Module03-08
C++ 面向对象编程：类与对象

- 楔子
- 语言基础
- 面向对象编程
- 泛型编程

面向对象编程

- 本单元的课程将从以下几个方面阐述 C++ 语言的抽象机制：
  - 类与对象
  - 操作符重载
  - 继承
  - 类层次结构
  - 异常处理

面向对象编程 - 类与对象

- 类与对象：
  - 类
  - 类对象

Those types are not "abstract";

they are as real as int and float .

– Doug McIlroy

类 - 关于类

- 关于类 (Classes)
  - 类用来描述拥有相同特性（属性）、行为（操作）的一组对象
  - C++ 语言中，类是一种支持用户自定义类型的机制
  - 通过封装、访问控制来实现信息的隐藏
  - 通过继承 (Inheritance) 、重写 (Overriding) 实现多态 (Polymiorphism)
  - 类对象用属性承载其状态，用操作来改变或获取其状态
  - 类定义了一个命名的作用域
- 关键字 struct 和 class
  - struct 是 class 的另一种称谓，只是有些许差异（后面的内容中描述二者之间的差异）

类 - 类的成员

- 一个简单的 Message 类以及一些关联的操作：

```
typedef char byte;
typedef unsigned short uShort;
typedef unsigned int uInt;

unsigned short msgVersion;

struct Message {
    uShort length;
    uInt sequenceNum;
    std::string content;
};

void makeMsg(Message& msg, const uInt& seqNum,
             const std::string& contnt);
void marshal(byte* result, const Message& msg);
void demarshal(Message& msg, const byte* result);
```

类 - 类的成员

- 将关联的操作纳入类的成员

```
typedef char byte;
typedef unsigned short uShort;
typedef unsigned int uInt;

unsigned short msgVersion;

struct Message {
    uShort length;
    uInt sequenceNum;
    std::string content;

    // 现在以下三个函数就是类 Message 的成员函数
    void makeMsg(const uInt& seqNum,
                 const std::string& contnt);
    void marshal(byte* result);
    void demarshal(const byte* result);
};
```

类 - 类的成员

- 类的成员
  - 数据成员 (Data Members)
    - 如类 Message 的数据成员：
      `length`、 `sequenceNum`、 `content`
  - 成员函数 (Member Functions)
    - 如类 Message 的成员函数：
      `makeMsg()`、 `marshal()`、 `demarshal()`
  - 成员的作用域
    - 类成员的作用域局限于该类，如 `Message` 中的 `length`
      - 不是全局的某个 `length`
      - 不是某个 `namespace` 下的 `length`
      - 不是某个其它类如 `Array` 等的 `length`
      - 不是某个函数局部的 `length`

类 - 成员函数

- 成员函数的定义
  - 成员函数可以在 struct 或 class 的 body 中定义，也可以在 struct 或 class 的 body 外定义，在外面定义需冠以类的名称，以示作用域

```
struct Message { /* ... */ };

void Message::makeMsg(const uInt& seqNum,
         const std::string& contnt) {
    sequenceNum = seqNum;
    content = contnt;
    length = 3 + 5 + 3 + content.length();
    if (msgVersion > 3) {
        length += 1;
    }
}
```

类 - 成员函数

- 成员的访问
  - 类对象：通过 . 访问
  - 类对象的指针：通过 -> 访问

```
Message m1;
m1.makeMsg(1200, "hello, Knuth!");

Message m2;
m2.makeMsg(1201, "hello, Joy!");

Message* mp = &m2;
cout << mp->length << endl;
```

类 - 访问控制

- Message 类定义的问题：
  - 我们可以使用成员函数 makeMsg() 来操作数据成员如：length、 sequenceNum、 content
    ```
    m1.makeMsg(1200, "hello, Knuth!");
    ```
  - 同时我们也可以（甚至是在调用 makeMsg() 后）直接操作这些数据成员：
    ```
    m1.length = 23;
    m1.content = "hello, Guy!";
    ```
  - 究竟怎样操作？不同的调用者，可能使用不同的方式，并且产生的结果不一定都正确。
- 怎么办？
  - 可以控制对类成员的访问！

类 - 访问控制

- 改造过的 Message 类

```
struct Message {  // 使用 ' class Message { ' 也一样
private: // 注意这里多了个label- private
    uShort length;
    uInt sequenceNum;
    std::string content;

public:  // 注意这里多了个label- public
    void makeMsg(const uInt& seqNum, const std::string& contnt);
    void marshal(byte* result);
    void demarshal(const byte* result);
};
```

```
// 现在情况变了！
Message m1;
m1.makeMsg(1200, "hello, Knuth"); // OK, makeMsg()是Message的public
成员
m1.length = 23; // 错误, length是Message的private成员
```

类 - 访问控制

- public 和 private
  - public 成员
    - 类对象的 public 接口，可以通过类对象不受限制的调用
    - public 成员可以是数据成员，也可以是成员函数
  - private 成员
    - 只能被该类的成员函数或该类的友元 (friend) 调用，除此之外，不能通过类对象直接调用
    - private 成员可以是数据成员，也可以是成员函数
  - 类定义中可以出现多个 public 或 private 标签

注意：涉及到类的继承体系，访问控制有额外的规则（后续课程的内容）

类 - struct 和 class

- 关键字 struct 和 class
  - 一个 struct 就是一个 class，两者之间有些微差别：
    - struct 中成员默认是 public 的，class 的成员默认是 private 的
    - 在类的继承体系中，struct 与 class 也有些许差异（以后的课程内容）
  - 下面两种类定义方式等效：

```
struct S {
    S();
private:
    int a;
    int b;
};
```

```
class S {
public:
    S();
private:
    int a;
    int b;
};
```

类 - 构造函数

- Message 类的 makeMsg()
  - makeMsg() 的工作：初始化 Message 的对象（为 3 个数据成员赋值，且有一些额外的逻辑）
  - 只有通过调用 makeMsg()， Message 对象的数据成员才能得到正确的初始化
  - 但是，不能保证调用者都会记住正确初始化 Message 对象的方法，比如：不调用 makesg()，或者多次调用 makeMsg()，都不是正确的途径，如（下面伪码中就没有调用 makeMsg()）：

```
Message m1; // 下面没有调用 makeMsg()初始化 m1
byte buf[m1.getLength()];
m1.marshal(buf);
send(peerAddr, buf, m1.getLength()); // 糟糕，发了个空消息给对端
```

类 - 构造函数

- 构造函数 (Constructor)
  - 构造函数是类中比较特殊的函数：
    - 函数名与类名相同
    - 函数不可声明有返回值（包括 void ）
    - 如果类中定义了构造函数，该类的所有对象的初始化都将通过对类的某个构造函数的调用完成
    - 如果在类中未定义任何构造函数，则编译器会根据其需要隐式的产生一个默认构造函数（无参数的构造函数）
  - 除此之外，构造函数如同一般函数：
    - 可以有或没有参数
    - 可以对构造函数进行重载
    - ...

类 - 构造函数

- 添加构造函数的 Message 类

```
class Message {
private:
    uShort length;
    uInt sequenceNum;
    std::string content;

public:
    Message();    // 注意：构造函数都没有返回值！
    Message(const uInt& seqNum, const std::string& content =
"hello!");
    void marshal(byte* result);
    void demarshal(const byte* result);
};
```

```
Message m1; // call Message()
Message m2(1200, "hello, Knuth!");  // call Message(const uInt&,
const std::string&);
```

类 - 静态成员

- 关注 Message 构造函数中的全局对象 msgVersion
  - 在之前的 makeMsg() 也就是后来的构造函数中，使用了一个全局对象 msgVersion，因为每个 Message 的对象需要知道消息协议的版本： msgVersion 。
    构造函数的定义：

```
Message::Message(const uInt& seqNum,
                 const std::string& contnt) {
    sequenceNum = seqNum;
    content = contnt;
    length = 3 + 5 + 3 + content.length() ;
    if (msgVersion > 3) {
        length += 1;
    }
}
```

考虑到数据的封装，还有全局对象的不便性，我们要消除 msgVersion 这个全局对象，但是怎样做？

类 - 静态成员

- 将 msgVersion 当作 Message 类的静态成员 (Static Member)

```
class Message {
private:
    static uShort msgVersion;    // 静态数据成员
    uShort length;
    uInt sequenceNum;
    std::string content;

public:
    Message();
    Message(const uInt& seqNum, const std::string& content =
"hello!");

    static uShort getVersion();    // 静态成员函数
    void marshal(byte* result);
    void demarshal(const byte* result);
    uShort getLength() const;
};
```

类 - 静态成员

- 关于静态成员
  - 类的静态成员属于该类的所有对象，换一种说法就是：类的静态成员不专属与该类的某个对象
  - 一个类的所有对象共享该类的静态数据成员的状态
  - 对类的静态成员访问，不必（但也可以）通过该类的某个对象
  - 类的静态成员：静态数据成员必须在类外某处另行定义

```
// 必须在类外定义msgVersion
uShort Message::msgVersion = 5;
uShort Message::getVersion() {
    return msgVersion;
}

uShort v1 = Message::getVersion();  // OK, v1 == 5
Message m1;
uShort v2 = m1.getVersion(); // OK, v2 == 5
```

类 - 类类型的 size

- 类类型的 size
  - 基本上是类的非静态数据成员的 size 之和（注意：是非静态数据成员）
  - 还有内存中字节对齐的影响，结果一般是指针类型 size 的整倍数
  - 示例：

```
class A {    // sizeof(A) == 8
    char sx;
    int ix;
};

class B {    // sizeof(B) == 8
    static int ssx;
    int sx;
    int ix;
};
```

类 - 类类型的 size

- 位域 (bit-field)
  - 指示类的数据成员存放在指定大小的 bits 中，如：

```
// Intel x87 FPU control word.
struct fpu_control {
    enum precision_control { single, double_prec = 2, extended };
    enum rounding_control { nearest, down, up, truncate };

    int :4; // Reserved
    rounding_control round_ctl :2;
    precision_control prec_ctl :2;
    int :2; // Reserved
    bool precision :1;
    bool underflow :1;
    bool overflow :1;
    bool zero_divide :1;
    bool denormal :1;
    bool invalid_op :1;
};
```

类 - 类类型的 size

- 位域 (bit-field) （续）
  - 带位域数据成员跟一般方式的数据成员类似，但需注意以下几点：
    - 不能是静态数据成员、不能是非整型类别的数据成员
    - 无法对位域的数据成员取地址
    - 不能将位域数据成员用于非 const 引用的初始化
    - 当我们将位域数据成员用于 const 引用的初始化，编译器总是会先产生一个临时对象，用临时对象为 const 引用初始化
    - 位域的大小可以指定得比数据成员的 size 大，如为（ 32bits 的实现中） int 型的数据成员指定成 36 ，但会超出的部分不是数据成员的值的一部分
    - 位域数据成员的放置次序和对齐方式是 C++ 实现定义的（ C++ 标准并无规定）

类 - 类类型的 size

- 再看类类型的 size

```
class A { // sizeof(A) == 4
    char sx :2;
    int ix :6;
};
```

类 - 类类型的 size

- 空类 (empty class) 或无非静态数据成员的类
  - 如果一个类没有非静态数据成员，或甚至无任何成员，其 size 是多大？会是 0 吗？

```
class C { // sizeof(C) == ?
    static double d;
public:
    void f();
};

class D { // sizeof(D) == ?
};
```

类 - 类对象的复制

- 复制构造函数 (Copy Constructor)
  - 复制构造函数的形式（以 Message 为例）：
    - `Message(const Message& msg);`
  - 关于复制构造函数：
    - 如果在类中没有定义复制构造函数，编译器将根据其需要隐式的产生一个
    - 复制构造函数不能被重载

```
Message m1;
Message m2 = m1; // call Message(const Message&)
Message m3(m2); // call Message(const Message&)
```

类 - 类对象的复制

- 赋值复制 (Assignment Copy)
  - 赋值操作符函数的形式（以 Message 为例）：
    - `Message& operator=(const Message& msg);`
  - 关于赋值赋值操作符：
    - 如果类中没有定义赋值操作符函数，编译器将视其需要隐式的产生一个

```
Message m1;
Message m2;
// ...
m2 = m1; // call Message& operator=(const Message&)
```

类 - 类对象的复制

- 类数据成员的复制
  - 默认按类的数据成员复制
  - 我们也可以定义自己的复制逻辑（后续课程内容）

类 - const 成员函数

- 关于类的 const 函数
  - 对于不用或不允许更改数据成员值的操作，我们通常定义为 const 成员函数
  - const 函数表明，其不会修改类的数据成员
  - const 关键字是函数类型的一部分
  - const 或者非 const 对象都可以调用该类的 const 成员函数，但非 const 成员函数则只能被非 const 对象调用
  - 静态成员函数不能定义成 const

类 - const 成员函数

- const 成员函数示例：

```
const Message m1;
uShort len = m1.getLength(); // OK, getLength()为const

byte ba[len];
m1.marshal(ba); // Error
Message m2;
m2.marshal(ba); // Error
m2.getLength(); // OK
```

类 - 自引用与 this 指针

- 假设有这么一个 Buffer 类

```
class Buffer {
    int _size;
    int gOffset;
    int pOffset;
    char* array;
public:
    enum {
        BUF_SIZE = 512
    };
    Buffer(const int& bufSize =
BUF_SIZE);
     ~Buffer();

    int size() const;
    char get() const;
    void put(char c);
};
```

```
Buffer::Buffer(const int&
bufSize) :
    _size(bufSize), gOffset(0),
pOffset(0) {
    array = new char[_size];
}

void Buffer::put(char c) {
    array[pOffset++] = c;
}

int main() {
    // 往Buffer中添加字符
    Buffer buf;
    buf.put('A');
    buf.put('B');
    // ....
}
```

类 - 自引用与 this 指针

- Buffer::put() 可以这么改

```
Buffer& Buffer::put(char c) {
    array[pOffset++] = c;
    return *this;
}

int main() {
    // 往Buffer中添加字符
    Buffer buf;
    // 连续put
    buf.put('A').put('B').put('X');
    // ....
}
```

类 - 自引用与 this 指针

- this 指针
  - 在非静态成员函数中， this 是一个指针，指向调用该函数时的那个对象，比如上例中， this 指针指向对象 buf
  - 在类 T 的非 const 成员函数中， this 的类型是 T* ；在 const 成员函数中， this 的类型是 const T*
  - 不能取得 this 指针的地址，也不能为 this 指针赋值
  - 在非静态成员函数中， this 指针的使用都是隐含的，如 put 的定义，也可以显式的使用 this ：

```
Buffer& Buffer::put(char c) {
    this->array[pOffset++] = c; // 显式的使用this指针
    return *this;
}
```

类 - mutable 数据成员

- 局部修改 const 对象
  - 一些情况下，想修改 const 对象的某些数据成员（一小部分，不是全部），如 Buffer 类的 get() 函数，需要更改成员数据 gOffset，但 get() 函数被定义为 const，也就是不能更改类数据成员的状态，事实上 get() 只是提取一个字符而已，并没改变 Buffer 中的内容
  - 使用 const_cast，临时将对象的 const 属性去掉，但是不安全：

```
char Buffer::get() const {
    Buffer* tmp = const_cast<Buffer*>(this);
    return tmp->array[tmp->gOffset++];
}
```

类 - mutable 数据成员

- mutable 数据成员
  - 将数据成员声明为 mutable，表明即使是 const 对象的成员，该数据成员也是可变的，静态数据成员不能是 mutable 的
  - 可以使用 mutable 成员，避免破坏对象的 const 语义

```
class Buffer {
    int _size;
    mutable int gOffset;  // mutable成员
    int pOffset;
    char* array;
public:
    enum { BUF_SIZE = 512 };
    Buffer(const int& bufSize = BUF_SIZE);

    int size() const;
    char get() const;
    Buffer& put(char c);
};
char Buffer::get() const {
    return array[gOffset++]; // 修改gOffset
}
```

类 - 类的 inline 函数

- inline 函数
  - 如果类的一个成员函数在类 body 内定义，该函数默认被当作 inline 函数
  - 同普通函数一样，类的成员函数可以定义为 inline 函数
  - 注意：现代编译器并非对所有声明为 inline 的函数都按 inline 实现，对于 inline 函数，各家编译器有自己的优化算法

类 - 嵌套类

- 嵌套类 (Nested Class)
  - 在一个类 body 中定义的类，称为嵌套类

```
class IntList {
public:
    IntList();
    // ...
private:
    struct Node {
        int value;
        Node* prev;
        Node* next;
    };

    Node* _begin;
    Node* _end;
};
```

类 - 局部类

- 局部类 (Local Class)
  - 一个类定义不仅可以出现在全局、名字空间、另一个类的定义中（嵌套类），还可以出现在函数定义中的任意语句块中
  - 推广而言，任意自定义类型： class 、 union 、 enum 等的定义都可以出现在函数定义中的任意语句块中

类对象 - 析构函数

- 再看 Buffer 的数据成员 array
  - array 是一个 char* 类型，在 Buffer 的构造函数中通过 new 操作符初始化，指向堆区、一个维数为 _size 的数组
  - 如果一个 Buffer 对象被销毁， array 所指向堆区的内存空间如何处理？

类对象 - 析构函数

- 析构函数 (Destructor)
  - 如果在类中没有定义析构函数，编译器会隐式产生一个
  - 析构函数没有参数，同构造函数一样没有返回值
  - 不能被重载
  - 如果在构造对象过程中，申请了某些资源，如：打开文件、获取一个锁、自由存储（如通过 new 申请的堆内存），这种情况下就有必要定义析构函数
  - 当一个自动变量离开其作用域时、或一个指向动态对象的指针被 delete 时等情况下，析构函数会被隐式的调用
  - 没有特殊需要，不要显式的调用析构函数

类对象 - 析构函数

- Buffer 的析构函数

```
class Buffer {
    int _size;
    mutable int gOffset;  // mutable成员
    int pOffset;
    char* array;
public:
    enum { BUF_SIZE = 512 };
    Buffer(const int& bufSize = BUF_SIZE);
    ~Buffer();

    int size() const;
    char get() const;
    Buffer& put(char c);
};

// Buffer的析构函数
Buffer::~Buffer() {
    delete[] array;
}
```

类对象 - 默认构造函数

- 关于默认构造函数
  - 调用时不必提供参数的构造函数即是默认构造函数，如 Buffer 类的构造函数：

```
Buffer(const int& bufSize = BUF_SIZE);
```

  或 Message 类的无参数构造函数：

```
Message();
```

  - 如果一个类声明了一个默认构造函数，则会使用它；如果没有声明任何构造函数，则编译器会在必要的情况下设法生成一个。编译器生成的默认构造函数隐式的为类类型的成员和其基类调用相关的默认构造函数。

类对象 - 默认构造函数

- 示例：

```
struct Misc {
    int size;
    Buffer buf;
    Matrix m;
};
Misc miscObj;
```

  - 示例说明：
    - 由于类 Misc 没有任何构造函数，且需要生成对象 miscObj，所以编译器有必要为类 Misc 生成一个默认构造函数
    - 在生成的 Misc 默认构造函数中，会调用 Buffer 类和 Matrix 类的默认构造函数：
      - 如果 Buffer 和 Matrix 类有用户定义的默认构造函数，调用之；如果没有默认构造函数，且用户没有定义其它构造函数，则由编译器负责为它们生成默认构造函数；否则报错
    - 但生成的 Misc 默认不会试图初始化成员数据 size（非类对象）

类对象 - 默认构造函数

- 编译器生成默认构造函数
  - 编译器仅在必要的时候，才会为类产生一个默认构造函数：
    - 一个类中没有声明任何构造函数，且需要生成对象时
  - 当类中有 const 或引用类型的成员时，必须显式提供构造函数

```
// 类C必须显式提供构造函数，否则在创建对象时会报错
struct C {
    const int i;
    const int& ir;
    int& nr;
};

C c; // Error
```

类对象 - 对象的复制

- 对象的复制方式：
  - 调用复制构造函数
  - 调用赋值操作符（=）
- 显式提供复制构造和赋值操作符函数
  - 对象间默认复制行为是按数据成员复制
  - 如果类的数据成员中有引用、指针类型的对象，则有必要显式提供复制构造函数和赋值操作符函数

```
// 编译器产生的复制构造函数的复制动作，按成员复制
Buffer::Buffer(const Buffer& that) {
    _size = that._size;
    gOffset = that.gOffset;
    pOffset = that.pOffset;
    array = that.array; // 隐患！！
}
```

类对象 - 对象的复制

- 自定义的复制构造函数

```
// 自定义的复制构造函数
Buffer::Buffer(const Buffer& that) {
    _size = that._size;
    gOffset = that.gOffset;
    pOffset = that.pOffset;
    array = new char[_size];
    for (int i = 0; i < _size; ++i)
        array[i] = that.array[i];
}
```

类对象 - 对象的复制

- 自定义的赋值操作符函数

```
Buffer& Buffer::operator=(const Buffer& that) {
    // 务必做以下检查，防止自赋值
    if (this != &that) { // 是不是同一个对象？
        _size = that._size;
        gOffset = that.gOffset;
        pOffset = that.pOffset;

        // 先释放原先的堆空间内存，再创建新的内存空间
        delete[] array;
        array = new char[_size];

        for (int i = 0; i < _size; ++i)
            array[i] = that.array[i];
    }

    return *this;
}
```

类对象 - 阻止对象的构造

- 阻止对象的构造
  - 如果一个类定义中将所有构造函数、复制构造函数、赋值操作符声明为 private 或 protected，则该类对象的构造无法直接进行
  - GOF 设计模式中的 Singleton 模式也许用得着这个方案？

类对象 - 对象的构造与析构

- 对象的创建方式：
  1. 自动对象，当程序执行到其声明时创建、离开其作用域时销毁
  2. 自由存储对象，通过 new 操作符建立，通过 delete 操作符销毁
  3. 一个非静态成员对象 a ，作为另一个类对象 b 的成员，则 b 被创建或销毁， a 随之被创建或销毁
  4. 一个对象 a ，作为数组 arr 的元素，则 arr 被创建或销毁，对象 a 随之创建或销毁
  5. 局部静态对象，在程序执行到其声明处建立一次，程序终止时销毁一次
  6. 全局对象、名字空间对象、类的静态数据成员，只在程序开始时（ main 函数执行前）创建一次，程序终止时销毁一次
  7. 临时对象在作为表达式求值的一部分被创建、在完整表达式的最后被销毁

类对象 - 对象的构造与析构

- 对象的创建方式（续）：
  8. 通过 new（exprlist) type（exprlist）操作符放置的对象
  9. 一个 union 成员，不能有构造函数和析构函数

类对象 - 局部对象

- 局部对象
  - 局部对象的创建和销毁同基本类型

类对象 - 动态对象

- 动态对象（堆区对象）
  - 由 new 操作符创建，同基本类型
    - 先分配适当的堆区内存
    - 再调用对象的指定构造函数初始化对象
  - 由 delete 操作符销毁：
    - 先调用对象的析构函数（由编译器隐式调用）
    - 再释放对象指针指向的内存

类对象 - 类对象作为成员

- 如果一个类 A 的对象 a 中包含一个或多个类对象（数据成员）：
  - 构造：
    - 所有成员对象在对象 a 构造之前构造；
    - 这些成员对象的构造按它们在类 A 中声明的顺序构造
  - 析构：
    - 对象 a 的析构函数先被执行
    - 其成员对象的析构按与它们在类 A 中声明的顺序相反的次序执行

类对象 - 类对象作为成员

- 成员初始化的必要性
  - 类对象中有 const 成员、引用类型的成员
  - 类对象中有没提供默认构造函数的成员对象
  - 上述情况下，必须对类对象的成员进行显式的初始化

```
class A {
    int n;
public:
    A(const int& i);
};
class C {
    const int& ir;
    int& nr;
    A a;
public:
    C(int& y);
    C(const int& x,
      int& y,
      const int& n);
};
```

类对象 - 类对象作为成员

- 初始化列表
  - 构造函数和复制构造函数中有一种初始化的语法，叫初始化列表
  - 初始化语法形式的初始化可能会比赋值形式的初始化更有效率

```
C::C(int& y) :
    ir(0), nr(y), a(0) {
}

C::C(const int& x,
     int& y,
     const int& n) :
    ir(x), nr(y), a(n) {
}
```

类对象 - 类对象作为成员

- 成员常量
  - 静态且具有整型类型的成员，可以在声明时加上一个常量表达式作为初始式

```
struct Members {
    static const int i = 8; // OK
    static int n = 6; // Error, n不是const
    const int k = 8; // Error, k非static
    static const int j = f(0); // Error, 类中的初始式不是常量表达式
    static const double d = 3.2; // Error, d不是整型 ???
};
```

  - 在类定义中用枚举作为符号常量，通常更有效：
    - 如 Buffer 类中的 BUF_SIZE

类对象 - 数组或其它容器

- 类对象作为数组或其它容器（如 vector 、 list 等）的元素，需要类对象具备以下条件：
  - 有可用的默认构造函数

类对象 - 局部静态对象

- 局部静态对象的创建和销毁：
  - 同基本类型

类对象 - 非局部对象

- 全局对象、名字空间对象、类的静态数据成员的创建与销毁：
  - 同基本类型

类对象 - 对象的放置

- new 操作符的另一种形式

```
void* operator new(size_t size, void* block);
new (block) T;
new (block) T(args);
```

- 示例：

```
class T {
    int n;
public:
    T(){}
};

int main() {
    void* block = operator new(sizeof(T));
    T* tp = new(block) T; // 放置对象
    // ...
    // 注意：显式调用析构函数，销毁对象，不是delete tp;
    tp->~T();
    // ...释放内存等操作，略
}
```

类对象 - union

- 联合 (union)
  - 类似与 struct，但所有数据成员共用一个地址，同一时间只能调用一个成员
  - 不能有静态成员
  - union 的数据成员不能有构造、析构函数
  - 可以创建匿名 union

类与对象 - 助记

- 一个类的成员列表，通常由数据成员和成员函数组成
- 正常使用一个类，会涉及到的成员函数有：
  - 构造函数、复制构造函数、赋值操作符函数、析构函数
  - 如果没有显式定义任何构造函数，编译器将在其需要的情况下产生一个
  - 如果没有显式的定义复制构造函数、赋值操作符函数、析构函数，编译器将在其需要的情况下创建一个
- 类的 const 成员函数可以被该类的 const 或非 const 对象调用，但非 const 成员函数只能被非 const 对象调用

类与对象 - Bjarne's Advices

- Bjarne's Advices
  - 将不需修改对象状态的成员函数声明为 const 函数
  - 将不需要通过特定对象访问的成员声明成 static
  - 如果在构造函数中申请了资源，须在析构函数中释放该资源
  - 如果类中有指针或引用数据成员，则有必要定义复制构造和赋值操作符函数，作复制的动作
  - 如果一个类需要复制操作或析构函数，它多半还需构造函数、析构函数、赋值操作符、复制构造函数
  - 在定义赋值操作符函数时要检查自赋值的情况
  - 在写复制构造函数时，不要遗漏任何一个数据成员的初始化
  - 在向类中添加新成员时，注意检查，看是否需要更新自定义的构造函数，确认它们能够初始化新成员

类与对象 - Bjarne's Advices

- Bjarne's Advices （ continued ）
  - 在类定义中需要整型常量时，考虑使用枚举
  - 请记住，临时对象将在创建它们的那个完整表达式结束时销毁