments. The function concatenants or appends one string to another.

Syntax:

```
strcat(destination, source)
```

Letters in source will be copied to the end of the destination. There is no return value of this function, because the string @ destination has already been affected.

strcat can't detect if there is a overflow issue (i.e. the length of destination + source is larger than the defined length of destination). Another function, strncat, can solve this issue. The call of strncat function is as follows:

```
strcat(destination, source, maxi_character)
```

where the third argument is the maximum number of characters transferred to the end of destination (including the null terminator). The maximum number is determined by:

```
sizeof(destination) - (strlen(destination) + 1);
```

+1 is for the null character.

strcpy

The strcpy function is used to copy one string to another. During this process, the tartget string (first argument) is destroyed.

Syntax:

```
strcpy(destination, source)
```

This function also has a safer version:

```
strncpy(destination, source, maxi_character)
```

strstr

The strstr function searches for a string inside of a string. You have to include <cstring> header file to use this function. The prototype for strstr function is:

```
char *strstr(char *string, char *keyword);
```

As can be seen from the prototype, the strstr function has two arguments, the first one is where to search (the beginning memory address of the string that is to be searched), the second argument is the memory address of the keyword to be searched. The function will go over the string being searched and try to find out if there is a section equal to the keyword. If it finds such section, it will return the memory address of the beginning of this section. If it fails to find any match, it will return an address pointing to nullptr.

Calling strstr function:

```
char *str_ptr = nullptr;

str_ptr = strstr(string, keyword);
```

In the above example, str_ptr is defined to hold the return value of strstr function.

strcmp

C-strings are stored in char arrays, so you can't use the relational operator to compare two C-strings. The strcmp function can do this job. The prototype of strcmp function is as follows:

```
int strcmp(char *string1, char *string2);
```

It has two arguments, the first one is the address for string1, the second one is the address for string2. The return type is integer. Basically, what it does is to compare string1 and string2. It will return different values based on the result:

1) return 0: string1 is equal to string2

2) return 1: string1 is after string2

3) return -1: string1 is before string2

You can use stricmp function to let the function be case in-sensitive

C-strings: numeric conversion functions

The following functions require the <cstdlib> header file. (On my computer, I can use these functions without including the <cstdlib> header file)

atoi

This function accepts a C-string as an argument, and returns the integer in the string (as an integer value). It will return the integer until there is another character appears. For example:

```
std::cout << atoi("43.4");
```

The result will be: 43.

Pay attention that, atoi function does not work with single character or C++ string. If you want to use with C++ string, you can use string.c_str() to convert C++ string to a C-string.

atol

This function accepts a C-string as an argument, and returns the long integer in the string (as a long integer value). It will return the long integer until there is another character appears. (Different return data type compared with atoi.

atof
This function accepts a C-string as an argument, and returns the float number in the string (as a double value). It will return the double number until there is another character appears. For example:

std::cout << atoi(" 43.454as ");

The result will be: 43.454

to_string
The to_string function can be used to convert numeric numbers to a string object (pay attention, the return type is not a C-string, it is a string object, defined by std::string statement, you must include <string> head file to use the string object as well as the to_string function.) Incorrect example:

```
  char *num_C_string = nullptr;
  num_C_string = std::to_string(200); //this is wrong! char * does not equal to string
object!
Correct example:

  std::string num_string_object;
  num_string_object = std::to_string(200);
```

Comparisons between C++ string and C string

C++ string objects can be compared to C-strings with relational operators. For example:

```
#include <iostream>
#include <string>

int main()
{
  std::string str1 =  " abcde " ;
  char str2[] =  " abcdf " ;

  if (str1 > str2)
    std::cout <<  " str1 is greater than str2\n " ;
  else
    std::cout <<  " str1 is smaller than str2\n " ;

  return 0;
}
```

C++ string definition example

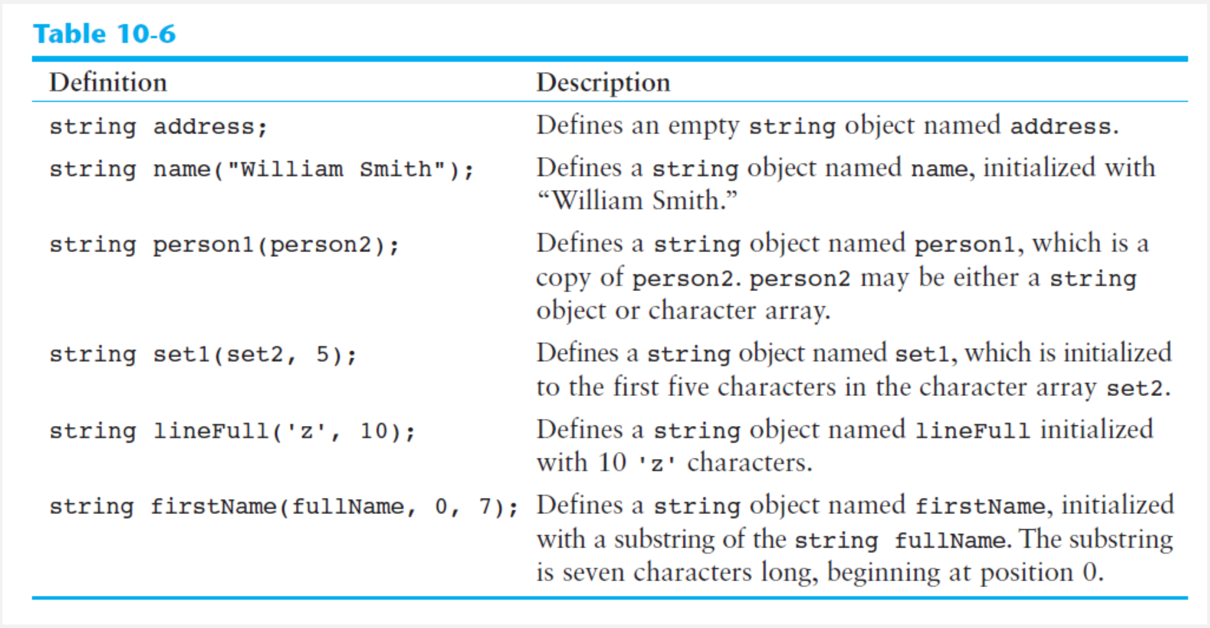


Table 10-6

Definition	Description
`string address;`	Defines an empty `string` object named `address`.
`string name("William Smith");`	Defines a `string` object named `name`, initialized with "William Smith."
`string person1(person2);`	Defines a `string` object named `person1`, which is a copy of `person2`. `person2` may be either a `string` object or character array.
`string set1(set2, 5);`	Defines a `string` object named `set1`, which is initialized to the first five characters in the character array `set2`.
`string lineFull('z', 10);`	Defines a `string` object named `lineFull` initialized with 10 `'z'` characters.
`string firstName(fullName, 0, 7);`	Defines a `string` object named `firstName`, initialized with a substring of the `string fullName`. The substring is seven characters long, beginning at position 0.

Note: in my compiler, the syntax on the fourth line is not working. The compiler will think you are using the last syntax: `string str1(str0, 0, 7)`, if you input `string str1(str0, 5)`, the compiler will create a string that has the content in str0 after `str0[5]` (all characters after the element 5, **NOT** the first five characters!!)

C++ string member function

There are many C++ string functions available to use. You need to include the <string> header file to use these member functions.

string.append()

The append member function will append other strings to the end of the target string. It has several overload functions.

string.append(n, ‘z’)

Appends n copies of ‘z’ to string.

string.append(str)

Appends str to string, str can be C++ string or C-string.

string.append(str, n)

Appends the first n characters in str to string. `//seems not working, if only one integer is provided, it means append the substring in str to string, start from the position this integer represents, look at following example.`

string.append(str, x)

Append the substring in str to string, start from the position x (position x is in str, not the target string) for example:

```
std::string str = " Patience and diligence like faith remove mountains " ;
std::string str2;
str2.append(str, 2);
```

```
std::cout << str2 << "\n";
```

Result: str2 = "tience and diligence like faith remove mountains"

string.append(str, x, n)

Appends n number of characters from str to string, starts at position x. If the destination (string) is too small, the function will copy as many characters as possible. Example:

```
std::string str = "Patience and diligence like faith remove mountains";
std::string str2;
str2.append(str, 2);
std::cout << str2 << "\n";
```

Result: str2 = "`tie`".

string.assign()

The assign member function will assign values to the string. The content stored previously will be cleared.

string.assign(n, 'z')

Assigns n copies of 'z' to string.

string.assign(str)

Assigns str to string, str can be C++ string or C-string.

string.assign(str, n)

Assigns the first n characters in str to string.

(The first n characters, or begin from nth character?)

string.assign(str, x, n)

Assigns n number of characters from str to string, starts at position x. If the destination (string) is too small, the function will copy as many characters as possible.

(isn't the string object can allocate memory dynamically?)

string.back()

The .back() member function will return the last character in the string.

string.begin()

Returns an iterator pointing to the first character in the string.

string.end()

Returns an iterator pointing to the last character in the string.

string.c_str()

Converts the contents of string to a C-string, and returns a pointer to the C-string. In my compiler, the returned pointer is a `const char *` type, so the C-string it transfers is a constant string. You have to define a pointer to constant to hold this returned pointer.

This is because C_string is an array of characters. The name of the array should be a constant pointer (the memory address stored in the pointer don't change, the pointer itself is constant pointer).

string.capacity()

Returns the size of the storage allocated for the string (number of characters can hold, not including the null character?). When defining a string, the system will allocate certain storage spaces to the string. The minimum storage is 2^4-1 (minus one is for the extra null character?), if the number of characters exceed the storage limit (for example, if you append other strings to it) the limit will rise automatically, the increased size is 2^4 at a time, to the smallest number of (n*2^4-1) that is big enough to hold the new string.

(in xcode, the sequence is: 22 – 31 – 47 – 63)

string.clear()

Clears the string by deleting all the characters stored in it. This is similar with vector.clear(). Although all the characters are deleted, the memory storage allocated to this string remains the same. Thus string.length() or string.size() will become 0, while string.capacity() remains the same.

string.compare(str2)

Performs a comparison like the strcmp function, with same return values (1: string is after str2; -1: string is before str2; 0: string is equal with str2). str2 can be C++ strings and C-string. (the comparison of C++ string and C-string can be done by relational operator).

string.compare(x, n, str2)

Compares string and str2, starting at position x, and continuing for n characters. The return value is the same (from x to x+n-1, 1: string is after str2; -1: string is before str2; 0: string is equal with str2).

string.copy(str2, x, n)

Copies the <u>character array (not std::string object)</u> str2 to string, beginning at x, for n characters. (My compiler is not working with this function). On the other hand, you can use '=' operator to copy C-string or C++ string to the target string.

string.empty()

Returns true (1) if the string is empty. Pay attention that this function is not to empty the string.

string.erase(x, n)

Erase n characters from string, beginning at position x. For example:

```
  std::string num =  " 12345 " ;
  num.erase(1, 2);
  std::cout << num;
result is 145.
```

string.find()

This function can find the keyword in C++ string. It has several overload functions. The return type of string.find() is unsigned int.

string.find(str2)

string.find(*"keyword"*)

string.find('c')

string.find("c")

Search the keyword in string from beginning and return an integer, which is the subscript of where the keyword begins. If no matches found, the function will return -1, or 4294967295, which is the largest unsigned int (2^32-1).

string.find(str2, x)

string.find(*"keyword"*, x)

string.find('c', x)

Search the keyword contained in str2 in string. This search begins at position x, and will return the subscript of where str2 starts (after the initial position x). If no matches found, the function will return -1, or 4294967295, which is the largest unsigned int (2^32-1).

string.front()

Returns the first character in the string. (Why not use string[0]?)

string.insert()

Inserts other strings or characters into string at a certain position.

string.insert(x, n, 'z')

Inserts 'z' n times into string at position x. If you want to insert character, use this! Remember the n argument.

string.insert(x, *"inserted string"*)

Inserts *"inserted string"* into string at position x. If you want to insert string, use this!

string.replace(x, n, str2)

Replaces the n characters in string beginning at position x with the characters in str2. n is the number that characters in destination string be replaced, not the number of characters in str2 being inserted (actually, all characters in str2 will be inserted into the destination). If n = = 0, no character in string will be replaced, but the str2 will still be inserted at position x. For example:

```
full_name.assign( " abcdefghijklmn " );
std::cout << full_name <<  " \n " ;
full_name.replace(0, 3,  " xxx " );
std::cout << full_name << std::endl;
```

after this replace, full name will be: `xxxdefghijklmn`. If we do: `full_name.replace(0, 0,  " xxx " );`. The string will be: `xxxabcdefghijklmn`. (because no character in the destination string will be replaced, and the insert starts at position 0).

string.resize(n, 'z')

Changes the **size** of the allocation in string to n. If n is less than the current size of the string, the string is truncated to n characters (however, the memory size will not change, i.e. string.capacity() will not change). If n is greater, the string is expanded and 'z' is appended at the end enough times to fill the new slots. During the expansion, the memory storage allocated to string can be increased too, at a stepsize of 2^4, to the smallest space that can hold the increase.

string.size()

Return the size of string. Not including the null terminator.

string.substr(x, n)

Returns a copy of a substring. The substring is n characters long and begins at position x of the original string. If n is larger than the character number of the original string, only whole string will be copied.

string.swap(str1)

Swap the contents of string with str1.

C++ stringstream class

The std::stringstream class is a stream class to operate on strings. Objects of this class use a string buffer that contains a sequence of characters. This sequence of characters can be accessed directly as a string object, using member str.

Characters can be inserted and/or extracted from the stream using any operation allowed on both input and output streams.

To use this class, you have to `#include <sstream>`

Contemplation

After you `#include <sstream>`, let's declare an object and call member str to "add" a string to it:

```
#include <sstream>
int main() {
  std::stringstream s1;
  s1.str( " 123 45   6 " );
}
```

Now, you can use s1 as you would use std::cin, for example, you want to print out the content separated by space:

```
  std::string line;
  while (getline(s1, line, '  ' )) {
    if (!line.empty())
      std::cout << line << ' \n' ;
  }
```

Output:

```
123
45
6
```

You can use it as an input stream, just as you would use the std::cin object.

Chapter 11. Structured Data

Abtract Data Types

Abstract data types (ADTs) are data types created by the programmer. ADTs have their own range (or domain) of data and their own sets of operations that may be performed on them (programmer can design his or her own specialized operatirons). ADTs are composed of one or more primitive data types.

Structures

Structure Declaration

In C++, you can group several variables together into a single item known as a structure. Below is how you declare a structure (package of several variables):

```
struct tag
{
  variable1 declaraction;
  variable2 declaraction;
  variable3 declaraction;
  variable4 declaraction;
  //more declarations may follow
};
```

The tag is the name of the structure, it can be used like a data type name (int, double etc). The variable declarations that appear inside the braces declare members of the structure. Following is the example of a structure declaration that holds the area calculation data.

```
struct Area_calc
{
  double length;
  double width;
  double radius;
  const double PI = 3.1415926;
  double area;
};
```

This declaration declares a structure named area_calc. The structure has five members: length, width, radius, PI and area. **Notice that a semicolon is required at the end of the declaration.** The first letter of the structure name is upper case, this can help us differentiates the structure name from variable name.

During the declaration, there is no variable be defined. It simply tells the compiler what a Area_calc structure is made of (looks like). The member variables are created in memory only when a structure variable is defined. Thus, you can't declare things like: area = length*width in the structure declaration section.

In essence, it creates a new data type named Area_calc. You can define variables of this type with simple definition statements, just the same as any other data type. For example, the following statement defines a variable named circle:

Area_calc circle_area;

The data type of circle_area is the Area_calc structure. It is structure variable, which is actually made up of other variables known as **members**. Because circle_area is an Area_calc structure, it contains the following members:

```
  double length;
  double width;
  double radius;
  const double PI = 3.1415926;
  double area;
```

In review, there are two steps to implementing structures in a program

1) Create the structure declaration. This process will establish the tag (or name) of the structure and a list of items that are members.

2) Define variables (or instances) of the structure data type and use them in the program to hold data.

Accessing Structure Members

The dot operator (.) allows you to access structure members in a program. With the dot operator, you can use member variables like regular variables.

You cannot compare two structure variables directly using relational operator, you have to compare them term by term (member by member).

Initializing a Structure Variable

When defining a structure variable, you can initialize its members with starting values. The syntax is similar with the initialization of an array (use = and {}), example:

```
struct City_info
{
  std::string city_name;
  std::string state;
  long population;
  int distance;
};

City_info location = { " Asheville " , " NC " , 50000, 28 };
```

This statement defines the City_info variable named location. The first value in the initialization list is assigned to the first declared member, the second value in the initialization list is assigned to the second declared member, and so on.

You do not have to provide initializers for all the members of a structure variable, however, you can not skip members in a structure. If you leave a structure member uninitialized, you must leave all the members that **follow** it uninitialized as well.

Notice: even if you defined a member of the structure as “constant variable”, it is not constant really. You can change the value it holds by initialization with another value. For example:

```
#include <iostream>

struct Area_rectangle {
  double length;
  const double PI = 3.1415;
  double width;
  double area = length * width;
};

int main() {
  Area_rectangle rect1 = {2, 3};
  std::cout << "PI is: " << rect1.PI << "\n";

  return 0;
}
```

The result is 3, rather than 3.1415.

PS: You can also assign values to the members of the structure variable using same syntax, for example:

```
struct Employee
{
  std::string name;
  int emp_id;
  double pay_rate;
  double hours;
  double gross_pay;
};
  Employee worker1;
  worker1 = { "Name", 12345, 13323 };
```

Defining and Initializing a Structure Array

You can define an array of structured variable.

To initialize a structure array, just use **nested {}**. For example:

```
struct movie
```

```
{
  string name;
  string director;
  string producer;
  string year;
};
movie favorite_movie[5] = { { " Cloud Atlas " , " asd " , " sdw " , " 2012 " }, { " The Last
Samuri " , " sww " , " okin " , " 2007 " } };
```

You still can't skip initialization of members or structure variables (element in the structure array).

Nested Structures

It's possible for a structure variable to be a member of another structure variable. For example:

```
struct Pay_info
{
  double pay_rate;
  double hours;
  double gross_pay;
};

struct Employee
{
  std::string name;
  int emp_id;
  Pay_info pay;
};
```

Just remember you have to declare a structure before you can use it. You access the inner member variable by using the dot operator several times.

Structures as Function Arguments

Structure variables may be passed as arguments to functions. This is the same as other variables. You can either passing individual members of a structure variable to functions, or passing the entire structure variable to the function.

Normally, structures are passed by value into a function, but you can add the & in front of the argument, so it can be passed by reference.

Before writing function prototype for functions with structure variable argument, you must declare the structure first, so the compiler knows what the structure looks like.

Constant Reference Parameters

When a structure variable is passed by value, a copy of the structure variable is created. However, sometimes the structure variable is very big. The copy will decrease the performance of the program. When the structure variable is passed by reference, however, no copy is created, a reference that points to the original argument is passed instead. This will increase the performance. At the same time, when passing by reference, the original argument may be altered or corrupted.

One way of solving this is to pass the argument as a constant reference. For example, if Product is a structure, you can define a function like:

```
void show_info(const Product &p);
```

Returning a Structure from a Function

A function may return a structure variable (you have to define the structure before the function prototype, so the compiler knows what is this function returning). The returned structure variable can be **assigned** to other structure variable with the same type.

By utilizing functions which return structure variable, you can return several result in one pack of structure variable.

Pointers to Structures

Like other variables, the structure variable has its own memory address. You can create pointer variables that points to structures to point to a certain structure variable.

```
Product *ptr = nullptr;
```

This statement defines a pointer points to a `Product` variable. If `part` is a `Product` variable (defined by: `Product part`), we can assign the memory address of `part` to `ptr`:

```
Product *ptr = &parts;
```

After this statement, you can use the ptr to access the data stored in parts: `(*ptr).description`, `(*ptr).part_num`, `(*ptr).cost`. The use of parenthese is because the dot operator has higher precedence than the indirection operator. If you write *ptr.description, the indirection operator tries to dereference ptr.description, not ptr.

C++ has a special operator for dereferencing structure pointers: the structure pointer operator, it consists a hyphen (-) followed by the greater-than symbol (>). Thus:

```
  ptr->description;
  ptr->cost;
  ptr->part_num;
```

Pay attention that there is no space between `ptr` and `->`.

When to use `.`, `->` and `*`
Sometimes structures contain pointers as members. Consider the following structure:

```
struct Grade_info
{
  std::string name;
  double *test_score;
  double average;
};
Grade_info student1;
```

Be clear in mind that the significance of the following notations:

```
  Grade_info *ptr = &student1;
  *student1.test_score; //this is the value test_score points to
  *ptr->test_score; //this is also the value test_score points to
```

Dynamically Allocating a Structure
You can also use a structure pointer and the new operator to dynamically allocate a structure. The same idea when you dynamically allocate an array of structures.

```
  Grade_info *ptr = new Grade_info;
```

Unions

General review
A union is like a structure, except all the members occupy the same memory area. This means that only one member can be used at a time. A union might be used in an application where the program needs to work with two or more values (of different data types), but only needs to use one of the values at a time. Unions conserve memory by storing all their members in the same memory location.

Declaration
Unions are declared just like structures, except the key word union is used instead of struct:

```
union Pay_source
{
  int hours;
  double sales;
};
```

Define a union variable:

```
  Pay_source employee1;
```

The union variable defined here has two members: hours (an int) and sales (a double). The entire variable will only take up as much memory as the largest member (in this case, a double). Thus, both members can't hold values at the same time.

Similarties with structure

Everything else you already know about structures applies to unions, for example:

1) you can define arrays of unions

2) you can define pointers to unions

3) members of the union referenced by the pointer can be accessed with the -> operator.

Anonymous unions

The anonymous unions have no names, but its members do have names. Below is an example of anonymous union declaration.

```
union
{
  int hours;
  double sales;
};
```

This declaration actually creates the member variables in memory (request a memory space large enough to hold the largest member in union). (For the unions with names, you have to define an union variable to create the member variable). Anonymous unions are simple to use because the members may be accessed without the dot operator.

You can think of the anonymous union as a fixed memory space that can hold many different variables with different data type. When you use one of the variables inside the declaration, the memory space is used to store that kind of variable.

If an anonymous union is declared globally, it must be declared static (the word static must appear before the word union). In this way, the content of variable inside the union will not change when being called from different blocks of codes. Otherwise, the union should be defined locally.

```
static union
{
  int hours;
  double sales;
};
```

To put it in a simple way, anonymous unions are a space that defines several variables. Those variables share the same memory space. When you declare an anonymous union, you defined all member variables, and you can use them directly in the following code (just as you've defined these variables, the only difference is these variables share the same memory space).

Enumerated Data Types

An enumerated data type is a programmer defined data type. It consists of values known as enumerators, which represent integer constants (works like tags for different integers).

Declaration of enumerated data type

You can declare the enumerated data type by using the `enum` key word. Following is the general syntax:

```
enum Type_name { one or more enumerators };
```

Data type defined by the `enum` keyword is called enumerated data type. Inside the braces is a list of identifiers. Here is an example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

The above declaration creates an enumerated data type named `Day`. The identifiers `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, which are listed inside the braces, are known as **enumerators**. They represent the values that belong to the Day data type (just like 1, 2, 3 are belong to int data type, or "patience and diligence, like faith, remove mountains." is belong to string data type).

Note that the enumerators are not enclosed in quotation marks, therefore they are not strings. Enumerators must be legal C++ identifiers.

Define variables of enumerated data type

Once you have created an enumerated data type in the program, you can define variables of that type. For example, the following statement defines `work_day` as a variable of the `Day` type:

```
Day work_day;
```

Because work_day is a variable of the Day data type, we may assign any of the enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY` to it. For example, the following statement assigns the value `WEDNESDAY` to the `work_day` variable.

```
work_day = WEDNESDAY;
```

C++ allows you to declare an enumerated data type and define one or more variables of the type in the same statement, for example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY } day1, day2;
```

`day1` and `day2` are defined when declaring `Day`.

Internal representation of enumerator

You can think of enumerators as named integer constants. Internally, the compiler assigns integer values to the enumerators, beginning with 0. The enumerator `MONDAY` is stored in memory as the number 0. `TUESDAY` is stored in memory as the number 1, and so forth. This means **you can use the enumerator as integers, like in mathematical calculations or array subscripts.**

However, you can not assign integer to enumerated variables. You can only use `static_cast` to transfer the integer to the corresponding enumerated data type, for example:

```
work_day = static_cast<Day>(3);
```

After the above statement, work_day will be assigned with `THURSDAY`, same result with following statement:

```
work_day = THURSDAY;
```

Although you cannot directly assign an integer value to an enum variable, you can directly assign an enumerator to an integer variable. For example:

```
  int i;
  i = FRIDAY;
```

or:

```
  int i;
  work_day = THURSDAY;
  i = work_day;
```

Comparing Enumerator Values

1) Compare enumerator values

Enumerator values can be compared using the relational operators. For example, using the Day data type we just defined, the following expression is true:

```
  FRIDAY > MONDAY;
```

This is because, FRIDAY is stored as 4 and MONDAY is stored as 0 internally.

2) Compare enumerator values with integer values

For example:

```
  if(MONDAY == 0)
```

You can even compare different enumerated data types, for example:

```
  enum Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
  enum Number {ZERO, ONE, TWO};

  if (MONDAY == ZERO)
    std::cout << " They are equal! " ;
```

Although you can compile and run this program, there will be a warning reminds you that you are comparing two different enumerators.

Anonymous Enumerated Types

When you do not need to define variables of an enumerated type, you can actually make the type anonymous. You just neglect the type name when you define the enumerated data type:

```
enum { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

After this statement, we can use MONDAY to represent integer 0, TUESDAY to represent integer 1, and so forth. We just can't use the data type to define variables because the type does not have a name. It works like you are defining a bunch of `const int` identifiers for 0, 1, 2…

Using math operators to change the value of an enum variable

Since you can not assign an integer to enum variable, you may have problem in the following situation:

```
Day day1 = MONDAY, day2;
day2 = day1 + 1;
```

The second statement is actually trying to assign integer value 1 (0+1) to `day2`. This is not correct because `day2`'s data type is `Day`, not `int`. You can use the `static_cast` to solve the problem:

```
day2 = static_cast<Day>(day1 + 1);
```

The same idea is for the increment or decrement operator. For example, the following code will not work:

```
Day day1 = MONDAY;
day1++;
```

Because the second statement is actually doing:

```
day1 = day1 + 1;
```

This has the same problem with `day2 = day1 + 1`, you have to use a cast to convert the type of `day1 + 1` to `Day`:

```
day1 = static_cast<Day>(day1 + 1);
```

Using enumerators to output values

You can output different values based on different enumerator values. For example, the enumerated variable can be used as integer expression in the switch statement:

```
  Day work_day;
  work_day = MONDAY;
  switch (work_day)
  {
  case MONDAY: std::cout << "It′s Monday\n"; break;
  case TUESDAY: std::cout << "It′s Tuesday\n"; break;
  case WEDNESDAY: std::cout << "It′s Wednesday\n"; break;
  case THURSDAY: std::cout << "It′s Thursday\n"; break;
  case FRIDAY: std::cout << "It′s Friday\n"; break;
  }
```

The program will print “It’s Monday” on screen.

Specifying integer values for enumerators

By default, the enumerators in an enumerated data type are assigned the integer values 0, 1, 2, and so forth (start with 0, each element is 1 greater than its previous element). However, you can specify the values to be assigned to the enumerators when you define the enum data type. For example:

```
enum Temperature { FREEZE = 0, COLD = 10, COOL = 20, PLEASANT = 25, WARM = 30, HOT =
55, SCORCHING = 75, BOIL = 100};
```

If you leave out the values assignment for one or more of the enumerators, it will be assigned with a default value (first one is 0, others are 1 bigger than the previous enumerator), for example:

```
enum Temperature { FREEZE, COLD = 3, COOL, PLEASANT = 25, WARM, HOT, SCORCHING, BOIL};
```

The assignment would be: `FREEZE = 0, COLD = 3, COOL = 4, PLEASANT = 25, WARM = 26, HOT = 27, SCORCHING = 28, BOIL = 29.`

Enumerators must be unique within the same scope

Enumerators are **identifiers** just like variable names, named constants and function names. As with all identifiers, they must be unique within the same scope (scope is within the block of code). For example, the following statements will cause an error:

```
enum Temperature { FREEZE, COLD = 10, COOL = 20, PLEASANT = 25, WARM = 30 };
enum Degree { FREEZE, COLD = 10, PLEASANT = 25 };
```

Since FREEZE, COLD and PLEASANT are declared more than once within the same scope, the above code can not be compiled.

```
enum Temperature { FREEZE, COLD = 10, COOL = 20, PLEASANT = 25 };
void FREEZE(int num);
```

In the above code, FREEZE is declared twice, both as an enumerator and a function name.

```
enum Temperature { FREEZE, COLD = 10, COOL = 20, PLEASANT = 25 };
const int FREEZE = 20;
```

In the above code, FREEZE is declared twice, both as an enumerator and a named constant.

Strongly Typed enums (C++ 11)

A new type of enum, known as a strongly typed enum (also known as an enum class), is introduced in C++ 11. One of the characteristic of enum class is that you can define multiple enumerators with the same name in different enum classes. For example:

```
enum class Temperature { FREEZE, COLD = 10, COOL = 20, PLEASANT = 25, WARM = 30 };
enum class Degree { FREEZE, COLD, PLEASANT };
const int FREEZE = 20;
```

There are three FREEZE in the above statements, two are in different enum classes, one is in named constant. The reason why compiler accepts this is you can only access one FREEZE if you call FREEZE in the program, which is the named constant. For example, following statement:

```
  std::cout << FREEZE << "\n";
```

will print 20 on screen, because the named constant `FREEZE` is 20. You need to prefix the enumerator FREEZE with the name of the enum, followed by the :: operator in order to access the enumerator named FREEZE:

```
  Temperature::FREEZE;
  Degree::FREEZE;
```

Enumerators in enum class are stored as integers like regular enumerators. However, if you want to retrieve a strongly typed enumerators' corresponding integer value, you must use a cast operator.

Also, when you declare a enum class, you can optionally specify any integer data type as the underlying type. You just need to write a colon (:) after the enum name, followed by the underlying type. For example:

```
enum class Day : char { MONDAY = 65, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

under this declaration, the follow code will display A.

```
std::cout << static_cast<char>(Day::MONDAY) << "\n";
```

Notice that, even if the underlying data type is set to character, the default starting index is still 0, which corresponds to the first character in ASCII table.

Chapter 12. Advanced File Operations

Using the fstream Data Type

Definition

You define the fstream object just as you define object of other data types:

```
std::fstream data_file;
```

Open a file

As with ifstream and ofstream objects, you use an fstream object's open function to open a file (or do so during the definition of fstream object, same as ifstream and ofstream). However, an fstream object's open function requires two arguments:

1. the path of the file. This is similar with ifstream and ofstream objects.
2. a file access flag that indicates the mode in which you wish to open the file.

For example:

```
data_file.open( " D:\\example.txt " , std::ios::out);
```

Take a look on the second parameter, the file access flag. This tells C++ to open the file in output mode, which enables the object to write data into a file. The following statement uses the `std::ios::in` access flag, this can open the file in input mode, which allows data to be read from the file.

```
data_file.open( " D:\\example.txt " , std::ios::in);
```

There are many file access flags:

std::ios::app
Append mode. This mode can write data into the file.

If the file already exists, its contents are preserved and all output is written to the end of the file. Notice that under this mode, even if you use seekp() member function to move the file stream object to previous location, you can't change the data. Anything you add will still be added to the end of the original file. Because the beginning position (position = 0) is already the end of the file, so you can not do anything to the original content.

If you want to open the file while keeping the original content, and you also want to modify the original content. You can open the file in `out|in` mode. When you open in this mode, the original content of the file will not be deleted, and you can modify them as well.

If the file doesn't exist, this object can create the file.

std::ios::ate
If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.

std::ios::binary
Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text, rather than binary) (when to include this flag?)

std::ios::in
Input mode. Data will be read from the file. If the file doest not exist, it will not be created and the open function will fail.

std::ios::out
Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.

std::ios::trunc

If the file already exists, its contents will be deleted (truncated) (is this the only function?). This is the default mode used by std::ios::out. The deletion happens when the file has been opend. As long as the file has been opened, even if no data is written into the file, the file's content has been deleted.

| operator

Several flags may be used together if they are connected with the | operator. For example:

```
file.open( " D:\\example.txt " , std::ios::out | std::ios::in);
```

This statement opens the file example.txt in both input and output modes. This means data may be written to and read from the file.

Also, pay attention to the following characteristics:

1) when std::ios::out is used by itself, it can delete the file's content (default mode: std::ios::trunc). When std::ios::out is used with std::ios::in, the file's existing contents are preserved.
2) when std::ios::in is used by itself, if the file does not exist, the open function will fail. when std::ios::in is used with std::ios::out, if the file does not exist, it will be created.

Another example:

```
file.open( " D:\\example.txt " , std::ios::out | std::ios::binary);
```

The file is opened in both output and binary modes.

By using different combinations of access flags, you can open files in many possible modes.

Optional Parameter for istream and ofstream

You cannot change the fact that ifstream files may only be read from and ofstream files may only be written to. You can, however, vary the way operations are carried out on these files by providing a file access flag as an optional parameter to the open function. For example:

```
std::ofstream cout;
cout.open( " D:\\example.txt " , std::ios::app);
```

The `std::ios::app` changed the default `std::ios::trunct`, so the content of the file will be preserved, and the input will start from the end of the file.

Checking File's Existence

Sometimes you want to check whether a file exists or not. Below are two ways of checking.

1) Fail() member function

First, attempt to open the file for input. If the file doesn't exist, the open operation will fail, the failure can be detected by fail() member function. For example:

```
//creating file streamer: fstream
```

```
  std::fstream file;
  //open the file for input
  file.open( " D:\\example.txt " , std::ios::in);
  if (file.fail())
    std::cout <<  " The open failed.\n " ;
  else
    std::cout <<  " Open succeed.\n " ;
```

2) Place the file stream object directly into the expression of if statement.

```
  file.open( " D:\\example.txt " , std::ios::in);
  if (!file)
    std::cout <<  " The open failed.\n " ;
  else
    std::cout <<  " Open succeed.\n " ;
```

Passing File Stream Objects to Functions

File stream objects may be passed by reference to functions (they should always be passed by reference, otherwise, the compiler won't compile).

Following is an example to print ASCII code to a text file.

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

void WriteASCII(std::fstream &file);


int main() {
  std::fstream file;
  file.open( " example.txt " , std::ios::out);
  WriteASCII(file);
  file.close();
  return 0;
}

void WriteASCII(std::fstream &file) {
  for (int i = 0; i < 127; i++) {
    file <<  " ASCII#:  "  << i <<  " --- " ;
    file << static_cast<char>(i) <<  " \n " ;
  }
}
```

Member Functions for Reading and Writing Files

File stream objects have member functions for more specialized file reading and writing.

More Detailed Error Testing

All stream objects have error state bits that act as flags, which indicates the condition of the stream. The file stream object can have more than one error state bits. Different types of error bits are shown below:

Error State Bits

std::ios::eofbit
Set to true when the end of an input stream is encountered.

std::ios::failbit
Set to true when an attempted operation has failed or was invalid.

Examples of failed attempt:

1) file.open() failed. (file not exist when opening with read mode).
2) file.seekg(-2, std::ios::beg). This statement asks for the file stream object go beyond the initial position. This is a valid operation (you can do this with the file stream object), however, this operation will fail)
3) if a file stream object opens a file in read only mode, another statement wants to use this file stream object to write something into the file, the operation will fail because the file stream object in std::ios::in mode can't write data into the file. This will also be considered as an invalid operation.

std::ios::hardfail
Set to true when an uncoverable error has occurred.

std::ios::badbit
Set to true when an invalid operation has been attempted.

Example of invalid operation (case_3 in failbit example):

```
file.open("test.txt", std::ios::in);
file << " 23 ";
```

After this operation, both failbit and badbit will be set.

std::ios::goodbit
Set to true when all the flags above are not set, indicate the stream is in good condition. (it will set to false when any one of the above flag sets to true)

Error State Bits Test Member Function

The error state bits can be tested by following member functions (use . to attach to the end of the file stream object).

eof()
Returns 1 if eofbit flag is set, otherwise returns 0 (return value is an integer).

fail()
Returns 1 if failbit flag is set, otherwise returns 0.

bad()
Returns 1 if badbit flag is set, otherwise returns 0.

good()
Returns 1 if goodbit flag is set, otherwise returns 0.

clear()
When called with no arguments, clear all the flags mentioned above.

When called with the specific error state bits, only that bit is cleared.

getline()

When the whitespace characters are read by the >> operator, it considers them as delimiters, it will stop after them and not read them (the next reading starts at the first non-whitespace character). This problem can be solved by using the getline function:

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

The first parameter is the name of the file stream object, it specifies the stream object from which the data is to be read.

The second parameter is the name of a string object. The data read from the file will be stored here.

The third parameter is the delimiter character of your choice. If this delimiter is encountered, it will cause the function to stop reading. This argument is optional, if it is left out, '\n' is the default.

get()

The get member function reads a single character from the file. The syntax is similar with cin.get(), here you put: file.get(ch), where file is the file stream object, ch is the variable where character data stored in.

put()

The put member function writes a single character to the file. It is used with output file stream object. For example:

```
file.put(ch);
```

Working with Multiple Files

It’s possible to have more than one file open at once in a program. You have to use multiple file stream object to keep multiple files connected to the program, which is the basis for manipulating multiple files at the same time.

Binary Files and Their Manipulation

Binary File

Binary files contain data that is not necessarily stored as ASCII text. To work with data and file in its pure, binary format, the first step is to open the file in binary mode. This is accomplished by using the ios::binary access flag (see here), one example is as follows:

```
file.open( " D:\\number.txt " , std::ios::out | std::ios::binary);
```

The above statement causes the file is opened in both output and binary modes.s

write() and read() Member Functions

write()

The file stream object’s write member function is used to write binary data to a file. The general format of the write member function is:

```
file.write(address, size);
```

where address is the starting address of the section of memory that is to be written on the file. This argument is expected to be the address of a char (or the pointer to a char).

size is the number of bytes of memory to write, this value must be an integer value.

For example:

```
std::fstream file( " example2.txt " , std::ios::out | std::ios::binary);
char line[] =  " ABCDEFGHIJKLMNOPQRSTUVWXYZ " ;
file.write(line, sizeof(line));
file.close();
```

read()

The read member function is used to read binary data from a file into memory (into a variable which can hold the content you read). The general format of the read member function is:

file.read(address, size);

where address is the starting address of the section of memory where the data being read from the file is to be stored. This argument is expected to be the address of a char (or the pointer to a char).

size is the number of bytes of memory to write, this value must be an integer value.

PS: in my computer and my compiler, even if I don’t open the file in binary mode, I can still use write, read member function to write or read binary data to a file (or operate the file in a binary fashion)

Working Other Data Types with Binary Files

The first argument of write and read member functions should be a pointer to a `char`, when you want to read or write items that are other data types, you have to use a type cast.

To convert a pointer from one type to another you should use the `reinterpret_cast` type cast. The general format of the type cast is:

```
reinterpret_cast<data_type>(value);
```

`data_type`: the data type you are converting to.

`value`: the value that you are converting.

For example, if you want to use `reinterpret_cast` to convert a memory address of an integer to the memory address of a character (so that you can use a pointer to character to store the address of an integer variable), you can do:

```
int num = 20;
char *ptr = nullptr;
ptr = reinterpret_cast<char *>(&num);
```

After these statements, ptr can be used as the argument in write or read member functions.

Or you can directly put the `reinterpret_cast<char *>()` into the parenthese of write or read function.

Creating Records with Structures

Structures may be used to store fixed-length records to a file. After you declared the structure and defined a structure variable, you can write the data package into the file by using a file stream object (in out | binary mode) and the `reinterpret_cast<char *>()` type cast:

```
struct Person_info
{
  char name[20];
  char phone[20];
  char address[100];
  char email[30];
  int age;
};
  Person_info  person1  =  {  " Yu  Miao " ,  " 8505245269 " ,  " 1839  Portland  Ave " ,  "
ym15d@my.fsu.edu " , 27 };
  file.write(reinterpret_cast<char *>(&person1), sizeof(person1));
```

PS: if you have an integer array: num[10], you can’t use above code to save the content of an array to the file:

```
  file.write(reinterpret_cast<char *>(num), sizeof(num)); //not working!
```

You have to go over the array, term by term:

Pay attention that structures containing pointers cannot be correctly stored to disk using the techniques introduced in this section. This is because if the structure is read into memory on a subsequent run of the program, the memory addresses of the variables may change. So the pointer may not still pointing to the original variables.

String class objects contain implicit pointers, they cannot be a part of a structure that has to be stored.

(if you want to write structure variable into file, you have to use write member function, i.e. you have to do it in a binary fashion, otherwise, you can just use the '<<' operator)

Binary File Read Example

To clarify how binary file storage works, let's see an example. Suppose a txt file which has content: 1234. Now you open it in append and binary mode:

```
std::fstream file( " D:\\test.txt " , std::ios::app | std::ios::binary);
```

Define an integer, assign 5 to the integer:

```
int num = 5;
```

In the actual memory, this variable is saved as:

0000 0000	0000 0000	0000 0000	0000 0101

Use the write member function to write this num variable to the test.txt file (in binary mode):

```
file.write(reinterpret_cast<char *>(&num), sizeof(num));
```

When representing the integer variable, the four memory slots are held together (the size of an integer variable is 4 bytes). However, when saved to file, the four memory slots will be manipulated one by one. Additionally, the first slots will be saved last, while the last slots will be saved first. So after saving to file, the record in file memory space will be:

0000 0101	0000 0000	0000 0000	0000 0000

If now you open the file, the content of each memory slot will be interpreted based on ASCII table:

1234ENQNULNULNUL

1234 is the original content. ENQ is the fifth element in ASCII table, this is because the very first element after the original content is: $\boxed{0000\ 0101}$, which is 5 in decimal. The three NUL correspond to the following three $\boxed{0000\ 0000}$.

Actually, when the program save data to a file, the default mode is saved as ASCII text, which means the characters will be transformed to ASCII text before being saved to the file (in this process, data saving is not done through binary mode). This transformation won't happen if you use write function to save data to file (in this approach, data saving is done through binary mode, so the data is written into the file as it is in pure binary formate). So the content of the file is actually:

'1'	'2'	'3'	'4'	'ENQ'	'NUL'	'NUL'	'NUL'
0011 0001	0011 0010	0011 0011	0011 0100	0000 0101	0000 0000	0000 0000	0000 0000
49	50	51	52	5	0	0	0

Normally, when you read data from a file to a variable, this transformation will also happen, just to make sure you get the original value rather than the character value. For example, you read the test.txt file (containing 1234) to an integer variable, ~~the program will read four bytes of data stored in test.txt file (size being read is determined by the data type, integer is four bytes)~~. (Actually its not four bytes. Four bytes is the final size, not the size of reading. For example, when you have: 123456, you have to read 6 bytes and translate the six characters to the integers, the detail of this process is not known yet. However, if you read in binary mode, you'll only read four bytes.). you'll get 1234 stored in the integer variable:

0000 0000	0000 0000	0000 0100	1101 0010

If you read the file in binary mode (use read member function), the first slot you read will be placed to the last slot of the variable (reverse again). So what you stored in the integer variable will be:

'4'	'3'	'2'	'1'
0011 0100	0011 0011	0011 0010	0011 0001

If you print this integer out, you'll get:

875770417

This is exactly the decimal form of 0011 0100 0011 0011 0011 0010 0011 0001.

Random Access Files

seekp and seekg

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are `seekp` and `seekg`. `seekp` is used with files opened for output (data output to the file), and `seekg` is used with files opened for input (data input from the file to a variable that can hold the data). ("p" stands for "put" and "g" stands for "get", put means you put data into the file, get means you get data from the file).

When you open the file in write|read mode (`file.open( "D:\\12-20.dat" , std::ios::in | std::ios::out)`), there is only one file stream object position (in and out). If you use seekp to change the write position, the read position will also be changed.

Below is an example of seekp's usage:

```
file.seekp(20L, std::ios::beg);
```

The first argument is a long integer representing an offset into the file (in my practice, you can also use integers). It is the number of the byte you wish to move. In this example, 20L is used. (The suffix L forces the compiler to treat the number as a long integer). This statement moves the file's write position to byte number 20. Sometimes, the argument can be a negative number, this means the offset is performed backward. However, if the offset is backward, you can't use sizeof() function at the same time, example:

```
//this is not working
file.seekg(-2*sizeof(Item), std::ios::end);
```

and:

```
//this is working
int size = sizeof(Item);
file.seekg(-2*size, std::ios::end);
```

The second argument is the mode of the offset performs. It designates where to calculate the offset from. There are three flags (which are three modes):

1) `std::ios::beg` from the beginning (position 0)
2) `std::ios::end` from the end (the end element is the EOF character)
3) `std::ios::cur` from the current position

If a program has read to the end of a file, you must call the file stream object's clear member function before calling seekg or seekp. This clears the file stream object's eof flag. Otherwise, the seekg or seekp function will not work. (From my experience with problem 4, if you move the file stream object beyond the legit position, you have to clear it before using it. Think of you broke the file stream object while doing these invalid thing, i.e. `file.good() = 0`).

tellp and tellg

`tellp` and `tellg` are member functions of file stream objects, which my be used for random file access. When used, they can return the position of current write or read position.

Openning a File for Both Input and Output

You can perform input and output on an fstream file without closing it and reopening it. In order to do this, you have to open the file in both input and output mode:

```
file.open( " D:\\12-20.dat " , std::ios::in | std::ios::out);
```

Under this mode, the position of write and read are the same. And also, the `std::ios::out` won't clear the original content of the file.

The book says when opening a file in both in and out mode, if the file doesn't exist, it will be created. However, in my computer, the file won't be created. You have to make sure the file is exist before opening it in in|out mode.

Chapter 13. Introduction to Classes

Two Styles of Programming

There are two styles of programming: procedural programming and object-oriented programming.

Procedural programming
Center on creating procedures or functions that work on data. Data and function are separated.

Object-oriented programming
Center on creating objects. An object is an software entity that contains both data and procedures. Data in an object is known as *attributes* of the object; Procedure in an object is known as *member functions* of the object.

Object-oriented programming addresses problems that can result from the separation of code and data through encapsulation and data hiding.

Encapsulation
refers to the combining of data and code into a single object.

Data Hiding
The object can hide its data from code that is outside of the object, only the object's member functions may directly access and make changes to the object's data. However, outside code can access the member functions of an object, in this way the attributes (data) of an object can still be affected indirectly.

Classes and Objects

Before an object can be created, the programmer must determine the attributes and member functions that are necessary and then creates a class. A class is not an object, it is a section of code that specifies the attributes and member functions of an object. It's like a blueprint from which object can be created.

Think of the following analogy. A class is like a blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. We can build several identical houses from the same blueprint, each house is a separate instance of the house described by the blueprint.

Thus, a class is not an object, but it is a description of an object. When the program is running, it uses the class to create, in memory, as many objects of a specific type as needed. Each object created from a class is called an **instance** of the class.

Basics of Classes

Declaration of Class

In C++, the class is the construct primarily used to create objects. It is similar to a structure. It is a data type defined by the programmer, consisting of variables and functions. The general format of a class declaration is as follows:

```
class Class_name
{
  declaration;
  // ...more declaration
  // may follow
};
```

Pay attention that there is a semicolon at the end of }.

Private and Public Members of Class

In C++, a class's private members are hidden and can be accessed only by functions that are members of the same class. While a class's public members may be accessed by code outside the class. The members of a class are private by default.

If data is stored in private member of a class, it can be protected from code outside the object. In object-oriented programming, an object should protect its important data by making it private and providing a **public interface** to access that data.

On the other hand, public members of a class are actual the interface to the class object. They are the only members that may be accessed by any application that uses the object.

Access Specifiers

You can use key words *private* and *public* in class declarations. They can specify how class members may be accessed. For example:

```
class Class_name
{
private:
  //declarations of private
  //members appear here
public:
  //declarations of public
  //members appear here
};
```

Note that even though the default access of a class is private, it's still a good idea to use the private key word to explicitly declare private members. This clearly documents the accee speficication of all the members of the class.

It is not required that all members of the same access specification be declared in the same place, for example, you can:

```
class Class_name
{
private:
  //declarations of private
  //members appear here
public:
  //declarations of public
  //members appear here
private:
  //additional private
  //members declared here
};
```

However, you should adopt a consistent standard. Most programmers choose to group member declarations of the same access specification together.

Public Member Functions

Declaration of Public Member Function (Member Function Prototypes)

To allow access to a class's private member variables, you create public member functions that work with the private member variables. These public member functions provide an interface for code outside the class to access private member variables inside the class.

Take the rectangle area calculation class as an example, now we want to add some public member functions inside the declaration of the class:

```
class Rectangle
{
private:
  double width;
  double length;
public:
  void set_width(double);
  void set_length(double);
  double display_width() const;
  double display_length() const;
  double display_area() const;
};
```

Notice that the key word const appears in the declarations of the display_width, display_length and display_area member functions. When const appears after the parenthese in a member function declaration, it specifies that the function will not change any data stored in the calling object (the object it belongs to). If you inadvertently write code in the function that changes the calling object's data, the compiler will generate an error. The const key word must also appear in the function header as well.

If your member function's return value type is pointer, and you don't want the function change data stored in the calling object, you have to claim the return type as pointer to constant. No other part can use this pointer to change value it pointes to. (check: pointer to constant).

All the member functions (private or public) used in member function labeled "const" must also be labeled with "const".

Defining Member Functions
The definition of member function of a class is written outside the class declaration. Below is the general format of the function header of any member function defined outside the declaration of a class:

```
returntype class_name::function_name(parameter_list)

{


}
```

The operator :: are called the scope resolution operator. It identifies the function is a member of the class. (Things on right of :: is a member of thing on left of ::) (Tells the compiler that this function is the member function of the class).

If you declare a member function as const in class declaration, you have to also include a const key word in the function header:

```
returntype class_name::function_name(parameter_list) const

{


}
```

It is a good practice to mark all accessor functions as const in case of inadvertently changing of data.

Private Member Functions

A private member function may only be called from a function that is a member of the same class. You can use the private member function for internal processing (and these member function should not be called from code outside the class. For example, functions that will allocate new memory space for a pointer. It should only be called in constructor, if other code calls this function again, the pointer will be allocated with new memory space without deallocating the memory that it currently points to).

Defining an Instance of a Class

After the class is declared, class objects can be defined in a similar way with normal variables and structure variables. For example:

```
Rectangle rec1;
```

Defining a class object is called the instantiation of a class. In the above statement, rec1 is an instance of the Rectangle class.

After you define an instance of a class (the object), you can access it by the dot operator (both for attributes(member variables) and member functions)

An object's *state* is simply the data that is stored in the object's attributes at any given moment.

Avoiding Stale Data

It is better that do not include member variables that are dependent on other member variables in that class. This is because if this variable is not updated when other member variables change, it becomes stale.

For this reason, when designing a class, you should take care not to store in a member variable calculated data that could potentially become stale. Instead, provide a member function that returns the result of the calculation.

Pointers to Objects

<u>Normal Pointers</u>

You can define pointers to class object:

```
Circle moon;
Circle *cir_ptr = &moon;
cir_ptr->set_radius();
```

As shown above, you can use -> operator to access member variables and call member functions (just as accessing the structure variable).

Class object pointers can be used to dynamically allocate objects (same as previous dynamically allocate example):

<u>Smart Pointers</u>

You can use a std::unique_ptr to dynamically allocate memory, and not worry about deleting the memory when you are finished using it. A std::unique_ptr automatically deletes a chunk of dynamically acclocated memory when the memory is no longer being used.

To use the std::unique_ptr data type, you have to include the memory header file:

```
#include <memory>
```

Below is an example of the syntax for defining a std::unique_ptr that points to a dynamically allocated Circle object:

```
std::unique_ptr<Circle> moon_ptr(new Circle);
```

Separating Class Specification from Implementation

To work with class in a program, you have to include the following component:

1) Class declaration

In this section, you have to declare the name of the class, private members and public members of the class (attributes and member functions).

2) Member function definition

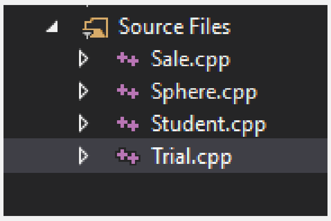
In this section, you have to define the details of each member function you declared in section 1.

Sometimes, it is desirable to split the class entity into separate sections, i.e., split the class into declaration of the class and the definition of the member function of the class. Then store these two parts in separate files. You can store class declaration into its own header file, the name of the file can be the same as the class, with the .h extension. This file is called the *class specification* file (specify the blueprint of the class, any program that uses the class should #include the class's header file). Member function definitions are stored in their own .cpp files, the name of the file can be the same as the class, with the .cpp extension. This file is called the *class implementation* file (because in class implementation file, you need to use class, so you have to include the class's header file when you write the class Implementation file).

After you finish the class header file and class implementation file, as well as the main program, you are now ready to compile the program. You have to first compile class implementation file and the main program, after this two steps you'll get two object files (object files contain machine language instructions, it needs to be further translated to machine language by adding the details of run time library routine machine codes. This step is done by another program called linker, which combines the object file with the necessary library routines). Then, these two object file is linked by linker and the file containing complete machine language instruction is created (the executable file).

Using other function when defining member function (writing class implementation)

When you define member functions, you can use other functions. You just need to include function prototype and function body in the implementation file. If you have multiple .cpp file in your project folder, like:



```
Source Files
  Sale.cpp
  Sphere.cpp
  Student.cpp
  Trial.cpp
```

And you include one function in one of these implementation file. If you want to use this specific function in another implementation file, you don't need to add function body again (this will cause an error: xxx function already defined). You just need to add a function prototype at the head.

An example:

1) class specification

```
//This is the specification of class Student

//this ifndef section is served as include guard, it can ensure that the declaration of class Student will only occur once
//if a certain constant has been found defined, it means the class has already been defined before, just skip that

#ifndef Student_h
#define Student_h

//class declaration
class Student
{
private:
  double score1;
  double score2;
  double score3;
public:
  void set_score(double s1, double s2, double s3);
  double get_average() const;
};

#endif
```

The above class declaration only execute once each time a program runs, because it is embeded in #ifndef-#endif, which will first check if there is a constant defined. If not, it will define the constant and then proceed to class declaration. If a constant is already defined, it means the class has already been declared somewhere else, this declaration will be skipped.

2) class implementation

```
//This is the implementation file of the class Student, it contains the definition of member functions
#include  " Student.h "

//set score function
void Student::set_score(double s1, double s2, double s3)
{
  score1 = s1;
  score2 = s2;
  score3 = s3;
}


double Student::get_average() const
{
  return (score1 + score2 + score3) / 3;
}
```

In order to let compiler know your function header, you have to include the Student.h header file in the beginning of the class implementation file. Then you can build up the definition of the member function.

3) main program

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

int main()
{
  //define a Student object
  Student Zack;

  //call set_score function to set scores for Zack
  Zack.set_score(59.9, 58, 44.4);

  //call get_average function to display the average score for Zack
  std::cout <<  " The average score for Zack is:  "  << Zack.get_average() <<  " \n " ;

  return 0;
}
```

In main program, you have to include the class header file in order to use the class to create object.

Inline Member Functions

When the body of a member function is written inside a class declaration, it is declared inline.

Constructors

Concept

A constructor is a member function that is automatically called when a class object is created. The name of the constructor is the same as the class. It is helpful to think of constructors as initialization routines, they are very useful for initializing member variables or performing other setup operations.

Following code shows an example of declaring a constructor for a class:

```
class Student
{
private:
  double score1;
  double score2;
  double score3;
public:
  Student();
  void set_score(double s1, double s2, double s3);
  double get_average() const
  {
    return (score1 + score2 + score3) / 3;
  }
};
```

It should be noticed that, the constructor does not have return type, not even void. This is because the constructor is not executed by explicit function calls and cannot return a value. The general format of the function header of a constructor's external definition is as follows:

Class_name::Class_name(Parameter_list);

PS: If you write a class with no constructor whatsoever, when the class is compiled C++ will

automatically write a default constructor that does nothing.

PS-2: If you define a pointer points to a class object, the constructor will not execute, because at this moment no actual object is created in memory. The execution of constructor happens when you use “new” to allocate a memory space to a object.

Passing Arguments to Constructors

Constructors that takes no arguments is called default constructor. However, like regular functions, constructors may accept arguments, have default arguments, be declared inline, and be overloaded.

When constructor accepts arguments, because the constructor is automatically called when an object is created, the arguments must be provided as part of the object definition, just the same as providing arguments when calling a function. For example:

```
Rectangle *rec1_ptr = new Rectangle(1, 2);
```

Otherwise, a compiler error will result.

Additional notes

[Gained when doing Mystring class]

Take Mystring class as an example. If you have a constructor like:

Mystring(); //default constructor

Mystring(char *initial); //constructor with arguments

If you write a statement like:

Mystring s1 = " abcdefg " ;

The default constructor will not be called, but the second constructor will be called. Because you provided a C-string literal, which is exactly the parameter type of the second constructor. The assignment operator in this initialization statement works like putting the thing on the right into the function parameter list of the constructor.

Overloading Constructors

A class can have more than one constructor, that is, the overloaded constructor. They all have different parameter lists.

Default Constructor

Suppose a class has several overloading constructors. Now we define an object based on this class. The default constructor is the constructor that will be automatically called when the object is defined without an argument list for its constructor. For this reason, a class may have only one default constructor (it there were more than one constructor that could be called without an argument, the compiler would not know which one to call by default).

Notice that if a constructor has a default argument for all its parameters, it is considered a default constructor. It would be an error to create a constructor that accepts no parameters along with another constructor that has default arguments for all its parameters (the compiler would not know which one to call by default).

For the above reason, a class should only have one default constructor.

Destructors

Destructor is a member function that is automatically called when an object is destroyed. The name of the destructor is the same name as the class, preceded by a tilde character (~).

When is the object be destroyed?

When the program exit (if the object is defined in the program).

When the function exit (if the object is defined in the function).

When using delete to clear dynamically allocated object.

PS: destructor can be called in the program directly. Although it will be called again when the object is destroyed.

Another three points should be mentioned:

1) destructor has not return type

2) destructor cannot accept arguments, so they never have a parameter list

3) a class should only have one destructor (no overloading destructors)

Arrays of Objects

You may define and work with arrays of class objects. Below is an example of defining an array of Student objects:

```
const int STUDENT_NUM = 100;
Student class_083[STUDENT_NUM];
```

When you create an array of objects, the default constructor is called for each object in the array (if the class does not have a default constructor, you must provide an initializer for each object in the array).

If you wish to define an array of objects and call a constructor that requires arguments, you must specify the arguments for each object individually in an initialization list (first term in initialization list corresponds to first element in the array, the default constructor is called for the non initialized object element).

For example, if you want to call a constructor that requires one argument, you can:

```
const int STUDENT_NUM = 100;
```

```
 Student class_083[STUDENT_NUM] = {  " Student1 " ,  " Student2 "  };
```

If you want to call a constructor that requires more than one argument, you have to use the form of a function call:

```
Student class_083[STUDENT_NUM] = { Student( " Yu Miao " , 100, 99.5),

 Student( " Yingying " , 95.5, 90.5) };
```

It isn't necessary to call the same constructor for each object in an array, for example:

```
Student class_083[STUDENT_NUM] = { Student( " Yu Miao " , 100, 99.5),  " Jack " ,
Student( " Yingying " , 95.5, 90.5) };
```

Creating Abstract Data Type Using Object-Oriented Programming

You can utilize the object-oriented programming to create abstract data types that are improvements on built-in data types. For example, you can create a class that has array-like characteristics and performs bounds checking.

Unified Modeling Language (UML)

The Unified Modeling Language provides a standard method for graphically depicting an object-oriented system.

Syntax of a class in UML:

1) Structure of class

Name of class
Member variables of the class (attributes)
Member function of the class (member functions)

2) Label private or public (access specification)

Private member: place a – in front of it;

Public member: place a + in front of it;

3) Data type

Member variable: place a colon followed by the name of the data type after the name of the variable. For example: - width : double

Return type of a member function: same as the member variable. After the function's name, place a colon followed by the return type. For example: + get_length() : double

Parameter variables: may be listed inside a member function's parentheses, for example:

+ set_length(len : double) : void

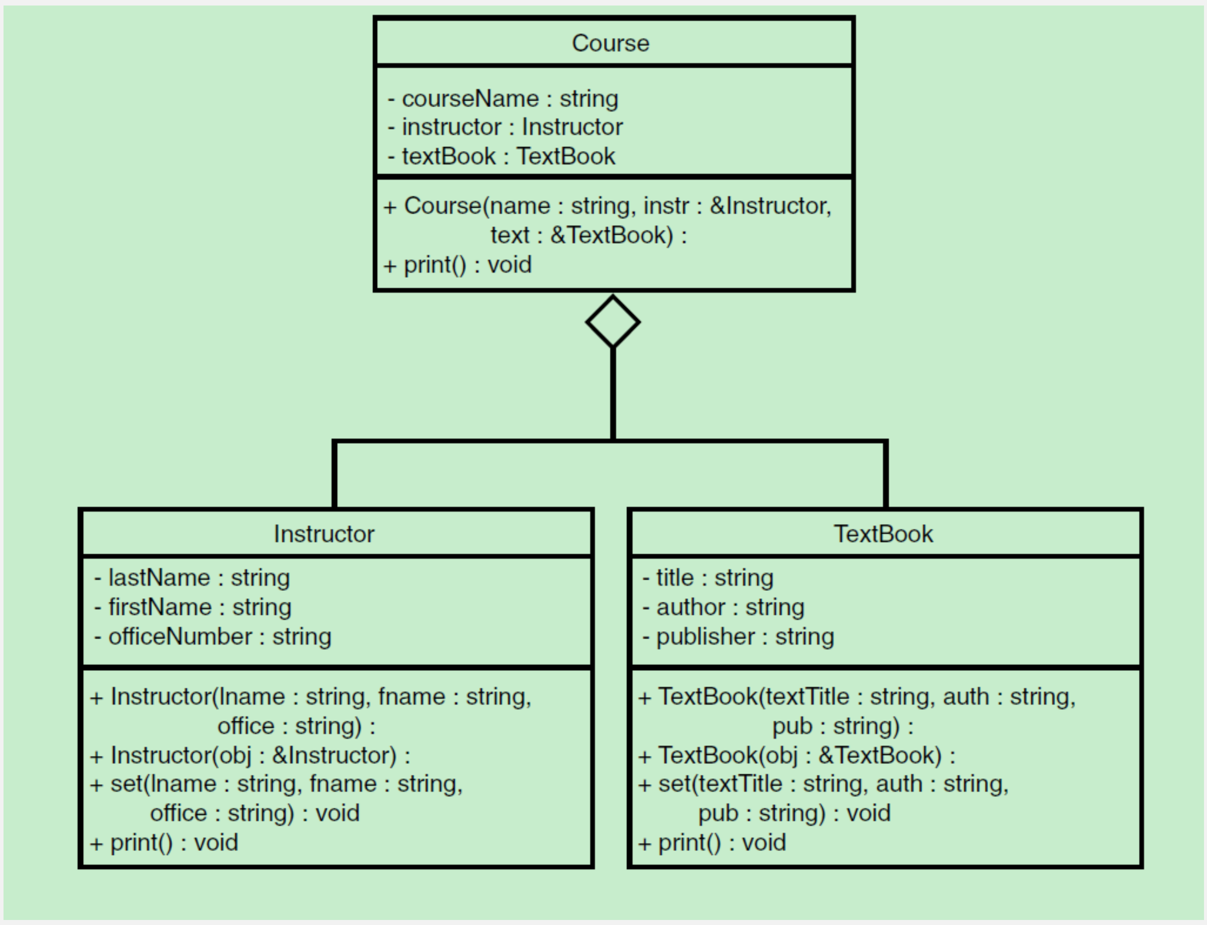
4) Constructors and destructor

You can show a constructor just as any other function, except you list no return type. For example:

+ Account() :

5) Aggregation in UML

You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that is the aggregate, example:


Course
- courseName : string
- instructor : Instructor
- textBook : TextBook
+ Course(name : string, instr : &Instructor, text : &TextBook) :
+ print() : void
Instructor
- lastName : string
- firstName : string
- officeNumber : string
+ Instructor(lname : string, fname : string, office : string) :
+ Instructor(obj : &Instructor) :
+ set(lname : string, fname : string, office : string) : void
+ print() : void
TextBook
- title : string
- author : string
- publisher : string
+ TextBook(textTitle : string, auth : string, pub : string) :
+ TextBook(obj : &TextBook) :
+ set(textTitle : string, auth : string, pub : string) : void
+ print() : void

Finding the classes and their responsibilities.

Finding the class

When creating an object-oriented application, the first step is to analyze the problem and determine the classes that are necessary and their responsibilities within the application.

Typically, your goal is to identify the different types of real-world objects that are present in the problem and then create classes for those types of objects within your application. A simple and popular technique involves the following steps:

1) Get a written description of the problem domain

The problem domain is the set of real-world objects, parties, and major events related to the problem. The problem domain description should include any of the following:

   a. Physical objects such as vehicles, machines, or products;
   b. Any role played by a person, such as manager, employee, customer, teacher, student, etc;
   c. The results of a business event;

2) Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class (literally, every nouns, and list them together).
3) Refine the list to include only the classes that are relevant to the problem.
   a. Eliminate nouns that refer to the same thing;
   b. Eliminate nouns that do not need to be concerned with in order to solve the problem;
   c. Eliminate nouns that might represent objects, not classes
   d. Eliminate nouns that might represent simple values which can be stored in a variable and do not require a class (ask yourself: a. would you use a group of related values to represent the item's state? b. are there any obvious actions to be performed by the item? If both questions are no, then the noun probably represents a value that can be stored in a simple variable.)

Identifying a class's responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's responsibilities are:

1) The things that the class is responsible for knowing
   (Attributes of the class)
2) The actions that the class is responsible for doing
   (Member functions of the class)

It is helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?". It is important to realize that designing an object-oriented application is an iterative process. As the design process unfolds, you will gain a deeper understanding of the problem, and consequently you will see ways to improve the design.

Chapter 14. More About Classes

Instance and Static Members

Each instance of a class has its own copies of the class's instance variables.

However, you can make a "universal" variable in a class, so each instance of that class have access to that variable (single variable). You do this by declaring a member variable as static.

On the other hand, if a member function is declared static, it may be called without any instances of the class being defined.

Instance variables

Instance variables are the variable of each instance (object) of a class that are holding unique data for specific instance. Remember class can be described as a blue print, and you can build different objects based on the same blue print (these objects are called instances). According to the blue print, an instance can have several variables to hold data. If these variables hold unique data for each object, they are instance variables.

Instance variable only serves the object which it belongs to, it will not server other object even built on the same class blue print.

Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as a static member variables and static member functions.

Static member variable and static member function can be accessed by any instance of the class. They are independent with any specific instances. You don't have to define an instance of the class in order to access the static member variable or static member function.

Static Member Variabls

When you write the class member, you can declare a member variable with the keyword static (add static before the declaration of the member variable). There will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. This copy of the static member variable will be shared by all instances of the class.

For example, the following Tree class uses a static member variable to keep count of the number of instances of the class that are created:

```
class Tree
{
private:
  static int tree_count; //static member variable
public:
  Tree()
  {
    tree_count++;
  }

  //access tree_count
  int GetTreeCount() const
  {
    return tree_count;
  }
};
```

//definition of static member variable is written outside of the class

```
int Tree::tree_count = 0;
```

Notice that the last line, `int Tree::tree_count = 0;`, You have to define the static member variable outside of the class declaration. This is required to establish a copy of static member function in memory. If you forget to define such variable, you'll receive following error message when compiling the program:

error LNK2001: unresolved external symbol " private: static int Tree::tree_count " (?tree_count@Tree@@0HA)

1>C:\Users\My\source\repos\Trial\Debug\Trial.exe : fatal error LNK1120: 1 unresolved externals

Normally, you put the definition of static member variable of the class in .cpp file, not in the header file (see the following problem I encountered if I put the definition in header file). However, if you define all your member functions inline, i.e., the definition of the function is in header file, you can put the definition of static member variable in the header file too.

PS: it is a good practice to initialize static variable with certain value (even 0, although if you don't provide a value, the default static variable will be initialized with value 0.). In this way, it is clear to anyone reading the code that what the variable starts with.

Problem:

When doing the budget class problem, if I declare a static variable, I have to write all public function inline (in header file). If there is any function definition in the corresponding .cpp file, the compiler will give following error:

1>main.obj : error LNK2005: " private: static int Budget::total_budget " (?total_budget@Budget@@0HA) already defined in Budget.obj

1>C:\Users\My\source\repos\Trial\Debug\Trial.exe : fatal error LNK1169: one or more multiply defined symbols found

It looks like the static int variable has been defined twice or so.

I don't know why this problem happens.

Update:

If you put the definition statement for the static member variable into the .cpp file, you will not have this problem (ie, you can put the definition of function in .cpp file, and leave the function prototype in the .h file).

Static Member Functions

You declare a static member function by placing the static keyword in the function's prototype. A static member function cannot access any nonstatic member data in its class (**can only access static member variables in the class**).

Static member function can be called and manipulate static member variable before any instance of class has been defined, this gives you the ability to create very specialized setup routines for class objects.

Format of declaring static member function:

```
static ReturnType FunctionName(ParameterTypeList);
```

Static member functions are called by connecting the function name to the class name with the scope resolution operator. If there is a class object defined, you can also call the static member function by the class object name and the dot operator.

Friends of Classes

Concept

A friend is a function or class that is not a member of a class, but has access to the private members of the class. A friend to a class can be a regular stand-alone function, or a member of another class, or an entire class.

You can declare friend of a class when you working on that class. Classes keep a list of their friends (inside the class specification). A function is declared a friend by placing the key word friend in front of a prototype of the function, general form (don't forget the return type!):

```
friend ReturnType FunctionName(ParameterTypeList)
```

You have to make this declaration inside the class's .h file to grant it access to the class.

Example 1: implementing a member function from class 2 as friend of class 1

In this example, two classes will be used to demonstrate how to declare a member function in class 2 as a friend of class 1. After the implementation, the member function in class 2 can access the private members of class 1 (private member variable and private static member variable).

Example overview

Class 1 is the Budget class, it's specification file is as followings:

```
//Budget class

#ifndef BUDGET_H

#define BUDGET_H


class Budget

{

private:

  static int total_budget; //hold total budget, it is a static variable

  int division_budget; //hold individual division budget
public:
 Budget(int b); //constructor, initialize division_budget
 int GetDivisionBudget() const; //return division budget
 int GetTotalBudget() const; //return total budget
 static void MainOffice(int b); //static member function, add b to total budget. It only
works with static member variable
};



#endif
```

This class describes the budget for a section of a company, and it uses a static member variable to keep track of the sum of all budgets.

The cpp file for class 1 is as follows:

```
#include  " Budget.h "


int Budget::total_budget = 0; //initialize static member variable


Budget::Budget(int b) //constructor: initialize with division_budget
{
 division_budget = b;
 total_budget += b;
}
int Budget::GetDivisionBudget() const //return division_budget stored in this class
{
 return division_budget;
}
int Budget::GetTotalBudget() const //return total_budget (a static member variable)
{
 return total_budget;
}
void Budget::MainOffice(int b) //static member function, add b to total budget
```

```
{
  total_budget += b;
}
```

it contains:

1) definition of static member variable (to establish a space for the variable in memory)

2) definition of public member functions.

Class 2 is the Auxiliary_office class. For simplicity, it only contains one public member function. The purpose of this member function is to add additional auxiliary office budgets to total_budget in Budget class. This means it has to be granted the access to the private static member variable (total_budget) of Budget class, thus this member function in Auxiliary_office class has to be declared as a friend of Budget class.

Steps for implementing friends

1) Setup in .h file where friend function belongs

To summarize, we have to declare a member function in `Auxiliary_office` class as a friend of `Budget` class. We write the header file for `Auxiliary_office` class as follows:

```
#ifndef AUXILI_H

#define AUXILI_H


class Budget;


class Auxiliary_office

{

public:

  void AddBudget(int b, Budget &B); //a friend of Budget

};


#endif
```

First, we take a look at the function prototype of the function:

```
void AddBudget(int b, Budget &B);
```

It has two parameters, one is an integer b, which will be added to the total_budget variable in Budget class.

Another is `Budget &B`, a reference object of the Budget class. This is to access any private member in an instance of Budget class (either static or normal member function). You have to add this parameter even if you don't use the reference object in the function at all (say, if you only want to access the static member variable by Budget::total_budget). If you don't include this reference object declaration, the compiler will remind you that the member is not accessible. This parameter is necessary if you want to declare the function as a friend of the Budget class. You can consider it as a label stating this member function is a friend of the `Budget` class.

Since you have such statement as:

Budget &B

You have to let compiler know what is Budget. At this time, the compiler doesn't there is a class named Budget, so you have to make this clear. Notice: `class Budget;` at the beginning, this is a forward declaration. It tells compiler that, a class named `Budget` will be declared later in the program.

Pay attention that, we don't use `#include "Budget.h"` here. Notice we didn't use any functions in Budget class, we only used the name of Budget when declaring the parameter. Under this circumstance, we use `class Budget;` to make the compiler know `Budget &B` is a reference object of class Budget.

If you are to declare a function that belongs `Budget` class in this `Auxiliary_office` class, you have to `#include "Budget.h"`.

2) Setup in .cpp file where friend function belongs

The .cpp file for `Auxiliary_office` class is:

```
#include "Budget.h"
#include "Auxiliary_office.h"

void Auxiliary_office::AddBudget(int b, Budget &B)
{
  Budget::total_budget += b;
}
```

or:

```
#include "Budget.h"
#include "Auxiliary_office.h"

void Auxiliary_office::AddBudget(int b, Budget &B)
{
  B.total_budget += b;
}
```

You can access the static member variable of class either way (class name + :: operator or object name + dot operator). However, you can only use the latter way to access a normal member variable in an object, because unlike static member variable, they are individual to each instance of the class.

Also, notice that you have to `#include "Budget.h"`. This is because you are accessing variables in `Budget` class.

3) Setup in .h file where friend function needs access

Now, back to header file of class 1: Budget class. As mentioned before, we have to declare function `Auxiliary_office::AddBudget(int b, Budget &B)` as a friend of Budget class, we should do this in **<u>Budget class's header file</u>** (specification file). Here is how:

First, we add `#include "Auxiliary_office.h"` in Budget class's specification file (header file). This is to enable the Budget class to declare a member function in `Auxiliary_office` class (as its friend). As mentioned above, if you want to use member function in `Auxiliary_office` class here, you should `#include "Auxiliary_office.h"` here, do not use `class Auxiliary_office`, because you are not declaring any parameter that is from `Auxiliary_office` class.

Then, you declare the member function in `Auxiliary_office` class as a friend of `Budget` class, you declare the function prototype of the member function (the same you wrote in header file for `Auxiliary_office` class) in public section of `Budget` class, and add `friend` before the return type of the function:

friend void Auxiliary_office::AddBudget(int b, Budget &B);

Now, you declared the member function `Auxiliary_office::AddBudget(int b, Budget &B)` as a friend of `Budget` class. The modified header file for budget is:

```
//Budget class
#ifndef BUDGET_H
#define BUDGET_H

#include "Auxiliary_office.h"

class Budget
{
private:
  static int total_budget; //hold total budget
  int division_budget; //hold individual division budget
public:
  Budget(int b);
  int GetDivisionBudget() const;
  int GetTotalBudget() const;
  static void MainOffice(int b); //static member function, add b to total budget
  friend void Auxiliary_office::AddBudget(int b, Budget &B);
};


#endif
```

Now you finished the job: successfully defined a member function from `Auxiliary_office` class as a friend of `Budget` class.

Example 2: implementing a stand-alone function as friend of class 1

Now we declare another friend of `Budget` class. This time, the friend is a stand-alone function:

```
void DirectModify(int b, Budget &B);
```

We can use this function to directly modify total_budget of the class. First, we declare this function as a friend of `Budget` class. We add following declaration in the public section of `Budget` class (same as example 1)

```
friend void DirectModify(int b, Budget &B);
```

Then we define this stand-alone function in main program, just as regular functions:

```
#include  " Auxiliary_office.h "

#include  " Budget.h "

#include <iostream>


void DirectModify(int b, Budget &B); //function prototype


int main()

{

   //main function declaration

}


void DirectModify(int b, Budget &B) //function definition

{

  B.total_budget = b;

}
```

We don't have to use statement like `class  Budget` to enable the use of `Budget  &B` in parameter of the function (I don't know why, is it because the class is included in main program?)

Example 3: implementing a class as friend of class 1

It is possible to make an entire class a friend of `Budget` class. You can do this by adding following declaration in `Budget` class:

```
friend class Auxiliary_office;
```

This statement will make every member function of `Auxiliary_office` (including ones that may be added later) would have access to the private members of `Budget`. Because of this, when writing function declaration and definition in `Auxiliary_office` class, you don't have to include `Budget &B` as the parameter (the forward declaration).

However, for this reason, declaring an entire class as a friend of another class is not encouraged. The best practice is to declare as friends only those functions that must have access to the private members of the class.

Memberwise Assignment

Concept

The = operator can be used between objects (objects are of the same class)

1) Assign one object's data to another object

2) Initialize one object with another object's data

By default, each member of one object is copied to its counterpart in the other object.

When you use = operator to initialize a newly defined object, the constructor of that object will not be called, otherwise, it will be bypassed (because memberwise assignment is conducted). You need a special constructor: copy constructor if you want something done when such situation happens (the copy constructor allows you to define more operations than just member wise assignment when you define an object using = operator).

Copy Constructors

Concept

A copy constructore is a special constructor. It will not be called under normal definition of a new object. However, if a **new** object is created and initialized with another object's data, the copy constructor will be called.

Pay attention that copy constructor will not be called when you assign an already existed object with other object of the same type. This because copy constructors are just constructors, they are only invoked when an object is created.

When you use = operatorm like:

obj_1 = obj_2;

Memberwise assignment will occur. If you want to change the way the assignment operator works, it must be overloaded (covered later).

Syntax

The syntax for defining a copy constructor is:

```
Class_name(const Class_name &obj)
{
  //memberwise assignment
  attribute_1 = obj.attribute_1;
  attribute_2 = obj.attribute_2;
```

```
    ...

    //special code to handle situation that
    //memberwise assignment is not applicable
    ......
  }
```

The flag for a copy constructor is its parameter list, which is a referenced class object of the same type (if a constructor has this type of parameter, then it is a copy constructor). You have to perform the memberwise assignment manually, then add codes that deal with situations that memberwise assignment is not applicable.

Notice the `const` in the parameter list. This is added because there is no reason the constructor should modify the argument's data. The `const` keyword ensures that the function cannot change the contents of the parameter. This will prevent you from inadvertently writing code that corrupts data.

Copy Constructors and Function Parameters

Copy constructor will be called when initialize object 2 with object 1.

Suppose you have a function like: function(Class_name a)

When you call the function, you pass an object by value (say you pass object named B to the function), another object (object a) will be created in memory, and a will be initialized with B's value (memberwise assignment). If the class has a copy constructor, it will be called during this process (copy constructor in a).

Back to the copy constructor:

Class_name(Class_name obj)

If the parameter is not passed by reference, but by value. Suppose you have following initialization:

Class_name obj_2 = obj_1;

You pass `obj_1` to the copy constructor by value. Another object, `obj`, will be created in memory. And `obj` will be initialized with `obj_1`'s value. The class do have a copy constructor (you are writing it right now!), so the constructor will be called. This means that, each call of the copy constructor will result in a creation of a object, followed by a new call of the copy constructor, it is like:

copy constructor-> create object_1 -> call copy constructor to finish creating object_1 -> create an object_2 -> call copy constructor to finish creating object_2 -> ...

or simpliy:

copy constructor -> create an object -> call copy constructor -> create an object -> call copy constructor -> create an object ->...

This is an infinite loop, which will continue until the available memory fills up.

To prevent the copy constructor from calling itself an infinite number of times, C++ requires its parameter to be a reference object. As long as the parameter is a reference object, there is no need to create a copy of object, thus the copy constructor will not be called during the pass of parameter.

<Work flow>: write the student example, check whether the passing of an object to a function can keep array information (under this circumstance, the copy constructor will be called, and the pointer to the array address will be re-defined, I want to check it out) The code is stored in program sketch file, work_flow_1 <Work flow>

The Default Copy Constructor

If a class doesn't have a copy constructor, C++ creates a default copy constructor for it. The default copy constructor performs the memberwise assignment automatically (this is the operation done when a class doesn't have a copy constructor but is initialized with = operator).

Operator Overloading

Concept

C++ allows you to redefine how standard operators work when used with class objects. This is done by working on the special member function called *operator function.*

Overloading the = operator

In order to change the way the assignment operator works with an object, it must be overloaded, i.e. you define an *operator function* for that operator. The operator function is then executed any time the operator is used with an object of the class.

Let's take a look on how we overload the = operator.

First, we should declare an operator function in the class header. Take the `Student` class as an example. To overload the = operator, we write the function head as:

```
void operator= (const Student &right);
```

`void` is the return type.

`operator=` is the function name, this specifies that the function overloads the = operator.

`const Student &right` is the parameter, declared as constant reference. This parameter is the term on the right side of the operator (it is why the name is `right.` However, you can name the parameter anything you wish, it will always take the object on the operator's right as its argument). It is not required that the parameter of an operator function be a reference object. However, a constant reference parameter has the following advantage:

1) The reference (`&`) can prevent the compiler from making a copy of the argument, thus increses the efficiency;
2) The `const` keyword can prevent accidentally changing the contents of the argument;

Because it is a member of the `Student` class, this function will be called only when the program encounters the following statement:

s1 = s2;

Where `s1` and `s2` are objects of `Student` class.

The next step is clear. We write the definition of the operator function. The operator = will do everything we defined in the operator function. For this particular example, we want to first copy the content of the object on the right of the = operator to object on the left, then we request a new memory space to the pointer:

```
void Student::operator=(const Student &obj)
{
  student_num++;
  name = obj.name;
  delete[] score; //to delete array previously stored
  score = new double[obj.score_num]; //request new space
  for (int i = 0; i < obj.score_num; i++)
    score[i] = DEFAULT_SCORE;
}
```

It is helpful to know that the following two statements do the same thing:

student2 = student1; // Call operator= function

student2.operator=(student1); // Call operator= function

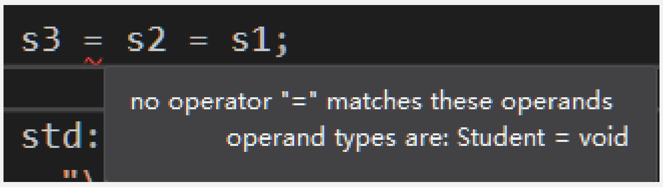
In the second statement you can see exactly what is going on in the function call.

The = Operator's Return Value

C++'s built-in operator allows multiple assignment statements such as:

a = b = c;

In this statement, the expression `b = c` causes `c` to be assigned to `b` and then returns the value of `c`. The return value is then assigned to `a`. In our version of the overloaded function, we didn't declare the return tuype of the function, so we can't do such multiple assignment. If you write statements like this, compiler will notice you that:



```
s3 = s2 = s1;
std:
no operator "=" matches these operands
        operand types are: Student = void
```

If a class object's overloaded = operator is to function the multiple assignment, it must have a valid return type. The operator function for = can be written as:

```
const Student operator= (const Student &right)
{
   //function definition
   return *this;
 }
```

It specifies that a `const Student` object is returned. Notice the last statement: `return *this`. This statement returns the value of a dereferenced pointer: `this`. It is a special built-in pointer that is available to a class's member functions. It always points to the instance of the class calling the member function. Actually, the `this` pointer is passed as a hidden argument to all nonstatic member functions.

General Issues of Operator Overloading

1) Changing an operator's entire meaning is not a good programming practice

2) You cannot change the number of operands taken by an operator. For example, the = symbol must always be a binary operator, ++ and – must always be unary operators.

3) You cannot overload all the C++ operators, following shows all of the C++ operators that may be overloaded

```
+  –   *   /   %   ^   &   |   ~   !   =   <
>   +=   -=   *=   /=  %=   ^=  &=  |=   <<   >>   >> =
<<=  ==   !=   <=  >=   &&   ||  ++   --   ->*   ,   ->
[]   ()   new   delete
```

The only operators that cannot be overloaded are:

```
?:   .   .*   ::   sizeof
```

Overloading Math Operators

You can also overload math operator to give more potential to classes.

Following italic notes and code are from :

http://en.cppreference.com/w/cpp/language/operators:

Binary operators are typically implemented as non-members to maintain symmetry (for example, when adding a complex number and an integer, if operator+ is a member function of the complex type, then only complex+integer would compile, and not integer+complex). Since for every binary arithmetic operator there exists a corresponding compound assignment operator, canonical forms of binary operators are implemented in terms of their compound assignments:

```
class X
{
public:
  X& operator+=(const X& rhs) // compound assignment (does not need to be a member, but
often is, to modify the private members)
  {
    /* addition of rhs to *this takes place here */
    return *this; // return the result by reference
  }

  // friends defined inside class body are inline and are hidden from non-ADL lookup.
Passing lhs by value helps optimize chained a+b+c. otherwise, both parameters may be
const references
  friend X operator+(X lhs, const X& rhs)
  {
    lhs += rhs; // reuse compound assignment
    return lhs; // return the result by value (uses move constructor)
  }
};
```

Additional notes:

Take a binary operator (+) as an example (example from Mystring work experience):

friend Mystring operator+ (const char* lhs, Mystring &rhs);

This function is declared as the friend of `Mystring` class. There are two parameters:

`const char* lhs`: this means an argument of `const char*` type will be on the left hand side of the operator (this is actually the caller of the operator).

`Mystring &rhs`: this means an argument of `Mystring` type will be on the right hand side of the operator.

If the program encounters following situation, this overloaded operator function will be called and do such calculation:

(const char*) + (Mystring)

Because the `Mystring` object is not the caller of this function, you should declare it as the friend of the `Mystring` class.

Additional notes:

<Got this experience when I work with the Mystring class>

I'm writing a `Mystring` class which does the similar thing as the `string` class. I want to overload the operator +, so that the following expression is possible:

Mystring + " c "

I found that even if I don't overload the operator + to accept expression like this, it can work too. The reason is I wrote a constructor which accepts a character parameter:

Mystring(const char *initial);

Thus, `"c"` will be automatically converted to `Mystring`, and because I already overloaded + for `Mystring + Mystring`, it can be calculated. So:

Mystring + `"c"` -> Mystring + Mystring

You can check what is the definition used by the + operator by right clicking the operator and choose "Peek definition" (in Visual Studio 2017).

On the other hand. if you overloaded + for `Mystring + "c"`, the `"c"` will not be converted.

Overloading the Prefix and Postfix ++ (or - -) Operator

The difference between prefix and postfix ++ operator is that, for prefix operator, the operand's value increases first and then returned. While for postfix operator, the operand's current value is returned first, then increases. When you redefine the prefix and postfix ++ operator, you should keep this in mind.

Overloading prefix ++ operator is like (take `Feet_inch` class as example)

```
Feet_inch Feet_inch::operator++ ()
{
  inch++;
  if (inch < 0 || inch >= 12)
    Simplify();
  return *this;
}
```

Remember that the ++ (or - -) operator is unary, so there is just one operand, the calling object itself, so there is no parameter inside the paranthese.

Overloading postfix ++ operator is slightly different from prefix ++ operator, because the current value of the object should be returned, and then the value increased. We have to use a temporary object to hold the current value.

```
Feet_inch Feet_inch::operator++ (int)
{
  Feet_inch temp = *this; //temp object to hold current value
  inch++;
  if (inch < 0 || inch >= 12)
    Simplify();
  return temp; //what you return is the value before increment
}
```

Notice that there is a *dummy parameter* in the parenthese of the function header. The keyword `int` establishes a nameless integer parameter. When C++ sees this parameter in an operator function, it knows the function is designed to be used in postfix mode.

Overloading Relational Operators

Relational operators may be overloaded, which allows classes to be compared in statements that use relational expression.

Overloaded relational operators are implemented like other binary operators. The only difference is that a relational operator function should always return a true or false value.

Overloading the << and >> Operators (stream insertion and extraction operator)

By overloading the stream insertion operator (<<). you could send the object to cout in following manner, and have the screen output automatically formatted in the correct way:

std::cout << distance;

By overloading the stream extraction operator (>>), you could take values directly from cin, as shown below:

std::cin >> distance;

Overloading << and >> is done in a slightly different way than overloading other operators. These operators are actually part of the ostream and istream classes defined in the C++ runtime library, (the cout and cin objects are instances of ostream and istream). You must write operator functions to overload the ostream version of << and the istream version of >>, so they work directly with a class such as `Feet_inch`.

<u>Overloading the << Operator</u>

The function that overloads the << operator is as follows:

```
std::ostream& operator<< (std::ostream &strm, const Feet_inch &obj)
{
  strm << obj.feet << "  feet, " << obj.inches << "  inches " ;
  return strm;
}
```

First, let's take a look on the parameters of this function:

`std::ostream &strm`: this is a reference to the actual ostream object on the left side of the << operator.

`const Feet_inch &obj`: this is a reference to the actual object on the right size of the << operator.

This function tells C++ how to handle any expression that has the following form:

`std::ostream object << Feet_inch object`

So, when C++ encounters the following statement, it will call the overloaded operator << function.

std::cout << distance;

Second, let's take a look on the return type of this function:

std::ostream&

This means that the function returns a reference to an `std::ostream` object. When:

return strm;

executes, it doesn't return a copy of `strm`, but a reference to it. This allows you to chain together several expressions using the overloaded << operator, such as:

```
std::cout << distance1 << "  " << distance2;
```

Overloading the >> Operator

The function that overloads the >> operator is as follows:

```
 std::istream& operator>> (std::istream &strm, Feet_inch &obj)
 {
   std::cout << " Please input the feet: " ;
   strm >> obj.feet;
   std::cout << " Please input the inch: " ;
   strm >> obj.inch;
   obj.Simplify(); //to simplify
   return strm;
 }
```

Similar with overloading the << operator, this function has two parameters:

`std::istream &strm`: this is a reference to the actual istream object on the left side of the >> operator.

`Feet_inch &obj`: this is a reference to the actual object on the right size of the >> operator.

This function tells C++ how to handle any expression that has the following form:

std::istream object >> Feet_inch object

So, when C++ encounters the following statement, it will call the overloaded operator >> function.

```
std::cin >> distance;
```

Once again, the function return value is:

std::istream&

It returns a reference to an istream object, so several of these expressions may be chained together.

Make overloaded << and >> to work

As you may have noticed, the two overloading operator functions above are actually not members of the `Feet_inch` class, thus they don't have access to the private member variables of `Feet_inch` class, such as `feet` and `inch`.

To give them access to the private member, you declare the overloaded << and >> operator functions as friends of the `Feet_inch` class:

```
friend std::ostream& operator<< (std::ostream &strm, const Feet_inch &obj)
friend std::istream& operator>> (std::istream &strm, Feet_inch &obj)
```

After you declare `friend` in header file, you can either write the function inline, or write the function in cpp file. If you decide to write in cpp file, remember don't add `Feet_inch::` in front of the function name (`operator<<`) as you would for other operator functions (see below), because this is actually not a member of class!

~~friend~~ std::ostream& ~~Feet_inch::~~operator<< (std::ostream &strm, const Feet_inch &obj)

(You don't have to input `friend`, because this is a keyword used inside a class).

Overloading the [ ] Operator

You can overload the [ ] operator to write classes that have array-like behaviors. An example is as follows:

```
int & Int_array::operator[] (const int &index) //overloaded [] operator, add a boundary check function
{
  if (index >= size || index < 0) //over the boundary
  {
    std::cout << " Error: subscript out of range!\n" ;
    exit(0);
  }
  else
    return num[index];
}
```

The `operator[]` function can have only a single parameter:

const int &index

This is a constant reference to an integer. This parameter holds the value placed inside the brackets in an expression (the subscript).

The `if` statement provides a boundary check. If the subscript is illegal, exit(0) will be called. If the subscript is within range, the function uses it as an offset into the array and returns a **reference** to the value stored at that location.

The return type of this function is: `int &`. You may wonder why the return type has to be a reference. The reason why the return type should be a reference rather than an integer value is that sometimes, you need to modify the returned "thing" by this [ ] operator, for example:

num[5] = 55;

Remember the built-in = operator (assignment operator) requires the object on its left to be an lvalue. An lvalue must represent a modifiable memory location, such as a variable. If the return type of [ ] operator is integer, it is not an lvalue. And a reference to an integer is an lvalue.

Additional notes:

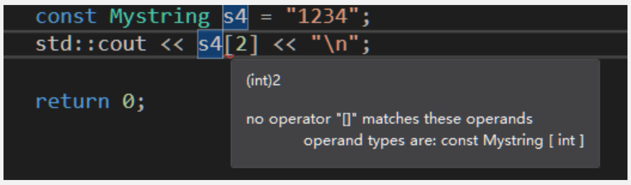
If you want to use [ ] operator on constant data type, you have to declare that the return type as constant. Take the Mystring class as an example:

```
char& operator[] (unsigned int index);     // returns L-value
const char& operator[] (unsigned int index) const;  // read-only return
```

The second one is used for const type. You have to declare this if you want to use it in the following situation:

```
const Mystring s4 = " 1234 " ;
std::cout << s4[2] << " \n " ;
```

If you don't declare the const type, in the above situation (use [ ] on a constant `Mystring` object), the compiler will remind you that:



```
const Mystring s4 = "1234";
std::cout << s4[2] << "\n";

return 0;
```

(int)2

no operator "[]" matches these operands
operand types are: const Mystring [int]

Object Conversion

Special operator functions may be written to convert a class object to any other type (automatic type conversion). Take the `Feet_inch` class as an example. The purpose is to achieve:

d = distance;

where `d` is a double variable, and `distance` is an object of `Feet_inch` class. After the statement, the decimal equivalent of the data stored in `distance` will be stored in `d`. For example, if `distance` is 4 feet, 6 inches, `d` will be: 4.5

Step_1: declare a conversion function in header file (class specification).

The syntax of declaring the conversion operator function is:

operator Data_type();

No return type is specified in the function header. Because the function converts the calling object to the data type specified by `Data_type`, it will always return this data type. The only argument for this function is the calling object, which is implicitly passed to the function, so there are no parameters.

Pay attention, this is written in the header file, inside the definition of class. `Data_type` is the data type you want to convert the class into. It can be built-in data types (like int, double etc) or user-defined data types (structures and classes). If you want to use user-defined data type, you have to include the header file that gives the definition of your data type. For example, if you want to convert `Feet_inch` object to `Student` object, you have to include `Student.h` in the header file of `Feet_inch` class when you declare: `operator Student();`

Step_2: Write the definition of the conversion function (either inline or in .cpp file)

The next step is to write your conversion logic, come up with your own way of how to convert `Feet_inch` object to `Student` object. If you choose to write in the cpp file, you have to declare the function is from the `Feet_inch` class, so function header is:

Feet_inch::operator Student()

After the above two steps, you can use assignment operatpor to convert one object to other data type in your defined way.

Aggresion

Aggresion occurs when a class contains an instance of another class as its attribute.

A good design principle is to separate related items into their own classes.

Making an instance of one class an attribute of another class is called *object aggregation*. The word aggregate means “a whole that is made of constituent parts.” A class can be called an aggregate class if it’s instance is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a “has a” relationship between the classes. Take the course, textbook and instructor as an example:

One instance of `course` class has an instance of `instructor` class;

One instance of `course` class has an instance of `textbook` class;

“Has a” relationship is sometimes called a *whole-part relationship* because one object is part of a greater whole.

Class Collaborations

It is common for classes to interact, or collaborate, with one another to perform their operations. Sometimes, one object will need the services of another object in order to fulfill its responsibilities.

Determining class collaborations with CRC cards.

CRC stands for class, responsibilities, collaborations.

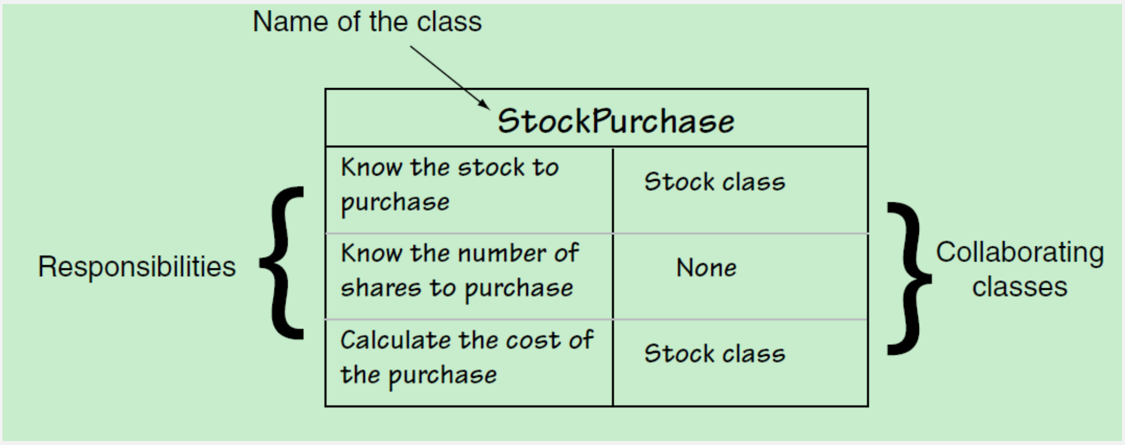
One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which is discussed in Chapter 13), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as:

> Will an object of this class need to get data from another object in order to fulfill this responsibility? (need data from other object?)

> Will an object of this class need to request another object to perform an operation in order to fulfill this responsibility? (need operation from other object?)

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. Following figure shows an example CRC card for `Stockpurchase` class.


Name of the class
StockPurchase
Responsibilities
Know the stock to purchase
Stock class
Know the number of shares to purchase
None
Calculate the cost of the purchase
Stock class
Collaborating classes

Chapter 15. Inheritance, Polymorphism and Virtual Functions

Inheritance

Concept

- Inheritance allows a new class to be based on an existing class;
- The new class inherits all the member variables and functions of the class it based on
- The new class won't inherit constructors and destructor of the class it based on

Origin of Inheritance: Generalization and Specialization

In the real world you can find many objects that are specialized versions of other more general objects. For example, *dog* is a specialized object of *animal*.

Inheritance allows abstraction of this kind of relation: creating new class that is based on an existing class. When one object is a specialized version of another object, there is an "*is a*" relationship between them, for example: a tree *is a* plant, a dog *is an* animal.

When an "*is a*" relationship exists between classes, it means that the specialized class has **all** of the characteristics of the general class, **plus additional** characteristics that make it special. In object-oriented programming, inheritance is used to create an "*is a*" relationship between classes.

Inheritance involves a *base class* and a *derived class*. The *base class* is the general class and the *derived class* is the specialized class.

The derived class has following features:

- it inherits the member variables and member functions of the base class without any of them being rewritten (including private and public members)
- it does not inherit constructors and destructor
- it can be added with new member variables and functions, making it more specialized than the base class

Syntax

Suppose you have a `Base` class. Now you want to make a derived class from `Base` named `Derived`. You declare the `Derived` class in following way:

```
class Derived : public Base
{
  // class definition
  // goese there
};
```

This declaration tells you that, `Derived` *is a* `Base`. Similarly, we can have:

```
class Dog : public Animal; // Dog is an Animal
class Tree : public Plant; // Tree is a Plant
class Ginkgo : public Tree; // Ginkgo is a Tree
```

You have to `#include` the base class header file in the derived class's header file.

The word, `public` which precedes the name of the `Base` class, is the *base class access specification*. It affects how the members of the `Base` class are inherited by the `Derived` class. When you create an object of a `Derived` class, you can think of it as **being built on top of** an object of the `Base` class. The members of the `Base` class object becomes members of the `Derived` class object automatically. How the `Base` class members appear in the `Derived` class is determined by the *base class access specification*. Base class access specification will be covered in detail in next section (Class Access).

If you declare the base class access specification as `public`, you can access the `public` members of the `Base` class without any additional declarations. For example, you can call `public` member functions of the `Base` class. Although you can't access the `private` member of the `Base` class directly in the `Derived` class, you can access them via the interface defined in `public` of the `Base` class.

Inheritance does not work in reverse. It is not possible for a base class to call a member function of a derived class.

Protected Members and Class Access Specifications

Protected members

Except `private` and `public`, C++ provides a third *access specification*, `protected`. It is similar with `private` members, except they may be accessed by functions in a derived class. It is inaccessible to the rest of the program (just like `private`).

More about base class access specification

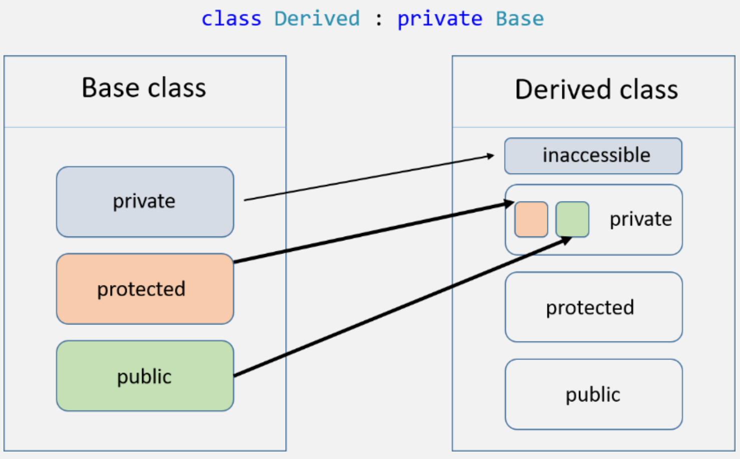
Make sure you know the difference between base class access specification and member access specification:

- base class access specification: determines how inherited base class members are accessed by the derived class
- member access specification: determines how members defined within the class are accessed by code outside of the class

When you create an object of a derived class, it will inherit all the members of the base class (including `private`, `protected` and `public`), i.e. it can have its own `private`, `protected` and `public` members (inherited from `Base` class).

However, the access of this `Derived` class to those inherited members is determined by the base class access specification. Following shows the details.

1) class Derived : private Base



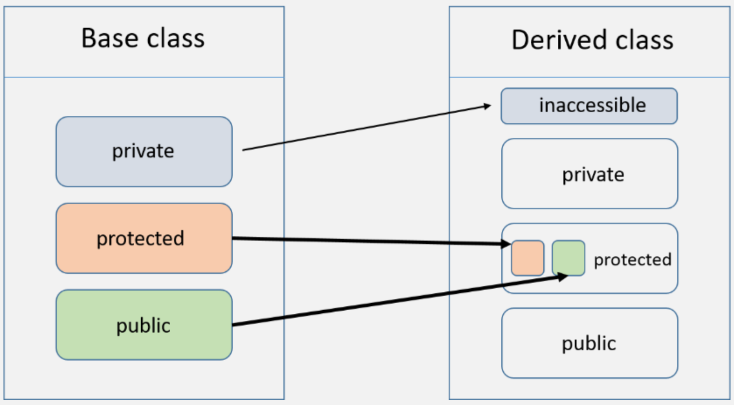
`private` members in `Base` class will become inaccessible to `Derived` class;

`protected` members in `Base` class will become `private` member in `Derived` class;

`public` members in `Base` class will become `private` member in `Derived` class;

2) class Derived : protected Base

```
class Derived : protected Base
```



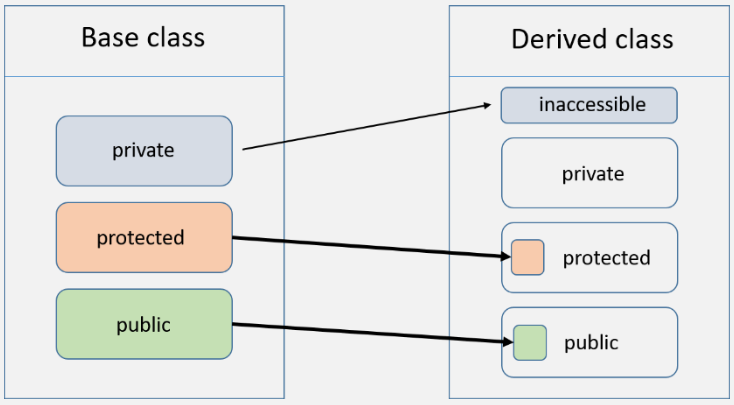
`private` members in `Base` class will become inaccessible to `Derived` class;

`protected` members in `Base` class will become `protected` member in `Derived` class;

`public` members in `Base` class will become `protected` member in `Derived` class;

3) class Derived : public Base

```
class Derived : public Base
```



`private` members in `Base` class will become inaccessible to `Derived` class;

`protected` members in `Base` class will become `protected` member in `Derived` class;

`public` members in `Base` class will become `public` member in `Derived` class;

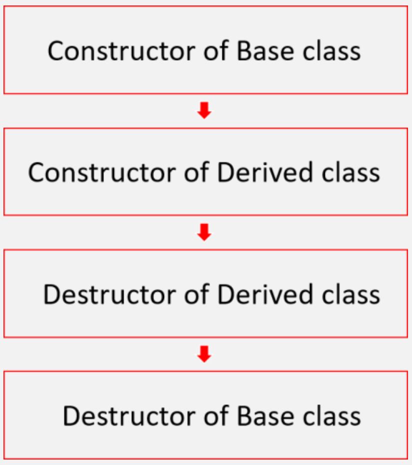
PS: if the base class access specification is left out of a declaration, the default access specification is private:

class Derived : Base

Constructors and Destructors in Base and Derived Classes

Calling order

Suppose you have a `Derived` class and a `Base` class. Now you are defining an object of a `Derived` class. Then you exit the program so the `Derived` object is destroyed. The order of calling constructors and destructor is as follows:


Constructor of Base class
Constructor of Derived class
Destructor of Derived class
Destructor of Base class

Passing Arguments to Base Class Constructors

Consider following scenario:

```
class Derived : public Base
```

When you are creating a `Derived` object, you will first call the constructor of `Base` class. In order to pass arguments to `Base` class constructor, you have to let the `Derived` class constructor pass arguments to the `Base` class constructor.

Now, suppose the `Base` class has a constructor that needs two arguments: `length` and `width`. And `Derived` class has a constructor that needs one argument: `height`. When you write the parameter list for constructor of `Derived` class, you can also include the parameter for constructor of `Base` class. Here is how you should write **when you implement the constructor (NOT the prototype. The prototype in .h file remains the same):**

```
Derived::Derived() : Base() // default constructor
Derived::Derived(double len, double w, double h) : Base(len, w) // constructor with parameter
```

// default constructor

Notice the added notation in the header of the constructor. A colon is placed after the `Derived` class constructor's parenthese, followed by a function call to a base class constructor. When this `Derived` class constructor is executed, it will first call the `Base` class's default constructor.

// constructor with parameter

You can also pass arguments to the `Base` class constructor. As we can see from the above, the

`Derived` class constructor has three parameters: `double len, double w, double h`. The first two is passed to `Base` class constructor as arguments. The third is the argument for `Derived` class constructor.

The order of the parameter in `Derived` class constructor is not required. You just need to put the argument in the followed `Base` class constructor in right order.

Note_1:

Pay attention that you only write this notation in the definition of a constructor, not in a prototype (when you write the constructor inline, you have to write this notation in .h file; when you define the constructor in .cpp file, you have to also write this to .cpp file, and write a normal prototype in .h file).

Note_2:

The `Base` class constructor is always executed before the `Derived` class constructor. It will take the argument it needs in the argument list of `Derived` class constructor, and executes. When the `Base` constructor finishes, the `Derived` class constructor is then executed.

Note_3:

If the `Base` class has no default constructor, then the `Derived` class must have a constructor that calls one of the `Base` class constructors. Because normally, if all the `Derived` class constructor don't call `Base` class constructor, when a `Derived` class object is made, the default `Base` class constructor will be called.

Redefining Base Class Functions

You can redefine a `Base` class member function in its `Derived` class. Its like an overloaded member function (because the function has the same name), however, there is a distinction between redefining a function and overloading a function:

- Overloading functions must have different ***function signature*** (same function name, but different parameter list)
- Redefining function must have the same ***function signature***. Redefining happens when a `Derived` class has a function with the same name and same parameter list as a `Base` class function.

Suppose you have follow classes:

```
class Derived : public Base
```

There is a function: `func(int num)` in `Base` class. Now you want to redefine `func` in `Derived` class. You can directly declare it in header file of `Derived` class:

```
class Derived : public Base
{
```

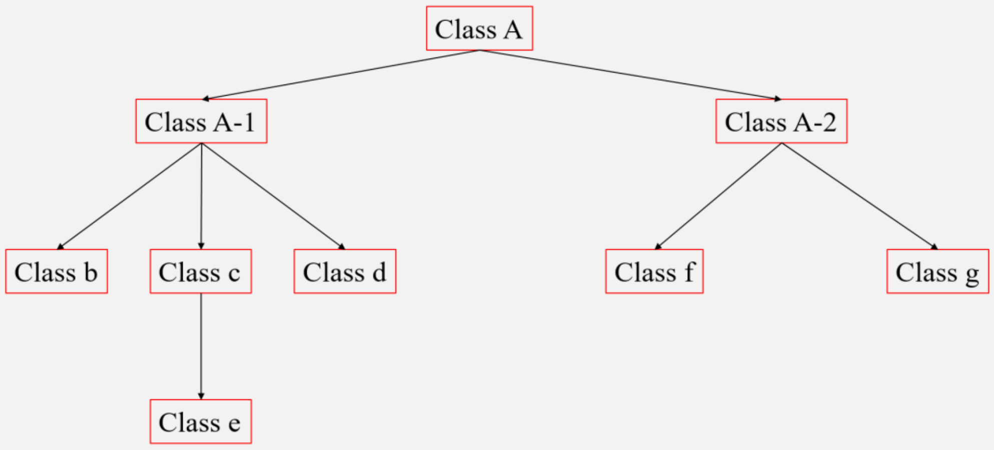
```
private:
  // declaration of private member
public:
  void func(int num2) // define the func() directly
  {
    int num = num2 / 2;
    Base::func(num); // you can call the
           // original func() in Base
           // using scope resolution operator
    // additional definition
    // goes here
  }
};
```

If you redefine the function. You have to use the scope resolution operator :: to call the original function in `Base` class.

C++ cast static binding to re-defined member function. That is, function call and the corresponding function is bound together when compiling.

Class Hierarchies

A `Derived` class can be the `Base` class for another class, for example:



In the above figure, class e inherits class c's members, including the ones that class c inherited from class A-1, including the ones that class A-1 inherited from class A.

Redefined member function in class hierarchies

Suppose there is a function named `func()` in class A-1 and class c (redefined `func()`). Now an object of class e is calling the `func()` (without scope resolution operator):

e.func();

The function that is cloest to class e will be called automatically (class c). If you want to call `func()` in class A-1, you have to use the scope resolution operator:

e.A_1::func();

This is because, class e inherits the redefined `func()`.

Polymorphism and Virtual Member Functions

Polymorphism

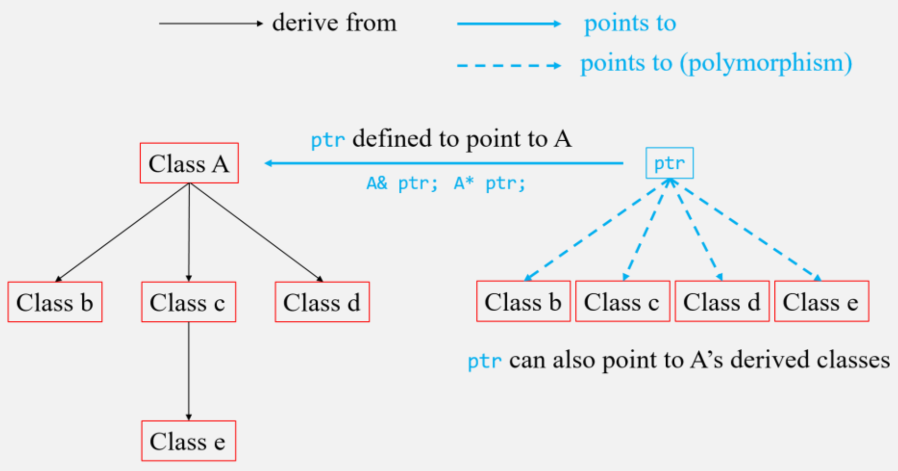
Concept

Polymorphism means the ability to take many forms.

Polymorphism allows an object reference variable or an object pointer to reference objects of different types and to call the correct member functions, depending upon the type of original object being referenced.

When the referenced object type is different from the original object reference type, it is required that the referenced object is **derived** from the original object reference type.

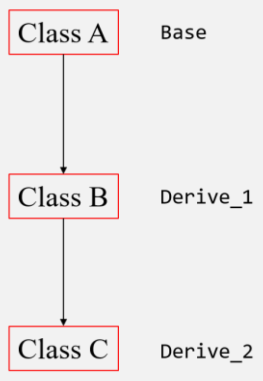
Following diagram shows the concept of polymorphism:



Polymorphism means an object reference variable or an object pointer is granted the ability to take many forms. Normally, a pointer can only points the data type declared in its definition. By polymorphism, an object reference variable or an object pointer can also point to object that is derived from the data type declared in its definition.

Example

Suppose you have following class hierarchy:



Now, you write a function who's parameter is constant reference of class A:

void Func(const A& object);

You can call the function by passing an A object, for example:

A obj;

Func(obj);

Since both class B and class C are derived from class A, polymorphism allows you to pass object of class B and C to `Func()` also:

B obj_b;

Func(obj_b); // valid

C obj_c;

Func(obj_c); // valid

Binding and Static Binding

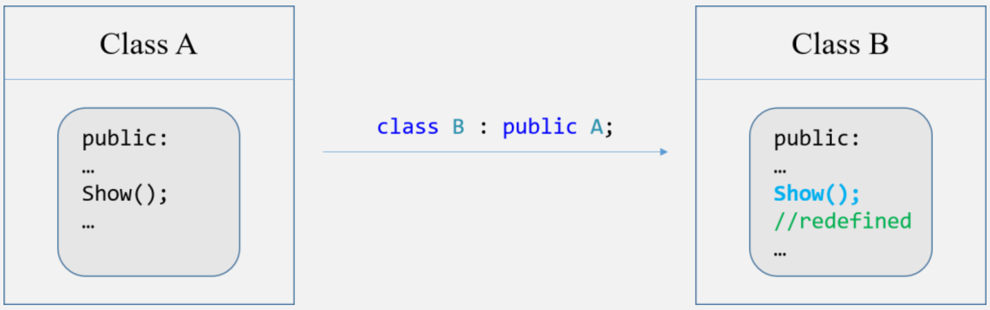
Consider the `Func()` as follows:

```
void Func(const A& object)
{
  // codes...
  object.Show();
  // codes...
}
```

The function accepts a constant reference variable of `A object` type.

The function will call a member function named `Show()` of `A object` in the function body.

Now, suppose class B is derived from class A, and the Show() function has been redefined in class B:



Now you call `Func()` and pass an object of B class to the function:

B obj_b;

Func(obj_b);

As the function reaches the step that calls `Show()`, it will call `Show()` function in class A, not the redefined `Show()` function in class B.

Short explain:

This is because, when compiling the program, C++ reads:

```
void Func(const A& object)
```

C++ then knows that if there is any member functions of `object` called in the function body, they should be member function of `A object`.

Explain (binding and static binding):

This behavior happens because of the way C++ matches function calls with the correct function. This process is known as ***binding***. Because the parameter is of the `A` type, the compiler binds the function call of `Show()` with the `A`'s `Show()` function. When the program executes, it has already been determined by the compiler that the `A`'s `Show()` function will be called.

The process of matching a function call with a function at compile time is called ***static binding.***

Base Class Pointers

As described above, pointers to a base class may be assigned the address of a derived class object.

However, you can only use this pointer to access the public member in base class. If you have any specialized public members in the derived class, you will not see them when you use a pointer to base class to access the derived class.

Also, pay attention that the "is-a" relationship does not work in reverse. You can't have a pointer (or reference variable) to a derived class points to the base class of the derived class directly. Take `A` and `B` as example. Class `B` is derived from class `A`.

```
B* ptr_B = nullptr;

A obj_A;


ptr_B = &obj_A; // invalid
```

However, you can use type cast (`static_cast`) to enable the assignment:

```
ptr_B = static_cast<A*>(&obj_A); // valid
```

After the above code, `ptr_B` points to `obj_A`. Since `ptr_B` is defined to point to `B`, it is logical for the compiler that it accesses some unique members in `B` class (only exists in `B`, not in `A`). This means that even if `ptr_B` points to `obj_A`, you can still write expression like:

```
ptr_B -> func_B();
```

where `func_B()` only exists in class `B`. The compiler will not think this is a mistake. However, this expression will cause runtime error (it may cause exit, or the pointer will point to garbage area).

Virtual Functions

Concept

It is possible to make a member function do specific things in specific derived class (in base class, the function can do this, in one derived class, the function can do that, the task completed by this function is different in different class). You just have to declare the member function as virtual member function. In this way, the static binding can be avoided.

C++ cast ***dynamic binding*** to virtual function, so it is dynamically bound to function calls. In dynamic binding, C++ determines which function to call at runtime, depending on the type of the object responsible for the call.

Take the above example:

If `Func()` is called, and an A class object is passed to the function as parameter, this A object is responsible for the call, C++ will execute `A::Show()`.

If `Func()` is called, and a B class object is passed to the function as parameter, this B object is responsible for the call, C++ will execute `B::Show()`.

Syntax

Virtual functions are declared by placing the key word `virtual` before the return type in the **base** class's function declaration, such as:

virtual void Show() const;

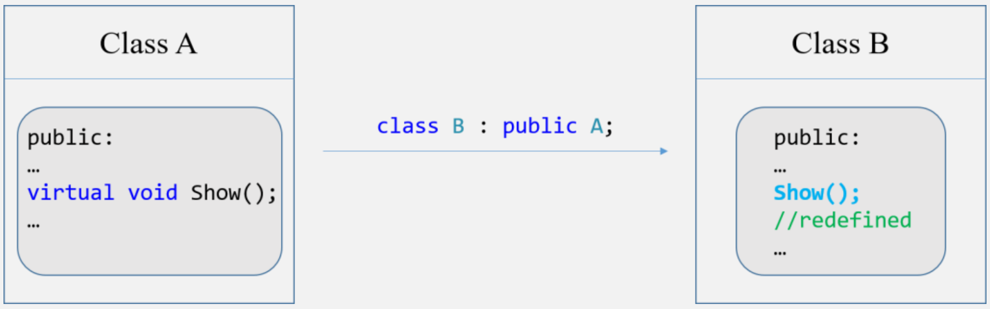
When compiler reads `virtual`, it expects `Show()` to be redefined in a derived class. Thus, the compiler will not cast ***static binding*** when it compiles the program (won't match function calls to the actual function). Instead, it allows the program to bind calls, at runtime, to the version of the function that belongs to the same class as the object responsible for the call.

Note:

- You place the `virtual` keyword only in the function's declaration or prototype (in header file). If the function is defined outside the class (in a .cpp file), you **do not** place the `virtual` key word in the function header in .cpp file. Because the `virtual` key word can only be placed inside a class specification.
- When a member function is declared `virtual` in a base class, any redefined versions of the function that appear in derived classes automatically become virtual. Thus you don't have to declare `virtual` in the redefined function in derived classes. However, it is a good idea to declare the function `virtual` in derived class for documentation purposes.

Example

Using the same example as in the last section. Now the function `Show()` is declared virtual, and this function is redefined in class B:



The function that calls `Show()` is defined as:

```
void Func(const A& object)
{
 // codes...
 object.Show();
 // codes...
}
```

Now, since `Show()` has been declared virtual, when an A object has been passed to the function, `A::Show()` will be called; when a B object has been passed to the function, `B::Show()` will be called:

```
A obj_a;
Func(obj_a);          //          A::Show()          will          be          called

B obj_b;
Func(obj_b); // B::Show() will be called
```

<u>Reference or Pointers Required</u>

Polymorphism requires references or pointers. The following object is passed by value:

```
void Func(A object);


B obj_b;
Func(obj_b); // invalid, because the parameter
      // is passed by value
```

When the object is passed by value, any objects that are passed to this function will be treated as object of `A` class. This means you can still pass `A`'s derived class when calling this function, but they will be treated as an object of `A` class. If there is any redefined member function called in this function, even if they are declared virtual, only the member function in `A` class will be called.

<u>Redefining vs. Overriding vs. Overloading</u>

When a class redefines a virtual function, it is said that the class overrides the function. In C++, the difference between overriding and redefining base class functions is that overridden functions are dynamically bound, and redefined functions are statically bound. Only virtual functions can be overridden.

Make sure you know the difference between following concept:

- Redefine: re-define a member function, *function signature* should be the same. Static binding.
- Override: re-define a virtual function. Dynamic binding.
- Overload: overload a member function, function signature is different (same function name, different parameter list)

Make sure your virtual function is overridden

When you are writing overriding functions, you must make sure you don't write the function into an overloading function (i.e. you didn't keep the function signature the same). You can place the `override` key word into the function header (or prototype), for example:

virtual char GetGrade() const override;

Unlike the `const` key word, you don't have to place the `override` key word in function definition in .cpp files. Just place it in function prototype.

The `override` key word tells the compiler that this function should be overridden. If the function does not actually override any functions, there will be a compiler error remind you that there is no overridden happen.

Prevent a member function from being overridden

In some derived classes, you might want to make sure that a virtual member function cannot be overridden any further down the class hierarchy. When a member function is declared with the `final` key word, it cannot be overridden in a derived class.

For example:

virtual char GetGrade() const final;

If a derived class attempts to override a `final` member function, the compiler generates an error.

<u>Virtual Destructors</u>

<From Textbook>

When you write a class with a destructor, and the class could potentially become a base class, you should always declare the destructor `virtual`. If the destructor is not declared as `virtual`, the compiler will perform static binding on the destructor (bind the destructor to the object type used when defining the pointer).

Look at following example:

class A;

```
class B : public A; // class B is derived from A


A* ptr_A = new B; // ptr_A is a base pointer, assign an address of B to ptr_A


delete ptr_A; // delete allocated memory
```

Suppose both `A` and `B` has destructor. And the destructor is not declared `virtual`. When `delete ptr_A;` is executed, only destructor in `A` is called. Because `ptr_A` is defined to point to `A`.

If you want C++ dynamically bind the destructor, you should declare the destructor in base class as virtual (same as how you declare virtual functions).

A good programming practice to follow is that any class that has a virtual member function should also have a virtual destructor. If the class doesn't require a destructor, it should have a virtual destructor that performs no statements. Remember, when a base class function is declared virtual , all overridden versions of the function in derived classes automatically become virtual. Including a virtual destructor in a base class, even one that does nothing, will ensure that any derived class destructors will also be virtual.

<From My Experience>

On my compiler (Microsoft Visual Studio 2017), even if I don't declare the destructor virtual, the destructor is dynamically bound. See calling orders.

Take the above example. When `delete ptr_A;` is executed, destructor in `B` is first called, then destructor in `A` is called.

Abstract Base Classes and pure Virtual Functions

Concept

- An abstract base class cannot be instantiated, but other classes are derived from it.
- A pure virtual function is a virtual member function of a base class that must be overridden.
- When a class contains a pure virtual function as a member, that class becomes an abstract base class

Consider the abstract base class as the "general concept" in real life. For example, "dog" can be an abstract base class. You can derive many classes from "dog", for example, husky "is a" dog, but you can't instantiate a "dog", because this is a general concept regard to a general kind of animal.

The abstract base class represents the generic, or abstract form of all the classes that are derived from it.

Defining Abstract Base Class and Pure Virtual Function

A class becomes an abstract base class when one or more of its member functions is a ***pure virtual function***. The syntax for defining the pure virtual member function is as follows:

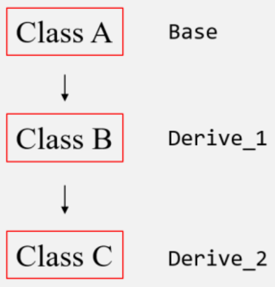
```
virtual void ShowInfo() const = 0;
```

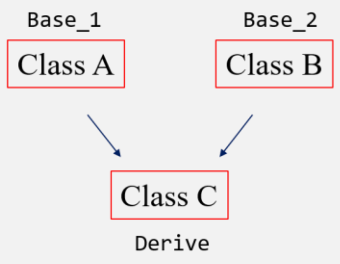
You add "`= 0`" at the end of the function header (after `const` specifier). The `= 0` notation indicates that `ShowInfo()` is a pure virtual function. **Pure virtual functions have no body, no definition in the abstract base class. They must be overridden in derived classes.** Because you can't instantiate a pure virtual function, the presence of a pure virtual function in a class prevents a program from instantiating the class. If you define an object of an abstract base class, the compiler will notify you that there is a pure virtual function in the class based on which you're trying to make an object.

Multiple Inheritance

Concept

Consider the following two inheritance scenario:





Chain inheritance

Multiple inheritance

These two ways can both combine classes: class C is the result of combining A and B, it has members from class A and class B. We've met chain inheritance before, now we focus on multiple inheritance.

In multiple inheritance, class C is directly derived from classes A and B and inherits the members of both. However, class A and class B has no interaction, i.e. neither class A nor class B inherits members from the other. Their members are only passed down to class C.

Syntax

Multiple inheritance declaration

The general format of the first line of a class declaration with multiple base classes is:

```
class DerivedClassName : AccessSpecification BaseClassName_1,
                  AccessSpecification BaseClassName_2 [, ...]
```

Notice that base classes are separated by a comma. The square brackets indicate that the list of base classes may be repeated, it is possible to have several base classes.

Take class A, B, C as an example, for multiple inheritance:

class C : public A, public B

Passing value to multiple base class constructor
When using multiple inheritance, the general format of a derived class's constructor header is:

DerivedClassName(ParameterList) : BaseClassName_1(ArgumentList),

BaseClassName_2(ArgumentList)[, ...]

All the parameter is still written in `DerivedClassName(ParameterList)`, then pass to the followed `BaseClassName(ArgumentList)`.

The order that the base class constructor calls appear in the list does not matter. They are always called in the order of inheritance. That is, they are always called in the order they are listed in the first line of the class declaration.

Distinguish Ambiguous Members
In multiple inheritance, two base classes may have member variables or functions of the same name. In situations like these, you should:

- redefine or override the member function, so the member function has a unique definition in the derived class
- use the scope resolution operator (::) to call the appropriate base class member function.
- use the scope resolution operator (::) to access the ambiguously named member variables of the correct base class.

Chapter 16. Exceptions, Templates, and the Standard Template Library (STL)

Exceptions

Concept
An exception is a value or an object that signals errors or unexpected events that occur while a program is running. It is not like the error signaling we did before (for example, print a message on the screen to let user know that there is something wrong). Exception can **let the program** know there are some errors or unexpected events happened. You can pre-define the ways of handling different exceptions.

Throwing an Exception
When the error occurs, an exception is "thrown". You do this by placing the `throw` keyword at the beginning of a line (following the `throw` keyword is an argument, which can be any value, this statement will be thrown). This line is known as the ***throw point***. If a `throw` statement is executed, control is passed to another part of the program known as an *exception handler*. When an exception is thrown by a function, the function aborts.

For example, the following code will throw a `char` variable when the condition is met:

```
  if (condition_C is true)
  {
    char error_code = ' C' ;
    throw error_code;
  }
```

If `throw error_code;` is executed, the program will go to exception handler to see if there is any protocol regarding a `char` variable type exception.

Handling an Exception (Exception handler)

You have to build a *try/catch* construct in the program to handle an exception. The general format of the try/catch construct is:

```
try
{
  // code here calls functions or object member
  // functions that might throw an exception
}

catch (ExceptionParameter)
{
  // code here handles the exception
}

// repeat as many catch blocks as needed
```

The first part of the construct is the *try block*. It starts with the keyword `try` and is followed by a block of code executing any statements that might directly or indirectly cause an exception to be thrown.

The second part of the construct is the *catch block*. It starts with the keyword `catch`, followed by a set of paretheses containing the definition of an exception parameter. If the thrown exception matches the exception parameter in one of the catch block, the statement in that catch block will be executed. After the catch block has finished, the program resumes with the first statement after the try/catch construct.

If no exception is thrown in the try block, all the catch blocks are to be skipped.

Exceptions That Is Not Caught

There are two possible ways for a thrown exception to go uncaught:

- the try/catch construct doesn't contain catch blocks with the matching exception parameter (an exception is thrown, but there is no catch block can "catch" it)
- the exception is thrown from outside a try block

In either case, the exception will cause the **entire** program to abort execution.

Object-Oriented Exception Handling with Classes

Exception throwing and handling can be implemented in object-oriented manner. Which means what you throw is not a single variable or a single left value, but an object of a class.

Take the `Rectangle` class as an example. The `Rectangle` class has two private member variables: width and length, which store the length info for the rectangle. The class has mutator that accept double argument and save the value to width and length. When the mutator receive a negative value, the class should throw a “singaling object” (the exception object), then abort the program. When you write the `Rectangle.h` file, you can define an exception class in the public section:

```
/* Exception class */
class NegativeSize
{ }; // empty class declaration
```

In this example, the `NegativeSize` class has no members. The only important part of the class is its name, which will be used in the exception-handling code.

In the function definition of mutator for width and length, you define the rules of exception (rules of when to throw an exception). As we discussed before, when the mutator receives a negative value, it is considered that an error occurred, and exception has to be thrown. The code for throwing a singaling object is as follows:

```
if (l < 0)
  throw NegativeSize();
```

The `throw` statement’s argument, `NegativeSize()`, causes an instance of the `NegativeSize` class to be created and thrown as an exception. After this `throw` action, the program knows an exception has been thrown, and will turn to the section that handles the exception.

Any code that uses the `Rectangle` class must simply have a catch block to handle the `NegativeSize` exceptions that the `Rectangle` class might throw. Following code shows the way of defining a catch block that catches an exception of a `NegativeSize` class type:

```
catch (Rectangle::NegativeSize)
{
  std::cout << " Error: value should be positive.\n " ;
}
```

You simply put the name of the `NegativeSize` class in the parameter parenthese of the `catch` statement. Because the `NegativeSize` class is declared inside the `Rectangle` class, we have to fully qualify the class name with the scope resolution operator. Notice that only the type name has been declared in the `catch` statement, we didn’t write things like:

```
catch (Rectangle::NegativeSize obj)
```

This is because we don't use any info from the thrown exception, we only need to know that an exception of type `Rectangle::NegativeSize` has been thrown. Thus, the `catch` statement only needs to specify the type of exception it handles.

Multiple Exceptions

C++ allows you to throw and catch multiple exceptions. The only requirement is that each different exception must be of a **different type**. You have to also build a separate catch block for each type of exception that may be thrown in the try block.

When an exception is thrown by code in the try block, C++ searches the try/catch construct for a `catch` statement that can handle the exception. If the construct contains a `catch` statement whose parameter is the same as the type of the exception, control of the program is passed to the catch block.

Extracting Data from the Exception Class

Take the object-oriented exception handling as an example. You can define a non-empty class as the class being thrown. It can contain variable that hold information which you want the class carry when being thrown to the exception handling zone of the program. Make sure you define appropriate accessor to access the information stored in the thrown exception object. Also, when you throw the object, make sure you store the necessary information in it.

Take the exception class discussed above as an example:

```
/* Exception class */
class NegativeSizeL
{
private:
  double value_; // to hold the negative value
public:
  /* Constructor */
  // hold the negative value
  NegativeSizeL(double v)
  {
    value_ = v;
  }

  /* Accessor */
  double value() const
  { return value_; }
 };
```

Unwinding the Stack

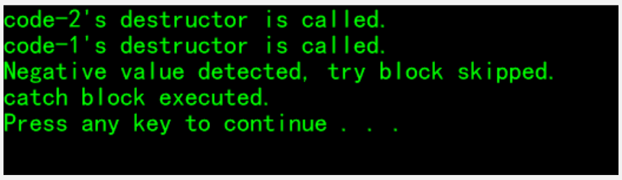
Once an exception has been thrown, the program cannot jump back to the throw point. The function that executes a `throw` statement will immediately terminate. If that function was called by another function, and the exception is not caught, then the calling function will terminate as well. This process, known as *unwinding the stack*, continues for the entire chain of nested function calls, from the throw point, all the way back to the try block (all the way back to where can catch the exception). If there is no try/catch block that can catch this rampaging exception, then the main function (`int main()`) will be aborted.

If an exception is thrown by the member function of a class object, then the class destructor is called. If statements in the try block or branching from the try block created any other objects, their destructors will be called as well. This is because if an exception is thrown in side a block, program will exit that block and go to the exception handling area. All objects defined in that block will be "destructed".

PS: if an exception is thrown in the constructor of a class (when defining an instance of that class), then the destructor of that instance will not be called. Because the program exited the constructor before it is finished, so the definition of this object is not considered finished, thus no destructor of this object being called. Example:

```
#include <iostream>


class Try
{
private:
 int code_;
public:
 Try(int num)
 {
  code_ = num;
  if (num < 0) // might throw an exception in constructor
   throw num;
 }


 ~Try()
 {
    std::cout << " code- " << code_ << " ′ s destructor is called.\n " ;
  }
};

int main()
{
  try
  {
    Try t1(1);
    Try t2(2);
    Try t3(-1); // exception will be thrown in constructor
  }

  catch (int)
  {
    std::cout << " Negative value detected, try block skipped.\n " ;
  }

  std::cout << " catch block executed.\n " ;

  return 0;
}
```

The result of the above code is:



```
code-2's destructor is called.
code-1's destructor is called.
Negative value detected, try block skipped.
catch block executed.
Press any key to continue . . .
```

This demonstrates that the destructor in t3 was not triggered. (t3’s constructor threw the exception).

Rethrowing an Exception

It is possible for try blocks to be nested. With nested try blocks, it is sometimes necessary for an inner exception handler to pass an exception it received to an outer exception handler. Sometimes, both an inner and an outer catch block must perform operations when a particular exception is thrown. These situations require that the inner catch block *rethrow* the exception so the outer catch block has a chance to catch it.

A catch block can rethrow an exception with the `throw;` statement.

Handling the `bad_alloc` Exception

When the `new` operator fails to allocate memory, a `bad_alloc` exception is thrown. The `bad_alloc` exception type is defined in the `new` header file, so any program that attempts to catch this exception should have the following directive:

#include <new>

Be advised that the `bad_alloc` exception is in the `std` namespace.

Below is the general format of a try/catch construct that catches the `bad_alloc` exception:

```
try
{
  double* d_ptr;
  d_ptr = new double;
}

catch (std::bad_alloc)
{
  std::cout << " Memory allocation failed.\n" ;
}
```

Function Templates

Concept

A function template is a "generic" function that can work with any data type. The programmer writes the specification of the function, but substitutes parameters for data types. When the compiler encounters a call to the function, it generates code to handle the specific data type(s) used in the call.

A similar concept is overloaded functions, which all have the same function name, only the parameters are different (parameter data types and number of the parameters). For overloaded function, however, each of them must still be written individually.

When the only difference between a set of functions are the data type of the parameter and the return values, it is more convenient to write a *function template* than an overloaded function. Function templates allow you to write a single function definition that works with many different data types, instead of writing a separate function for each data type used.

Syntax

A function template is not an actual function, but a mold the compiler uses to generate one or more specific functions.

When writing a function template, you do not have to specify actual types for the parameters, return values or local variables. Instead, you use a *type parameter* to specify a generic data type. When the compiler encounters a call to the function, **it examines the data types of its arguments** and generates the function code that will work with those data types (the generated code is known as a *template function.*)

Take square() function as an example. It accepts a number, and will return the square of that number. Following is a function template for this square() function:

```
template <class T>
T square(T number)
{
  return number*number;
}
```

Since the compiler must already know the template's contents when it encounters a call to the template function, therefore, templates should be placed near the top of the program or in a header file. (A function template is merely the specification of a function and by itself does not cause memory to be used. An actual instance of the function is created in memory when the compiler encounters a call to the template function)

The beginning of a function template is marked by a `template` keyword. This `template` keyword is required for every function template you write. Next is a set of angled brackets that contain one or more generic data types used in the template. A generic data type starts with the key word `class` followed by a parameter name that stands for the data type. The example just given only has one, which is named `T`. Multiple generic data types will be separated by commas (`class` key word is needed for each generic data type). Each type parameters declared in the template prefix must appear at least once in the function parameter list, and must be used somewhere in the template definition.

After this, the function definition is written as usual, except the type parameters are substituted for the actual data type names. Let's take a look on the function header:

```
T square(T number)
```

`T` is the type parameter, or generic data type. The header defines `square` as a function that returns a value of type `T` and uses a parameter (whose type is also `T`), the parameter's name is `number`. The compiler examines each call to `square` and fills in the appropriate data type for `T`. For example, the following call uses an `int` argument:

```
int y, x = 4;
y = square(x);
```

This code contains a call to `square()`, the compiler will exam the data type of `x`, then decide that the generic data type for the parameter of `square()` defined in its templates should be of `int`. Thus, following code will be generated for the `square()` function called here:

```
int square(int number)
```

```
{
  return number * number;
}
```

If the data type of y and x are `double`, then following code will be generated:

```
double square(double number)
{
  return number * number;
}
```

Using Operators in Function Templates

The type of variables are not defined in template, i.e. no specific data type is declared for these variables. It is ok if operators are used on those variables when they are of a primitive data type such as `int`, `double`. However, when a user-defined class object is passed to the function, the operators are not available unless the class contain code for overloaded operators. Otherwise, the compiler will generate a function with an error (undefined operators used).

For this case, always remember that a class object passed to a function template must support all the operations that function will perform on the object.

Overloading with Function Templates

Function templates may be overloaded by having different parameter lists (not parameter types, because the type of the parameter is just generic data type. The only way to make the function signature different is to change the number of parameter). You should write `template` key word for each overloaded templates.

For example:

```
template <class T>
T Sum(T n1, T n2)
{
  return n1 + n2;
}

template <class T>
T Sum(T n1, T n2, T n3)
{
  return n1 + n2 + n3;
}
```

There are other ways to perform overloading with function templates as well. For example, a program might contain a regular (nontemplate) version of a function as well as a template version. As long as each has a different parameter list, they can coexist as overloaded functions.

Convert Existing Function into Template

It is easier to convert an exiting function into a template than to write a template from scratch. You can design a function template by writing it as a regular function in the beginning. After the function is properly tested and debugged, you can convert it into a template in following steps:

- Check how many generic data types you need, each data type is counted as one generic data type;
- Add the template prefix header, including all generic data types you need
- Replace the specific reference data type into generic data type

Class Templates

Concept

Templates may also be used to create generic classes and abstract data types. Class templates allow you to create one general version of a class without having to duplicate code to handle multiple classese with the only difference be the data types.

When do you need to use class templates? If your data has different types, and your classes that handle these data have the same logic, only the data type is different. Then you can define a generic class (which will not bother the specific data type, but will focus on the core logic), then use this generic class to create different class (which uses different types of data to achieve the same purpose).

Syntax

Declaring a class template is very similar to declaring a function template.

The header file for an example class is shown below:

```
#ifndef CLASSNAME_H
#define CLASSNAME_H

template <class T, class S>
class ClassName
{
private:
  T var1;
  S var2;
  // variable definition goes here
public:
  // class member function definition goes here
  // ...
};

#endif
```

Again, you place the `template` keyword before the class declaration (but inside the declaration guard). Similarly, you don't use the specific data type when you are defining the variable or writing the class member function, you use the generic data type you declared in the parenthese of the template prefix (in this example, the generic data types are: `class T, class S`). Other than that, the process of building a class is the same.

Now let's take a look on how to define the member function. Normally, we give the definition in a separate .cpp file. However, for the class template, you can't put the definition in the .cpp file (see the following link for reason: https://stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file). A easy way is to put the definition in the header file:

- declare all member function inline
- include the full definition of the member function in the header file (in this way, you still need to use the scope resolution operator, if you place the function definition outside of the class).

Take the second method as an example:

For every member function definition you write, you have to include the template prefix before the function header:

```
template <class T, class S>
```

Then, normally, you write the header via scope resolution operator like:

```
ClassName :: ClassName()
```

However, when writing function definition for class template, you have to include the generic argument list. For example, you write the header for the constructor as follows:

```
ClassName<T,S> :: ClassName()
```

Otherwise, the compiler will notify that: argument list for class template "ClassName" is missing.

Then you define the member function in the same way you do for normal class member function, except using the generic data type `T` and `S`.

Defining Objects of the Class Template

Class template objects are defined like object of ordinary classes, with one small difference: you have to specify the data type you wish to pass to the type parameter in the class template. For example, if the two data type you want to use to define the class is `int` and `char`, then you write the definition of the object as follows:

```
ClassName<int, char> obj_name;
```

Actually, the program is not simpliy defining an object, it first defining a class, then it define an object based on this class. You need to provide the specific data type when defining the class.

(this format is the same as the vector defining format: `std::vector<int> num;`)

Example: Making a SimpleVector class template with range check [ ]

The vector is like an array, however, the size of the vector can be changed when new element is added or deleted (by the push_back() and pop_back() member function). This is different from the traditional array.

This example will build a simple vector class, which has the characteristics of the vector type: it can hold an array of data, its size can be changed, it can report its size, it has range check function in the [ ] operator.

(Check program example 16-11 for more details)

Inherited Class Templates

You can define a derived class template from a base class template.

Normally, the inheritance syntax is:

```
class Derived : public Base
{
  // class definition
  // goese there
};
```

For class template, there is some minor differences:

```
template <class T> // the template prefix
class Derived : public Base<T>
{
  // class definition
  // goese there
};
```

For inheriting class template, you have to add the template prefix before the class header. And for each use of `Base` class name, you have to include the generic data type parameter list as well: `Base<T>` (i.e., replace `Base` by `Base<T>`). This is because `Base` is a class template, not an actual class. The type parameter must be passed to it.

Example: build a `SearchableVector` class, which is derived from `SimpleVector`.

(PS: in the definition of a const labeled function, you can only use const function or operator.)

The above section shows how to build a derived class template from a base class template. In addition, class templates may be derived from ordinary classes, and ordinary classes may be derived from class templates.

Specialized Templates

After creating a class template, it can work with different data types. However, if you want to use a different template for a certain data type, you can declare a specialized template that works specifically for the chosen data type.

In the declaration, the actual data type is used instead of a type parameter. For example, the declaration of a specialized version of the `SimpleVector` class might start like this:

```
  template <~~class T~~> // should be template<>, you are defining the specialized template, so the generic data type is not necessary. But you have to provide template<> to let the program know that this is a specialized template of a template
  class SimpleVector<char *>
```

In this way, the compiler would know that this version of the `SimpleVector` class is intended for the `char *` data type. Anytime an object is defined of the type `SimpleVector<char *>`, the compiler will use this template to generate the code for the class and build an objct.

When you write the definition for members of this specialized template, you don't add `template<>`, because you have declared: `class SimpleVector<char *>`, this means `SimpleVector<char *>` is not a class template, but an ordinary class (it is a specialized "template").

Alternatively, if you want to just explicitly specialize specific member in `SimpleVector<T>`, you can just define an explicit version of it (when you are writing the definition for its members, outside of the class).

```
 template<>
 void SimpleVector<char*>::Print();
```

The program will call this function when the generic data type is `char*`.

Details and examples: https://stackoverflow.com/questions/14178440/class-template-state-data-member-not-an-entity-that-can-be-explicitly-specializ

Standard Template Library (STL)

Concept

The Standard Template Library contains many generic templates for implementing abstract data types (ADTs) and algorithms. The most important data structures in the STL are *containers* and *iterators*.

Abstract Data Types (ADT)

Containers

- Container: a class that stores data and organizes it in some fathion.
  - Sequence container: organizes data in a sequential fashion similar to an array
    - ⇨ `vector`
    - ⇨ `deque`
    - ⇨ `list`
  - Associative container: organizes data with keys, which allow rapid, random access to elements stored in the container
    - ⇨ `set`
    - ⇨ `multiset`
    - ⇨ `map`
    - ⇨ `multimap`

Description of different sequence containers.

Sequence container	Description	Performance
`vector`	Expandable array; Add or remove value from the end or middle of a vector;	Efficient: add value to its end; Inefficient: insert value to other position
`deque`	Like a vector; Can also add or remove values from the front	Efficient: add value to its front and end Inefficient: insert value to other position
`list`	A doubly linked list of data elements; Insert or remove values from any position	Efficient: insert values anywhere in its sequence; DO NOT provide random access

Description of different associative containers

Associative container	Description	Performance
`set`	Stores a set of keys; No duplicate values are allowed;	
`multiset`	Stores a set of keys; Duplicate values are allowed;	
`map`	Maps a set of keys to data elements; Only one key per data element is allowed; No duplicate are allowed;	
`multimap`	Maps a set of keys to data elements; Multiple keys per data element are allowed; Duplicates are allowed;	

Syntax:

When you define a container, you have to include the corresponding header file (for example, if you want to use vector, you have to include <vector>).

The type name of the container is in std namespace, so you have to add std:: in front of the container's type name when you define it.

Take vector as an example:

```
std::vector<char> list;
```

A set of angled bracket is followed by the container's name, you have to specify the data type of the element in the container. After that, you input a space, and then the name for the container.

Iterators

Elements in a container can be accessed by the container's member functions (for example, the operator[]() function). On the other hand, iterators can also be used to access and manipulate container elements.

The characteristic and type of iterator is as follows:

- Iterator: an object that behaves like a pointer (generalizations of pointers), which is used to access the individual data elements stored in a container.

  Iterators are associated with containers. The type of container you have determines the type of iterator you use. For example, `vectors` and `deques` require random-access iterators, while `lists`, `sets`, `multisets`, `maps` and `multimaps` require bidirectional iterators.
  - Forward: can only move forward in a container (uses the ++ operator)
  - Bidirectional: can move forward or backward in a container (uses the ++ and – operator)
  - Random-access: can move forward and backward, and can jump to a specific data element in a container
  - Input: can be used with an input stream to read data from an input device or a file
  - Output: can be used with an output stream to write data to an output device or a file

Syntax:

Defining an iterator is closely related to define a container (because iterators are associated with containers). Take an iterator for a vector as an example. Following code shows the definition of an iterator for a vector (of `int` type):

```
std::vector<int>::iterator iter;
```

`iter` act like a pointer, which can point to an element in a `vector<int>` type entity.

If you want iter to point to the beginning of a vector, you can assign an iterator pointing to the beginning of that vector to iter:

```
std::vector<int> num;
std::vector<int>::iterator iter;
iter = num.begin();
```

The compiler will automatically choose the right type iterator (for vector, the iterator is of random-access type).

On the third line, the container's begin() member function has been used, this member function will return an iterator pointing to the beginning of the vector. The above statement causes iter to point to the first element in the num.

Following statement causes iter to point to the last element in the num:

```
iter = num.end() - 1;
```

Pay attention that the member function end() returns an iterator referring to the *past-the-end* element in the vector container. The *past-the-end* element is the theoretical element that would follow the last element in the vector. It does not point to any element, and thus shall not be dereferenced.

begin() and end() are often used together to specify a range including all the element in a container.

The memory position iter points to can be changed by ++ or -- operator, for example, iter++ causes iter to point to the next element in the num vector.

To access the value stored in the memory position pointed by iter, you can use the * operator to dereference the iterator, just like a pointer.

Algorithms

The algorithms provided by the STL are implemented as function templates. Specific algorithms are sealed in those function templates, which can be used to perform various operations on elements of containers.

To use these algorithms, you have to #include <algorithm>.

binary_search

(detailed: http://www.cplusplus.com/reference/algorithm/binary_search/)

Performs a binary search for an object (the keyword, returns true if the object is found.

Example: std::binary_search(iter1, iter2, value);

In this statement, iter1 and iter2 point to elements in a container (iter1 points to the first element in the range, iter2 points to the last element in the range).

The function will perform a binary search on the range of [iter1, iter2), to see if it contains value. It returns true if value is found, false if not found.

count

(detailed: http://www.cplusplus.com/reference/algorithm/count/)

Returns the number of times a value appears in a range.

Example: num = std::count(iter1, iter2, value);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

The function will count the number of elements that is equal to `value` on the range of [`iter1`, `iter2`), and then return it. The return type is a signed integral type.

find

(detailed: http://www.cplusplus.com/reference/algorithm/find/)

Finds the first element in a container that matches a value and returns an iterator to it.

Example: iter3 = std::find(iter1, iter2, value);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

The function will search the range of [`iter1`, `iter2`), to see if it contains `value`. If `value` is found, the function returns an iterator to the first element containing it. It no matches found, the function returns `iter2`, which points to the first element that following the last element in the range

for_each

(detailed: http://www.cplusplus.com/reference/algorithm/for_each/)

Executes a function for each element in a container.

Example: std::for_each(iter1, iter2, func);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

The third parameter, func, is the name of a unary function that accepts an element in the range as argument. The statement calls the function func for each element in the range, passing the element as an argument. Range: [`iter1`, `iter2`).

The function's return value, if any, is ignored.

max_element

(detailed: http://www.cplusplus.com/reference/algorithm/max_element/)

Returns an iterator to the largest object in a range.

Example: iter3 = std::max_element(iter1, iter2);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

The statement returns an iterator to the element containing the largest value in the range: [`iter1`, `iter2`).

min_element
(detailed: http://www.cplusplus.com/reference/algorithm/min_element/)

Returns an iterator to the smallest object in a range.

Example: iter3 = std::min_element(iter1, iter2);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

The statement returns an iterator to the element containing the smallest value in the range: [`iter1`, `iter2`).

random_shuffle
(detailed: http://www.cplusplus.com/reference/algorithm/random_shuffle/)

Rearranges the elements in the range randomly.

Example: std::random_shuffle(iter1, iter2);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

This statement will randomly reorders the elements in the range: [`iter1`, `iter2`). (Pay attention, not including element pointed by `iter2`).

Normally, the program will use the default random generator (same as the default rand()), i.e. a set of fixed ordered random number. This means each time, the sort order is the same.

However, you can seed the random generator by running:

std::srand(unsigned(std::time(0)));

The random generator will be seed with `time(0)`, and the random order will be different.

sort
(detailed: http://www.cplusplus.com/reference/algorithm/sort/)

Sorts the elements in the range in ascending order.

Example: std::sort(iter1, iter2);

In this statement, `iter1` and `iter2` point to elements in a container (`iter1` points to the first element in the range, `iter2` points to the last element in the range).

This statement will sort the element in the range in ascending order, range: [`iter1`, `iter2`).

Chapter 17. Linked List

Introduction to the Linked List

Concept

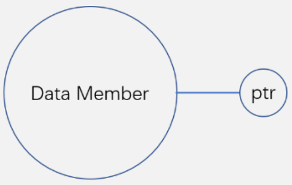
Suppose you design a program that will dynamically allocate data structures, and you want to manage them. You can use linked list to do this job.

Dynamically allocated data structures may be linked together in memory to form a chain (linked list), each dynamically allocated data structure in that linked list is called ***node***. A linked list can easily grow or shrink in size. Also, compared with vector, it is very efficient for a linked list to insert an element into the middle of the list, or delete an element that is in the middle. For a vector, to access a middle element you have to move all the element after that position, while for a linked list, none of the other nodes have to be moved.

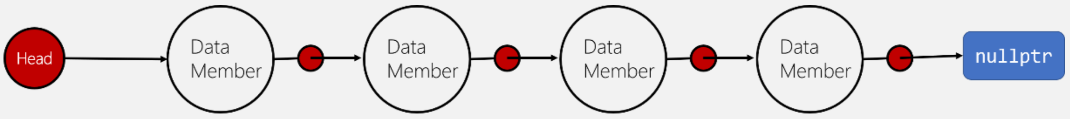
Composition of a Linked List

A linked list is a series of connected *nodes*, where each node is a data structure.

By saying "connected", it means that there is a pointer in each node that points to next node, thus the term "connected". Actually, a *node* can be depicted as follows:



Data members hold the data that this node contain (one or more members). In addition to the data, each node contains a pointer, which can point to another node (the next node). A linked list is formed when each node points to the next node, this is depicted below:



This list contains four nodes, plus a pointer known as the *list head*. This head simplily points to the first node in the list. Each node, in turn, points to the next node in the list. The last node in the list will point to the null address, this can signify the end of a list.

Declarations

In this section, you'll learn how to create a linked list in C++. The first step is to build a data type that can serve as the node of the linked list. As you read in the above section, the node for a linked list should have at least two sections: data member to hold data and a pointer to point to next node. The following structure could be used to create a list where each node holds a double value:

```
struct ListNode
{
  double value; // hold the value in one node
  ListNode* next; // a pointer of the same type, to point to the next node
  };
```

This `ListNode` structure contains a pointer to an object of the same type as being declared, it is known as a *self-referential data structure*. This structure makes it possible to create nodes that point to other nodes of the same type.

The next step is to define a pointer to serve as the list head (to point to the first node):

```
 ListNode* head;
```

You have to initialize `head` to `nullptr`, this indicates that `head` is pointing to an empty list.

A linked list can have many operations on its nodes, the general operation includes:

- Append a node to the end
- Display each node in the linked list
- Insert a node (based on descending or ascending order)
- Search and delete a node from the list (also release memory space of the deleted node)
- Destroy the list (release memory spaces for each node)

These operations will be covered in the next section. A class `NumberList` will be used as an example to give the details of the different algorithms. The class specification is as follows:

```
#ifndef NUMBERLIST_H
#define NUMBERLIST_H
#include <iostream>

class NumberList
{
private:
  // declaring a structure for the list
  struct ListNode
  {
    double value; // hold the value in one node
    ListNode* next; // a pointer of the same type, to point to the next node
  };

  // define a list head pointer
  ListNode* head;

public:
  /* Constructor */
  // default constructor
  // the head will point to nullptr, this is an empty linked list
  NumberList()
  {
    head = nullptr;
  }

  /* Destructor */
  // deletes all nodes of the linked list (list head: head)
  ~NumberList();

  /* Linked list operations */
  void AppendNode(double num);
  void InsertNode(double num);
  void DeleteNode(double num);
  void DisplayList() const;
};


#endif
```

Linked List Operation Algorithms

Append a Node

To append a node means you add a node to the end of the list.

The steps are as follows:

- Dynamically allocate a node data entity (first define a pointer to node data type: new_node)
- Store data into the node
- Check if the list is empty, if so, make the new node the first node `(head -> new_node, new_node->next = nullptr)`
- If not, traverse the list to find the last node:

```
  // find the last node
  node_ptr = head;
  while (node_ptr->next != nullptr)
    node_ptr = node_ptr->next;
```

- Insert new_node to the back of the last node:

```
node_ptr->next = new_node;
new_node->next = nullptr;
```

Display a Node

You need a pointer to traverse the linked list. For each stay, you display the information you wish to display.

The steps are as follows:

- Define a pointer to traverse: `node_ptr`
- Assign `head` to `node_ptr`
- Check if the list is empty: `node_ptr == nullptr?`
- If not, display the content in `node_ptr`, then move to next node: assign `node_ptr->next` to `node_ptr`
- Repeat the above process until `node_ptr->next` is `nullptr`

Inserting a Node

Inserting a node into the list. Suppose the values in a list are sorted. When you insert a new node, you have to insert it in the proper position. This means that you have to:

1. Find out the sorting rule used by this specific linked list
2. Find the proper position of the `new_node`
3. Insert the `new_node` into the right position

The steps are as follows:

- Define a pointer `new_node`, and allocate a new node variable
- Store value to `*new_node`
- Check if the list is empty, if so, store `*new_node` to the first node; if not:
- Check if the `head->next` is `nullptr`. If so, insert `*new_node` into the list in the default sorting mode (ascending or descending). You can compare `head->value` and `new_node->value` to decide the proper position to insert. If not, the list has at least two nodes, which means there is a pre-defined sorting order. You have to find out the sorting order.
- Define an integer variable: `sort_mode`, which can be used to hold sorting mode. You can define the corresponding integer value to represent the different sorting order. For example, 1: ascending; 0: all equal; -1: descending; 2: sorting mode not determined yet. You should initialize this variable with the code that represent "sorting mode not determined" state (`sort_mode = 2;`)
- Define two "navigation pointers": `ptr_1, ptr_2;` initialize them with:
  ```
  ptr_1 = head;
  ptr_2 = head->next;
  ```
- Use a while loop to find out the sorting order of the list: `while (sort_mode == 2)`
- If `ptr_1->value > ptr_2->value`: this list is in descending order, assign -1 to `sort_mode`, then end loop
- If `ptr_1->value < ptr_2->value`: this list is in descending order, assign 1 to `sort_mode`, then end loop
- If `ptr_1->value == ptr_2`: can't tell the sorting order of this list. Check if `ptr_2` is the end of the loop. If so, then all element is equal in this list, assign 0 to `sort_mode`. If not, `ptr_1 = ptr_1->next; ptr_2 = ptr_2->next;` (move two navigation pointers to next position, and start the loop again)
- After the above while loop, `sort_mode` contains the sorting order code. Now move `ptr_1` and `ptr_2` back to beginning:

  ptr_1 = nullptr;
  ptr_2 = head;
  The next step will be finding the right positi on for the `*new_node`.
- `switch (sort_mode)`
- `case 1: sort_mode == 1;` The list is ascending order. The two navigation pointers should keep moving forward until either of the following condition is true: 1, `ptr_2 ==`

nullptr (marks the end of the list; 2, ptr_2->value < new_node->value; (marks the right position for *new_node is after ptr_2). Once the condition met, you have to first check if ptr_1 == nullptr. If so, it means *new_node should be the new first node. If not, insert the *new_node between ptr_1 and ptr_2.

- case -1: sort_mode == -1; The list is descending order. The two navigation pointers should keep moving forward until either of the following condition is true: 1, ptr_2 == nullptr (marks the end of the list; 2, ptr_2->value > new_node->value; (marks the right position for *new_node is after ptr_2). Once the condition met, you have to first check if ptr_1 == nullptr. If so, it means *new_node should be the new first node. If not, insert the *new_node between ptr_1 and ptr_2.
- case 0: sort_mode == 0; All nodes in the list are equal. You can insert *new_node in the default sorting order. For example, if the default sorting order is ascending, then you compare head->value and new_node->value. If head->value >= new_mode->value, insert *new_node to the first node. Otherwise, append *new_node to the list. Same idea for default descending mode.

The insert member function can be called each time when you add a new node to the list. This can keep the linked list in order (ascending or descending).

Deleting a Node

To delete a node, you first search the list to find out the target node, then you delete it from the list and release the memory space it occupies.

Steps:

- Check if the list is empty, if so, do nothing (return)
- Define two pointers: ptr_1 and ptr_2, to navigate the list.
- Check if the first node is the target. If so:

```
ptr_1 = head;
 if (head->value == num)
 {
  head = head->next;
  delete ptr_1; // ptr_1 is pointing to the original head
  return;
 }
```

- Check node in the middle. Use a while loop to move the two navigation pointer forward. The two navigation pointers should keep moving forward until either of the following condition is true: 1, ptr_2 == nullptr (marks the end of the list; 2, ptr_2->value == num; (marks ptr_2 is pointing to the target node). Once any condition is met, the loop is broken and proceed to next step.
- Check if ptr_2 is the nullptr or not. If so, it means no match found in the list. Otherwise, *ptr_2 is the target node, delete it from list and release the memory occupied by it:

```
if (ptr)
{
  ptr_1->next = ptr_2->next;
  delete ptr_2;
  return;
}
```

Destroying the List

To destroy the list, each node should be released by `delete`, because they are all dynamically allocated entity. For this example, the destructor of the `NumberList` class should release all nodes. This is done node by node.

Steps:

- Check the list is empty or not. If so, no operation needed. If not:
- Define two navigation pointer: `ptr_1`, `ptr_2`, initialize them: `ptr_1 = nullptr; ptr_2 = head;`
- Make `ptr_1` point to the node which is now pointed by `ptr_2`: `ptr_1 = ptr_2;` Then move `ptr_2` to next node: `ptr_2 = ptr_2->next;` Then delete the node that is pointed by `ptr_1` (this is the node that is previously pointed by `ptr_2`). This loop is executed as long as `ptr_2` is not pointing to nullptr. If `ptr_2` is pointing to null, it means the end of the list has been reached, and now `ptr_1` is pointing to the last node. So, after the loop, you delete the last node by executing: `delete ptr_1;`
- These code should be put in the destructor section so the list can be destroyed properly

Linked List Template

In the above example (`NumberList` class), a linked list that stores double type data has been created. Actually you can create a template, so the linked list can store any data type you want to.

It should be noted that, the data type you choose when you making the linked list should support the operators used in the linked-list member functions. For example, in the `NumberList` class, you used:

- relational operators: ==, !=, <, >, <=, >=
- << operator (to displate node)

You can create a linked list of the user defined data type as long as it overloads these operators.

Using a Class Node Type

In the above example (`NumberList` class), the data type for node is a structure. You can also use a class or a class template as the node's data type. The class template is separate from the `NumberList` class that are using it. You can define it in `NumberList` class or outside of `NumberList` class. Its two separate entities.

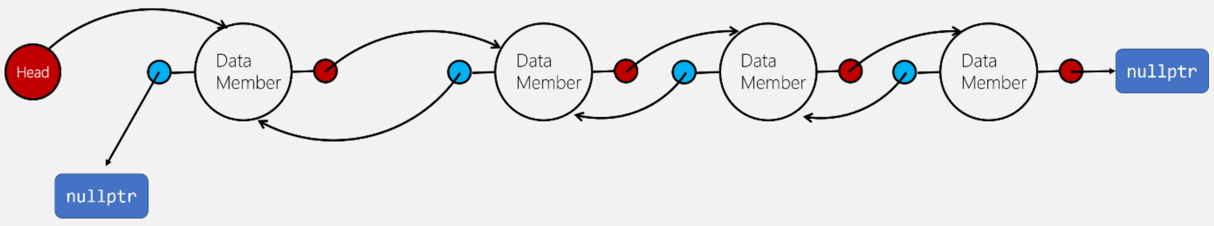
Also, you can declare the member variable of the node class as public. It is more convenient because the `NumberList` class needs to access the member variable of the node class in linked list operations.

Variations of the Linked List

In general, the idea of linked list is to link the dynamically allocated data structures (entities) together. The linked list we discussed above is an example of linking the dynamically allocated data structures (entities) together. It is a kind of singly linked lists. There are other ways of linking data structures (entities) together.

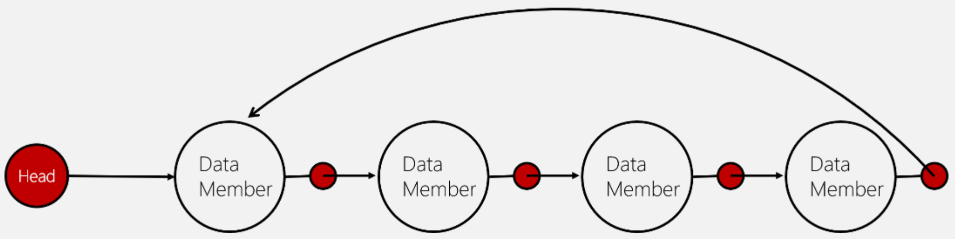
Doubly Linked List

The node in the doubly linked list is linking not only to the next node, but also the previous one. The node of such linked list has two pointers, one points to the next node, the other points to the previous node:



Circularly Linked List

In circularly linked list, the last node doesn't point to nullptr. Instead, it points to the first node:



The STL `list` Container

Concept

The Standard Template Library provides a linked list container template (just like the vector container). It is a template version of a doubly linked list. STL lists can insert elements or add elements to their front quicker than vectors can because lists do not have to shift the other elements. Lists are also efficient at adding elements at their back because they have a built-in pointer to the last element in the list (no traverse required).

Member function of the list template

The details of each member function of list can be found here:

http://www.cplusplus.com/reference/list/list/insert/

Following member functions and their explanations are just a simple version.

list.back(), list.front()

The `back()` member function returns a reference to the last element in the list. It is a reference, so you can access the element directly.

The `front()` member function will return a reference to the first element in the list.

list.empty()

The `empty()` member function returns `true` if the list is empty, otherwise it returns false.

list.end(), list.begin()

The `end()` member function returns a bidirectional iterator to the past-the-end element of the list.

list.erase()

list.erase(iter)

Will remove the list element pointed by the iterator `iter`.

list.erase(first_iter, last_iter)

Will remove the list elements from `first_iter` to `last_iter`.

list.insert(iter, x)

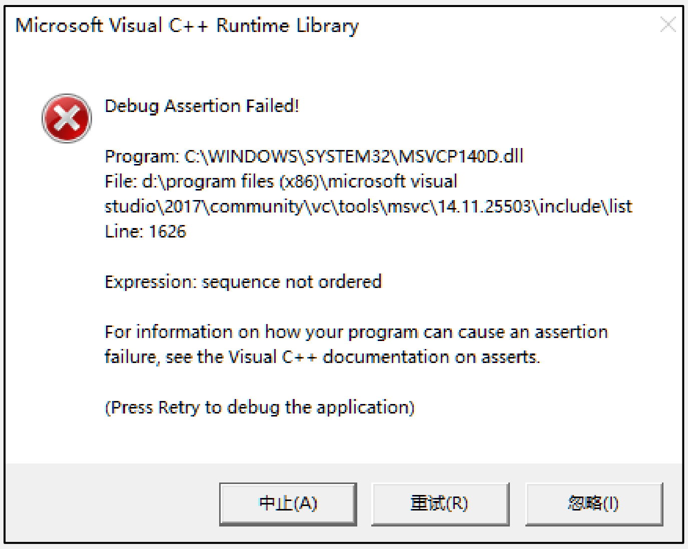
The `insert()` member function inserts an element into the list. It will insert element `x` just before the element pointed to by `iter`. `x` is of the same type with the node in the list.

Pay attention that there is no sorting contained in this insert member function.

list_1.merge(list_2)

The `merge()` member function inserts all the items in `list_2` into `list_1`. `list_2` will be empty after the execut of this function.

`list_1` and `list_2` is expected to be sorted before calling the `merge()` function. If not, following error will occur:


```
Microsoft Visual C++ Runtime Library

Debug Assertion Failed!

Program: C:\WINDOWS\SYSTEM32\MSVCP140D.dll
File: d:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.11.25503\include\list
Line: 1626

Expression: sequence not ordered

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)
```

Pay attention that, the sorting order of list_1 and list_2 should be the same. If one is ascending and the other is descending, the above error will also occur.

When `list_2` is inserted into `list_1`, the elements are inserted into their right position, so the resulting is also sorted.

This operation does not construct or destroy any element, they are just transferred.

list.pop_back()
Removes the last element of the list.

list.pop_front()
Removes the first element of the list.

list.push_back(x)
Inserts an element (x) to the end of the list.

list.push_front(x)
Inserts an element (x) to the beginning of the list.

list.reverse()
Reverses the order in which the elements appear in the list.

list.size()
Returns the number of elements in the list.

list_1.swap(list_2)
Swaps the elements stored in two list: `list_1` and `list_2`. Elements in `list_1` and `list_2` will be exchanged.

list.unique()
Removes all but the first element from every consecutive group of equal elements in the container. Notice that an element is removed from the list container if it compares equal to the element immediately preceding it. Thus, this function is especially useful for sorted lists.

Example:

before: 1, 1, 2, 3, 2, 4, 5

after: 1, 2, 3, 2, 4, 5

The redundant 2 was not removed because 3 != 2.

Chapter 18. Stacks and Queues

Stack ADT

Concept

- A stack is a data structure that holds a sequence of elements.
- A stack stores and retries elements in a last-in-first-out manner. When a program retrieves elements from a stack, the last element inserted into the stack is the first one retrieved.
- There are two types of stack data structure: static and dynamic.
  Static stacks: have fixed size and are implemented as arrays;
  Dynamic stacks: grow in size as needed and are implemented as linked lists;

Applications of Stacks

Stacks are useful data structures for algorithms that work first with the last saved element of a series. For example, you have a program that looks like this:

```
{
 {
  {
   {
    {
     // the innermost function is doing something here
     // local variable created here will be the last in the stack
     // but will be the first to out the stack
     // because it will be destroyed when coming out from this block
    }
   }
  }
 }
}
```

It is made of several nested functions. Each layer will have their own local variable defined, and return address saved. The computer system will use stacks while executing programs. The innermost function will access the stack in the last to store its local variable and return address. When the innermost function finishes, its local variable will be destroyed (before anyother function). A stack structure can make sure its local variable (which is the last-in element in stack) is accessed first, this is more efficient.

Stack Operations

push and pop: Manipulating elements in the stack

A stack has two primary operations to manipulate the element: push and pop.

Push: store a value to a stack (push onto the stack)

Pop: retrieves a value from the stack (the element will be removed from the stack)

is_full operation for a static stack

For a static stack (one with a fixed size), a Boolean is_full operation is needed to give people a chance to determine if the stack is full or not. It will return true if the stack is full, and false otherwise.

This operation is necessary to prevent stack overflow in the event that a push operation is attempted when all the stack's elements have values stored in them.

Dynamic stacks don't need this because it's size will change dynamically.

is_empty operation

The is_empty operation is a Boolean operation, it returns true when the stack is empty and false otherwise. This prevents an error from occurring when a pop operation is attempted on an empty stack.

Static Stack Class

The size of a static stack is pre-determined, not changeable.

Following table shows an example of a static stack class.

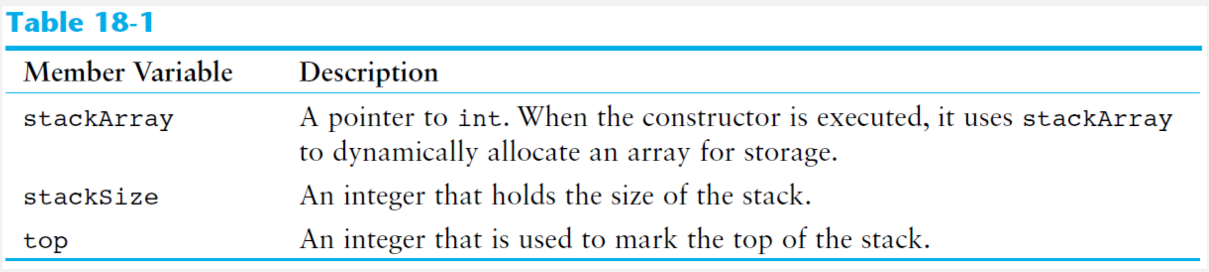


Table 18-1

Member Variable	Description
`stackArray`	A pointer to `int`. When the constructor is executed, it uses `stackArray` to dynamically allocate an array for storage.
`stackSize`	An integer that holds the size of the stack.
`top`	An integer that is used to mark the top of the stack.

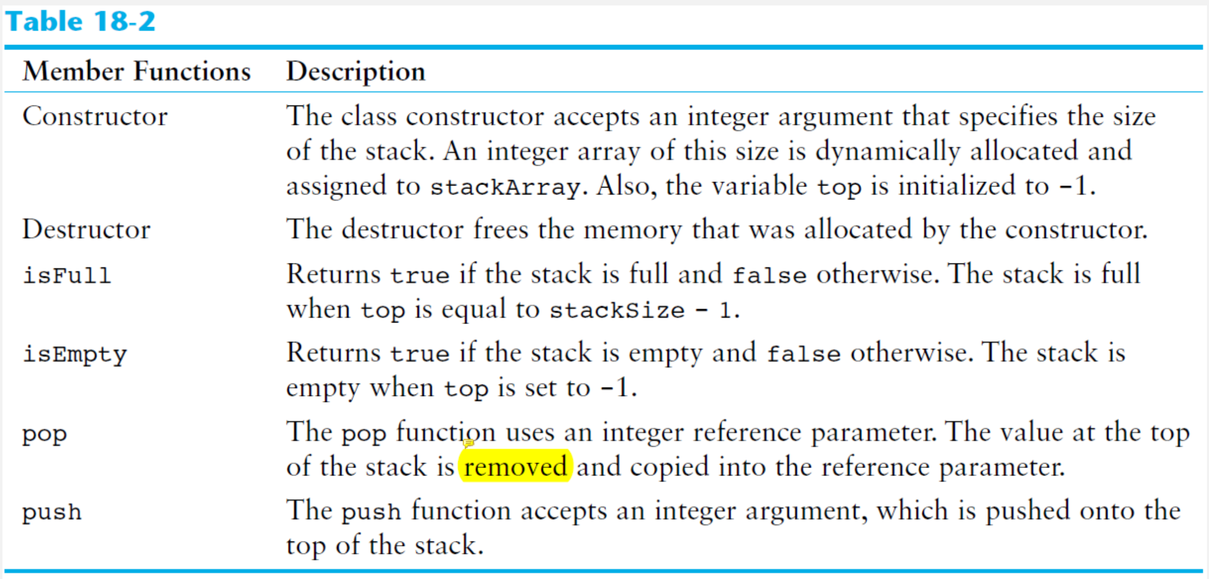


Table 18-2

Member Functions	Description
Constructor	The class constructor accepts an integer argument that specifies the size of the stack. An integer array of this size is dynamically allocated and assigned to `stackArray`. Also, the variable `top` is initialized to -1.
Destructor	The destructor frees the memory that was allocated by the constructor.
`isFull`	Returns `true` if the stack is full and `false` otherwise. The stack is full when `top` is equal to `stackSize - 1`.
`isEmpty`	Returns `true` if the stack is empty and `false` otherwise. The stack is empty when `top` is set to -1.
`pop`	The `pop` function uses an integer reference parameter. The value at the top of the stack is removed and copied into the reference parameter.
`push`	The `push` function accepts an integer argument, which is pushed onto the top of the stack.

The implementation of a static stack class template can be found in this chapter's folder (number_stack_template.h).

Dynamic Stack

<u>Concept</u>

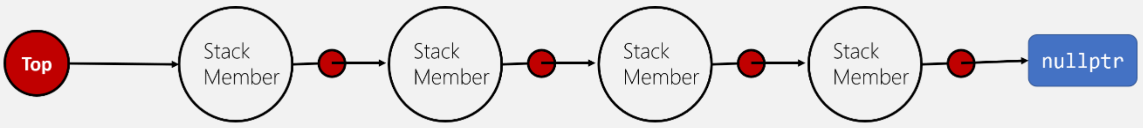
The size of a dynamic stack is not fixed, it will expand or shrink with each push or pop operation. A linked list-based stack offers two advantages over an array-based stack:

- No need to specify the starting size of the stack (start as an empty linked list)
- Won't be full (as long as the system has enough free memory)

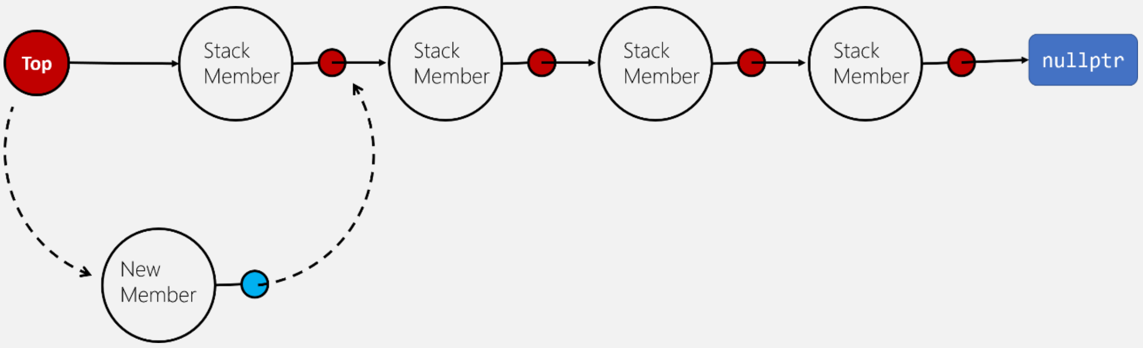
<u>Building of a dynamic stack class</u>

The dynamic stack class is built based on linked list rather than array. The basic component in the dynamic stack class should include:

- A stack node. It can be a class or structure. It should contain terms that hold data, and a pointer that point to the same type entity.
  It is like The declaration of this stack node can be either inside or outside of the dynamic stack class.
- A pointer to a stack node type (top). This pointer will always point to the "top" element in the dynamic stack. The push() and pop() member function should make sure that top always point to the "last accessed element". The structure of the dynamic stack should look like:



- Constructor. When a dynamic stack is declared, it should be an empty stack. This is represented by top pointing to nullptr.
- Destructor. When a dynamic stack is destroyed, all the dynamically allocated node should be released.
- Push() member function. This will add an element to the dynamic stack. This element should be added to the front of the linked list (so the last added element can be accessed first)



- Pop() member function. It accepts a reference of a variable (the data type of the variable should be the same as the variable that holds data in stack node) as the parameter. After popping an element, it should be deleted from the linked "stack list", after its content being "extracted" to the reference variable, it should be released by `delete`.
- IsEmpty() member function. The return type of this function is Boolean value. If the stack is empty (top pointing to nullptr), it should return true, otherwise, it should return false.

Refer to `dynamic_stack_template.h` for more details.

The STL `stack` Container

Details: http://www.cplusplus.com/reference/stack/stack/

Concept

The Standard Template Library offers a stack template, which may be implemented as a vector, a list or a deque. because the stack container is used to adapt these other containers, it is often referred to as a *container adapter*. Container adapters are classes that use an encapsulated object of a specific container class as its underlying container (such as vector, list or a deque), providing a specific set of member functions to access its elements which are stored in the underlying container, in a "stack way" (e.g. LIFO, last in first out).

To work in a "stack way", the container shall support the following operations:

`empty`

`size`

```
back
push_back
pop_back
```

Syntax

The header of the stack template can be written as:

```
template <class T, class Container = deque<T> >
class stack;
```

Notice that the default container is `deque`. When instantiate a specific stack, if no container class is specified, the standard container `deque` is used.

Below is three examples of how to define a stack of `ints`, implemented as a `vector`, a `list` and a `deque`:

```
#include <stack>

#include <vector> // for vector based stack

#include <list> // for list based stack


int main()

{

  std::stack<int> stack_1; // no container class specified, deque will be used

  std::stack<int, std::vector<int> > stack_2; // vector based stack

  std::stack<int, std::list<int> > stack_3; // list based stack

}
```

Member function

stack.empty()

Returns true if the stack is empty, false otherwise.

stack.pop()

Removes the element at the top of the stack, effectively reducing its size by one. This calls the removed element's destructor.

The remove is done by effectively calling the member function `pop_back` of the underlying container object.

stack.push(x)

Pushes an element with the value *x* onto the stack.

stack.size()
Returns the number of elements in the stack.

stack.top()
Returns a reference to the element at the top of the stack. This function is used to retrive the element at the top. This member function effectively calls member `back` of the underlying container object.

Queue ADT

Concept
A queue is a data structure that stores and retrieves items in a first-in-first-out manner (FIFO). This is the primary difference between queues and stacks: the way data elements are accessed.

Queues can be static and dynamic too, just as the static stack and dynamic stack. Dynamic queue has no fixed size, its size is flexible. You do not need to determine the size of the queue when you instantiate one.

Think of queue has a front and a rear. When an element is added to the queue, it is added to the rear (known as *enqueuing*). When an element is removed (accessed) from a queue, it is removed from the front (known as *dequeuing*).

Static Queue
Functions
A simple static queue template should support following functions:

- empty: to show if the queue is empty
- full: to show if the queue is full
- size: to show the current number of elements stored in the queue
- front: enable access to the next element of the queue. Next element is the “oldest” element stored in queue. Based on FIFO protocol, it should be popped first
- back: enable access to the last element of the queue. Last element is the “newest” element stored in queue. Based on FIFO protocol, it should be popped last.
- push: inserting element after the last element (newest) of the queue
- pop: remove the next element (the oldest) of the queue

Structure
The sketch for the static queue is as follows. Dynamically allocated array will be used to hold data of the queue. The size of the array is determined by the parameter passed to the constructor of the queue during instantiation.

It is a template, so don’t forget: `template <class T>`

Member variables:

- A pointer to `class T`, this is used to dynamically allocate the array
- An integer to hold the current number of elements in the queue
- An integer to hold the capacity of the queue, this is the parameter passed to the constructor of the queue during instantiation
- An integer to hold the subscript of the oldest element
- An integer to hold the subscript of the newst element

Member functions

- Constructor should accept an integer specifying the capacity of the queue, then allocate the memory dynamically; should initialize the member variable as well
- Should have a copy constructor. This is to prevent copying pointer from memberwise assignment
- Destructor should release the memory allocated to hold the array (the underlying container)
- Empty(): returns true if the queue is empty, otherwise returns false
- Full(): returns true if the queue is full, otherwise returns false
- Size(): returns the current number of element in the queue
- Capacity(): returns the total capacity of the queue
- Front(): returns reference to the next element of the queue
- Back(): returns reference to the newest element of the queue
- Push(): insert element after the newest element
- Pop(): remove the oldest element in queue

Dynamic Queue

Functions

A simple dynamic queue template should support following functions:

- empty: to show if the queue is empty
- size: to show the current number of elements stored in the queue
- front: enable access to the next element of the queue. Next element is the "oldest" element stored in queue. Based on FIFO protocol, it should be popped first
- back: enable access to the last element of the queue. Last element is the "newest" element stored in queue. Based on FIFO protocol, it should be popped last.
- push: inserting element after the last element (newest) of the queue
- pop: remove the next element (the oldest) of the queue

Structure

The sketch for the dynamic queue is as follows. Linked list structure will be used to hold data of the queue.

It is a template, so don't forget: `template <class T>`

Queue Node

Since linked list is the underlying container, the node of the linked list should be built. It contains the data-holding section and the pointer to point to the next node.

Member variables:

- front: A pointer to QueueNode. This pointer will always point to the next element of the queue, which is the oldest element in the queue.
- back: A pointer to QueueNode. This pointer will always point to the next element of the queue, which is the newest element in the queue (point to the rear of the queue).
- size_: An integer to hold the current number of elements in the queue

Member functions

- Constructor should initialize size_ to be zero. Also the front_ and back_ pointer should be initialized to `nullptr`.
- Should have a copy constructor. The copy constructor should replicate the target queue. Each node in the target node should be replicated by dynamically allocation and value assignment. A pointer to QueueNode should be used to navigate the whole list of the target.
- Destructor should release the memory allocated. Two pointers to QueueNode should be used to navigate through the linked list, delete each node's memory.
- Empty(): returns true if the queue is empty, otherwise returns false
- Size(): returns the current number of element in the queue
- Front(): returns reference to the data object stored in next element of the queue (notice that this is not the reference to the QueueNode, but reference to the data object stored in QueueNode).
- Back(): returns reference to the data object stored in the newest element of the queue
- Push(): insert QueueNode after the newest element
- Pop(): remove the oldest element in queue

The STL deque and queue Containers

The Standard Template Library provides two containers, `deque` and `queue`, for implementing queue-like data structures.

- deque: a double-ended queue. It is similar to a vector, but allows efficient access to values at both the front and the rear (vector is only efficient at accessing element at the end). Notice that deque does not provide the FIFO feature (it is not a container adapter)
- queue: a container adapter, similar with the stack ADT.

deque

Concept

details: http://www.cplusplus.com/reference/deque/deque/

You have to `#include <deque>` to use the `deque` ADT. The full name for deque is double-ended-queue. It is sequence container with dynamic sizes that can be expanded or contracted on both ends (front and back). Remember that the vector container is only efficient at deleting or inserting new element at the end. deque is efficient at deleting and inserting new element at the end and also the beginning. Elements in deque can also be accessed by the [ ] operator.

Both vectors and deques provide a very similar interface and can be used for similar purposes. However, vector and deque is working in quite different ways:

- vector uses a single array that needs to be occasionally re-allocated for growth. Elements are stored in contiguous storage locations (elements in a vector can be accessed by offsetting a pointer to another element).
- deque's elements are scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time (complexity) and with a uniform sequential interface (through iterators). Elements in a deque can not be accessed by offsetting a pointer to another element, this will cause undefined behavior.

deque is not efficient for operations that involve frequent insertion or removals of elements at positions other than the beginning or the end.

Syntax
You need to `#include <deque>` to use the `deque` ADT.

You define a deque by following statement:

std::deque<int> mydeque;

You specify the data type this deque is going to hold in the angled bracket.

Member functions
The details of the member function can be looked up here:

http://www.cplusplus.com/reference/deque/deque/

queue

Concept
The `queue` ADT is a container adapter, which is similar with the `stack` ADT. Container adapters are classes that use an encapsulated object of a specific container class as its underlying container (such as vector, list or a deque), providing a specific set of member functions to access its elements which are stored in the underlying container, in a "queue way" (e.g. FIFO, first in first out).

The queue container adapter can be built upon `vectors`, `lists` or `deques`. The default underlying container is `deque`.

Syntax
To use `queue`, you have to `#include <queue>`.

The header of the queue template can be written as:

```
template <class T, class Container = deque<T> >
```

```
class queue;
```

Notice that the default container is `deque`. When instantiate a specific queue, if no container class is specified, the standard container `deque` is used.

Below is three examples of how to define a queue of `ints`, implemented as a `vector`, a `list` and a `deque`:

```
#include <queue>

#include <vector> // for vector based queue

#include <list> // for linked-list based queue


int main()

{

  std::queue<int> queue_1; // no container class specified, deque will be used

  std::queue<int, std::vector<int> > queue_2; // vector based queue

  std::queue<int, std::list<int> > queue_3; // list based queue

}
```

Member functions
Details: http://www.cplusplus.com/reference/queue/queue/

queue.empty()
Returns true if the queue is empty, otherwise, returns false.

queue.size()
Returns the number of elements in the underlying container.

queue.front()
Returns a reference to the next element in the queue. The next element in the queue is also the element that is popped out when member function pop is called.

queue.back()
Returns a reference to the last element in the queue. This is the newest element in the queue.

queue.push(const value_type& val)
Inserts a new element (the element "val" already exist before the call of this function) at the end of the queue, after its current last element.

queue.pop()

Removes the next element in the queue, which is the oldest element in the queue. The value of this element can be retrieved by calling member function front().

Chapter 19. Recursion

Concept

A *recursive* function is one that calls it self.

When you define a function, in the function definition, you can call the function itself (even though it is not complete yet). For example:

```
void Show()
{
 std::cout << " This is a recursive function. time: \n " ;
 Show();
}
```

When the above function is called, it will call itself again. It is like an infinite loop because there is no code to stop it from repeating. The system stores temporary data on stack each time a function is called. Eventually, the above recursive function calls will use up all available stack memory and cause it to overflow.

Like a loop, a recursive function must have some method to control the number of times it repeats. The following example is the updated version of Show() function, it can control the depth of recursion:

```
void Show(int times)
{
 if (times > 0)
 {
  std::cout << " This is a recursive function, recursive depth: " << times << " \n " ;
  Show(times - 1);
 }
}
```

Depth of recursion

The depth of recursion is the time the function calls itself (not including the initial function call outside of the function).

When the recursive function reachese its depth of recursion (the condition for recursing any deeper is not met), the control path will return to the point in the immediate upper recursion instance where the recursive function is called, and continue to perform rest statement. When this layer of recursion instance returns, the control path will return to the upper layer again (to where the recursive function call appears). All the recursive function “layer” will return like this until the initial call is reached and returned.

Problem Solving with Recursion

A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

It should be noted that recursion is nenver absolutely required to solve a problem. Any problem that can be solved recursively can also be solved iteraively, with a loop. Actually, recursive algorithms are usually less efficient than iterative algorithms because recursion requires building multiple function instances, which requires operations like allocating memory for parameters and local variables, storing the address of the program location where control returns after the function terminates.

We use recursion because some repetitive problems are more easily solved with recursion than iteration (the designing of a recursive algorithm is easier and faster than iterative algorithm).

The general working process for a recursive function is depicted as follows:

- At the beginning of a certain recursive layer, check if the problem can be solved or not now without any further recursion.
- Yes: problem can be solved without additional recursion, solve it and return
- No: problem can't be solved now, it needs additional recursion, which means the problem needs to be simplified, being reduced to a smaller but similar problem. Then the function will simplify the problem, and then call itself again to begin a new run.

In order to apply this approach, we first identify at least one case in which the problem can be solved without additional recursion. This is known as the *base case* . Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case*. In the recursive case, we **must always reduce the problem to a smaller version of the original problem**. By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop. Usually a problem is reduced by updating the value of one or more parameters with each recursive call.

Direct and Indirect Recursion

Direct recursion means the recursive function directly call themselves:

```
void A()

{

 A();

}
```

Indirect recursion means the recursive function does not directly call itself, but will call another function, which in turn calls the recursive function. For example:

```
void A()
```

```
{
  B();
}

void B()
{
  C();
}

void C()
{
  A();
}
```

Example

Following example problems can be solved easily by recursion, refer to textbook for details:

Calculate the Factorial of a Number

Count Characters

Find the Greatest Common Divisor of Two Numbers

The Euclid's algorithm is used to calculate the greatest common divisor of two numbers.

Algorithm:

The GCD of two positive integers, x and y, is:

```
gcd(x, y) = y; // if x % y == 0
gcd(y, x%y); // otherwise
```

Fibnacci Series

Recursive Linked List Operations

Recursion can be used to traverse the nodes in a linked list.

Two example operation can be implemented by recursive function:

- Count the number of nodes in the linked list
- Display the list reversely

Recursive Binary Search Function

The binary search algorithm can be defined as a recursive function.

Recursive Algorithm for Solving the Towers of Hanoi

Algorithm

The following statement describes the overall solution to the problem:

Move n discs from starting peg to receiving peg using a temporary peg;

The general algorithm for solving the Towers of Hanoi is:

If n > 0, then:

- Move (n – 1) discs from starting peg to temporary peg, using the receiving peg as transfer;
- Move the remaining disc from the starting peg to receiving peg;
- Move (n – 1) discs from temporary peg to receiving peg, using the starting peg as transfer

End if

The first step is the preparation step. Before the actual move, (n – 1) discs should be placed on peg B;

The second step is the actual step that "move" a disc

The third step is what needs to be done after the second step

Actually, the general solution (the similar solving technique for arbiturary step) is: move one disce from a peg to another peg, use a third peg as a temporary peg (don't think about which is the first peg, which is the last peg).

n == 0 is the base case of this problem: there is no disc to be moved;

The problem is divided into a smaller problem by calling Move(n – 1, from, to, temporary) function.

Tips

Tips for dealing with recursive way of solving problem. You have to treat the recursive function as a black box, do not try to trace the details of what happened, you just think it as a function that will give you the desired result. For example, in the three steps of the solution of towers of Hanoi, you call: move(n – 1, from, to, temporary) to prepare for moving the largest disc from A to C. Do not think how the function achieve this, just trust the function that it can handle by itself. After calling the function, you're ready to move the disc.

The QuickSort Algorithm

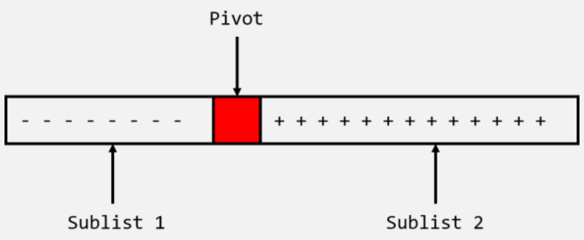
The QuickSort Algorithm uses recursion to efficiently sort a list.

QuickSort Algorithm contains two section (composed of two functions): partition and quick_sort.

Partition

Partition means divide the list (array) into two parts.

It is done by first selecting an element in the list, which is known as pivot. Then the order of the whole list is rearranged so that each element smaller than the pivot is on its left hand side, each element greater than the pivot is on its right hand side. In this way the list is divided into two sublists, shown below:



The negative sign means each element in sublist 1 is less than pivot; The positive sign means each element in sublist 2 is greater than the pivot.

The partition section does the actual sorting operation (place each element in the array to where it should be, if an element is smaller than pivot, then it should be placed on the left hand side; otherwise, it should be placed on the right hand side;)

You can pick the middle element as the pivot, whose subscript is:

```
(start_index + end_index) / 2.
```

Code:

// Partition function

// it accepts an array, the start and end index of the array

// the function will pick a pivot first, then partition the array so all the elements before the pivot are smaller (or larger) than the pivot, all the elements after the pivot are larger (or smaller) than the pivot

// then it returns the subscript of the pivot element in the array

// the function does the actual work of sorting

// first, determine a pivot based on a certain rule, for example, the value in the middle will be used as the pivot

// then, swap the pivot element with the first element in the array. This is to move the pivot to the head of the array. This can make it easy to place the remaining element before or after the pivot

// scan through the array (from start + 1 to end). Partition the array. Take an ascending array as an example:

// initialize the pivot value and the pivot index (= start). The pivot index should point to the element that has the following properties: the left element is smaller than the pivot value; the right element is larger than the element;

// Scan the array, if an element is greater than the pivot value, do nothing (because the pivot is on the head, so all the elements are on the right of the pivot by default); if an element is smaller than the pivot, first increment the pivot_index, then swap the smaller element and the element being pointed by pivot_index (after the increment, now this element should be the first that is greater than pivot)

// after the scan, swap array[start] with array[pivot_index]. array[start] contains the actual pivot value, array[pivot_index] contains the last element that is smaller than pivot

// to this point, the array is partitioned, return the pivot value for additional partition of the array (by the recursive QuickSort function)

```
template <class T>
int Partition(T* set, int start, int end)
{
 int mid = (start + end) / 2;
 Swap(set[start], set[mid]);

 int pivot_value = set[start];
 int pivot_index = start;

 // go over the array and sort
 for (int scan = start + 1; scan <= end; scan++)
 {
  if (set[scan] < pivot_value)
  {
   pivot_index++;
   Swap(set[pivot_index], set[scan]);
  }
 }

 Swap(set[start], set[pivot_index]);

 return pivot_index;
}
```

Pay attention that the partition function does not initially sort the values into their final order. Its job is only to move the values that are less than the pivot to the pivot's left, and move the values that are greater than the pivot to the pivot's right.

The ultimate sorting order of the entire list is achieved cumulatively, through the revursive calls to QuickSort.

QuickSort

This is a recursive function that sort the list recursively by calling partition function we defined and itself.

Remember the result of this function: it will sort the array sent to it.

The basic idea is, it accepts an array, then it will check if the array has only one element, if so it will do nothing but return. If the array contains more than one element (in this case, it should be sorted), it will call partition function to split the array into two sublists, then call itself to sort the two sublists.

The base case for the recursion is that the array has only one element, this is the point that the sorting problem can be solved without calling any recursion (you don't have to sort the array that has only one element).

By each recursion, the list is divided into two smaller sublists, this is where the problem has been simplified.

Code:

```
// QuickSort() function

// this is a recursive function

// it accepts: a pointer to an array, the start and end subscript

// this function will sort the array from position dictated by [start, end]

// first it calls partition function to split the array into two sub-arrays

// then it calls itself twice to sort the two sub-arrays: one is smaller than the pivot value,
and one is larger than the pivot value

// base case: when the sub-array has only one element (start_index is no longer smaller
than end_index)

// the problem is simplified each time the array is divided into two sub-arrays

template <class T>

void QuickSort(T* set, int start, int end)

{

 if (start < end)

 {

  int pivot = Partition(set, start, end); // do the splitting and sorting

  QuickSort(set, start, pivot - 1); // sort the first sublist

  QuickSort(set, pivot + 1, end); // sort the second sublist
 }
}
```

Exhaustive Algorithms

An exhaustive algorithm can find a best combination of items by looking at all the possible combinations.

Example: change coin

Recursion vs. Iteration

Any algorithm that can be coded with recursion can also be coded with an iterative control structure (for, while loop), and vice versa.

From book:

There are several reasons not to use recursion. Recursive algorithms are certainly less efficient than iterative algorithms. Each time a function is called, the system incurs overhead that is not necessary with a loop. Also, in many cases an iterative solution may be more evident than a recursive one. In fact, the majority of repetitive programming tasks are best done with loops.

Some problems, however, are more easily solved with recursion than with iteration. For example, the mathematical definition of the GCD formula is well-suited for a recursive approach. The QuickSort algorithm is also an example of a function that is easier to code with recursion than iteration. The speed and amount of memory available to modern computers diminishes the performance

impact of recursion so much that inefficiency is no longer a strong argument against it. Today, the choice of recursion or iteration is primarily a design decision. If a problem is more easily solved with a loop, that should be the approach you take. If recursion results in a better design, that is the choice you should make.

Chapter 20. Binary Trees

Definition and Applications of Binary Trees

Concepts and Definitions

A binary tree is a linked structure that links many nodes together;

Each node may point to one or two other nodes;

Due to the above characteristic, a binary tree is nonlinear;

Every node but the root node has a single predecessor;

Root node doesn't have a predecessor, it is pointed by a tree pointer

A node can have children or child node. Each of the children has its own set of two pointers and can have its own children.

A node that has no children is called a *leaf node*.

All pointers in the nodes that do not point to a node are set to nullptr.

Binary trees can be divided into subtrees.

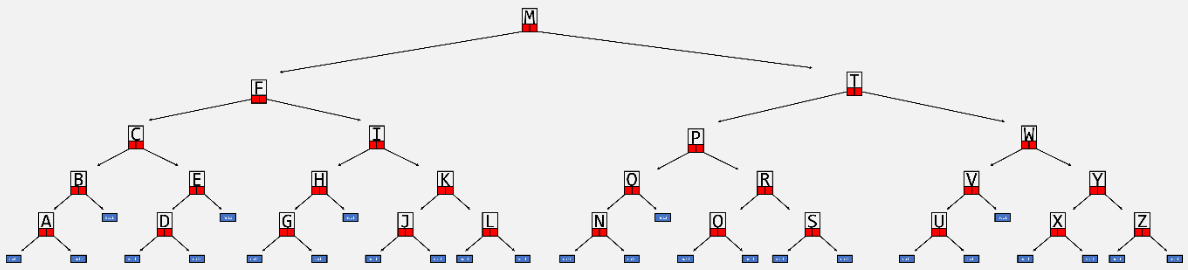
Applications of Binary Tree

Binary trees are excellent data structures for searching large amounts of data. They are commonly used in database applications to organize key values that index database records. When used to facilitate searches, a binary tree is called a binary search tree.

Using binary search tree, you can perform binary search algorithm on linked nodes data structures, just like you perform binary search on a sorted array. (pay attention that a sorted linear linked list is not able to do a binary search:

https://stackoverflow.com/questions/5281053/how-to-apply-binary-search-olog-n-on-a-sorted-linked-list).

For example, following is a binary search tree where each node stores a letter of the alphabet.



The nodes were stored into the binary search tree in a way that a node's left child holds data whose value is less than the node's data, and the node's right child holds data whose value is greater than the node's data.

When an application is searching a binary tree, it starts at the root node. If the root node does not hold the search value, the application branches either to the left or right child, based on the result of comparing the search value with the value stored in root node (if seach value is less than root value, turns left; if search value is larger than root value, turns right). This process continues until the value is found (or a nullptr is met, which indicates there is no match).

Binary Search Tree Operations

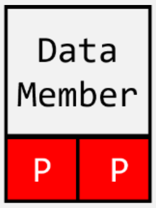
This section will briefly introduce creating a binary search tree and implement following operations:

- Inserting nodes
- Finding nodes
- Deleting nodes

Creating a Binary Tree

Node class

Before creating a binary tree, you have to create its node. Similar with the node of a linked list, the node for the binary tree has following structures:



Similarly, you can declare a class to hold data, together with two pointers. The constructor of the node should accepts at least one object (which is used to store data), also it should let two pointers point to nullptr.

Binary search tree class member

tree_pointer

It should contain a tree pointer, which will points to the root node (the constructor of this class should make tree_pointer points to nullptr).

Counstructor

The constructor of the tree class should make tree_pointer points to nullptr.

Copy constructor

It replicates the tree object passed in by parameter (`const BinarySearchTree<T>& obj`). Only values in each node will be copied, not the memory address stored in node.left and node.right pointer.

The copy constructor performs replication by calling a private member function: `TreeNode<T>* BinarySearchTree<T>::copy(const TreeNode<T>* node_ptr)` , and passing `obj.tree_pointer` to it. The return value of the `copy()` function is a pointer to a `TreeNode`. In the copy constructor, the `tree_pointer` is assigned with this return value.

The `copy()` function works in a recursive way. It accepts a pointer to a TreeNode (`node_ptr`). Its return type is `TreeNode<T>*`, a pointer to TreeNode (it returns a pointer, or a "link" to the node it just created. In this way, the nodes of the tree can be linked together).

Its basic operation is to create a TreeNode using values in its parameter pointer (`TreeNode<T>* ptr = new TreeNode<T> (node_ptr->value);`). Then it should call itself to copy left branch of the node (`ptr->left = copy(node_ptr->left);`), then call it again to copy the right branch of the node (`ptr->right = copy(node_ptr->right);`). At last, it should return the pointer pointing to this newly created node (`return ptr;`).

The base case is when the `node_ptr` passed to the function is `nullptr`, which means end of the branch has been reached. It shoud return `nullptr` as well.

Destructor

It releases all memory space allocated to the nodes of the tree.

The destructor performs memory releasing by calling a private member function (`void BinarySearchTree<T>::remove_memory(TreeNode<T>* node_ptr)`), and passing `tree_pointer` to it.

The `remove_memory()` function works in a recursive way. It accepts a pointer to TreeNode (`node_ptr`). Its function is to delete all memory allocated to itself and its children. It does this by first calling itself to delete all memory allocated to its left branch (`remove_memory(node_ptr->left);`), then calling itself again to delete all memory allocated to its right branch (`remove_memory(node_ptr->right);`), then deleting memory allocated to itself (`delete node_ptr;`).

The base case is when the `node_ptr` passed to the function is `nullptr`, which means end of the branch has been reached. It shoud do nothing but return.

Inserting node function

If the value being inserted is equal to a specific node value, insert it on the right of the node.

A node should be inserted to its proper position.

A recursive function is used to find out the proper position of the node and insert it. The function prototype of this recursive function is:

void BinarySearchTree<T>::add(TreeNode<T>*& node_ptr, T val)

It accepts a **reference** to a `TreeNode` pointer (passing by reference is a must because you may change the memory address stored in the pointer variable), and a value `T val`. The first parameter indicates where should the node be inserted, the second parameter indicates the value of the node be inserted. It works like this. First, it will check if `node_ptr` points to `nullptr`. If so, this is the proper position to insert (base case). If not, this means the node should be inserted to either the left of the node or right, depends on which is bigger: `val` or `node_ptr.value`. If (`val > node_ptr.value`), call the function to insert to right of the node. If (`val < node_ptr.value`), call the function to insert to left of the node. If (`val == node_ptr.value`), insert to the node's right (this is another base case, don't call the function to perform this operation, but write the code to do this work).

Write an "interface" function that will call this recursive function and pass the starting point (`tree_pointer`) to it. The purpose of the function is just to pass `tree_pointer` to it (it is a recursive function, starting point should be passed in).

Pay attention that the shape of the tree is determined by the order in which the values are inserted.

Searching node function

This is a Boolean function. It accepts a value, and will traverse the binary tree to search for the value. If the value is found in the binary tree, it will return true, otherwise it will return false.

The prototype of the function is as follows:

`bool BinarySearchTree<T>::Search(T val)`

It will call a recursive private member function and pass the starting point (`tree_pointer`) to it. The prototype of the private member function is:

`bool BinarySearchTree<T>::search(TreeNode<T>* node_ptr, T val)`

There are two base cases for this recursive function:

- The pointer (`node_ptr`) is `nullptr`. This means there is no match in the binary tree, return false.
- The value at the pointer address (`node_ptr->value`) is equal to `val`. This means match found, return true

Except from these two base cases, the function will call itself to find the value in either the left branch or right branch of the binary tree.

Deleting node function

This can be designed as a Boolean function. It accepts a value, and will traverse the binary tree to search for the value. If the value is found in the binary tree, it will delete it from the binary tree, and then return true. Otherwise it will return false.

The prototype of this function is:

`bool BinarySearchTree<T>::Delete(T val)`

It will call a recursive private member function and pass the starting point (`tree_pointer`) to it. This function will search from the starting point to delete the node. The prototype of this recursive member function is:

`bool BinarySearchTree<T>::remove(TreeNode<T>*& node_ptr, T val)`

When deleting a node, the subtree of that node must be preserved. You have to consider the following two cases:

- The node has only one child
  In this case, attach the node's child directly to the node's parent.
- The node has two children
  In this case, attach the node's right subtree to the parent, then attach the node's entire left subtree to its right subtree (should be attached to the leftmost position, because each node in right subtree is greater than any node in left subtree).

There are three possible scenarios:

- The node is the target node, delete it as discussed above
- The value is greater than the node's value, call itself to delete the node in left branch of the node
- The value is smaller than the node's value, call itself to delete the node in right branch of the node

Display function

This function will traverse the binary tree and display value in each node.

It will call a recursive private member function and pass the starting point (`tree_pointer`) to it. The private member function has following prototype:

`void BinarySearchTree<T>::display(const TreeNode<T>* node_ptr) const`

The base case for this function is when `node_ptr == nullptr`. In this case, just return. Other than that, the function calls itself to display the node's left branch, then write statement to display itself, then calls itself to display the node's right branch.

There are three common methods for traversing a binary tree and processing the value of each node: *in order, preorder postorder*. The order described above is in-order, because the order is from small node to big node. Algorithms for the three orders are described below:

- Inorder: traverse left subtree -> process this -> traverse right subtree
- Preorder: traverse this -> traverse left subtree -> traverse right subtree
- Postorder: traverse left subtree -> traverse right subtree -> process this