第五节课类和对象

对象的成员变量（私有）是每个对象自己的，成员函数（公有）是放在公有区域的

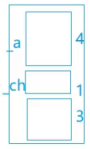
内存对齐规则：

对齐数=取min（默认对齐数，该成员大小）

比如先char ch，再int a，VS的默认对齐数为8，所以ch先放在第一个位置（第0号）

然后int，int是4个字节，和8相比较小，所以int对齐数是4，int要对齐到4的整数倍，所以char之后要空3个位置，第五个位置（第4号）

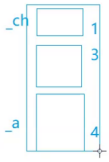
```
// 类中既有成员变量，又有成员函数
class A1 {
public:
    void f1() {}
private:
    int _a;
    char _ch;
};
```



7.3 结构体内存对齐规则

1. 第一个成员在与结构体偏移量为0的地址处。
2. 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处。
   注意：对齐数 = 编译器默认的一个对齐数 与 该成员大小的较小值。
   VS中默认的对齐数为8
3. 结构体总大小为：最大对齐数（所有变量类型最大者与默认对齐参数取最小）的整数倍。
4. 如果嵌套了结构体的情况，嵌套的结构体对齐到自己的最大对齐数的整数倍处，结构体的整体大小就是所有最大对齐数（含嵌套结构体的对齐数）的整数倍。

```
class A1 {
public:
    void f1() {}
private:
    char _ch;
    int _a;
};
```



结构体的总大小为，最大对齐数的整数倍

为什么要内存对齐：（提高访问效率）

硬件读取数据的设计，一次性读整字大小的空间，一个字是四个字节（32位系统），或8个字节（64位系统），甚至是两个字节（16位系统）

如果不对齐的话，某些数据可能会被隔成两段，一个数据就会被读取两次，这样效率太低了，如果对齐了的话，读取次数就少了

```
39  // 类中仅有成员函数
40  class A2 {
41  public:
42      void f2() {}
43  };
44
45  // 类中什么都没有---空类
46  class A3
47  {};
48
49  int main()
50  {
51      //cout << sizeof(A1) << endl;
52      cout << sizeof(A2) << endl;
53      cout << sizeof(A3) << endl;
54
```

对于类中只有成员函数的和空类，sizeof大小是1，因为虽然没有，但是也要占一个地方表示这里有东西，可以打印出通过该类实例化出的对象的地址，发现是有地址的，有地址说明有空间。

```
49  int main()
50  {
51      //cout << sizeof(A1) << endl;
52
53      // 没有成员变量的类对象，需要1byte，是为了占位，表示对象存在
54      // 不存储有效数据
55      cout << sizeof(A2) << endl;
56      cout << sizeof(A3) << endl;
57      A2 aa1;
58      A2 aa2;
59      cout << &aa1 << endl;
60      cout << &aa2 << endl;
61
62      return 0;
63  }
```

面试题：

【面试题】

1. 结构体怎么对齐？ 为什么要进行内存对齐?
2. 如何让结构体按照指定的对齐参数进行对齐？能否按照3、4、5即任意字节对齐?
3. 什么是大小端？如何测试某台机器是大端还是小端，有没有遇到过要考虑大小端的场景

对于类中的函数，函数中使用的参数是隐含地带有this指针的

```
void Print()
{
    cout << _year << "-" << _month << "-" << _day << endl;
}
```

```
// 编译器会成员函数的处理
void Print(Date* this)
{
    cout << this->_year << "-" << this->_month << "-" << this->_day << endl;
}
```

不同对象使用这个函数时，函数是同一个函数，但参数就不一定是同一个参数了。

C++中不允许在形参和实参的位置显式传递，但显式地去用，

```
void Print(Date* this)
{
    cout << _year << "-" << _month << "-" << _day << endl;
}
```

比如这种情况就是不允许的

但下面这种情况是允许的

```cpp
void Print()
{
    // this不能在形参和实参显示传递，但是可以在函数内部显示使用
    cout << this << endl;
    cout << this->_year << "-" << _month << "-" << _day << endl;
}
```

注意：this不可以被修改，比如this = nullptr 这种是不允许的，因为this是* const this属性，不可以改变指针（但可以改变指针指向的对象的内容）

this指针存在于哪里?

存在于函数调用的栈帧里，因为this本质是形参，参数在函数调用过程中压栈

```cpp
void Print()
{
    cout << _year << "-" << _month << "-" << _day << endl;
}
```

```cpp
// 编译器会成员函数的处理
void Print(Date* this)
{
    cout << this->_year << "-" << this->_month << "-" << this->_day << endl;
}
```

this指针存在哪里? 对象里面 栈 堆 静态区 常量区

this是形参，所以this指针是跟普通参数一样存在函数调用的栈帧里面。

再细化一下的话，每个编译器的处理方式不同，以vs为例，因为this指针比较频繁调用，所以放到了寄存器里，寄存器比较快，以下面为例就是把d1的地址（this指针）放到了ecx里

```cpp
97  int main()
98  {
99      Date d1, d2;
100     d1.Init(2022, 1, 11);
101     d2.Init(2022, 1, 12);
102     d1.Print();
```

```
003E26EF  push        0Bh
003E26F1  push        1
003E26F3  push        7E6h
003E26F8  lea         ecx,[d1]
003E26FB  call        Date::Init (03E143Dh)
    d2.Init(2022, 1, 12);
```

vs下面对this指针传递，进行优化，对象地址是放在ecx，ecx存储this指针的值

小题测试：

```cpp
// 1.下面程序编译运行结果是？ A、编译报错 B、运行崩溃 C、正常运行
class A
{
public:
    void Print()
    {
        cout << "Print()" << endl;
    }
private:
    int _a;
};

int main()
{
    A* p = nullptr;
    p->Print();
    return 0;
}
```

p调用Print，不会发生解引用，因为Print的地址不在对象中。p会作为实参传递给this指针。

```cpp
// 1.下面程序编译运行结果是？ A、编译报错 B、运行崩溃 C、正常运行
class A
{
public:
    void PrintA()
    {
        cout<<_a<<endl;
    }
private:
    int _a;
};

int main()
{
    A* p = nullptr;
    p->PrintA();
    return 0;
}
```

p->Print() 这里不会解引用，所以p为空指针的话不会在这里报错，因为Print是公有函数，地址在公用代码段，不在对象里，所以即使通过对象指针解引用后也是找不到的，所以这不是解引用。

this指针本质是，Print函数调用时的形参，所以p就传给了this，所以this为空，所以第二张图的this->_a就报错了，这才是空指针。

A：：Print（）也是错误的，因为没有可以传进去的指针以供this指针可用

130，131行这个也是错误的，因为this不能在形参和实参显式传递

```
125  int main()
126  {
127      A* p = nullptr;
128      p->Print();
129
130      A::Print(nullptr);
131      A::Print(p);
132
133
134      return 0;
135  }
```

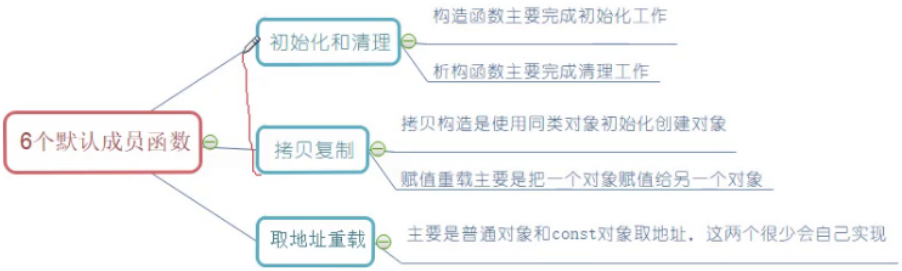
类和对象（中）：：：：

C++的类里面有6个默认成员函数，在创造出类的时候就已经存在了。

1.类的6个默认成员函数

> 如果一个类中什么成员都没有，简称为空类。
>
> 空类中真的什么都没有吗？并不是，任何类在什么都不写时，编译器会自动生成以下6个默认成员函数。
>
> 默认成员函数：用户没有显式实现，编译器会生成的成员函数称为默认成员函数。

```
class Date {};
```



构造函数特性：

> **构造函数**是特殊的成员函数，需要注意的是，构造函数虽然名称叫构造，但是构造函数的主要任务**并不是开空间创建对象，而是初始化对象。**
>
> **其特征如下：**
>
> 1. 函数名与类名相同。
> 2. 无返回值。
> 3. 对象实例化时编译器**自动调用**对应的构造函数。
> 4. 构造函数可以重载。

下面写一个构造函数：函数名与类名相同，参数写需要用到的形参，调用时在外面相应位置上写实参

```
typedef int DataType;
class Stack
{
public:
    Stack(int capacity = 4)
    {
        _array = (DataType*)malloc(sizeof(DataType) * capacity);
        if (NULL == _array)
        {
            perror("malloc申请空间失败!!!");
            return;
        }

        _capacity = capacity;
        _size = 0;
    }
```

构造函数可以重载，因为可能有多种初始化的方式，可以有多个构造函数。

析构函数：：函数名和类名相同，但是要在前面加个波浪号~，无参无返回值，构造函数是有参无返回值

3.析构函数

3.1 概念

> 通过前面构造函数的学习，我们知道一个对象是怎么来的，那一个对象又是怎么没呢的?
>
> 析构函数：与构造函数功能相反，析构函数不是完成对对象本身的销毁，局部对象销毁工作是由编译器完成的。而**对象在销毁时会自动调用析构函数，完成对象中资源的清理工作。**

3.2 特性

> **析构函数**是特殊的成员函数，其**特征**如下：
>
> 1. 析构函数名是在类名前加上字符 ~。
> 2. 无参数无返回值类型。

析构函数是在对象销毁的时候自动调用，构造函数是在创建对象的时候自动调用。

好处：：不会忘记初始化和销毁了，而且比较简化

1、初始化和销毁经常忘记
2、有些地方写起来很繁琐。

Stack有了构造和析构，就不怕忘记写初始化和清理函数了，也简化了

```
class Solution {
public:
    bool isValid(string str) {
        Stack st;
        //st.Init();
        const char* s = str.c_str();
        while(*s)
        {
            if(*s == '(' || *s == '[' || *s == '{')
            {
                st.Push(*s);
            }
            else
```


如果类中没有显式定义构造函数或析构函数，编译器会自动生成一个，一旦显式定义了编译器就不再生成。

数据类型有两大类，内置类型（基本类型）和自定义类型（struct等）

1、内置类型/基本类型、语言本省定义的基础类型
int/char/double/指针等等
2、自定义、用struct/class等等定义的类型

我们不写，编译器默认生成构造函数，内置类型不做处理，自定义类型会去调用他的默认构造。

编译器自动生成的构造函数会调用结构体的默认构造。对于内置类型，有的编译器会进行处理，有的编译器不会处理，但自定义类型都会处理。基本上都有内置类型，老老实实自己写吧。

所以，我们写不写构造函数，取决于有没有内置类型，如果全部都是自定义类型，就可以考虑让编译器自己生成构造函数进行处理。

后来C++11打了个补丁，内置类型成员可以给缺省值来提供给默认的构造函数，对内置类型进行处理。（不是初始化，因为这里根本还没给空间），如果在主函数中创建对象而不给值的话，默认是那个缺省值，给值的话都用给的那个。

```
297 int main()
298 {
299     //Date d1;
300     Date d2(2023, 1, 1);
301     //d1.Print();
302     d2.Print();
303
304     return 0;
305 }
306
```

```
306     // 构造函数的调用跟普通函数也不一样
307     Date d1;
308     Date d2(2023, 1, 1);
309     d1.Print();
310     d2.Print();
311
312     return 0;
313 }
314
```

不可以这样写：：会跟普通的函数声明有点冲突，编译器不好识别。

```
303
304 int main()
305 {
306     // 构造函数的调用跟普通函数也不一样
307     //Date d1;
308     Date d1(); // 不可以这样写，
```

而可以这样写：因为这样一看就不是普通的函数声明

```
310     Date d2(2023, 1, 1);
311     d1.Print();
312     d2.Print();
313
314     return 0;
315 }
316
```

不传参就可以调用的就是默认构造函数，使用默认的构造函数时，调用的时候按照参数声明的顺序。？？？？

```
268 class Date
269 {
270 public:
271     Date()
272     {
273         _year = 1;
274         _month = 1;
275         _day = 1;
276     }
277
278     Date(int year = 1, int month = 1, int day = 1)
279     {
280         _year = year;
281         _month = month;
282         _day = day;
283     }
284
```

上面这两个Date，可以构成重载，无参和全缺省的可以构成重载，能构成归能构成，但是调用的时候会存在歧义，所以调用的时候也不能同时存在，也会报错。

默认构造函数：

7. 无参的构造函数和全缺省的构造函数都称为默认构造函数，并且默认构造函数只能有一个。注意：无参构造函数、全缺省构造函数、我们没写编译器默认生成的构造函数，都可以认为是默认构造函数。

```
_year << "-" << _month << "-" << _day << endl;
```

不传参就可以调用的就是默认构造函数

默认成员函数是我们不写，编译器会自动生成的函数，而默认构造函数是不用自己传参就可以调用的函数，包括无参或者全缺省的函数，以及编译器自动生成的函数，与默认成员函数并不完全一样。

```
class MyQueue {
public:
    // 可以不写构造函数，编译器默认生成构造函数会完成两个栈对象的初始化
    // 可以不写析构函数，编译器默认生成析构函数会完成两个栈对象的资源清理释放

    void push(int x) {}

    int pop() {}

    int peek() {}

    bool empty() {}

    Stack _pushst;
    Stack _popst;
};
```