Assignment(徐子翔-2022141470095-assignment2)

0 Class

类（class）是结构体的拓展，不仅能够拥有成员元素，还拥有成员函数

在面向对象编程（OOP）中，对象就是类的实例，也就是变量

C++ 中 `struct` 关键字定义的也是类， **结构体** 的定义来自 C。因为某些历史原因，C++ 保留并拓展了 `struct`

定义类

类使用关键字 `class` 或者 `struct` 定义

```
class ClassName {
  ...
};
// Example:
class Object {
 public:
  int weight;
  int value;
} e[array_length];

const Object a;
Object b, B[array_length];
Object *c;

```

访问说明符:

- `public`：该访问说明符之后的各个成员都可以被公开访问，简单来说就是无论 **类内** 还是 **类外** 都可以访问。
- `protected`：该访问说明符之后的各个成员可以被 **类内**、派生类或者友元的成员访问，但类外 **不能访问。**
- `private`：该访问说明符之后的各个成员 **只能** 被 **类内** 成员或者友元的成员访问，**不能** 被从类外或者派生类中访问。

对于 `struct`，它的所有成员都是默认 `public`。对于 `class`，它的所有成员都是默认 `private`。

成员函数

```
class Class_Name {
  ... type Function_Name(...) { ... }
};

// Example:
class Object {
 public:
  int weight;
```

```
  int value;

  void print() {
    cout << weight << endl;
    return;
  }

  void change_w(int);
};

void Object::change_w(int _weight) { weight = _weight; }

Object var;

```

和函数类似，对于成员函数，也可以先声明，在定义，如第十四行（声明处）以及十七行后（定义处）。

如果想要调用 `var` 的 `print` 成员函数，可以使用 `var.print()` 进行调用。

重载运算符

```
class Vector {
 public:
  int x, y;

  Vector() : x(0), y(0) {}

  Vector(int _x, int _y) : x(_x), y(_y) {}

  int operator*(const Vector& other) { return x * other.x + y * other.y; }

  Vector operator+(const Vector&);
  Vector operator-(const Vector&);
};

Vector Vector::operator+(const Vector& other) {
  return Vector(x + other.x, y + other.y);
}

Vector Vector::operator-(const Vector& other) {
  return Vector(x - other.x, y - other.y);
}
```

该例定义了一个向量类，并重载了 `* + -` 运算符，并分别代表向量内积，向量加，向量减

重载运算符的模板大致可分为下面几部分

```
/*类定义内重载*/ 返回类型 operator符号(参数){...}

/*类定义内声明，在外部定义*/ 返回类型 类名称::operator符号(参数){...}

```

对于自定义的类，如果重载了某些运算符（一般来说只需要重载 `<` 这个比较运算符），便可以使用相应的 STL 容器或算法（sort等）

默认构造函数

```
class ClassName {
  ... ClassName(...)... { ... }
};

// Example:
class Object {
 public:
  int weight;
  int value;

  Object() {
    weight = 0;
    value = 0;
  }
};
```

若无显式的构造函数，则编译器认为该类有隐式的默认构造函数。换言之，若无定义任何构造函数，则编译器会自动生成一个默认构造函数，并会根据成员元素的类型进行初始化（与定义 内置类型 变量相同）

在这种情况下，成员元素都是未初始化的，访问未初始化的变量的结果是未定义的（也就是说并不知道会返回和值）

销毁(析构函数)

析构函数（Destructor）将会在该变量被销毁时被调用。重载的方法形同构造函数，但需要在前加 `~`

```
class Object {
 public:
  int weight;
  int value;
  int* ned;

  Object() {
    weight = 0;
    value = 0;
  }

  ~Object() { delete ned; }
};

```

实践：可变长数组类

```
//建立可变长度动态数组类
#include <iostream>
#include <cstring>
using namespace std;

class carray
{
    int size;                                   //数组元素的个数
    int * ptr;                                  //指向动态数组的指针
```

```
    public:
        carray(int s = 0);                      // 构造函数，s:数组元素个数，缺省时默认为0，对应实现数组特性1
        carray(carray & a);                     // 复制构造函数
        ~ carray();                             //析构函数

        //建立构造该类特性的成员函数
        void push_back(int v);                                          //在尾部添加一个整型元素v，无返回值
        carray & operator = (const carray & a);          //用于数组对象间的赋值，该函数的返回类型为该类的引用
        int length() const {return size;}                       //返回数组元素个数
        //上一行中的const关键字表示：length()是只读函数，不能修改类中的成员变量的值，只能引用。此写法只能用于成员函数

        int & operator [] (int i)               //重载运算符[]，操作对象是i，使得该类支持根据下标访问数组元素
        {
            return ptr[i];                          //利用数组指针的性质
        }
};

carray :: carray(int s) : size(s)           //在此位置，不能将缺省使得默认值=0写上
{
    if(s == 0)
        ptr = NULL;                             //若s为0，则指针为空指针
    else
        ptr = new int[s];                       //若s不为0，则指针指向新开辟的int[s]的动态内存空间
}
carray :: carray(carray & a)
{
    if(!a.ptr)
    {
        ptr = NULL;                             //若a.ptr为空，则使新初始化对象中的ptr为空，size为0，且结束该函数
        size = 0;
        return;                                 //不带返回值的return语句，用于void函数，运行至此即不再继续此函数
    }
    ptr = new int[a.size];                  //若a.ptr不为空，则为ptr开辟一个与a.ptr同样大小的动态空间
    memcpy(ptr, a.ptr, sizeof(int) * a.size);
    //memcpy函数用于将a.ptr中的内容逐字节拷贝到ptr中，拷贝的字节数为sizeof(int)*a.size
    //存放于<cstring>头文件中，与strcpy不同之处在于，不是遇到“\0”结束，而是一定会拷贝完给定的字节数
    size = a.size;
}
carray :: ~ carray()                            //析构函数在对象消亡时释放掉ptr指向的内存空间，防止其变为内存垃圾
{
    if(ptr)
```

```
        delete [] ptr;
}
carray & carray :: operator = (const carray & a)
{
    //重载后“=”可以使左边对象中存放的数组大小和内容与右边对象一样
    if(ptr == a.ptr)                                    //当出现“a=a"这样的错误赋值时，结束函数并且返回指向对象a的this指针
        return * this;
    if(a.ptr == NULL)                                   //当a.ptr为空时，ptr同样应为空
    {
        if(ptr)                                         //ptr如果已有内容，应当将该片内存空间释放
            delete [] ptr;
        ptr = NULL;
        size = 0;
        return * this;
    }
    if(size < a.size)                                   //若原内存空间不够大，需分配新的空间（释放掉之前的内存空间）
    {
        if(ptr)
            delete [] ptr;
        ptr = new int[a.size];                          //分配的新内存空间应当和右边对象一致
    }
    memcpy(ptr, a.ptr, sizeof(int) * a.size);
    size = a.size;
    return * this;
}
void carray :: push_back(int v)                         //数组尾部添加一个元素v
{
    if(ptr)                                             //若ptr不为空
    {
        int * tempptr = new int[size + 1];              //重新分配空间
        memcpy(tempptr, ptr, sizeof(int) * size);       //复制原数组内容到临时tempptr数组中
        delete [] ptr;                                  //释放ptr指向的空间
        ptr = tempptr;                                  //ptr指向新的内存空间
    }
    else                                                //数组本来为空
        ptr = new int[1];
    ptr[size ++] = v;                                   //加入新的数组元素
}

int main()
{
    carray a;
    for(int i=0; i<5; ++i)
        a.push_back(i);
    carray a2, a3;
```

```
    a2 = a;
    for (int i=0; i< a.length(); ++i)
        cout << a2[i] << " ";
    a2 = a3;
    for(int i=0; i<a2.length(); ++i)
        cout << a2[i] << " ";
    cout << endl;
    a[3] = 100;
    carray a4(a);
    for(int i=0; i < a4.length(); ++i)
        cout << a4[i] <<" ";
    cout << endl;
    return 0;
}
```

1 Makefile & Cmake

1.0 基本操作

准备好对应的$xxx.h$ $xxx.cpp$ $main.cpp$ $CMakelist.txt$ 文件

然后执行如下操作

```
mkdir build
cd build
cmake ..
make
```

1.1 代码实现

main.cpp

```
#include <bits/stdc++.h>
#include "stuinfo.h"
//#include "stuinfo.cpp"
using namespace std;
stuinfo stu[20];
int n;
int main (){
    inputstu(stu,n);
    showstu(stu,n);
    sortstu(stu,n);
    showstu(stu,n);
    string ch;
    cout<<"Please input information"<<'\n';
    cin>>ch;
    if(findstu(stu,n,ch)) cout<<"The student is in the student list.";
    else cout<<"The student is NOT in the student list.";
}
```

stuinfo.cpp

```cpp
#include <bits/stdc++.h>
#include "stuinfo.h"
using namespace std;
void inputstu(stuinfo stu[] , int &n){
    cout<<"Please input the number of students:"<<'\n';
    cin>>n;
    cout<<"Please input information of :name score1 score2 score3"<<'\n';
    for(int i=1;i<=n;i++){
        cin>>stu[i].name>>stu[i].score[0]>>stu[i].score[1]>>stu[i].score[2];
        stu[i].ave=(stu[i].score[0]+stu[i].score[1]+stu[i].score[2])/3.0;
    }
}
void showstu(stuinfo stu[] , int n){
    for(int i=1;i<=n;i++)
        cout<<stu[i].name<<"  ave  "<<stu[i].ave<<"  scores  "
<<stu[i].score[0]<<' '<<stu[i].score[1]<<' '<<stu[i].score[2]<<'\n';
}
void sortstu(stuinfo stu[] , int n){
    sort(stu+1,stu+1+n);
}
bool findstu(stuinfo stu[] , int n, string ch){
    for(int i=1;i<=n;i++){
        if(stu[i].name==ch) return 1;
    }
    return 0;
}
```

stuinfo.h

```cpp
#include <bits/stdc++.h>
using namespace std;
struct stuinfo
{
    string name;
    double score[3];
    double ave;
    bool operator < (const stuinfo &others)const {
        return ave>others.ave;
    }
};
void inputstu(stuinfo stu[] , int &n);
void showstu(stuinfo stu[] , int n);
void sortstu(stuinfo stu[] , int n);
bool findstu(stuinfo stu[] , int n, string ch);
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(stuinfo)

aux_source_directory(. srcs)

add_executable(stuinfo.exe ${srcs})
```

1.2 运行情况

```
root@8fd9ee166d63:/ws/cm# ls
CMakeLists.txt  main.cpp  stuinfo.cpp  stuinfo.h
root@8fd9ee166d63:/ws/cm# mkdir build
root@8fd9ee166d63:/ws/cm# cd build
root@8fd9ee166d63:/ws/cm/build# cmake ..
-- The C compiler identification is GNU 11.2.0
-- The CXX compiler identification is GNU 11.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/local/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /ws/cm/build
root@8fd9ee166d63:/ws/cm/build# make
Scanning dependencies of target stuinfo.exe
[ 33%] Building CXX object CMakeFiles/stuinfo.exe.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/stuinfo.exe.dir/stuinfo.cpp.o
[100%] Linking CXX executable stuinfo.exe
[100%] Built target stuinfo.exe
root@8fd9ee166d63:/ws/cm/build# ls
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake  stuinfo.exe
root@8fd9ee166d63:/ws/cm/build#
```

```
root@8fd9ee166d63:/ws/cm/build# ls
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake  stuinfo.exe
root@8fd9ee166d63:/ws/cm/build# ./stuinfo.exe
Please input the number of students:
2
Please input information of :name score1 score2 score3
xzx 1 2 3
dyh 0 0 -1
xzx  ave  2  scores  1 2 3
dyh  ave  -0.333333  scores  0 0 -1
xzx  ave  2  scores  1 2 3
dyh  ave  -0.333333  scores  0 0 -1
Please input information
xzx
The student is in the student list.root@8fd9ee166d63:/ws/cm/build#
```

1.3 补充学习

sudo apt install tree：安装树状图目录

tree -L x 调出x层树状图

CMake 详细语法

语法的基本原则

```
1. 变量使用${}方式取值，但是在IF控制语句中是直接使用变量名
```

2. 指令(参数1参数2...)参数使用括弧括起，参数之间使用空格或分号分开。以上面的ADD_EXECUTABLE指令为例，如果存在另外一个func.cpp源文件就要写成：ADD_EXECUTABLE(hello main.cpp func.cpp)或者ADD_EXECUTABLE(hello main.cpp;func.cpp)
3. 指令是大小写无关的，参数和变量是大小写相关的。但推荐全部使用大写指令

```
PROJECT(HELLO xxx)
#工程名为HELLO，支持语言xxx(xxx留空为不限制语言)

SET(SRC_LIST main.cpp)
#SET关键字，用来指定变量
#SRC_LIST变量就包含了main.cpp，也可以SET(SRC_LIST main.cpp t1.cpp t2.cpp)

ADD_EXECUTABLE(hello ${SRC_LIST})
#ADD_EXECUTABLE 生成可执行文件
#生成的可执行文件名是hello，源文件读取变SRC_LIST中的内容
#也可以直接写 ADD_EXECUTABLE(hello main.cpp)

ADD_SUBDIRECTORY(src bin)
#ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
#这个指令用于向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置
#将src子目录加入工程并指定编译输出(包含编译中间结果) 路径为bin目录
#如果不进行bin目录的指定，那么编译结果(包括中间结果)都将存放在build/src目录

INSTALL(FILES COPYRIGHT README DESTINATION /ws/0309/cmake02)
#安装文件COPYRIGHT和README
# FILES：文件 DESTINATION：绝对路径

INSTALL(PROGRAMS runhello.sh DESTINATION bin)
#PROGRAMS：非目标文件的可执行程序安装(比如脚本之类)

INSTALL(DIRECTORY doc/ DESTINATION /ws/0309/cmake02)
#安装doc中的hello.txt

ADD_LIBRARY(hello SHARED ${LIBHELLO_SRC})
# hello：就是正常的库名，生成的名字前面会加上lib，最终产生的文件是libhello.so
# SHARED，动态库；STATIC，静态库
# ${LIBHELLO_SRC}：源文件
```

使用CMake安装过程：cmake ..; make; make install

静态库和动态库的区别

静态库和动态库的区别静态库的扩展名一般为“.a”或“.lib”；动态库的扩展名一般为“.so”或“.dll”

静态库在编译时会直接整合到目标程序中，编译成功的可执行文件可独立运行

动态库在编译时不会放到连接的目标程序中，即可执行文件无法单独运行。

构建库的实例

```
├──build
├──CMakeLists.txt
└──lib
    ├── CMakeLists.txt
    ├──hello.cpp
    └──hello.h
```

hello.h中的内容

```
#ifndef HELLO_H
#define Hello_H
void HelloFunc();
#endif
```

hello.cpp中的内容

```
#include "hello.h"
#include <iostream>
void HelloFunc(){ std::cout<<"Hello World"<<std::endl;}
```

项目中的cmake内容

```
PROJECT(HELLO)
ADD_SUBDIRECTORY(lib bin)
```

lib中CMakeLists.txt中的内容

```
SET(LIBHELLO_SRC hello.cpp)
ADD_LIBRARY(hello SHARED ${LIBHELLO_SRC})
```

2 Types

2.1 问答题

2.1.1 static 用法和作用

在C++中，static是一个关键字，它可以用于多个上下文中，具有不同的用法和作用。下面是static的常见用法和作用：

1. 静态变量：当static用于函数内部的变量时，会将该变量声明为静态变量。这意味着，**该变量只被初始化一次，并且在函数调用之间保持其值**。静态变量通常用于需要在函数调用之间保存状态的情

况。

```
void myFunction()
{
    static int myStaticVar = 0;
    myStaticVar++;
    cout << "myStaticVar: " << myStaticVar << endl;
}

int main()
{
    myFunction(); // 输出：myStaticVar: 1
    myFunction(); // 输出：myStaticVar: 2
    myFunction(); // 输出：myStaticVar: 3
    return 0;
}

```

2. 静态函数：当static用于函数的声明时，它将该函数声明为静态函数。这意味着该函数只能在当前文件中被访问，不能被其他文件中的函数访问。静态函数通常用于实现私有函数或辅助函数。

```
// 在当前文件中定义静态函数
static void myStaticFunction()
{
    // do something
}

// 在当前文件中声明静态函数
static void myStaticFunction();

int main()
{
    // 调用静态函数
    myStaticFunction();
    return 0;
}

```

3. 静态类成员：当static用于类的成员变量或函数时，它将该成员声明为静态成员。这意味着，该成员与类本身相关，而不是与类的实例相关。静态成员可以被类的所有实例共享，并且可以在不创建类的实例的情况下访问。

```
class MyClass
{
public:
    static int myStaticVar;
    static void myStaticFunction();
};

int MyClass::myStaticVar = 0; // 静态变量的初始化
void MyClass::myStaticFunction()
{
    // do something
}
```

```
13
14 int main()
15 {
16     MyClass::myStaticVar = 10; // 直接访问静态变量
17     MyClass::myStaticFunction(); // 直接调用静态函数
18     return 0;
19 }
20
```

2.1.2 什么是隐式转换, 如何消除隐式转换

隐式转换是指在表达式中使用不同类型的变量时，编译器会自动将其中一种类型转换为另一种类型的过程。例如，在将一个整数和一个浮点数相加时，编译器会将整数隐式转换为浮点数，然后执行加法操作。

消除隐式转换的方法：

1. 显式类型转换：使用类型转换运算符强制将一个类型转换为另一个类型
2. 使用构造函数或转换函数

2.2 解释程序

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4  int main() {
5      int num1 = 1234567890;
6      int num2 = 1234567890;
7      int sum = num1 + num2;
8      cout << "sum = " << sum << endl;
9      //sum = -1825831516 一个int类型的变量存在上界2147483647，爆int(整形溢出)后变为
   负数
10
11     float f1 = 1234567890.0f;
12     float f2 = 1.0f;
13     float fsum = f1 + f2;
14     cout << "fsum = " << fsum << endl;
15     cout << "(fsum == f1) is " << (fsum == f1) << endl;
16     //fsum = 1.23457e+009 精度过高的小数用科学计数法表示
17     //(fsum == f1) is 1 float精度有限，丢失精度后二者的数据相同
18
19     float f = 0.1f;
20     float sum10x = f + f + f + f + f + f + f + f + f + f;
21     float mul10x = f * 10;
22     cout<<"sum10x = "<< sum10x << endl;
23     cout<<"mul10x = "<< mul10x << endl;
24     cout<<"(sum10x == 1) is "<< (sum10x == 1.0) << endl;
25     cout<<"(mul10x == 1) is "<< (mul10x == 1.0) << endl;
26     //sum10x = 1
27     //mul10x = 1
28     //(sum10x == 1) is 1
29     //(mul10x == 1) is 1
30     //结果符合感性理解，但判等浮点数最好还是使用eps
31     return 0;
```

```
}

```

```
#include <iostream>
using namespace std;
int main(){
    cout << fixed;
    float f1 = 1.0f;
    cout<<"f1 = "<<f1<<endl;

    float a = 0.1f;
    float f2 = a+a+a+a+a+a+a+a+a+a;
    cout<<"f2 = "<<f2<<endl;

    if(f1 == f2) cout << "f1 = f2" << endl;
    else cout << "f1 != f2" << endl;
    return 0;
}
/*
f1 = 1.000000
f2 = 1.000000
f1 = f2
在float类型下被判等，符合人的感性判断
*/
```

```
#include <iostream>
using namespace std;
int main(){
    int a, b;
    double c, d;
    a = 19.99 + 21.99;//a=41.98 向下取整得到a=41
    b = (int)19.99 + (int)21.99;// b=19+21 得到b=40
    c = 23 / 8; // 23/8向下取证 得到c=2
    d = 23 / 8.0; // d = 2.875

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    cout << "0/0= " << 0/0 << endl;
    return 0;
}
/*
    //tempCodeRunnerFile.cpp:15:25: 警告：被零除 [-Wdiv-by-zero]
    15 |     cout << "0/0= " << 0/0 << endl;
    0/0被警告
*/

```

3 Structs

3.1结构体对齐问题

alignas 失效的情形解释：当变量或类型的大小超过指定的对齐值时，该函数失效。

3.2 二维计算几何结构

4 C++动态内存申请

4.1 C++的内存分区

在C++中，程序运行时分配的内存可以分为以下几个区域：

1. 栈（Stack）：存放函数的参数、局部变量和返回地址等信息，由系统自动分配和释放。栈是一个连续的内存区域，先进后出。
2. 堆（Heap）：由程序员动态分配和释放，用于存储程序运行时动态申请的内存。堆是一个不连续的内存区域，没有固定的分配和释放顺序。
3. 全局区（Static Data Area）：存放全局变量、静态变量和常量等，它们在程序运行期间始终存在，直到程序结束才被释放。
4. 常量区（Constant Data Area）：存放程序中的常量，例如字符串常量等，它们也在程序运行期间始终存在，直到程序结束才被释放。
5. 代码区（Code Area）：存放程序的可执行代码，也称为文本区。它是只读的，不允许进行写操作。

```
class A {
    int num;
}
static int a; // 存储在全局区
auto b=0; // 存储在全局区
int main()
{
    char s[]="abc"; //存储在栈区
    char *p="123456"; //存储在栈区，指向的字符串常量存储在全局区

    p1= (char *)malloc(10); // 存储在栈区，指向的内存块存储在堆区
    A *a = new A(); // 存储在栈区，指向的对象存储在堆区
}

```

4.2 问答题

4.2.1 new 和 malloc 的区别

`new` 和 `malloc` 都可以在堆上动态分配内存，但它们的行为和用法是不同的：

1. `new` 是 C++ 中的运算符，`malloc` 是 C 语言中的库函数。
2. `new` 会调用对象的构造函数，而 `malloc` 不会。因此，使用 `new` 分配的内存可以直接初始化对象，而使用 `malloc` 分配的内存需要手动初始化对象。
3. `new` 返回的是对象的指针，而 `malloc` 返回的是 `void` 指针，需要进行强制类型转换。
4. `new` 可以重载，可以为特定类型定义自己的 `new` 实现，从而控制内存的分配和初始化方式。
5. `new` 在分配内存时，会调用 `operator new` 函数，该函数可以被重载，实现更复杂的内存管理。
6. `new` 和 `delete` 都是运算符，可以被重载，而 `malloc` 和 `free` 是 C 语言库函数，不能被重载。
7. `new` 分配的内存大小由编译器自动计算，而 `malloc` 需要手动指定分配的内存大小。

总之，使用 `new` 可以更方便、安全地分配和初始化对象，而 `malloc` 则更为灵活，适合于一些特殊的内存分配场景。当使用 `new` 分配内存时，需要记得使用 `delete` 进行内存释放，而当使用 `malloc` 分配内存时，需要使用 `free` 进行内存释放。

4.2.2 delete p、delete[ ] p、allocator 的作用

`delete p`、`delete[] p` 和 `allocator` 都是用来释放动态分配的内存的工具。

1. `delete p` 用于释放使用 `new` 分配的单个对象的内存。如果 `p` 指向一个数组，则只会释放数组的第一个元素，而其他元素的内存不会被释放。同时，如果 `p` 为 `nullptr`，则不会有任何操作。
2. `delete[] p` 用于释放使用 `new[]` 分配的数组的内存。如果 `p` 指向单个对象，则行为未定义。同时，如果 `p` 为 `nullptr`，则不会有任何操作。
3. `allocator` 是 C++ STL 中的一个内存分配器，可以用于在堆上动态分配内存，并返回指向该内存的指针。相比于 `new` 和 `malloc`，`allocator` 可以更灵活地控制内存的分配和释放，而且可以为容器提供自定义的内存管理策略。

总之，使用 `delete` 和 `delete[]` 可以释放动态分配的内存，并避免内存泄漏。使用 `allocator` 可以更灵活地管理内存，并提供自定义的内存管理策略。需要注意的是，在使用 `delete` 和 `delete[]` 释放内存时，必须保证指针指向的内存是动态分配的，并且没有被释放过。如果多次释放同一块内存，会导致程序崩溃或者出现其他未定义的行为。

4.2.3 malloc 申请的存储空间能用 delete 释放吗

`malloc` 分配的内存应该使用 `free` 函数来释放。而在 C++ 中，应该使用 `new` 和 `delete` 或者 `new[]` 和 `delete[]` 来进行内存分配和释放。这样可以确保内存的分配和释放是匹配的，并且可以避免出现内存泄漏或者非法内存访问等问题。

4.2.4 malloc 与 free 的实现原理

`malloc` 函数的作用是在堆上动态分配指定大小的内存，并返回指向该内存的指针。其实现原理一般分为以下几个步骤：

- 首先，从堆上分配一块足够大的内存空间，该内存空间一般包含一个头部结构体和指定大小的空闲内存块。
- 然后，根据用户请求的内存大小，从空闲内存块中选择一块大小合适的内存块，并将其分配给用户。
- 最后，将该内存块的状态标记为已分配，并返回指向该内存块的指针。

`free` 函数的作用是释放由 `malloc` 分配的内存，并将其返回到堆中以便后续的内存分配。其实现原理一般分为以下几个步骤：

- 首先，将要释放的内存块的状态标记为未分配，以便后续的内存分配。
- 然后，将该内存块的指针加入到空闲内存块链表中，以便后续的内存分配。
- 最后，对空闲内存块链表进行合并，以便更好地利用空闲内存块。

5 Debug 和 Release

5.1 对数组访问越界的总结

内存越界指的是程序在访问内存时，访问了超过所申请内存空间边界的内存，导致出现各种异常情况，如程序崩溃、数据损坏等问题。内存越界一般发生在数组越界访问、指针操作不当等场景下。

为了防止内存越界，操作系统和编译器都提供了相应的机制。具体来说，操作系统通过虚拟内存机制对程序进行隔离，将每个进程的虚拟地址空间映射到物理内存上，并对访问内存的指针进行边界检查和访问权限检查，避免程序访问非法的内存空间。而编译器则通过代码检查和优化等手段，检测和修复潜在的内存越界问题，生成更加安全和高效的代码。

除了依赖操作系统和编译器的机制，程序员也需要养成良好的编程习惯，以尽量避免内存越界问题的出现。具体来说，可以采取以下措施：

1. 熟悉内存分配和释放的相关函数，如 `malloc`、`new`、`free`、`delete` 等，避免内存泄漏和重复释放等问题。
2. 注意数组下标越界和指针操作不当等情况，及时进行边界检查和错误处理。
3. 使用 STL 等现代 C++ 标准库，避免手动管理内存和容器等数据结构，减少内存越界的概率。
4. 避免使用裸指针和 C 风格字符串等容易出现内存越界问题的代码结构，尽可能使用智能指针、字符串类等高级语言特性。
5. 通过静态代码分析和动态内存检测等工具，及时发现和修复内存越界问题，确保代码的安全和稳定。