Introduction to Computer and Program Design

Lesson 8

C++

James C.C. Cheng
Department of Computer Science
National Chiao Tung University
Textbooks

- C++ Primer, 5ed
  - By Stanley B. Lippman, Josée Lajoie, Barbara E. Moo
  - Publisher: Addison Wesley
  - Features:
    - Introduction to OOP
    - C++11
A simple C++ program

- What are the differences between C and C++?

```cpp
#include <iostream>    // for cin & cout
using namespace std;   // for cin & cout
int main()       // function definition
{
    int x = 0;  // object declaration
    cin >> x;   // object input
    x *= 1 + 2 * 3 / 4;  // arithmetical expression
    cout << x << endl; // object output
    return 0;  // function return
}  // block end
```
## C/C++ Operators - 1, 2

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1(highest)</td>
<td>::</td>
<td>Scope resolution (C++ only)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>2</td>
<td>++</td>
<td>Suffix increment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>Suffix decrement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>()</td>
<td>Function call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[]</td>
<td>Array subscripting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>Element selection by reference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt;</td>
<td>Element selection through pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeid()</td>
<td>Run-time type information (C++ only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const_cast</td>
<td>Type cast (C++ only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dynamic_cast</td>
<td>Type cast (C++ only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reinterpret_cast</td>
<td>Type cast (C++ only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>static_cast</td>
<td>Type cast (C++ only)</td>
<td></td>
</tr>
</tbody>
</table>
### C/C++ Operators - 3

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>++</td>
<td>Prefix increment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>Prefix decrement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>Unary plus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Unary minus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>Logical NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>Bitwise NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type)</td>
<td>Type cast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Indirection (dereference)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>Address-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sizeof</td>
<td>Size-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new, new[]</td>
<td>Dynamic memory allocation (C++ only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete, delete[]</td>
<td>Dynamic memory deallocation (C++ only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sizeof...</td>
<td>Get the number of arguments (C+11)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>alignof</td>
<td>Get the number of bytes for memory alignment (C++11)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>noexecpt</td>
<td>Does the expression throw an exception? (C++11)</td>
<td></td>
</tr>
</tbody>
</table>
## C/C++ Operators – 4-7

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>.*</td>
<td>Pointer to member (C++ only)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td>-&gt;*</td>
<td>Pointer to member (C++ only)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>Multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>Division</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Modulo (remainder)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>Addition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&lt;&lt;</td>
<td>Bitwise left shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>Bitwise right shift</td>
<td></td>
</tr>
</tbody>
</table>
## C/C++ Operators – 8-14

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>&lt;</td>
<td>Less than</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>Greater than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>==</td>
<td>Equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>Not equal to</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>^</td>
<td>Bitwise XOR (exclusive or)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>Bitwise OR (inclusive or)</td>
</tr>
<tr>
<td>13</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# C/C++ Operators – 15-18

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>? :</td>
<td>Ternary conditional (see ?:)</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>16</td>
<td>=</td>
<td>Direct assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+=</td>
<td>Assignment by sum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-=</td>
<td>Assignment by difference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*=</td>
<td>Assignment by product</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/=</td>
<td>Assignment by quotient</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%=</td>
<td>Assignment by remainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;=</td>
<td>Assignment by bitwise left shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;=</td>
<td>Assignment by bitwise right shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;=</td>
<td>Assignment by bitwise AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^=</td>
<td>Assignment by bitwise XOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=</td>
<td>Assignment by bitwise OR</td>
</tr>
<tr>
<td>17</td>
<td>throw</td>
<td>Throw operator (exceptions throwing, C++ only)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>,</td>
<td>Comma</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>
Function overloading

- Multiple function with the name but different parameter
  - Ex:
    ```c
    int F1(void);
    int F2();
    void F3(void);
    void F3(int); // In C++, OK, but error in C
    int F4(int, int, char);
    or
    int F4(int x, int y, char c);
    ```
Function definition

- Default argument
  - Only in C++
  - A default argument is a value given in the declaration that the compiler automatically inserts if you don’t provide a value in the function call.
  - A default argument cannot be placed before non-default argument
  - The declaration of a default argument either in global function declaration or in function definition.
  - Local function declaration can has its own default argument list.

```cpp
void F1(int a, int b = 10, int c = 2)
{ 
cout << a << ", " << b << ", " << c << endl;
}

int main()
{
F1(1, 2, 3); // a = 1; b= 2; c =3 
F1(1); // a = 1; b= 10; c =2 
void F1(int a, int b =20, int c = 30); // Local decl. 
F1(1); // a = 1; b= 20; c =30 
F1(1, ,3); // Compiler error
}
Function definition

- Default argument

```c
void F2(int a, int b = 10, int c){ }     // Compiler error
void F3(int a, int b = 10, int c=5){ }   // Compiler error

void F1(int a, int b, int c);
int main(){
    F1(1); // Compiler error: default argument missing
}
void F1(int a, int b = 10, int c = 2){
    cout << a << "", " << b << "", " << c << endl;
}
void F1(int a, int b = 1, int c = 2);
int main(){
    F1(1);
}
void F1(int a, int b = 4, int c = 4) // Compiler error: redefinition
{ cout << a << "", " << b << "", " << c << endl; }
```
Standard Output in C++

- Std IO C and C++

```c
#include <cstdio>
int main()
{
    int x = 10;
    double y = 1.23456789;
    char *s = "Hello world";
    printf("%d, %0.8f, %s\n", x, y, s);
}
```

```cpp
#include <iostream>
int main()
{
    int x = 10;
    double y = 1.23456789;
    char *s = "Hello world";
    std::cout.precision(9);
    std::cout << x << " , ";
    std::cout << y << " , ";
    std::cout << s << std::endl;
}
```
Standard Output in C++

- It means that put the x into the *stdout*
- *cout* is an object that represents the *stdout*
- To save typing, we can use the “using namespace” command.

```
std::cout << x ;
```

```
std::cout.precision(9) ;
```

- **Namespace**
- **Scope operator**
- **Left object**
- **Insertion operator** (a binary operator)
- **Right object**
- **Object**
- **Member-selection operator**
- **Member method**
Standard Output in C++

- The manipulators of cout

```cpp
int x = 255;
std::cout << std::oct;
std::cout << x << std::endl; // 377  
std::cout << std::dec;
std::cout << x << std::endl; // 255  
std::cout << std::showbase << std::hex << x << std::endl; // 0xff
std::cout << std::uppercase << x << std::endl; // 0xFF
std::cout << std::noshowbase;
std::cout.width(10);
std::cout << std::dec << x << std::endl; // 255
std::cout.width(10);
std::cout << std::right << x << std::endl; // 255
std::cout.width(10);
std::cout << std::left << x << "!" << std::endl; // 255
float y = 123.56789;
std::cout.precision(5);
std::cout << y << std::endl; // 123.57
```
Standard Output in C++

- The list of manipulators
  - `boolalpha` - Insert or extract bool objects as "true" or "false".
  - `noboolalpha` - Insert or extract bool objects as numeric values.
  - `fixed` - Insert floating-point values in fixed format.
  - `scientific` - Insert floating-point values in scientific format.
  - `internal` - Internal-justify.
  - `left` - Left-justify.
  - `right` - Right-justify.
  - `dec` - Insert or extract integer values in decimal format.
  - `hex` - Insert or extract integer values in hexadecimal (base 16) format.
  - `oct` - Insert or extract values in octal (base 8) format.
Standard Output in C++

• The list of manipulators
  ◆ noshowbase - Do not prefix value with its base.
  ◆ showbase - Prefix value with its base.
  ◆ noshowpoint - Do not show decimal point if not necessary.
  ◆ showpoint - Always show decimal point when inserting floating-point values.
  ◆ noshowpos - Don't insert plus sign (+) if number >= 0.
  ◆ showpos - Do insert plus sign (+) if number >=0.
  ◆ noskipws - Do not skip initial white space on extracting.
  ◆ skipws - Skip initial white space on extracting.
  ◆ nouppercase - Don't replace lowercase letters by uppercase equivalents.
  ◆ uppercase - Replace lowercase letters by uppercase equivalents.
  ◆ unitbuf - Flush buffer after an insert.
  ◆ nounitbuf - Don't flush buffer after each insert.
Standard Output in C++

- `ios::flags`
  - `setf(long)` and `unsetf(long)`, to set and unset the flag.
  - Using `flags()` to recover all setting

```cpp
int x = 255;
long OrgFlag = std::cout.flags();  // Record the origin setting
std::cout.unsetf( ios::dec );       // unset
std::cout.setf( ios::oct );
std::cout << x << std::endl;        // 0377
std::cout.unsetf( ios::oct );      // unset
std::cout.setf( ios::dec );
std::cout << x << std::endl;        // 255
std::cout.unsetf( ios::dec );      // unset
std::cout.setf( ios::hex | ios::showbase | ios::uppercase );
std::cout << x << std::endl;        // 0xFF
std::cout.flags( OrgFlag );
std::cout << x << std::endl;        // Recover the flag
```
Standard Input in C++

- `std::cin` with the extraction operator, `>>`
  - `cin` is an object that represents the `stdin`

```cpp
char c1=0, c2 =0, c3=0;

std::cin >> c1 >> c2;
std::cout << c1 << " , " << c2 << std::endl;

std::cin >> c3;
std::cout << c3 << std::endl;
```

Notice that the delimit characters are white space keys.
Standard Input in C++

- `std::cin` and EOF
  - Ctrl+D in UNIX
  - Ctrl+Z in Windows

```cpp
#include <iostream>
int main(){
    int x = 10;
    while( !((std::cin >> x).eof()) )
        std::cout << x << std::endl;
    std::cout << "END" << std::endl;
}
```
Standard Input in C++

- Abnormal input
  - All wrong input must be removed before the next input
  - The fail state of `cin` must be reset after stdin buffer cleared

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

int main(){
    int x = 10;
    while(!(cin >> x).eof()){ // Abnormal input
        if(cin.fail()){ // Abnormal input
            std::cin.clear(); // Clear the fail state
            while(cin.get()!='
' && !cin.eof()); // Clearing
        }
        else{ // Normal input
            cout << x << endl;
        }
    }
    cout << "END" << endl;
}
```
Memory management in C++

- malloc and free in C

```cpp
#include <cstdlib> // for malloc, calloc, realloc, and free
#include <cstring> // for memset, memcpy, memmove, and memcmp
#include <iostream>
using namespace std;

int main()
{
    size_t n = 0;
    cin >> n;

    // Dynamic memory allocation without initialization
    int *pi = (int *)malloc( sizeof(int) * n);
    memset(pi, 0, sizeof(int) * n);

    // Dynamic memory allocation with zero initialization
    double *pd = (double *)calloc(n, sizeof(double) * n);

    // Deallocations
    free(pi); free(pd);
}
```
Memory management in C++

- new and delete in C

```cpp
#include <cstdlib> // for malloc, calloc, realloc, and free
#include <cstring> // for memset, memcpy, memmove, and memcmp
#include <iostream>
using namespace std;

int main(){
    size_t n = 0;
    cin >> n;

    int *pi = new int[n];
    memset(pi, 0, sizeof(int) * n);

    double *pd = new double[n];
    memset(pd, 0, sizeof(double) * n);

    // Deallocation
    delete [] pi;
    delete [] pd;
}
```
Dynamic memory allocation

- Case study: recording the weights and heights (dynamic memory alloc)

```cpp
size_t n = 0;
cout << "n="; cin >> n; getchar(); // n=

// Memory allocation, watch the array pointer declaration
float (*A)[2] = new float[n][2];
int i=0;
for(i=0; i<n; ++i){ // Data input.
    cout << "W H: ";
    cin >> A[i][0] >> A[i][1];
}
qsort(A, n, sizeof(float) * 2, cmpW ); // By weight
for(i=0; i<n; ++i)
    cout << A[i][0] << ":" << A[i][1] << ", ";
cout << endl;
qsort(A, n, sizeof(float) * 2, cmpH ); // By height
for(i=0; i<n; ++i)
    cout << A[i][0] << ":" << A[i][1] << ", ";
cout << endl;
delete [] A;
```
Struct

- To design a custom data type
- To group multiple relevant data
- Heterogeneous array

```c
struct Student{
    char name[8];
    float w, h;
};
```

```c
void InputStudent(Student* px)
{
    cin >> px->name >> px->w >> px->h; getchar();
}

void PrintStudent(const struct Student* px){
    cout << "Name:" << px->name << ", ";
    cout << "w:" << px->w << ", ";
    cout << "h:" << px->h << endl;
}
```
Struct

- Initialization
  -Initializer list

```cpp
struct Student{
  char name[8];
  float w, h;
};

Student x = {0};
Student y = {"James", 75.2f, 175.6f};
// Notice the order
```

- Uniform initialization
  - C++11 and GCC
  - Visual C++ 2010, 2012 not support

```cpp
Student x{0};
Student y{"James", 75.2f, 175.6f};
// Notice the order
```
File I/O in C++

- **fstream**, a class of File I/O
  - its usage is similar to `cin` and `cout`

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

int main()
{
    fstream file("Test.txt", ios::out); // Open for writing
    for(int i=0; i<10; ++i)
    {
        file << "LINE" << i << endl;
    }
    file.close(); // Don't forget to close!

    file.open("Test.txt", ios::in); // Open for read
    while(!file.eof())
    {
        char buf[255] = {0};
        file >> buf;
        cout << buf << endl;
    }
    file.close(); // Don't forget to close!
}
```
File I/O in C++

- Opening mode
  - `ios::app`, appending to the end of file.
  - `ios::ate`, to seek to the end of file.
  - `ios::binary`, binary mode
  - `ios::in`, read mode.
  - `ios::out`, write mode.
  - `ios::trunc`, to delete contents of an existing file.
File I/O in C++

- Input (read) / Output (write)

```cpp
istream& fstream::read(char* buf, streamsize size);
ostream& fstream::write(const *char buf, streamsize size);

- `buf`: Storage location for data
- `size`: Item size in bytes.

```cpp
fstream file("Test.txt", ios::out);
int A[3] = {1, 2, 3};
file.write((const char*)A, sizeof(A));
file.close();

int B[3] = {0};
file.open("Test.txt", ios::in);
file.read((char*)B, sizeof(B));
cout << B[0] << endl;
cout << B[1] << endl;
cout << B[2] << endl;
file.close();
```
File I/O in C++

- Stream I/O
  - Other Stream I/O functions:
    - `streampos fstream::tellg();`
      - Get position of get pointer
    - `streampos fstream::tellp();`
      - Get position of put pointer
    - `istream& fstream::seekg(streamoff off, ios_base::seekdir dir );`
      - Sets the position of the get pointer
      - `dir = ios_base::beg` beginning of the stream buffer
      - `ios_base::cur` current position in the stream buffer
      - `ios_base::end` end of the stream buffer
      - notice that the eof or other error state must be cleared before seekg() or seekp()
    - `ostream& fstream::seekp(streamoff off, ios_base::seekdir dir );`
      - Sets the position of the put pointer
    - `bool fstream::eof( ) const;`
      - Returns true if the cursor is at the end of file
References

- Syntax:

```cpp
typename &alias = object name;
```

- A reference is just an alias of an object.

```cpp
int x = 10;
int &refx = x;
++refx;
cout << x << endl; // 11
```

- A reference must be some object’s alias
- A const reference can refer a constant

```cpp
const int &r = 123;
cout << r << endl; // 123;
r = 456; // Error
const int x = 10;
r = x; // Error
```

```cpp
int &r1; // Error!
int &r2 = 123; // Error
int x = 10, y = 20;
int &r = x;
cout << r << endl; // 10
int &r = y;
cout << r << endl; // 20
```
References

Why we need references?

- Multiple output of function

```c
void Swap(int *pa, int pb)
{ int t = *pa; *pa = *pb; *pb = t; } // Too complicated!
...
int x = 10, y = 20;
Swap(&x, &y);
printf("%d, %d\n", x, y); // 20, 10 ... OK!
```

```c
void Swap(int &a, int &b)
{ int t = a; a = b; b = t; } // Very simple!
...
int x = 10, y = 20;
Swap(x, y);
printf("%d, %d\n", x, y); // 20, 10 ... OK!
// Call by name (or reference):
// passing the parameter by alias name
```
References

- Type cast of references
  - Be careful of the size of object

```cpp
float x = -0.0f;
int &rx = (int &)x;
cout << hex << x << endl; // 80000000

char y = 0;
int &ry = (int &y);
ry = 257; // Runtime error!
```
Reference cannot be reset to refer other object.

```cpp
int x = 10, y = 20;
int &r = x;
cout << r << endl; // 10
r = y;
cout << x << " , " << y << endl; // 20, 20
r = (int & )y;
cout << x << " , " << y << endl; // 20, 20
r = 30;
cout << x << " , " << y << endl; // 30, 20
```

There is no straight method to reset a reference.
## References

- **Reference and Array**

```cpp
void printArray(int (&A)[10]){
    for(int i=0; i<10; ++i) cout << A[i] << endl;
}

int main(){
    int M = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    printArray(M);
}
```

```cpp
void printArray(int (&A)[] ){ // Error! Unknown bound!
    for(int i=0; i<10; ++i) cout << A[i] << endl;
}

void printArray(int &A[10] ){ // Error! 10 integer references?
    for(int i=0; i<10; ++i) cout << A[i] << endl;
}

void printArray(int &A ){ // Error! Only one integer reference!
    for(int i=0; i<10; ++i) cout << A[i] << endl;
}
```
Class

- Only in C++
- As the same as struct, class also can contains member functions
- Actually, the usage of class is the same as struct’s besides the default member access privilege and initializer.

- **struct: public**

```cpp
struct: public
```

- **class: private**

```cpp
class Student{
    char name[8];
    float weight, height;
    float bmi();
};
float Student::bmi(){
    float h = height / 100.0f;
    return weight / (h * h);
}
```

```cpp
Student x;
x.weight = 80;  // Error! w is a private member
x.bmi();       // Error! bmi() is a private member
```
Class

- There are three member access privileges
  - private:
    - Accessed by member functions, friend functions, and friend classes
  - protected:
    - Accessed by *inherited classes*, member functions, friend functions, and friend classes
  - public:
    - Accessed by *derivative objects*, inherited classes, member functions, friend functions and friend classes
- To access a private member data, you should design a public function
Class

- DO NOT declare member variable as public! Just **private** or **protected**.
- To access a private member data, you should design a public function

```cpp
class Student{
private:
    char name[8];
    float weight, height;
public:
    void setHeight (float _h){height= _h; }
    void setWeight (float _w){weight= _w; }
    void setName(const char * _name) {strcpy(name, _name);} 
    float getHeight () const { return height; }
    float getWeight () const { return weight; }
    const char* getName() const{ return name; }
    float bmi();
};

float Student::bmi(){
    float h = height / 100.0f;
    return weight / (h * h);
}
```
“this” pointer

Questions:
- Why `setHeight(float _h)` can pass `_h` to `x.height` or `y.height`?
- Why `bmi()` can calculate `x's bmi` or `y's bmi`?

```cpp
void Student::setHeight(float _h)
{ height = _h; }

float Student::bmi()
{
    float h = height / 100.0f;
    return weight / (h * h);
}
```

```cpp
Student x, y;
x.setWeight(70);
x.setHeight(170);
cout << x.bmi() << endl;

y.setHeight(75);
y.setHeight(170);
cout << y.bmi() << endl;
```
“this” pointer

- The answer is, your code will be changed by the compiler:

```cpp
void Student::setHeight(float _h)
{ height= _h; }

float Student::bmi()
{
    float h = height / 100.0f;
    return weight / (h * h);
}
```

```cpp
void Student::setHeight(Student *this, float _h)
{this->height= _h; }

float Student::bmi(Student *this)
{
    float h = this->height / 100.0f;
    return this->weight / (h * h);
}
```

- The added pointer, this, is used to represent the members owner.
“this” pointer

- Certainly, the member function calling also be changed by compiler:

```cpp
Student x, y;
x.setWeight(70);
x.setHeight(170);
cout << x.bmi() << endl;

y.setHeight(75);
y.setHeight(170);
cout << y.bmi() << endl;
```

```cpp
Student x, y;
x.setWeight(&x, 70);
x.setHeight(&x, 170);
cout << x.bmi(&x) << endl;

y.setHeight(&y, 75);
y.setHeight(&y, 170);
cout << y.bmi(&y) << endl;
```
“this” pointer

- "this" is an reserved key word.
- Therefore, you cannot naming any object as "this"
- In any **non-static** member functions, you can use *this* pointer to indicate the other member.

```cpp
void Student::setHeight( Student *this, float _h) // error
{this->height= _h; } // OK

float Student::bmi(Student *this){// error
    float h = this->height / 100.0f; // OK
    return this->weight / (h * h); // OK
}
```
constant methods

- You should design the member function as a constant method, if all member data won't be changed in the function.
- put `const` key word before {

```cpp
class Student{
  ...
  float getHeight () const;
  float getWeight () const;
  const char* getName() const;
  float bmi() const;
};
```

```cpp
float Student::getHeight() const {return height;}
float Student::getWeight () const {return weight;}
const char* Student::getName() const{return name;}
float Student::bmi() const{
  float h = height / 100.0f;
  return weight / (h * h);
}
```
constant methods

- However, the following member functions cannot be constant member methods:
  - Any member variable is modified in the function body
  - Any non-constant member function is called in the function body
  - Static member function
constant methods

- *this* pointer in the constant method

```cpp
float Student::getHeight(const Student* this) const
{return this->height;}
float Student::getWeight (const Student* this) const
{return this->weight;}
const char* Student::getName(const Student* this) const
{return this->name;}
float Student::bmi(const Student* this) const
{
    float h = this->height / 100.0f;
    return this->weight / (h * h);
}
```

- const Typename* p;
  - The memory space pointed by p is read-only.
  - But p can be modified.
  - Considering these five pointers:

        const int *p;           int const *p;           int * const p;
        const int const *p;     const int *const p;
Static members

- When a class member is **static**, only one memory location is allocated
  - All members of the class share a single storage location for a static data member of that same class
  - When you create a non-static variable within a function, a new variable is created every time you call that function
  - When you create a static variable, the variable maintains its memory address and previous value for the life of the program
Static members

- Static member variable must be **instantiated**

**Class declaration**

class Student {
    ...
    static float bmiM, bmiF;

public:
    static float StandBmiM();
    static float StandBmiF();

    static float setStandBmiM( float v);
    static float setStandBmiF( float v);
    ...
};
Static members

- Static member variable must be instantiated

Implementation

```cpp
// Static variables memory allocation
float Student::bmiM = 22.0f;
float Student::bmiF = 19.0f;

// Static member functions implementation
float Student::StandBmiM() { return bmiM; }
float Student::StandBmiF() { return bmiF; }
float Student::setStandBmiM(float v) { bmiM = v; }
float Student::setStandBmiF(float v) { bmiF = v; }
```

Notice that there is no static keyword in the function implementation
Static members

- A static member does not belong to any object ➔ a common member
  - It is called **class variables**, **class fields**, or **class-wide fields**

```cpp
int main(){
    Student Lin, Chen;
    cout << Lin.StandBmiF() << endl;     // 19.0
    cout << Chen.StandBmiF() << endl;     // 19.0
    cout << Student::StandBmiF() << endl; // 19.0
    // You can use the static memebr by the scope operator.
}
```

- No `this` pointer in the static member function

```cpp
float Student::setStandBmiM (float v) {
    this->bmiM = v;
}
// Error! The static member function doesn’t have this pointer
```
Static members

- Other non-static member function also can use the static members

Class declaration

```cpp
class Student {
private:
    int g;  // gender
    ...
    static float bmiM, bmiF;

public:
    ...
    void getGender() const;
    int setGender(int _g);
    float StandBmi() const;
    float bmiDiff() const;
};
```

Implementation

```cpp
void getGender() const {return g;}
int setGender(int _g){g = _g;}
float Student::StandBmi() const{
    if( g == 0) return bmiF;
    else return bmiM;
}
float Student::bmiDiff() const
{ return bmi() - StandBmi(); }
```

- But, DO NOT use `this` pointer to access the static members

```cpp
float Student::StandBmi() const {
    if( this->g == 0) return this->bmiF;
    else return this->bmiM;
} // Error!
```
const static members

- Unchangeable static member data
  - They are read-only
  - They must be assigned a initial value in the class declaration
  - They do not need memory allocation

Class declaration

```cpp
class Student {
    ...
    const static float bmiM = 22.0;
    const static float bmiF = 19.0;
public:
    static float StandBmiM();
    static float StandBmiF();

    static float setStandBmiM( float v);
    static float setStandBmiF( float v);
    ...
};
```
const static members

Implementation

```
// Static variables memory allocation
float Student::bmiM = 22.0f; // Error!
float Student::bmiF = 19.0f; // Error!

// Static member functions implementation
float  Student::StandBmiM() { return bmiM; }
float  Student::StandBmiF() { return bmiF; }
float  Student::setStandBmiM(float v)
{ bmiM = v; } // Error! bmiM is unchangeable!
float  Student::setStandBmiF(float v)
{ bmiF = v; } // Error! bmiF is unchangeable!
```
Constructors and Destructor

- **Constructor**, ctor
  - A member function is automatically invoked after object is created
  - Its name is the same as the type name
  - No return type
  - The number of parameters $\geq 0$
  - You can design many ctors with different parameters for a datatype

- **Destructor**, dtor
  - A member function is automatically invoked before object is deleted
  - The name of dtor is $\sim$typename
  - It has no return type
  - It has no any parameter
  - It is unique
Constructors

• Default constructor: ctor without any argument

```cpp
class Student{
    char name[8];
    float weight, height;

public:
    // ctor
    Student();
};

Student::Student()
{
    weight = height = 0.0f;
    memset(name, 0, sizeof(char) * 8);
}
```

```cpp
Student x;
cout << x.getName() << " : ";
cout << x.getWeight() << ", " << x.getHeight() << endl;
```
Constructors

- Default constructor

```cpp
class Student{
    char name[8];
    float weight, height;
public:
    Student ();
    Student (float _w, float _h, const char* _name);
};
...
Student::Student(float _w, float _h, const char* _name){
    weight = _w;
    height = _h;
    if(_name != NULL) strcpy(name, _name);
    else memset(name, 0, sizeof(char) * 8);
}

Student x(70.5f, 175.5f, "James");
cout << x.getName() << ": ";
cout << x.getWeight() << ", " << x.getHeight() << endl;
```
Constructors

- In C++11, we can use initializer list or uniform initialization to call ctor
  - Visual C++ 2010 and 2012 not support

```cpp
class Student{
    char name[8];
    float weight, height;
public:
    // ctors
    Student ()
    Student (float _w, float _h, const char* _name);
};
```

```cpp
Student x = {70.5f, 175.5f, "James"};
    // Initializer list, Only in GCC C++11
Student y{65.5f, 180.5f, "Lin"};
    // Uniform initialization, Only in GCC C++11
cout << x.getName() << " : ";
cout << x.getWeight() << ", " << x.getHeight() << endl;
cout << y.getName() << " : ";
cout << y.getWeight() << ", " << y.getHeight() << endl;
// That is the main difference between class and struct
```
Constructors

- Initialization list of ctor

```cpp
class Student{
    char name[8];
    float weight, height;
public:
    Student ();
    Student (float _w, float _h, const char* _name);
};
...
Student::Student(float _w, float _h, const char* _name)
: weight(_w), height(_h)  // Notice the order!
{
    if(_name != NULL) strcpy(name, _name);  // For array
    else memset(name, 0, sizeof(char) * 8);
}

Student x(70.5f, 175.5f, "James");
cout << x.getName() << " : ";
cout << x.getWeight() << ", " << x.getHeight() << endl;
```

Only single object can be initialized in the initialization list.
Constructors

- Better default ctor

```cpp
class Student{
    char name[8];
    float weight, height;
public:
    Student(float _w = 0, float _h = 0,
            const char* _name = NULL);
};
...
Student::Student(float _w, float _h, const char* _name)
    : weight(_w), height(_h)            // Notice the order!
    {
        if(_name != NULL) strcpy(name, _name); // For array
        else memset(name, 0, sizeof(char) * 8);
    }

Student x;
cout << x.getName() << " : ";
cout << x.getWeight() << ", " << x.getHeight() << endl;
```
Constructors

- Dynamic member and ctor

```cpp
class Student{
    char *name;
    float weight, height;
public:
    Student(float _w=0, float _h=0, const char* _name=NULL);
};

Student::Student(float _w, float _h, const char* _name)
: name(NULL), weight(_w), height(_h) // Notice the order!
{  setName(_name); }

void Student::setName (const char* _name)
{  if(name) delete [] name; // delete the previous new
    name = NULL;
    if(_name != NULL){
        name = new char[strlen(_name) + 1];
        strcpy(name, _name);
    }
} // Where should you do the last delete[]?
```
Destructor

- Destructor, dtor
  - In most case, dtor is needed to be designed only when there is a dynamic member in the class

```cpp
class Student{
    char *name;
    float weight, height;
public:
    Student(float _w = 0, float _h = 0,
            const char* _name = NULL);
    ~Student();
};

Student::~Student(){  if(name) delete [] name; }
```
Copy constructor

- You must design the copy constructor if there is a dynamic member in the class

```cpp
class Student{
    char *name;
    float weight, height;
public:
    Student(float _w = 0, float _h = 0,
             const char* _name = NULL);
    Student(const Student& x);
    ~Student();
};

Student::Student(const Student& )
{ setName(x.name); } // Private access is allowed
```
EX: Stack

- Stack, a LIFO (Last-In-First-Out) data structure.

- ADT:

```cpp
class Stack{
private:
    int* data;
    int top, n;
public:
    Stack(int _n = 10);
    Stack(const Stack& s);
    ~Stack();
    int maxsize();
    int size();
    bool isEmpty();
    bool isFull();
    void push(int x);
    int pop();
    void out();
};
```
EX: Stack

- Member function definitions:

```cpp
// ctor
Stack::Stack(int _n): data(NULL), top(-1), n(_n) {
    if(n > 0) {
        data = new int[n];
        memset(data, 0, sizeof(int) * n);
    }
}

// Copy ctor
Stack::Stack(const Stack& s): data(NULL), top(s.top), n(s.n) {
    if(n > 0) {
        data = new int[n];
        for(int i=0; i<n; ++i)
            data[i] = s.data[i]; // memcpy is not recommended
    }
}

// dtor
Stack::~Stack() { if(data) delete [] data; }
```
EX: Stack

- Member function definitions:

```cpp
// properties
int Stack::maxsize() { return n; }
int Stack::size() { return top+1; }
bool Stack::isEmpty() { return top<0; }
bool Stack::isFull() { return top == n-1; }
```
Member function definitions:

```cpp
void Stack::push(int x) {
    if (!isFull()) data[++top] = x;
}

int Stack::pop() {
    if (!isEmpty())
        return data[top--];
    return 0;
}

void Stack::out() {
    cout << "STACK N = " << size() << ": ";
    for (int i = top; i >= 0; ) {
        cout << data[i--];
        if (i >= 0) cout << ", ";
    }
    cout << endl;
}
```
Namespace

- Only in C++
- Given a group name, namespace groups constants, variables, functions, enums, structs, unions, and classes.

```cpp
namespace boy {
    Student p;
    bool IsTooFat() { return p.bmi() >= 24.0f; }
}

namespace girl {
    Student p;
    bool IsTooFat() { return p.bmi() >= 22.0f; }
}
```
Namespace

- Using the scope operator :: to access the members in a namespace

```cpp
boy::p.setWeight(60.0f);
boy::p.setHeight(165.0f);
cout << boy::p.getWeight() << ", ";
cout << boy::p.getHeight() << ": ";
cout << boy::p.bmi() << ", ";
cout << boy::IsTooFat() << endl;

girl::p.setWeight(60.0f);
girl::p.setHeight(165.0f);
cout << girl::p.getWeight() << ", ";
cout << girl::p.getHeight() << ": ";
cout << girl::p.bmi() << ", ";
cout << girl::IsTooFat() << endl;
```
Namespace

- Keyword: using
  - To save typing, the keyword “using” allows the objects in a namespace to be used without the namespace-name.

```cpp
boy::p.setHeight(60.0f);
using namespace boy;
p.setWeight(165.0f);
cout << p.getWeight() << ", ";
cout << p.getHeight() << ": ";
cout << p.bmi() << ", ";
cout << IsTooFat() << endl;

using namespace girl;
p.setHeight(60.0f); // Error! Ambiguous
```

Notice that there is no way to turn off “using”!
### Namespace

- Local using

```cpp
using boy::IsTooFat;  // Notice! DO NOT add "()"
boy::p.setWeight(60.0f);
boy::p.setHeight(165.0f);
cout << boy::p.getWeight() << ", ";
cout << boy::p.getHeight() << ": ";
cout << boy::p.bmi() << ", ";
cout << IsTooFat() << endl;

using girl::IsTooFat;
girl::p.setWeight(60.0f);
girl::p.setHeight(165.0f);
cout << girl::p.getWeight() << ", ";
cout << girl::p.getHeight() << ": ";
cout << girl::p.bmi() << ", ";
cout << IsTooFat() << endl;  // Error! Ambiguous
```
Namespace

- Local using
  - Local using is more prior than using namespace

```cpp
using namespace boy;
p.setWeight(60.0f); p.setHeight(165.0f);
cout << p.getWeight() << "", "";
cout << p.getHeight() << ": ";
cout << p.bmi() << "", "";
cout << IsTooFat() << endl; // OK! boy's

using girl::p;
using girl::IsTooFat;
p.setWeight(60.0f); p.setHeight(165.0f);
cout << p.getWeight() << "", "";
cout << p.getHeight() << "": "";
cout << p.bmi() << "", "";
cout << IsTooFat() << endl; // OK! Girl's
```
**Namespace**

- **Local using**

```cpp
using boy::IsTooFat; // Notice! DO NOT add "()"
boy::p.setWeight(60.0f); boy::p.setHeight(165.0f);
cout << boy::p.getWeight() << ", ";
cout << boy::p.getHeight() << ": ";
cout << boy::p.bmi() << ", ";
cout << IsTooFat() << endl; // OK! boy's
{
    using girl::IsTooFat;
    girl::p.setWeight(60.0f); girl::p.setHeight(165.0f);
cout << girl::p.getWeight() << ", ";
cout << girl::p.getHeight() << ": ";
cout << girl::p.bmi() << ", ";
cout << IsTooFat() << endl; // OK! Girl's
}
boy::p.setWeight(60.0f); boy::p.setHeight(165.0f);
cout << boy::p.getWeight() << ", ";
cout << boy::p.getHeight() << ": ";
cout << boy::p.bmi() << ", ";
cout << IsTooFat() << end; // OK! boy's
```
Namespace

- using namespace ≠ local using

```cpp
using namespace boy;
p.setWeight(60.0f); boy::p.setHeight(165.0f);
cout << p.getWeight() << ", ";
cout << p.getHeight() << ": ";
cout << p.bmi() << ", ";
cout << IsTooFat() << endl; // OK! boy's
{
  using namespace girl;
  p.setWeight(60.0f); // Error! Ambiguous
}
```
Namespace

- Only objects in an namespace can be joined to local using

```cpp
int x = 0;
using girl;    // Error
using girl::;  // Error
using girl::x; // Error
```
Namespace

- global namespace

```cpp
Student p;
int main(){
    p.setWeight(60.0f); // Global's p
    p.setHeight(165.0f);
    cout << boy::p.getWeight() << "", "; // boy's p
    using namespace boy;
    cout << p.getHeight() << ": "; // Ambiguous
    cout << boy::p.bmi() << "", "; // Boy's
    cout << IsTooFat() << endl; // Boy's
}
```

```cpp
Student p;
int main(){
    p.setWeight(60.0f); // Global's p
    p.setHeight(165.0f);
    cout << boy::p.getWeight() << "", "; // boy's p
    using boy::p;
    cout << p.getHeight() << ": "; // Boy's
    cout << ::p.bmi() << "", "; // Global's
    cout << IsTooFat() << endl; // Error
}
```
Namespace

- global namespace

```cpp
Student p;
int main()
{
    p.setWeight(60.0f);  // Global's p
    p.setHeight(165.0f);
    {
        using boy::p;
        cout << p.getWeight() << "", ";  // boy's p
        cout << ::p.getHeight() << ": ";  // Global's
    }
    cout << p.bmi() << ", ";  // Global's
    cout << boy::IsTooFat() << endl;  // Error
}
```
Nested Namespace

```plaintext
namespace Normal{
    namespace boy {
        Student p;
        bool IsTooFat() { return p.bmi() >= 24.0f; }
    }
    namespace girl {
        Student p;
        bool IsTooFat() { return p.bmi() >= 22.0f; }
    }
}

namespace Model{
    namespace boy {
        Student p;
        bool IsTooFat() { return p.bmi() >= 21.0f; }
    }
    namespace girl {
        Student p;
        bool IsTooFat() { return p.bmi() >= 20.0f; }
    }
}
```
Nested Namespace

```
Normal::girl::p.setWeight(58.0f);
Normal::girl::p.setHeight(165.0f);
cout << Normal::girl::p.getWeight() << " , ";
cout << Normal::girl::p.getHeight() << ": ";
cout << Normal::girl::p.bmi() << " , ";
cout << Normal::girl::IsTooFat() << endl;

Model::girl::p.setWeight(58.0f);
Model::girl::p.setHeight(165.0f);
cout << Model::girl::p.getWeight() << " , ";
cout << Model::girl::p.getHeight() << ": ";
cout << Model::girl::p.bmi() << " , ";
cout << Model::girl::IsTooFat() << endl;
```
using namespace Normal::girl;
  p.setWeight(58.0f); // Normal::girl::p
  p.setHeight(165.0f);
  cout << p.getWeight() << "", ";
  cout << p.getHeight() << ": ";
  cout << p.bmi() << "", ";
  cout << IsTooFat() << endl;

using Model::girl::p;
using Model::girl::IsTooFat;
  p.setWeight(58.0f); // Model::girl::p
  p.setHeight(165.0f);
  cout << p.getWeight() << "", ";
  cout << p.getHeight() << ": ";
  cout << p.bmi() << "", ";
  cout << IsTooFat() << endl; // Model::girl::IsTooFat