# 程序设计与算法语言 · 静态与友元
# C/C++ Programming and Algorithms
# Static, Friend and Const

Dongke Sun (孙东科)
dksun@seu.edu.cn

东南大学机械工程学院
School of Mechanical Engineering
Southeast University

April 12, 2018

# 牢记80/20定律—The Pareto Principle

# 提纲

# 提纲

# 类的静态成员

## 类的静态成员有

- 静态数据成员
- 静态成员函数

# 类的静态成员

## 类的静态成员有

- 静态数据成员
- 静态成员函数

## 类的静态数据成员

一个类的不同对象，其普通数据成员的存储空间是相互独立的。

```
1  class Person                    // 数据成员+动作
2  {
3  private:
4      char name[20];              // 姓名
5      char gender;                // 性别
6      int  age;                   // 年龄
7  public:
8      void SetData(char [ ], char , int );
9      void GetName(char *);
10     char GetGender( );
11     int  GetAge( );
12 };
13 Person   ChengB("ChengBin", 'F', 20), ZhangJ("ZhangJie", 'M', 18);
```

类Person的两个对象ChengB和ZhangJ的内存空间是相互独立的

# 类的静态数据成员

- 如果将类的一个数据成员定义成静态成员，则这个类所有对象的该成员共用同一存储空间。

```
1  class Sample
2  {
3  private:
4      int a, b, c ;
5      static int d ;  // 定义静态数据成员
6      ...
7  }sA, sB, sC;
```

对象sA、sB、sC的数据成员a、b、c在内存空间是相互独立的，而数据成员d是属于类Sample的，不专属于任何对象。

- 静态数据成员实际上（本质上要看所在的内存分区）是类域中的全局变量。

  因此，类的静态数据成员的初始化 —— 必须在类体外进行：

# 类的静态数据成员

- 如果将类的一个数据成员定义成静态成员，则这个类所有对象的该成员共用同一存储空间。

```
1  class Sample
2  {
3  private:
4      int a, b, c ;
5      static int d ;  // 定义静态数据成员
6      ...
7  }sA, sB, sC;
```

对象sA、sB、sC的数据成员a、b、c在内存空间是相互独立的，而数据成员d是属于类Sample的，不专属于任何对象。

- 静态数据成员实际上（本质上要看所在的内存分区）是类域中的全局变量。

  因此，**类的静态数据成员的初始化** —— 必须在类体外进行：

# 类的静态数据成员

- 如果将类的一个数据成员定义成静态成员，则这个类所有对象的该成员共用同一存储空间。

```
1  class Sample
2  {
3  private:
4      int a, b, c ;
5      static int d ;  // 定义静态数据成员
6      ...
7  }sA, sB, sC;
```

对象sA、sB、sC的数据成员a、b、c在内存空间是相互独立的，而数据成员d是属于类Sample的，不专属于任何对象。

- 静态数据成员实际上（本质上要看所在的内存分区）是类域中的全局变量。

  因此，**类的静态数据成员的初始化** —— 必须在类体外进行：

```
8  int Sample::d(10);  // 初始化，直接初始化（direct initialization）
```

或

```
8  int Sample::d = 10; // 初始化，拷贝初始化（copy initialization）
```

# 静态数据成员的使用

- 类的静态数据成员必须在类体外进行初始化

```
1  #include <iostream>
2  using namespace std;
3  class Sample
4  {
5      int n;
6      static int sum;   // 静态成员
7  public:
8      Sample(int x)
9      {   n=x;     }
10     void add( ) const // 注意
11     {   sum+=n; }
12     void disp( )
13     {
14         cout<<"n="<<n<<",sum="<<
       sum<<endl;
15     }
16 };
17 int Sample::sum = 0;   // 初始化
18 int main( )
19 {
22     Sample a(2), b(3), c(5);
23     a.add( );       a.disp( );
24     b.add( );       b.disp( );
25     c.add( );       c.disp( );
26     cout <<"sizeof(a)␣:␣"<<
       sizeof(a)<<endl;
27     cout <<"sizeof(b)␣:␣"<<
       sizeof(b)<<endl;
28     cout <<"sizeof(c)␣:␣"<<
       sizeof(c)<<endl;
29     return 0;
30 }
```

程序运行结果是:

# 静态数据成员的使用

- 类的静态数据成员必须在类体外进行初始化

```
1  #include <iostream>
2  using namespace std;
3  class Sample
4  {
5      int n;
6      static int sum;   // 静态成员
7  public:
8      Sample(int x)
9      {   n=x;     }
10     void add( ) const // 注意
11     {   sum+=n; }
12     void disp( )
13     {
14         cout<<"n="<<n<<",sum="<<
       sum<<endl;
15     }
16 };
17 int Sample::sum = 0;   // 初始化
18 int main( )
19 {
22     Sample a(2), b(3), c(5);
23     a.add( );       a.disp( );
24     b.add( );       b.disp( );
25     c.add( );       c.disp( );
26     cout <<"sizeof(a)␣:␣"<<
       sizeof(a)<<endl;
27     cout <<"sizeof(b)␣:␣"<<
       sizeof(b)<<endl;
28     cout <<"sizeof(c)␣:␣"<<
       sizeof(c)<<endl;
29     return 0;
30 }
```

程序运行结果是：n=2, sum=2
n=3, sum=5
n=5, sum=10
sizeof(a) : 4
sizeof(b) : 4
sizeof(c) : 4

# 静态数据成员的使用

- 类的静态数据成员必须在类体外进行初始化

```
1  #include <iostream>
2  using namespace std;
3  class Sample
4  {
5      int n;
6      static int sum;
7  public:
8      Sample(int x)
9      {   n=x;    }
10     void add( ) const
11     {   sum+=n; }
12     void disp( )
13     {
14         cout<<"n="<<n<<",sum="<<
       sum<<endl;
15     }
16 };
17 int Sample::sum = 0;
```

如果将主函数改写为

```
18 int main( )
19 {
20     Sample a(2), b(3), c(5);
21     a.add(); b.add(); c.add();
22     a.disp();b.disp();c.disp();
23     return 0;
24 }
```

程序运行结果是:

# 静态数据成员的使用

- 类的静态数据成员必须在类体外进行初始化

```
1  #include <iostream>
2  using namespace std;
3  class Sample
4  {
5      int n;
6      static int sum;
7  public:
8      Sample(int x)
9      {   n=x;    }
10     void add( ) const
11     {   sum+=n; }
12     void disp( )
13     {
14         cout<<"n="<<n<<",sum="<<
       sum<<endl;
15     }
16 };
17 int Sample::sum = 0;
```

如果将主函数改写为

```
18 int main( )
19 {
20     Sample a(2), b(3), c(5);
21     a.add(); b.add(); c.add();
22     a.disp();b.disp();c.disp();
23     return 0;
24 }
```

程序运行结果是：n=2, sum=10
n=3, sum=10
n=5, sum=10

**静态数据成员的全局性**

- 静态数据成员实际上是类域中的全局变量，其定义（初始化）不应该被放在头文件中（接口与实现分离）。

- 静态数据成员可在const成员函数中被合法地改变，而普通数据成员则不可。
- 静态数据成员可作为类的成员函数的可选参数，而普通数据成员则不可以。

# 类的静态成员函数

**静态成员函数**属于类，就像静态数据成员属于类一样。

- 可以通过类名或对象名调用静态成员函数。
- 在静态成员函数中只能直接访问静态数据成员。

# 类的静态成员函数

**静态成员函数的使用**　下面程序运行结果是?

```
1  #include <iostream>
2  using namespace std;
3  class Sample
4  {
5      int a;    static int b;                    // 成员变量、静态数据成员
6  public:
7      static int c;
8      Sample(int x) {  a=x; b+=x;  }
9      static void disp(Sample s) // 类体外的函数定义不能指定关键字static
10     {  cout<<"a="<<s.a<<", b="<<b<<endl;  }
11 };
12 int Sample::b = 10;  int Sample::c = 20;      // 静态数据成员初始化
13 int main( )
14 {
15     Sample s1(2), s2(3);
16     Sample::disp(s1);              // 可改写为s1.disp(s1); 或s2.disp(s1);
17     Sample::disp(s2);              // 可改写为s1.disp(s2); 或s2.disp(s2);
18     cout<<"c="<<Sample::c<<endl; // Sample::c 可改写为s1.c 或s2.c
19     return 0;
20 }
```

# 类的静态成员函数

程序运行结果是：

# 类的静态成员函数

程序运行结果是：

```
n=2, sum=10
a=2, b=15
a=3, b=15
c=20
```

**关于静态成员的说明**

# 类的静态成员函数

程序运行结果是：

```
n=2, sum=10
a=2, b=15
a=3, b=15
c=20
```

## 关于静态成员的说明

- **静态成员之间**可以相互访问，包括：1) 类的静态成员函数访问类的静态数据成员；2）类的静态成员函数调用类的静态成员函数。
- **静态成员函数**由同一类中的所有对象共用，只能调用静态成员变量和静态成员函数；而非静态成员函数可以任意地访问静态成员函数和静态数据成员。
- **静态成员函数**没有this指针，它不能返回非静态成员。 这是因为除了对象会调用它外，类本身也可以调用。
- **静态成员函数**与类的其他函数相比调用效率上会有少许的提高，这是由于静态成员函数没有`this`指针的额外开销。

# 类的静态成员 · 详解

- 可以通过类名调用静态成员函数，但不能通过类名来调用类的非静态成员函数。

```
1  class Point
2  {
3  public:
4      void init()
5      {
6      }
7      static void output()
8      {
9      }
10 };
11 void main()
12 {
13     Point::init();
14     Point::output();
15 }
```

编译出错：error C2352: 'Point::init' : illegal call of non-static member function

# 类的静态成员 · 详解

- 可以通过对象调用类的静态成员函数和非静态成员函数。

```
1  class Point
2  {
3  public:
4      void init()
5      {
6      }
7      static void output()
8      {
9      }
10 };
11 void main()
12 {
13     Point pt;
14     pt.init();
15     pt.output();
16 }
```

编译通过。

# 类的静态成员 · 详解

- 类的静态成员函数不能调用类的非静态成员。

```
1  class Point
2  {
3  public:
4      void init() {  }
5      static void output() { printf("%d\n", m_x); }
6  private:
7      int m_x;
8  };
9  void main()
10 {
11     Point pt;
12     pt.output();
13 }
```

编译出错：error C2597: illegal reference to data member 'Point::m_x' in a static member function
因为静态成员函数属于整个类，在类实例化对象之前就已经分配空间了，而类的非静态成员必须在类实例化对象后才有内存空间，所以这个调用就出错了，就好比没有声明一个变量却提前使用它一样。

# 类的静态成员 · 详解

- 类的非静态成员函数可以调用静态成员函数，反之则不能。

```
1  class Point
2  {
3  public:
4      Point()  { m_nPointCount++; }
5      ~Point() { m_nPointCount--; }
6      static void output() { printf("%d\n", m_nPointCount); }
7  private:
8      static int m_nPointCount;
9  };
10 void main()
11 {
12     Point pt;
13     pt.output();
14 }
```

编译出错：error LNK2001: unresolved external symbol "private: static int Point ::m_nPointCount" (?m_nPointCount@Point@@0HA) 这是因为类的静态成员变量在使用前必须初始化。 在main()函数前加上int Point::m_nPointCount = 0; 再编译链接无错误，运行程序将输出1。

# 静态成员详解

- 类的静态成员变量使用前，必须先对其进行初始化。

```
1  class Point
2  {
3  public:
4      Point()  { m_nPointCount++; }
5      ~Point() { m_nPointCount--; }
6      static void output() { printf("%d\n", m_nPointCount); }
7  private:
8      static int m_nPointCount;
9  };
10 void main()
11 {
12     Point pt;
13     pt.output();
14 }
```

编译无误，但生成可执行文件时报链接错误： error LNK2001: unresolved external symbol "private: static int Point::m_nPointCount"(?m_nPointCount@Point@@0HA) 这是因为类的静态成员变量在使用前必须先初始化。 在主函数前加上`int Point::m_nPointCount = 0;`再次编译链接则无误，运行时输出1。

# 提纲

# 类的友元函数

**类的封装性和信息的隐蔽性**

- 在类内，类的任意成员函数均能够访问类的私有成员。
- 在类外，只能由公有成员函数才能访问类的私有成员。

**通过公有成员函数接口访问私有成员**

# 类的友元函数

## 类的封装性和信息的隐蔽性

- 在类内，类的任意成员函数均能够访问类的私有成员。
- 在类外，只能由公有成员函数才能访问类的私有成员。

## 通过公有成员函数接口访问私有成员

```
12  class Point
13  {
14      int x, y;
15  public:
16      Point(int a=0, int b=0) { x=
        a; y=b; }
17      ~Point() {  }
18      void Show( ) { cout<<"Point(
        "<<x<<','<<y<<")\n"; }
19      int Getx( )  { return x; }
20      int Gety( )  { return y; }
21  }p1, p2(1,1);
22  double distance(Point &p1, Point
        &p2 )//normal
```

```
1   {   //
2       return sqrt(
3           (p1.Getx( )-p2.Getx( ))
        *(p1.Getx( )-p2.Getx( ))
4         + (p1.Gety( )-p2.Gety( ))
        *(p1.Gety( )-p2.Gety( ))
5         );
6   }
7   int main( )
8   {
9       cout << distance(p1, p2) <<
        endl;
10      return 0;
11  }
```

# 类的友元函数

## 类的封装性和信息的隐蔽性

- 在类内，类的任意成员函数均能够访问类的私有成员。
- 在类外，只能由公有成员函数才能访问类的私有成员。

## 通过公有成员函数接口访问私有成员

```
12  class Point
13  {
14      int x, y;
15  public:
16      Point(int a=0, int b=0) { x=
        a; y=b; }
17      ~Point() {  }
18      void Show( ) { cout<<"Point(
        "<<x<<','<<y<<")\n"; }
19      int Getx( )  { return x; }
20      int Gety( )  { return y; }
21  }p1, p2(1,1);
22  double distance(Point &p1, Point
        &p2 )//normal
```

```
1   {   // 多次调用公有函数，开销较大
2       return sqrt(
3           (p1.Getx( )-p2.Getx( ))
        *(p1.Getx( )-p2.Getx( ))
4         + (p1.Gety( )-p2.Gety( ))
        *(p1.Gety( )-p2.Gety( ))
5         );
6   }
7   int main( )
8   {
9       cout << distance(p1, p2) <<
        endl;
10      return 0;
11  }
```

# 类的友元函数

## 类的封装性和信息的隐蔽性

- 在类内，类的任意成员函数均能够访问类的私有成员。
- 在类外，只能由公有成员函数才能访问类的私有成员。

## 通过公有成员函数接口访问私有成员

```
12  class Point
13  {
14      int x, y;
15  public:
16      Point(int a=0, int b=0) { x=
        a; y=b; }
17      ~Point() {  }
18      void Show( ) { cout<<"Point(
        "<<x<<','<<y<<")\n"; }
19      int Getx( )  { return x; }
20      int Gety( )  { return y; }
21  }p1, p2(1,1);
22  double distance(Point &p1, Point
        &p2 )//normal
```

```
1   {   // 多次调用公有函数，开销较大
2       return sqrt(
3           (p1.Getx( )-p2.Getx( ))
        *(p1.Getx( )-p2.Getx( ))
4         + (p1.Gety( )-p2.Gety( ))
        *(p1.Gety( )-p2.Gety( ))
5         );
6   }
7   int main( )
8   {
9       cout << distance(p1, p2) <<
        endl;
10      return 0;
11  }
```

有没有办法在普通函数distance()中直接访问Point的私有成员呢？

# 类的友元函数

## Solution

回答是肯定的：有！

- 只需将 函数distance( ) 说明为类 Point 的友元（friend）函数！

**通过公有成员函数接口访问私有成员**

```
12  class Point
13  {
14      int x, y;
15  public:
16      Point(int a=0, int b=0) { x=
        a; y=b; }
17      ~Point() {  }
18      void Show( ) { cout<<"Point(
        "<<x<<','<<y<<")\n"; }
19      int Getx( )  { return x; }
20      int Gety( )  { return y; }
21  }p1, p2(1,1);
22  double distance(Point &p1, Point
        &p2 )//normal
```

```
1   {   // 多次调用公有函数，开销较大
2       return sqrt(
3           (p1.Getx( )-p2.Getx( ))
        *(p1.Getx( )-p2.Getx( ))
4         + (p1.Gety( )-p2.Gety( ))
        *(p1.Gety( )-p2.Gety( ))
5         );
6   }
7   int main( )
8   {
9       cout << distance(p1, p2) <<
        endl;
10      return 0;
11  }
```

有没有办法在普通函数distance()中直接访问Point的私有成员呢？

# 类的友元函数

友元函数的定义方法是：

```
1  class  A
2  {
3      ...
4      friend void fun(...); //说明函数fun( )是类A的友元函数
5  };
6  void fun(...)
7  {
8      ...
9      class A a;              // 于是在此函数中，可以直接访问类A的私有成员
10 }
```

将前文的程序代码修改为

# 类的友元函数

友元函数的定义方法是：

```
1  class  A
2  {
3      ...
4      friend void fun(...); //说明函数fun( )是类A的友元函数
5  };
6  void fun(...)
7  {
8      ...
9      class A a;              // 于是在此函数中，可以直接访问类A的私有成员
10 }
```

将前文的程序代码修改为

```
1  class Point
2  {
3      int x, y;
4  public:
5      ...
6      friend double distance(Point&, Point& ); // 修改之处，也可以将主函
       数main声明为一个类的友元函数
7  };
```

# 类的友元函数

```
8  double distance(Point &p1, Point &p2 )
9  {
10     // 修改之处
11     return sqrt( (p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y) );
12 }
13 int main( ) // 也可以将主函数main声明为一个类的友元函数
14 {
15     Point p1, p2(1,1);
16     cout<<distance(p1,p2)<<endl;
17     return 0;
18 }
```

关于友元函数的几点说明：

# 类的友元函数

```
8  double distance(Point &p1, Point &p2 )
9  {
10     // 修改之处
11     return sqrt( (p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y) );
12 }
13 int main( ) // 也可以将主函数main声明为一个类的友元函数
14 {
15     Point p1, p2(1,1);
16     cout<<distance(p1,p2)<<endl;
17     return 0;
18 }
```

关于友元函数的几点说明：

- 友元函数不是类的成员函数，访问权限不能限制友元函数。
- 友元函数的说明可放在private、public、protected的任意段中。
- 友元函数可以在类外直接访问包含私有和保护成员在内的所有成员。
- 友元函数可提高程序的运行效率，但破坏了类中信息的隐蔽性。
- 慎用友元函数。

# 类的友元函数

- 普通函数可以做友元函数，类的成员函数也可以作为其他类的友元函数。

```
1  class N;  // 为什么用引用性声明?
2  class M
3  {
4      int a, b;
5  public:
6      M(int x, int y)
7      {   a=x; b=y;  }
8      void Print( )
9      {
10         cout<<"a="<<a<<'\t'<<"b=
       "<<b<<endl;
11     }
12     void Setab(N &);//类M的成员
13 };
14 class N
15 {
16     int c, d;
17 public:
18     N(int a, int b)
19     {   c=a;  d=b;  }
```

```
18     void Print( )
19     {
20         cout<<"c="<<c<<'\t'<<"d=
       "<<d<<endl;
21     }
22     friend void M::Setab(N&); //
       将M类的成员函数说明成类N的友
       元。它可直接访问类N的所有成员
23 };
24 void  M::Setab(N &obj) // 类N的
       友元，可直接访问类N的所有成员
25 {   a = obj.c;   b = obj.d;   }
26 int main( )
27 {
28     M  m(25, 40);  N  n(55, 66);
29     m.Setab(n);    m.Print();
30     return 0;
31 }
```

请问：程序运行结果是什么？

# 类的友元函数

## Solution

程序运行结果是： a=55 b=66

```
class M
{
    int a, b;
public:
    M(int x, int y)
    {   a=x; b=y;  }
    void Print( )
    {
        cout<<"a="<<a<<'\t'<<"b="<<b<<endl;
    }
    void Setab(N &);//类M的成员
};
class N
{
    int c, d;
public:
    N(int a, int b)
    {   c=a;  d=b;  }
    {
        cout<<"c="<<c<<'\t'<<"d="<<d<<endl;
    }
    friend void M::Setab(N&); //将M类的成员函数说明成类N的友元。它可直接访问类N的所有成员
};
void  M::Setab(N &obj) // 类N的友元，可直接访问类N的所有成员
{   a = obj.c;   b = obj.d;   }
int main( )
{
    M  m(25, 40);  N  n(55, 66);
    m.Setab(n);    m.Print();
    return 0;
}
```

请问：程序运行结果是什么？

# 类的友元函数

## Solution

程序运行结果是：　a=55 b=66　　请运行FriendMemberFunction.cpp

```
3  {
4      int a, b;
5  public:
6      M(int x, int y)
7      {   a=x; b=y;  }
8      void Print( )
9      {
10         cout<<"a="<<a<<'\t'<<"b=
       "<<b<<endl;
11     }
12     void Setab(N &);//类M的成员
13 };
14 class N
15 {
16     int c, d;
17 public:
18     N(int a, int b)
19     {   c=a;  d=b;  }
20         cout<<"c="<<c<<'\t'<<"d=
       "<<d<<endl;
21     }
22     friend void M::Setab(N&); //
       将M类的成员函数说明成类N的友
       元。它可直接访问类N的所有成员
23 };
24 void  M::Setab(N &obj) // 类N的
       友元，可直接访问类N的所有成员
25 {   a = obj.c;   b = obj.d;   }
26 int main( )
27 {
28     M  m(25, 40);  N  n(55, 66);
29     m.Setab(n);    m.Print();
30     return 0;
31 }
```

请问：程序运行结果是什么？

# 友元类

- 一个类可以作为另一个类的友元。当一个类作为另一个类的友元时，就意味着：这个类的所有成员函数都是另一个类的友元函数。
- 如果类B是类A的友元，则类B所有的成员函数均可以访问类A的所有（含私有）成员。

```
1  class   B;
2  class   A
3  {
4      int     i,j,k;          // 类A的私有成员
5      void    funcInA( );     // 类A的成员函数，不可访问类B的私有成员
6      friend class  B;        // B类是A类的友元
7  };
8  class   B
9  {
10     int x,y,z;              // 类B的私有成员
11 public:
12     void    funcInB( );     // 可访问类A的私有成员
13 };
```

- 如果类B是类A的友元，但类A不是类B的友元，则类A的成员函数未必可以访问类B的私有成员。未必：类A与类B可能存在继承与派生等其他关系。

# 友元类

- 将一个类说明为另一个类的友元类

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

class B;        // 引用性声明
class A
{
    int x;
    void Disp( )
    {
        cout<<"x="<<x<<endl;
    }
    friend B; // 声明B是A的友元类
};

class B
{
public:
    void Set(int n)
    {
        A a;
        a.x = n;    // 可访问A类对象的私有数据成员x
        a.Disp( ); // 可访问A类对象的私有成员函数disp( )
    }
};

int main(void)
{
    B b;
    b.Set(3);

    return 0;
}
```

程序运行结果：

# 友元类

- 将一个类说明为另一个类的友元类

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

class B;        // 引用性声明
class A
{
    int x;
    void Disp( )
    {
        cout<<"x="<<x<<endl;
    }
    friend B; // 声明B是A的友元类
};

class B
{
public:
    void Set(int n)
    {
        A a;
        a.x = n;    // 可访问A类对象的私有数据成员x
        a.Disp( ); // 可访问A类对象的私有成员函数disp( )
    }
};

int main(void)
{
    B b;
    b.Set(3);

    return 0;
}
```

程序运行结果：x=3

这表明：类B所有的成员函数均可访问类A中包含私有成员在内的所有成员。

# 提纲

# 常量数据成员和常量成员函数

- 私有数据成员可以通过公有成员函数接口修改。为了保证部分私有数据成员不被修改，C++提供了自动保护私有数据成员的机制，即常数据成员和常成员函数。

## 常量数据成员

方法是在类体中定义数据成员时，在前面加上关键字const。

```
1  class Point
2  {
3      int x, y;              // 点的坐标
4      const int color;       // 点的颜色，color是常数据成员
5  public:
6      Point(int a=0, int b=0, int c=0): color(c)
7      {
8          x = a; y = b;
9      }
10 };
```

以上代码，color的初始化只能通过构造函数的成员初始化列表实现。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词`const`。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词`const`。

## 关于常量成员函数的说明

- 常量成员函数可以访问类的所有成员，但不能修改任何数据成员（含常量数据成员）的值。
- 普通成员函数可以访问类的所有成员，但不能修改常量数据成员，但是能修改除常量数据成员外的其他数据成员的值。
- 关键词`const`是函数类型的一部分，在声明和定义常成员函数时都要加关键字`const`，而且`const`也参与重载函数的区分。
- 常对象只能调用常成员函数。
  常量成员函数就是声明自己不会修改被操作对象的常数，因为你的变量不允许修改内容，所以只能调用不会修改内容的常量成员函数。如果调用其它函数就有可能修改变量的内容，所以这是不允许的。
- 常量成员函数不能调用另一个非`const`成员函数。
  如果调用了，则常量成员函数可能通过非`const`成员函数修改数据成员的值。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词`const`。

## 关于常量成员函数的说明

- 常量成员函数可以访问类的所有成员，但不能修改任何数据成员（含常量数据成员）的值。
- 普通成员函数可以访问类的所有成员，但不能修改常量数据成员，但是能修改除常量数据成员外的其他数据成员的值。
- 关键词`const`是函数类型的一部分，在声明和定义常成员函数时都要加关键字`const`，而且`const`也参与重载函数的区分。
- 常对象只能调用常成员函数。
  常量成员函数就是声明自己不会修改被操作对象的常数，因为你的变量不允许修改内容，所以只能调用不会修改内容的常量成员函数。如果调用其它函数就有可能修改变量的内容，所以这是不允许的。
- 常量成员函数不能调用另一个非`const`成员函数。
  如果调用了，则常量成员函数可能通过非`const`成员函数修改数据成员的值。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词`const`。

## 关于常量成员函数的说明

- 常量成员函数可以访问类的所有成员，但不能修改任何数据成员（含常量数据成员）的值。
- 普通成员函数可以访问类的所有成员，但不能修改常量数据成员，但是能修改除常量数据成员外的其他数据成员的值。
- 关键词`const`是函数类型的一部分，在声明和定义常成员函数时都要加关键字`const`，而且`const`也参与重载函数的区分。
- 常对象只能调用常成员函数。
  常量成员函数就是声明自己不会修改被操作对象的常数，因为你的变量不允许修改内容，所以只能调用不会修改内容的常量成员函数。如果调用其它函数就有可能修改变量的内容，所以这是不允许的。
- 常量成员函数不能调用另一个非`const`成员函数。
  如果调用了，则常量成员函数可能通过非`const`成员函数修改数据成员的值。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词const。

## 关于常量成员函数的说明

- 常量成员函数可以访问类的所有成员，但不能修改任何数据成员（含常量数据成员）的值。
- 普通成员函数可以访问类的所有成员，但不能修改常量数据成员，但是能修改除常量数据成员外的其他数据成员的值。
- 关键词const是函数类型的一部分，在声明和定义常成员函数时都要加关键字const，而且const也参与重载函数的区分。
- 常对象只能调用常成员函数。
  常量成员函数就是声明自己不会修改被操作对象的常数，因为你的变量不允许修改内容，所以只能调用不会修改内容的常量成员函数。如果调用其它函数就有可能修改变量的内容，所以这是不允许的。
- 常量成员函数不能调用另一个非const成员函数。
  如果调用了，则常量成员函数可能通过非const成员函数修改数据成员的值。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词const。

## 关于常量成员函数的说明

- 常量成员函数可以访问类的所有成员，但不能修改任何数据成员（含常量数据成员）的值。
- 普通成员函数可以访问类的所有成员，但不能修改常量数据成员，但是能修改除常量数据成员外的其他数据成员的值。
- 关键词const是函数类型的一部分，在声明和定义常成员函数时都要加关键字const，而且const也参与重载函数的区分。
- 常对象只能调用常成员函数。
  常量成员函数就是声明自己不会修改被操作对象的常数，因为你的变量不允许修改内容，所以只能调用不会修改内容的常量成员函数。如果调用其它函数就有可能修改变量的内容，所以这是不允许的。
- 常量成员函数不能调用另一个非const成员函数。
  如果调用了，则常量成员函数可能通过非const成员函数修改数据成员的值。

# 常量数据成员和常量成员函数

## 常量成员函数

常量成员函数的定义方法：在函数头尾部、参数的右括号后加关键词const。

## 关于常量成员函数的说明

- 常量成员函数可以访问类的所有成员，但不能修改任何数据成员（含常量数据成员）的值。
- 普通成员函数可以访问类的所有成员，但不能修改常量数据成员，但是能修改除常量数据成员外的其他数据成员的值。
- 关键词const是函数类型的一部分，在声明和定义常成员函数时都要加关键字const，而且const也参与重载函数的区分。
- 常对象只能调用常成员函数。
  常量成员函数就是声明自己不会修改被操作对象的常数，因为你的变量不允许修改内容，所以只能调用不会修改内容的常量成员函数。如果调用其它函数就有可能修改变量的内容，所以这是不允许的。
- 常量成员函数不能调用另一个非const成员函数。
  如果调用了，则常量成员函数可能通过非const成员函数修改数据成员的值。

# 常量数据成员和常量成员函数

- 常量成员函数和一般成员函数对常量数据成员和一般数据成员的访问

```
1  #include <iostream>
2  using namespace std;
3  class Point                    // 注意类中的两个fun( )函数是不同的函数
4  {
5      int x;                     // 普通数据成员
6      const int y;               // 常量数据成员
7  public:
8      Point(int a=0, int b=0): x(a), y(b) { } // 默认构造函数
9      int fun( )                 // 一般成员函数
10     {
11         x = 5;                 // 修改一般数据成员x，合法
12         y = 6;                 // 修改常量数据成员，非法
13         return x+y;            // 访问x和y合法
14     }
15     int fun( ) const           // 常量成员函数，相当const this指针作函数参数
16     {
17         x = 5;                 // 修改一般数据成员，非法
18         y = 6;                 // 修改常量数据成员，非法
19         return x+y;            // 访问x和y，合法
20     }
21 };
```

# 常量数据成员和常量成员函数

- 关键词const参与区分重载函数；常量对象调用常量成员函数。

```
1  #include <iostream>
2  using namespace std;
3  class Point
4  {
5      int x;              const int y;
6  public:
7      Point(int a=0, int b=0): x(a), y(b) { }
8      int fun( );                 // 第1个fun函数
9      int fun( ) const;           // 第2个fun函数，类中声明有const
10 };
11 int Point::fun( )
12 {   return x+y;   }
13 int Point::fun( ) const      // 类外定义也必须有const
14 {   return x-y;   }
15 int main( )
16 {
17     Point p1(1,8);     const Point p2(1,8);
18     //
19     cout<<p1.fun( )<<',␣'<<p2.fun( )<<endl;
20     return 0;
21 }
```

# 常量数据成员和常量成员函数

- 关键词const参与区分重载函数；常量对象调用常量成员函数。

```
1  #include <iostream>
2  using namespace std;
3  class Point
4  {
5      int x;              const int y;
6  public:
7      Point(int a=0, int b=0): x(a), y(b) { }
8      int fun( );                 // 第1个fun函数
9      int fun( ) const;           // 第2个fun函数，类中声明有const
10 };
11 int Point::fun( )
12 {   return x+y;   }
13 int Point::fun( ) const      // 类外定义也必须有const
14 {   return x-y;   }
15 int main( )
16 {
17     Point p1(1,8);     const Point p2(1,8);
18     // p1调用的是第1个fun函数；p2调用的是第2个fun函数，即常量函数
19     cout<<p1.fun( )<<',␣'<<p2.fun( )<<endl;
20     return 0;
21 }
```

# 常量数据成员和常量成员函数

## 程序的运行结果

9, -7

```
2  using namespace std;
3  class Point
4  {
5      int x;              const int y;
6  public:
7      Point(int a=0, int b=0): x(a), y(b) { }
8      int fun( );                 // 第1个fun函数
9      int fun( ) const;           // 第2个fun函数，类中声明有const
10 };
11 int Point::fun( )
12 {   return x+y;   }
13 int Point::fun( ) const      // 类外定义也必须有const
14 {   return x-y;   }
15 int main( )
16 {
17     Point p1(1,8);     const Point p2(1,8);
18     // p1调用的是第1个fun函数；p2调用的是第2个fun函数，即常量函数
19     cout<<p1.fun( )<<',␣'<<p2.fun( )<<endl;
20     return 0;
21 }
```

# 提纲

# 关键词MUTABLE的用法

## 突破const函数的限制

# 关键词MUTABLE的用法

## 突破const函数的限制

- 若需要修改常对象中的某个数据成员，可将数据成员定义为mutable类型。mutable是为了突破const函数的限制而设置的：即使在一个const函数中，被mutable修饰的变量，将永远处于可变的状态。
- 关键词mutable的意义
  - 关键词const意思是“这个函数不修改对象的内部状态”。

    为了保证这一点，编译器也会主动替你检查，确保你没有修改对象成员变量——否则内部状态就变了。
  - 关键词mutable意思是“这个成员变量不影响对象的内部状态”。

# 关键词MUTABLE的用法

## 突破const函数的限制

- 若需要修改常对象中的某个数据成员，可将数据成员定义为mutable类型。mutable是为了突破const函数的限制而设置的：即使在一个const函数中，被mutable修饰的变量，将永远处于可变的状态。
- 关键词mutable的意义
  - 关键词const意思是“这个函数不修改对象的内部状态”。

    为了保证这一点，编译器也会主动替你检查，确保你没有修改对象成员变量——否则内部状态就变了。
  - 关键词mutable意思是“这个成员变量不影响对象的内部状态”。

## 关键词mutable的用法

- 为调试程序，编程者在常函数中定义一个变量用于统计某对象的访问次数。
- 编译器并不知道该变量并不影响对象的其他功用，会阻止在声明为const的函数中修改这个变量的行为。
- 如果把这个计数变量声明为mutable，编译器就明白了：这个变量不影响对象的内部状态、并不影响const语义，就不需要禁止const函数修改它。

# 关键词MUTABLE的用法

```
1  #include <iostream>
2  using namespace std;
3  class TestMutable
4  {
5  public:
6      TestMutable(){i=0;}
7      int Output() const
8      {
9          return i++; // error C2166:  l-value specifies const object
10     }
11 private:
12     int i;
13 };
14
15 int main()
16 {
17    TestMutable testMutable;
18    cout << testMutable.Output() << endl;
19    return 0;
20 }
```

显然i++在const修饰的函数里是编译通不过的。

# 关键词MUTABLE的用法

```
1  #include <iostream>
2  using namespace std;
3  class TestMutable
4  {
5  public:
6      TestMutable(){i=0;}
7      int Output() const
8      {
9          return i++;
10     }
11 private:
12     mutable int i;  // 请分析关键词mutable的作用
13 };
14
15 int main()
16 {
17    TestMutable testMutable;
18    cout << testMutable.Output() << endl;
19    return 0;
20 }
```

在 int i 前面加上mutable上面就能编译通过了。

# 提纲

# 总结与思考

## 选择题

1. 下面各函数中，（　）不是类的成员函数。
A. 构造函数　　B. 析构函数
C. 友元函数　　D. 拷贝构造函数

2. 下面有关静态数据成员的描述中，（　）是错误的。
A. 定义静态数据成员时，前面要加关键字static
B. 静态数据成员要在类体外进行初始化
C. 在类外访问公有静态数据成员时，要在静态数据成员名前加 <类名> 和作用域运算符
D. 静态数据成员不是所属类的所有对象共有的

3. 下面有关友元函数的描述中，错误的是（　）。
A. 友元函数不是成员函数
B. 友元函数加强了类的封装性
C. 在友元函数中可以访问所属类的私有成员
D. 友元函数的作用是提高程序的执行效率

# 总结与思考

## 选择题

1. 下面各函数中，（ C ）不是类的成员函数。
A. 构造函数　　B. 析构函数
C. 友元函数　　D. 拷贝构造函数

2. 下面有关静态数据成员的描述中，（　　）是错误的。
A. 定义静态数据成员时，前面要加关键字static
B. 静态数据成员要在类体外进行初始化
C. 在类外访问公有静态数据成员时，要在静态数据成员名前加 <类名> 和作用域运算符
D. 静态数据成员不是所属类的所有对象共有的

3. 下面有关友元函数的描述中，错误的是（　　）。
A. 友元函数不是成员函数
B. 友元函数加强了类的封装性
C. 在友元函数中可以访问所属类的私有成员
D. 友元函数的作用是提高程序的执行效率

# 总结与思考

## 选择题

1. 下面各函数中，（ C ）不是类的成员函数。
A. 构造函数　　B. 析构函数
C. 友元函数　　D. 拷贝构造函数

2. 下面有关静态数据成员的描述中，（ D ）是错误的。
A. 定义静态数据成员时，前面要加关键字static
B. 静态数据成员要在类体外进行初始化
C. 在类外访问公有静态数据成员时，要在静态数据成员名前加 <类名> 和作用域运算符
D. 静态数据成员不是所属类的所有对象共有的

3. 下面有关友元函数的描述中，错误的是（　　）。
A. 友元函数不是成员函数
B. 友元函数加强了类的封装性
C. 在友元函数中可以访问所属类的私有成员
D. 友元函数的作用是提高程序的执行效率

# 总结与思考

## 选择题

1. 下面各函数中，（ C ）不是类的成员函数。
A. 构造函数　　B. 析构函数
C. 友元函数　　D. 拷贝构造函数

2. 下面有关静态数据成员的描述中，（ D ）是错误的。
A. 定义静态数据成员时，前面要加关键字static
B. 静态数据成员要在类体外进行初始化
C. 在类外访问公有静态数据成员时，要在静态数据成员名前加 <类名> 和作用域运算符
D. 静态数据成员不是所属类的所有对象共有的

3. 下面有关友元函数的描述中，错误的是（ B ）。
A. 友元函数不是成员函数
B. 友元函数加强了类的封装性
C. 在友元函数中可以访问所属类的私有成员
D. 友元函数的作用是提高程序的执行效率

# 总结与思考

4. 下述关于成员函数特征的描述中，错误的是（　）。
A. 成员函数一定是内联函数
B. 成员函数可以重载
C. 成员函数可以设置参数的缺省值
D. 成员函数可以是静态的

5. 对C++编译器区分重载成员函数无任何意义的信息是（　）。
A. 参数类型　B. 参数个数　C. 返回值类型　D. 关键词`const`

## 编程题

6. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：1）设置实部值，2）设置虚部值，3）读取/返回实部值，4）读取/返回虚部值，5）输出复数。在主函数中定义这个复数类的一个对象，然后对所有成员函数进行测试，即调用所有成员函数。

7. 为上述题目的复数类增加一个友元函数。该友元函数的原型为`void print(Complex &)`，用于输出一个复数对象。在主函数中自行设计程序代码以测试该函数。

# 总结与思考

4. 下述关于成员函数特征的描述中，错误的是（ A ）。
A. 成员函数一定是内联函数
B. 成员函数可以重载
C. 成员函数可以设置参数的缺省值
D. 成员函数可以是静态的

5. 对C++编译器区分重载成员函数无任何意义的信息是（　）。
A. 参数类型　B. 参数个数　C. 返回值类型　D. 关键词`const`

## 编程题

6. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：1）设置实部值，2）设置虚部值，3）读取/返回实部值，4）读取/返回虚部值，5）输出复数。在主函数中定义这个复数类的一个对象，然后对所有成员函数进行测试，即调用所有成员函数。

7. 为上述题目的复数类增加一个友元函数。该友元函数的原型为`void print(Complex &)`，用于输出一个复数对象。在主函数中自行设计程序代码以测试该函数。

# 总结与思考

4. 下述关于成员函数特征的描述中，错误的是（ A ）。
A. 成员函数一定是内联函数
B. 成员函数可以重载
C. 成员函数可以设置参数的缺省值
D. 成员函数可以是静态的

5. 对C++编译器区分重载成员函数无任何意义的信息是（ C ）。
A. 参数类型　B. 参数个数　C. 返回值类型　D. 关键词`const`

## 编程题

6. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：1）设置实部值，2）设置虚部值，3）读取/返回实部值，4）读取/返回虚部值，5）输出复数。在主函数中定义这个复数类的一个对象，然后对所有成员函数进行测试，即调用所有成员函数。

7. 为上述题目的复数类增加一个友元函数。该友元函数的原型为`void print(Complex &)`，用于输出一个复数对象。在主函数中自行设计程序代码以测试该函数。

# 总结与思考

4. 下述关于成员函数特征的描述中，错误的是（ A ）。
A. 成员函数一定是内联函数
B. 成员函数可以重载
C. 成员函数可以设置参数的缺省值
D. 成员函数可以是静态的

5. 对C++编译器区分重载成员函数无任何意义的信息是（ C ）。
A. 参数类型　B. 参数个数　C. 返回值类型　D. 关键词`const`

**编程题**

6. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：1）设置实部值，2）设置虚部值，3）读取/返回实部值，4）读取/返回虚部值，5）输出复数。在主函数中定义这个复数类的一个对象，然后对所有成员函数进行测试，即调用所有成员函数。

7. 为上述题目的复数类增加一个友元函数。该友元函数的原型为`void print(Complex &)`，用于输出一个复数对象。在主函数中自行设计程序代码以测试该函数。

# 总结与思考

4. 下述关于成员函数特征的描述中，错误的是（ A ）。
A. 成员函数一定是内联函数
B. 成员函数可以重载
C. 成员函数可以设置参数的缺省值
D. 成员函数可以是静态的

5. 对C++编译器区分重载成员函数无任何意义的信息是（ C ）。
A. 参数类型　B. 参数个数　C. 返回值类型　D. 关键词`const`

**编程题**

6. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：1）设置实部值，2）设置虚部值，3）读取/返回实部值，4）读取/返回虚部值，5）输出复数。在主函数中定义这个复数类的一个对象，然后对所有成员函数进行测试，即调用所有成员函数。

7. 为上述题目的复数类增加一个友元函数。该友元函数的原型为`void print(Complex &)`，用于输出一个复数对象。在主函数中自行设计程序代码以测试该函数。

# 总结与思考

4. 下述关于成员函数特征的描述中，错误的是（ A ）。
A. 成员函数一定是内联函数
B. 成员函数可以重载
C. 成员函数可以设置参数的缺省值
D. 成员函数可以是静态的

5. 对C++编译器区分重载成员函数无任何意义的信息是（ C ）。
A. 参数类型　B. 参数个数　C. 返回值类型　D. 关键词`const`

**编程题**

6. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：1）设置实部值，2）设置虚部值，3）读取/返回实部值，4）读取/返回虚部值，5）输出复数。在主函数中定义这个复数类的一个对象，然后对所有成员函数进行测试，即调用所有成员函数。

7. 为上述题目的复数类增加一个友元函数。该友元函数的原型为`void print(Complex &)`，用于输出一个复数对象。在主函数中自行设计程序代码以测试该函数。