

【课件】C++入门

该内容是公众号 计算机视觉 life 旗下课程《C++》的课件编纂成的课程手册。持续更新。课程官网：cvlife.net

作者：肖老师，中科大硕士，他也是计算机视觉与 OpenCV 公众号博主。

修订时间：2022 年 8 月 15 日

扫描一下，或登录 cvlife.net 系统学习视觉/激光/多传感器融合 SLAM、三维重建等精品课程：



扫描学习SLAM、三维重建

关于课程及交流群：如果想要咨询、购买计算机视觉 life 课程，请添加下面微信，备注：“咨询或已购课程”。请按照格式备注，否则不予通过。



第一章 基础介绍

(1.1 节为课程直播，未参与的同学请回看。)

1.2 C++开发环境安装配置

主要讲解 Windows 和 Linux 上的开发环境配置。

1.2.1 Windows 环境安装配置

安装介绍

Windows 上开发最常用的 IDE(集成开发环境)为 Visual Studio。

(链接: <https://visualstudio.microsoft.com/zh-hans/>)

了解 Visual Studio 系列



The image shows a promotional banner for the Visual Studio ecosystem. It features three distinct cards with a light purple-to-blue gradient background. Each card includes the Visual Studio logo, the product name, version information, a brief description of its capabilities, a '了解更多' (Learn More) link, and a '下载' (Download) button with a dropdown arrow.

- Visual Studio**
版本 17.1
适用于 Windows 上 .NET 和 C++ 开发人员的最佳综合 IDE。完整打包了一系列丰富的工具和功能,可提升和增强软件开发的每个阶段。
了解更多 >
下载 Visual Studio ▾
- Visual Studio for Mac**
版本 8.10
为 .NET 开发人员提供的综合 IDE,其原生在 macOS 上。包括对 Web、云和游戏开发的顶级支持,以及可用于开发跨平台移动应用的出色工具。
了解更多 > 下一个候选发布 >
详细了解 正在测试你的许可证
下载 Visual Studio for Mac ▾
- Visual Studio Code**
版本 1.64
在 Windows、macOS 和 Linux 上运行的独立源代码编辑器。JavaScript 和 Web 开发人员的最佳选择,具有几乎可支持任何编程语言的扩展。
了解更多 >
使用 Visual Studio Code 即表示你同意其 许可 & 隐私声明
下载 Visual Studio Code ▾

下载社区版:



Visual Studio

Windows | 版本 17.1

适用于 Windows 上 .NET 和 C++ 开发人员的最佳综合 IDE。完整打包了一系列丰富的工具和功能，可提升和增强软件开发的每个阶段。

[了解更多 >](#)

下载 Visual Studio ▾

Community 2022

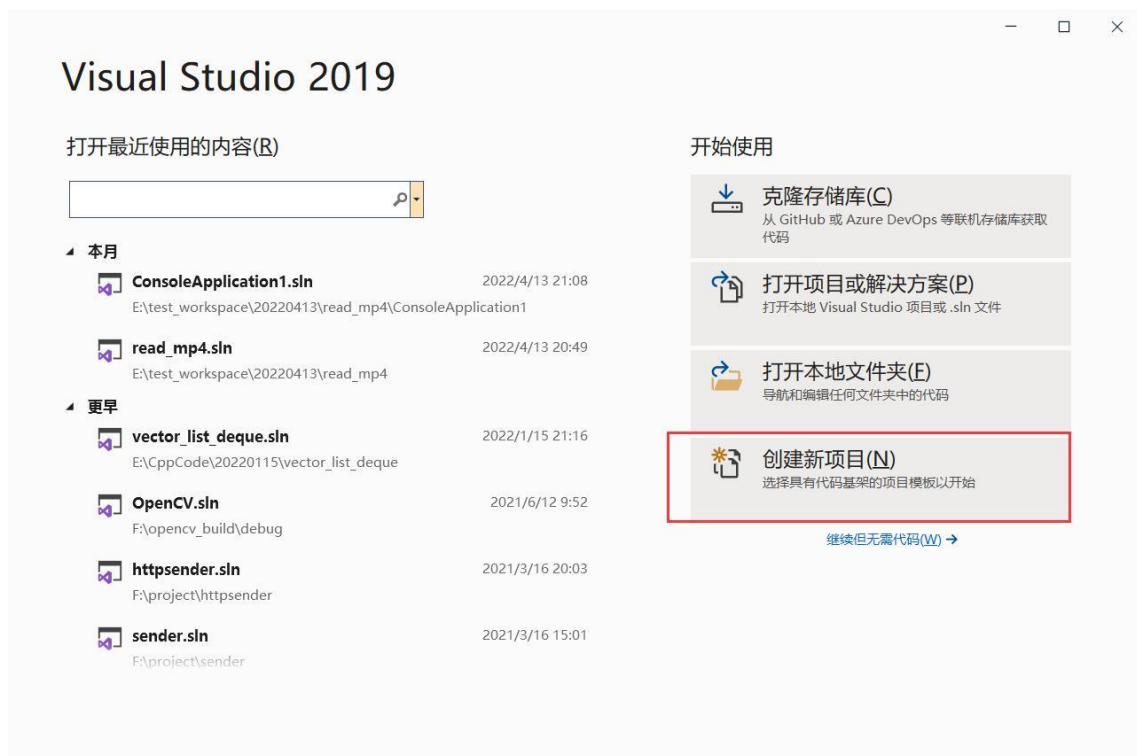
Professional 2022

Enterprise 2022

点击下载会下载“VisualStudioSetup.exe”安装文件（**推荐使用 VS2019**），默认安装即可。

环境测试

1、打开 vs2019



2、创建项目



3、添加项目名称与项目路径

配置新项目

控制台应用 C++ Windows 控制台

项目名称(N)
course1-2

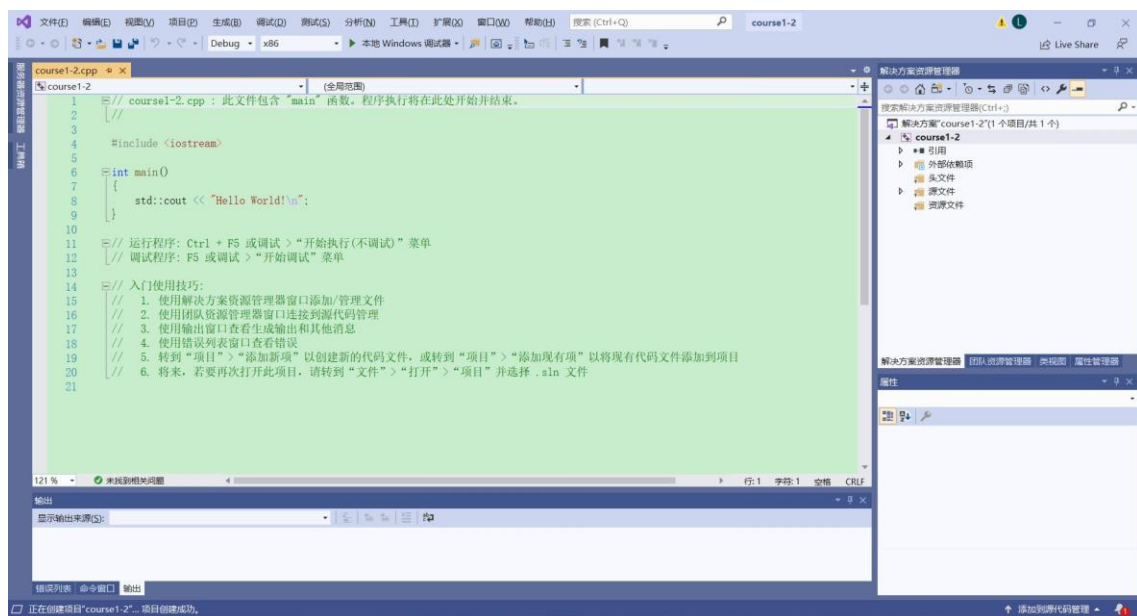
位置(L)
F:\Cpp_Course_Code\course1\ 注意最好不要有中文路径

解决方案名称(M)
course1-2

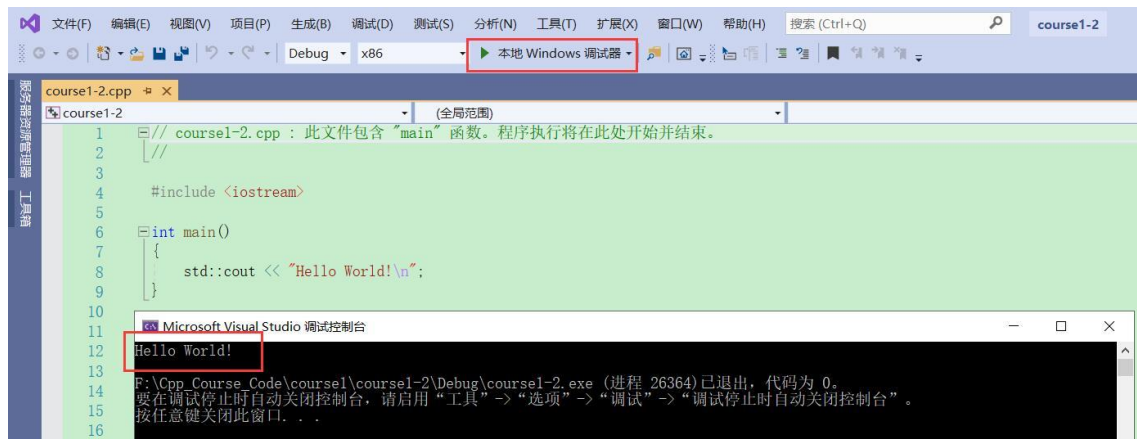
☒ 将解决方案和项目放在同一目录中(D)

上一步(B) 创建(C)

4、点击“创建”按钮，创建项目



5、项目运行并测试



1.2.2 Linux 环境安装配置

编译器为 g++ (C 语言编译器为 gcc), Linux 机器上一般会默认安装 (本课程讲解以 Ubuntu 为准)。

安装介绍

```
lxiao217@lxiao217:~$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.3.0-17ubuntu1~20.04' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch=32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-9-HskZEa/gcc-9-9.3.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```

如果版本太低可以通过命令升级。

环境测试

1、创建代码文件

```
Plaintext
gedit main.cpp
```

2、将如下代码复制粘贴到文件中, 并保存

```
C++
#include <iostream>
using namespace std;
int main(){
    // 输出: Hello World!
    std::cout << "Hello World!" << std::endl;
```

```
    return 0;  
}
```

3、测试运行

打开终端，输入命令编译代码：

```
C++  
g++ ./main.cpp
```

运行结果：



The screenshot shows a file explorer window with a dark theme. The top bar has tabs for '用户文件夹', 'mycode', 'course', and 'C++-2022'. The main area shows a list of files: 'a.out' (with a gear icon) and 'main.cpp' (with a C++ icon). Below the file explorer is a terminal window with a dark background. The terminal shows the following commands and output:

```
lxiao217@lxiao217:~/mycode/course/C++-2022$ ls  
a.out  main.cpp  
lxiao217@lxiao217:~/mycode/course/C++-2022$ ./a.out  
Hello World!  
lxiao217@lxiao217:~/mycode/course/C++-2022$
```

1.3 输入输出

1.3.1 iostream 数据流输入输出

iostream

iostream 是一个头文件，打开存储路径，可以看到文件：

Windows 端：

此电脑 > Windows (C:) > Program Files (x86) > Microsoft Visual Studio > 2019 > Community > VC > Tools > MSVC > 14.27.29110 > include >

名称	修改日期	类型	大小
invkprxy.h	2020/8/17 22:43	C/C++ Header	2 KB
iomanip	2020/8/17 22:43	文件	17 KB
ios	2020/8/17 22:43	文件	10 KB
iosfwd	2020/8/17 22:43	文件	10 KB
iostream	2020/8/17 22:43	文件	3 KB
isa_availability.h	2020/8/17 22:43	C/C++ Header	2 KB
iso646.h	2020/8/17 22:43	C/C++ Header	1 KB
istream	2020/8/17 22:43	文件	33 KB
iterator	2020/8/17 22:43	文件	29 KB
ivec.h	2020/8/17 22:43	C/C++ Header	37 KB
limits	2020/8/17 22:43	文件	28 KB
limits.h	2020/8/17 22:43	C/C++ Header	2 KB
list	2020/8/17 22:43	文件	70 KB
listing.inc	2020/8/17 22:43	Include File	5 KB
locale	2020/8/17 22:43	文件	9 KB
map	2020/8/17 22:43	文件	23 KB
memory	2020/8/17 22:43	文件	119 KB
memory_resource	2020/8/17 22:43	文件	34 KB
mm3dnow.h	2020/8/17 22:43	C/C++ Header	2 KB
mmintrin.h	2020/8/17 22:43	C/C++ Header	7 KB

Ubuntu 端:


```

1  // course1-2.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
2  //
3
4  #include <iostream>
5
6  int main() {
7
8      //整型输入
9      int a = 0;
10     std::cout << "请输入整型变量: " << std::endl;
11     std::cin >> a;
12     std::cout << a << std::endl;
13
14     //浮点型输入
15     double d = 0;
16     std::cout << "请输入浮点型变量: " << std::endl;
17     std::cin >> d;
18     std::cout << d << std::endl;
19
20     //字符型输入
21     char ch = 0;
22     std::cout << "请输入字符型变量: " << std::endl;
23     std::cin >> ch;
24     std::cout << ch << std::endl;
25
26     //字符串型输入
27     std::string str;
28     std::cout << "请输入字符串型变量: " << std::endl;
29     std::cin >> str;
30     std::cout << str << std::endl;
31
32     //布尔类型输入
33     bool flag = true;
34     std::cout << "请输入布尔型变量: " << std::endl;
35     std::cin >> flag;
36     std::cout << flag << std::endl;
37
38     std::cerr << "There is no errors!" << std::endl;
39     return 0;
40 }
41

```

使用时需要添加命名空间：**std**（第 6 章详细讲解命名空间）

cin-写数据

cin，表示标准输入(standard input)的 istream 类对象。cin 使我们可以从设备读出数据。

cout-读数据

cout，表示标准输出(standard output)的 ostream 类对象。cout 使我们可以向设备输出或者写数据。

cerr-报错误

cerr, 表示标准错误(standard error)的 ostream 类对象。cerr 是导出程序错误消息的地方, 它只能允许向屏幕设备写数据。

代码示例

```
Plaintext
#include <iostream>

int main() {

    //整型输入
    int a = 0;
    std::cout << "请输入整型变量: " << std::endl;
    std::cin >> a;
    std::cout << a << std::endl;

    //浮点型输入
    double d = 0;
    std::cout << "请输入浮点型变量: " << std::endl;
    std::cin >> d;
    std::cout << d << std::endl;

    //字符型输入
    char ch = 0;
    std::cout << "请输入字符型变量: " << std::endl;
    std::cin >> ch;
    std::cout << ch << std::endl;

    //字符串型输入
    std::string str;
    std::cout << "请输入字符串型变量: " << std::endl;
    std::cin >> str;
    std::cout << str << std::endl;

    //布尔类型输入
    bool flag = true;
    std::cout << "请输入布尔型变量: " << std::endl;
    std::cin >> flag;
    std::cout << flag << std::endl;

    std::cerr << "There is no errors!" << std::endl;
    return 0;
}
```

执行结果：

```
Microsoft Visual Studio 调试控制台
请输入整型变量:
10
10
请输入浮点型变量:
10.0
10
请输入字符型变量:
a
a
请输入字符串型变量:
helloworld
helloworld
请输入布尔型变量:
0
0
There is no errors!

F:\Cpp_Course_Code\course1\course1-2\Debug\course1-2.exe (进程 23848)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .
```

1.3.2 fstream 文件输入输出

使用场景：

程序执行结果保存

日志文件的保存

不方便调试时，增加打印信息，将打印信息保存到文件，方便数据分析

...

文件操作介绍

程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放，通过文件可以将数据持久化

文件操作头文件： `fstream`

文件类型分为两种：

1. 文本文件 - 文件以文本的 **ASCII** 码形式存储在计算机中
2. 二进制文件 - 文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂它们

操作文件的三大类：

1. `ofstream`：写操作
2. `ifstream`：读操作
3. `fstream`：读写操作

文本文件

写文件

写文件步骤如下：

- 1. 包含头文件

```
#include <fstream>
```

- 1. 创建流对象

```
ofstream ofs;
```

- 1. 打开文件

```
ofs.open("文件路径",打开方式);
```

- 1. 写数据

```
ofs << "写入的数据";
```

- 1. 关闭文件

```
ofs.close();
```

文件打开方式：

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置：文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

注意： 文件打开方式可以配合使用，利用|操作符

例如： 用二进制方式写文件 `ios::binary | ios:: out`

示例：

```

C++
#include <fstream>

void test01()
{
    std::ofstream ofs;
    ofs.open("test.txt", std::ios::out);

    ofs << "姓名: 张三" << std::endl;
    ofs << "性别: 男" << std::endl;
    ofs << "年龄: 18" << std::endl;

    ofs.close();
}

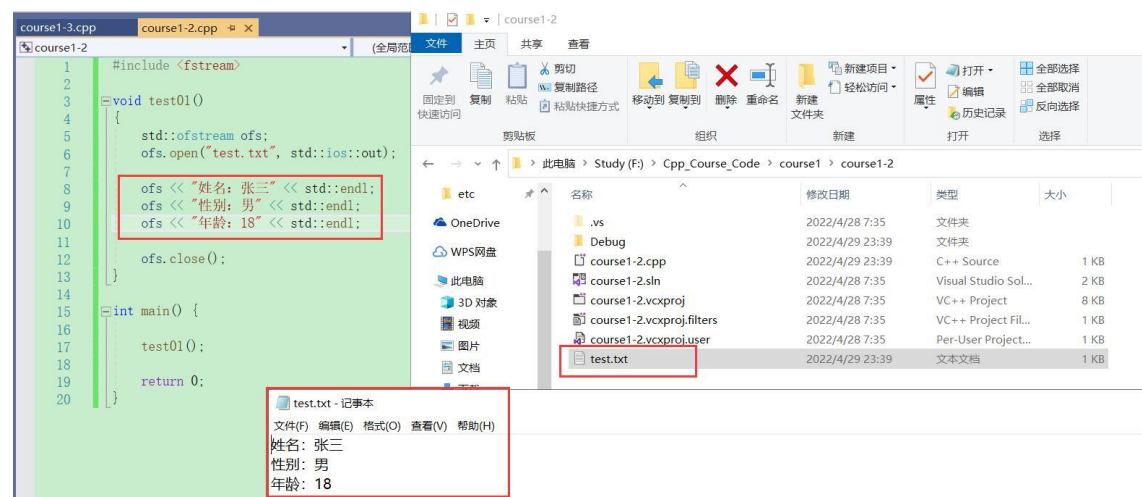
int main() {

    test01();

    return 0;
}

```

执行结果：



总结：

- 文件操作必须包含头文件 `fstream`
- 写文件可以利用 `ofstream`，或者 `fstream` 类
- 打开文件时候需要指定操作文件的路径，以及打开方式
- 利用 `<<` 可以向文件中写数据

- 操作完毕，要关闭文件

读文件

读文件与写文件步骤相似，但是读取方式相对于比较多。

读文件步骤如下：

1. 包含头文件

```
#include <fstream>
```

1. 创建流对象

```
ifstream ifs;
```

1. 打开文件并判断文件是否打开成功

```
ifs.open("文件路径",打开方式);
```

1. 读数据

四种方式读取

1. 关闭文件

```
ifs.close();
```

示例：

```
C++
#include <fstream>
#include <string>
#include<iostream>
using namespace std;
void test01()
{
    ifstream ifs;
    ifs.open("test.txt", ios::in);

    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
    }

    cout << "##### 方式一结果 #####" << endl;
    // 第一种方式，需要知道数据有多少（使用时选一种即可）
    char buf1[100] = { 0 };
    while (ifs >> buf1)
    {
```

```

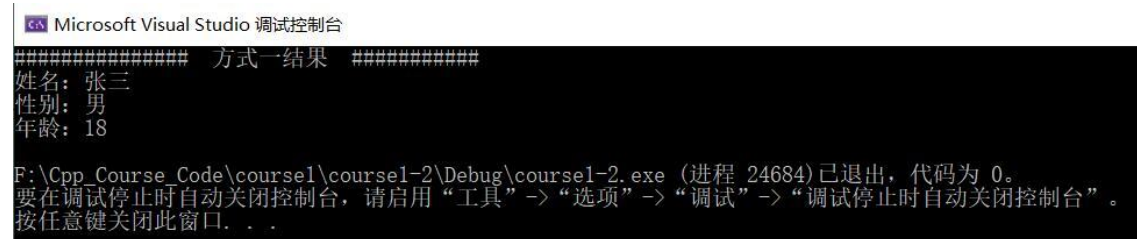
        cout << buf1 << endl;
    }
    cout << "##### 方式二结果  #####" << endl;
    // 第二种
    char buf2[1024] = { 0 };
    while (ifs.getline(buf2,sizeof(buf2)))
    {
        cout << buf2 << endl;
    }
    cout << "##### 方式三结果  #####" << endl;
    // 第三种
    string buf3;
    while (getline(ifs, buf3))
    {
        cout << buf3 << endl;
    }
    cout << "##### 方式四结果  #####" << endl;
    // 第四种
    char c;
    while ((c = ifs.get()) != EOF)
    {
        cout << c;
    }

    ifs.close();
}

int main() {
    test01();
    return 0;
}

```

执行结果：



```

Microsoft Visual Studio 调试控制台
##### 方式一结果 #####
姓名: 张三
性别: 男
年龄: 18
F:\Cpp_Course_Code\course1\course1-2\Debug\course1-2.exe (进程 24684)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .

```

总结：

- 读文件可以利用 ifstream , 或者 fstream 类
- 利用 is_open 函数可以判断文件是否打开成功

- close 关闭文件

二进制文件

以二进制的方式对文件进行读写操作

打开方式要指定为：ios::binary

写文件

二进制方式写文件主要利用流对象调用成员函数 write

函数原型：`ostream& write(const char * buffer, int len);`

参数解释：字符指针 `buffer` 指向内存中一段存储空间。`len` 是读写的字节数

示例：

```
C++
#include <fstream>
#include <string>
using namespace std;

//二进制文件 写文件
void test01()
{
    //1、包含头文件

    //2、创建输出流对象并打开
    ofstream ofs("test.txt", ios::out | ios::binary);

    // 也可以先创建对象再打开文件
    //ofs.open("test.txt", ios::out | ios::binary);

    char str[] = "I like study C++!";

    //3、写文件
    ofs.write(str, sizeof(str));

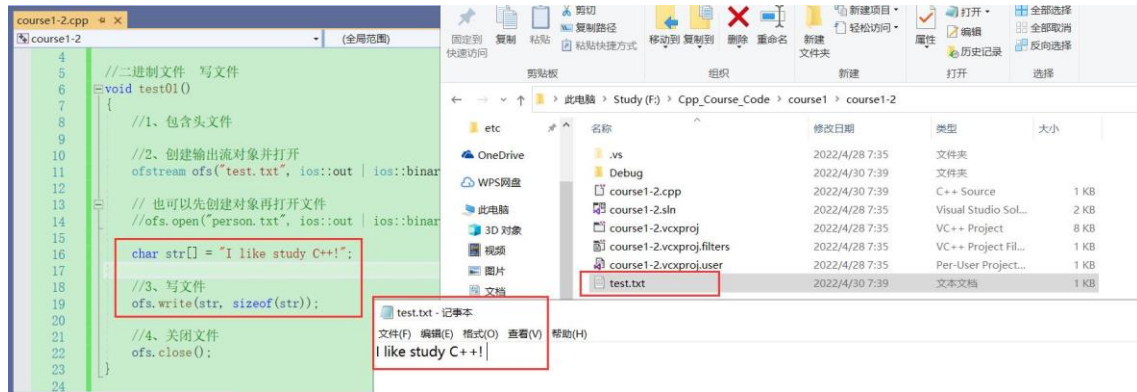
    //4、关闭文件
    ofs.close();
}

int main()
{
    test01();
}
```



```
    return 0;
}
```

运行结果：



总结：

- 文件输出流对象 可以通过 `write` 函数，以二进制方式写数据

读文件

二进制方式读文件主要利用流对象调用成员函数 `read`

函数原型：`istream& read(char *buffer,int len);`

参数解释：字符指针 `buffer` 指向内存中一段存储空间。`len` 是读写的字节数

示例：

```
C++
#include <fstream>
#include <string>
#include<iostream>
using namespace std;

//二进制文件  写文件
void test()
{
    //1、包含头文件

    //2、创建输出流对象并打开
    ifstream ifs("test.txt", ios::in | ios::binary);

    // 也可以先创建对象再打开文件
    //ifs.open("person.txt", ios::in | ios::binary);
```

```

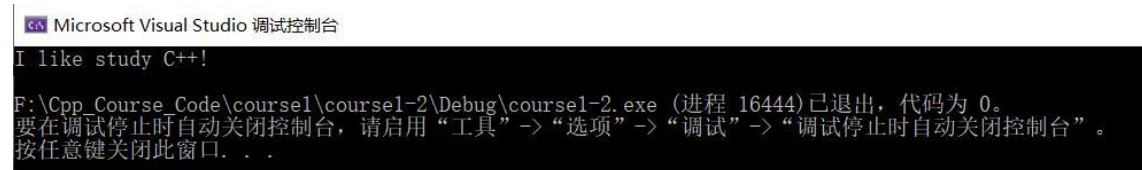
        //3、读取文件
        // 获取字节流的长度
        ifs.seekg(0, std::ios::end);
        int len = ifs.tellg();
        ifs.seekg(0, std::ios::beg);
        // 创建一段存储空间存储数据（第 11 章介绍）
        char* buff = new char[len];
        // 读取数据
        ifs.read(buff, len);
        for (int i = 0; i < len; i++)
            cout << buff[i];
        cout << endl;
        delete[]buff;

        //4、关闭文件
        ifs.close();
    }

int main()
{
    test();
    return 0;
}

```

运行结果：



```

Microsoft Visual Studio 调试控制台
I like study C++!
F:\Cpp_Course_Code\course1\course1-2\Debug\course1-2.exe (进程 16444) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .

```

总结：

- 文件输入流对象 可以通过 `read` 函数，以二进制方式读数据

第三章 表达式

算术运算符

- + 把两个操作数相加
- 从第一个操作数中减去第二个操作数
- * 把两个操作数相乘

/ 分子除以分母
% 取模运算符，整除后的余数
++ 自增运算符，整数值增加 1
-- 自减运算符，整数值减少 1

关系运算符

== 检查两个操作数的值是否相等，如果相等则条件为真。
!= 检查两个操作数的值是否相等，如果不相等则条件为真。
> 检查左操作数的值是否大于右操作数的值，如果是则条件为真。
< 检查左操作数的值是否小于右操作数的值，如果是则条件为真。
>= 检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
<= 检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。

逻辑运算符

&& 称为逻辑与运算符。如果两个操作数都 true，则条件为 true。
|| 称为逻辑或运算符。如果两个操作数中有任何一个 true，则条件为 true。
! 称为逻辑非运算符。用来逆转操作数的逻辑状态，如果条件为 true 则逻辑非运算符将使其为 false。

位运算符

位运算符作用于位，并逐位执行操作。&（按位与）、|（按位或）和 ^（按位异或）的真值表如下所示：

x	y	x&y	x y	x^y
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

~ 取反运算符，按二进制位进行"取反"运算。

<< 二进制左移运算符。将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补 0）。

>> 二进制右移运算符。将一个数的各二进制位全部右移若干位，正数左补 0，负数左补 1，右边丢弃。

赋值运算符

= 简单的赋值运算符，把右边操作数的值赋给左边操作数

+= 加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数

-= 减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数

*= 乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数

/= 除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数

%= 求模且赋值运算符，求两个操作数的模赋值给左边操作数

<<= 左移且赋值运算符

>>= 右移且赋值运算符

&= 按位与且赋值运算符

^= 按位异或且赋值运算符

|= 按位或且赋值运算符

成员访问运算符

.（点）和 ->（箭头）成员运算符用于引用类、结构和共用体的成员。

条件运算符

?:

condition? x1 : x2 条件运算符。如果 condition 为真，则值为 x1；否则值为 x2。

其他运算符相关操作

sizeof sizeof 运算符返回变量的大小。

， 逗号运算符

static_cast const_cast reinterpret_cast dynamic_cast 强制转换运算符把一种数据

类型转换为另一种数据类型。

& 指针运算符 **&** 返回变量的地址。

***** 指针运算符 ***** 指向一个变量。例如，***p**；将指向变量 **p**。

运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

如下为运算符的优先级列表

类别	运算符	结合性
后缀	() [] -> . ++ --	从左到右
一元	+ - ! ~ ++ -- (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与	&	从左到右
位异或	^	从左到右
位或		从左到右
逻辑与	&&	从左到右
逻辑或		从左到右
条件	?:	从右到左
赋值	= += -= *= /= %= >>= <<= &= ^= =	从右到左
逗号	,	从左到右

第四章 语句

条件语句

if/ if...else/ 嵌套 if 语句

```
Go
if(condition)
{
    // 语句块
}
else
{
    // 语句块
}
```

if 语句中 condition 为判断条件，满足条件则执行 if 后面语句块的内容，不满足的情况下，如果有 else 则执行 else 没有的话则跳出 if 语句结构。

switch 语句

switch 语句一般形式：

```
Go
switch(表达式)
{
    case 常量表达式 1:语句 1;
    case 常量表达式 2:语句 2;
    ...
    default:语句 n+1;
}
```

switch 先计算表达式的值，再逐个和 case 后的常量表达式比较，若不等则继续往下比较，若一直不等，则执行 default 后的语句；若等于某一个常量表达式，则从这个表达式后的语句开始执行，并执行后面所有 case 后的语句。

与 if 语句的不同：if 语句中若判断为真则只执行这个判断后的语句，执行完就跳出 if 语句，不会执行其他 if 语句；而 switch 语句不会在执行判断为真后的语句之后跳出循环，而是继续执行后面所有 case 语句。在每一 case 语句之后增加 break 语句，使每一次执行之后均可跳出 switch 语句，从而避免输出不应有的结果。

一个 switch 语句允许测试一个变量等于多个值时的情况。

嵌套 switch 语句您可以在一个 switch 语句内使用另一个 switch 语句。

循环语句

循环的实现可以使用 for、while 和 do...while 三种形式

```
Go
for(初始值;循环结束条件;循环量变更)
{
    //代码块
}
```

范围 for 循环：

```
Go
for(元素:集合)
{
    //代码块
}
```

范围 for 循环是从集合中取其中的元素，进行循环操作。

```
Go
while(循环结束条件)
{
    // 语句块
}

do
{
    // 语句块
}
while (循环结束条件)
```

注意：do...while 会先执行一次语句块的功能，然后再判断循环条件是否满足。

循环结构要非常注意循环变更是否能满足循环结束条件，不满足则出现死循环。

跳转语句

常见的跳转语句有：

break 用于循环和 switch 结构中，用于循环中则结束距离最近的循环；

`continue`: 用于循环结构，用于结束当前这一次的循环；

`return`: 用于函数的返回；

`goto`: 无条件跳转，容易破坏代码结构，不建议使用。

try 语句

异常是程序在执行期间产生的问题或者特殊错误，如除以零的操作。

C++提供了异常处理机制，异常处理涉及到三个关键字：`try`、`catch`、`throw`。

`try`: `try` 块中是尝试执行的代码，该代码有可能有异常抛出，它后面通常跟着一个或多个 `catch` 块。

`catch`: 用于处理程序捕获异常。

`throw`: 当代码执行出现问题时，可以使用 `throw` 关键字抛出一个异常。

```
Java
try
{
    // 可能有异常抛出的代码
}catch( ExceptionName e1 )
{
    // catch 块
}catch( ExceptionName e2 )
{
    // catch 块
}
```

C++ 提供了一系列标准的异常，定义在 `<exception>` 中，我们可以在程序中使用这些标准的异常。

也可以通过继承和重载 `exception` 类来自己定义异常。

异常	描述
std::exception	该异常是所有标准 C++ 异常的父类。
std::bad_alloc	该异常可以通过 new 抛出。
std::bad_cast	该异常可以通过 dynamic_cast 抛出。
std::bad_exception	这在处理 C++ 程序中无法预期的异常时非常有用。
std::bad_typeid	该异常可以通过 typeid 抛出。
std::logic_error	理论上可以通过读取代码来检测到的异常。
std::domain_error	当使用了一个无效的数学域时，会抛出该异常。
std::invalid_argument	当使用了无效的参数时，会抛出该异常。
std::length_error	当创建了太长的 std::string 时，会抛出该异常。
std::out_of_range	该异常可以通过方法抛出，例如 std::vector 和 std::bitset<>::operator[]()。
std::runtime_error	理论上不可以通过读取代码来检测到的异常。
std::overflow_error	当发生数学上溢时，会抛出该异常。
std::range_error	当尝试存储超出范围的值时，会抛出该异常。
std::underflow_error	当发生数学下溢时，会抛出该异常。

代码经验分享之 gdb 调试总结

一、gdb简介

GDB是一个由GNU开源组织发布的、UNIX/**LINUX操作系统** 下的、基于命令行的、功能强大的程序调试工具。对于一名Linux下工作的c/c++程序员，gdb是必不可少的工具；

二、gdb使用流程

这里用c程序做基本演示，c++程序也是一样的；

1、启动gdb

编译一个测试程序，-g表示可以调试，命令如下：

```
1 | gcc -g test.c -o test
```

启动gdb，命令如下：

```
1 | gdb test
2 | gdb -q test // 表示不打印gdb版本信息，界面较为干净；
```

2、查看源码

list(简写 l)：查看源程序代码，默认显示10行，按回车键继续看余下的。

测试如下：

```
1 | (gdb) list
```

3、运行程序

run(简写 r)：运行程序直到遇到 结束或者遇到断点等待下一个命令；

测试如下：

```
1 | (gdb) r
```

4、设置断点

break(简写 b)：格式 b 行号，在某行设置断点；

info breakpoints：显示断点信息

Num：断点编号

Disp：断点执行一次之后是否有效 keep：有效 dis：无效

Enb：当前断点是否有效 y：有效 n：无效

Address：内存地址

What：位置

```
1 | (gdb) b 5
2 | Breakpoint 3 at 0x400836: file write.c, line 5.
3 | (gdb) b 26
4 | Breakpoint 4 at 0x4008a6: file write.c, line 26.
5 | (gdb) b 30
6 | Breakpoint 5 at 0x4008c6: file write.c, line 30.
7 | (gdb) info breakpoints
8 | Num      Type           Disp Enb Address              What
9 | 3        breakpoint      keep y   0x0000000000400836 in main at write.c:5
10 | 4        breakpoint      keep y   0x00000000004008a6 in main at write.c:26
11 | 5        breakpoint      keep y   0x00000000004008c6 in main at write.c:30
12 | (gdb)
```

5、单步执行

使用 continue、step、next 命令

测试如下：

```
1 | (gdb) r
```

6、查看变量

使用 print、whatis 命令

测试如下：

```
1 | main () at write.c:28
2 | 28      if (fd<0)
3 | (gdb)
4 | 35      printf("写入的长度: %d\n写入文本内容: %s\n",size1,buf);
5 | (gdb) print fd
6 | $10 = 3
7 | (gdb) whatis fd
8 | type = int
9 | (gdb)
10 |
```

7、退出gdb

用quit命令退出gdb：

三、gdb基本使用命令

1、运行命令

run：简记为 r，其作用是运行程序，当遇到断点后，程序会在断点处停止运行，等待用户输入下一步的命令。

continue（简写 c）：继续执行，到下一个断点处（或运行结束）

next：（简写 n），单步跟踪程序，当遇到函数调用时，也不进入此函数体；此命令同 step 的主要区别是，step 遇到用户自定义的函数，将步进到函数中去运行，而 next 则直接调用函数，不会进入到函数体内。

step（简写 s）：单步调试如果有函数调用，则进入函数；与命令 n 不同，n 是不进入调用的函数的

until：当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

until+行号：运行至某行，不仅仅用来跳出循环

finish：运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。

call 函数(参数)：调用程序中可见的函数，并传递“参数”，如：call gdb_test(55)

quit：简记为 q，退出gdb

2、设置断点

break n（简写 b n）：在第 n 行处设置断点

（可以带上代码路径和代码名称：b OAGUPDATE.cpp:578）

b fn1 if a>b：条件断点设置

break func（break 缩写为 b）：在函数 func() 的入口处设置断点，如：break cb_button

delete 断点号 n：删除第 n 个断点

disable 断点号 n：暂停第 n 个断点

enable 断点号 n：开启第 n 个断点

clear 行号 n：清除第 n 行的断点

info b（info breakpoints）：显示当前程序的断点设置情况

delete breakpoints：清除所有断点：

3、查看源码

list：简记为 l，其作用就是列出程序的源代码，默认每次显示10行。

list 行号：将显示当前文件以“行号”为中心的前后10行代码，如：list 12

list 函数名：将显示“函数名”所在函数的源代码，如：list main

list：不带参数，将接着上一次 list 命令的，输出下边的内容。

4、打印表达式

print 表达式：简记为 p，其中“表达式”可以是任何当前正在被测试程序的有效表达式，比如当前正在调试C语言的程序，那么“表达式”可以是任何C语言的有效表达式，包括数字，变量甚至是函数调用。

print a：将显示整数 a 的值

print ++a：将把 a 中的值加1,并显示出来

print name：将显示字符串 name 的值

print gdb_test(22)：将以整数22作为参数调用 gdb_test() 函数

print gdb_test(a)：将以变量 a 作为参数调用 gdb_test() 函数

display 表达式：在单步运行时将非常有用，使用display命令设置一个表达式后，它将在每次单步进行指令后，紧接着输出被设置的表达式及值。如：display a

watch 表达式：设置一个监视点，一旦被监视的“表达式”的值改变，gdb将强行终止正在被调试的程序。如：watch a

whatis：查询变量或函数

info function：查询函数

扩展info locals：显示当前堆栈页的所有变量

表 3.9 gdb 的常用命令

命令	简写	功能
list	l	列出源码
break	b	设置断点
run	r	从头开始运行程序
continue	c	从停止处继续运行程序
next	n	向前执行一句（不进入被调用函数中）
step	s	向前执行一句（可进入被调用函数中）
return	ret	从当前函数返回
print	p	显示变量或表达式的值
x	x	显示内存值
backtrace	bt	显示调用栈
quit	q	退出 gdb

第五章

5.1 函数定义与声明

函数是一起执行一个任务的一组语句。每个程序（C/C++）都有一个主函数 main()，所有简单的程序都可以定义其他额外的函数。可以把代码划分到不同的函数中，函数划分通常是每个函数执行一个特定的任务来进行的。

函数定义的语法

函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

```
Go
int max(int x, int y)
{
    //函数体
}
```

返回类型：一个函数可以返回一个值。`return_type` 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，`return_type` 是关键字 `void`。

函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。

参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。

函数主体：函数主体包含一组定义函数执行任务的语句。

函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

如下是一个函数声明：

```
int max(int num1, int num2);
```

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

```
int max(int, int);
```

当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

函数声明告诉编译器函数的名称、返回类型和参数。

函数定义提供了函数的实际主体。

C++中常将声明与定义分开。

函数必须先声明或者定义才能使用。

函数调用

当程序调用函数时，程序控制权会转移给被调用的函数。

被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。

5.2 函数传参

函数传参有三种传参方式：传值、传址、传引用。

1、按值传递

(1) 形参和实参各占一个独立的存储空间。

(2) 形参的存储空间是函数被调用时才分配的，调用开始，系统为形参开辟一个临时的存储区，然后将各实参传递给形参，这是形参就得到了实参的值。

2、地址传递

地址传递与值传递的不同在于，它把实参的存储地址传送给形参，使得形参指针和实参指针指向同一块地址。因此，被调用函数中对形参指针所指向的地址中内容的任何改变都会影响到实参。

3、引用传递

引用传递是以引用为参数，则既可以使得对形参的任何操作都能改变相应数据，又使函数调用方便。引用传递是在形参调用前加入引用运算符“&”。引用为实参的别名，和实参是同一个变量，则他们的值也相同，该引用改变则它的实参也改变。

传值与传引用的差别：

- 1 传引用时，形参和实参是同一个变量，即使用相同的内存空间，二者有相同的地址。而传值时二者地址不同；
- 2 传引用时，由于没有新建变量，所以对于类对象参数，不会产生构造和析构。而如果是传值调用，调用时会进行构造，退出函数时会进行析构；
- 3 由于传引用使用的是原本实参的地址，所以对引用参数值的修改，会在退出函数后体现在主调函数中，而传值调用对参数的修改不会影响到主调函数。

5.3 函数中的 **const** 修饰符与命令行参数

const 修饰函数参数

防止传入的参数代表的内容在函数体内被改变，但仅对指针和引用有意义。因为如果是按值传递，传给参数的仅仅是实参的副本，即使在函数体内改变了形参，实参也不会得到影响。

const 修饰的函数参数是指针时,代表 在函数体内不能修改该指针所指的内容，起到保护作用，在字符串复制的函数中保证不修改源字符串的情况下，实现字符串的复制。

Go

```
void fun(const char * src, char * des){ //保护源字符串不被修改，若修改 src 则编译出错。
strcpy(des,src);
}
```

```

void main(){
char a[10]="china";
char b[20];
fun(a,b);
cout<<b<<endl;
}

```

const 修饰返回值

用 const 来修饰返回的指针或引用，保护指针指向的内容或引用的内容不被修改，也常用于运算符重载。归根究底就是 使得函数调用表达式不能作为左值。

```

Go
#include <iostream>
using namespace std;

class A {
private:
int i;
public:
A(){i=0;}
int & get(){
return i;
}
};

void main(){
A a;
cout<<a.get()<<endl; //数据成员值为 0
a.get()=1; //尝试修改 a 对象的数据成员为 1，而且是用函数调用表达式作为左值。
cout<<a.get()<<endl; //数据成员真的被改为 1 了，返回指针的情况也可以修改成员 i 的值，所以为了安全起见最好在返回值加上 const，使得函数调用表达式不能作为左值
}

```

命令行参数

执行程序时，可以从命令行传值给 C 程序。这些值被称为命令行参数，它们对程序很重要，特别是当您想从外部控制程序，而不是在代码内对这些值进行硬编码时，就显得尤为重要了。

命令行参数是使用 `main()` 函数参数来处理的，其中，`argc` 是指传入参数的个数，`argv[]` 是一个指针数组，指向传递给程序的每个参数。

使用命令行参数的方式传递参数时，最好检查命令行是否有提供参数，检查参数个数是否符合自己的预期，然后根据参数执行相应的动作。

命令行参数的 `main` 函数定义如下：

```
int main(int argc, char** argv)
```

或者

```
int main(int argc, char* argv[])
```

其中，`argc` 表示命令行输入参数的个数（以空白符分隔），`argv` 中存储了所有的命令行参数。

5.4 函数重载

C++ 允许在同一作用域中的某个函数指定多个定义，这就是函数重载。

重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

当调用一个重载函数时，编译器通过把你所使用的参数类型与定义中的参数类型进行比较，决定选用最合适的定义。选择最合适的重载函数的过程，称为重载决策。

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形参（如参数的个数、类型或者顺序）必须不同，不能仅通过返回类型的不同来重载函数。

如下代码中同名函数 `print()` 被用于输出不同的数据类型：

```
Go
void print(int i) {
    cout << "整数为: " << i << endl;
}

void print(double f) {
    cout << "浮点数为: " << f << endl;
}

void print(char c[]) {
    cout << "字符串为: " << c << endl;
}
```

注意，参数列表不同包括参数的个数不同、类型不同或顺序不同，仅仅参数名称不同是不可以的。函数返回值也不能作为重载的依据。

函数的重载的规则：

函数名称必须相同。

参数列表必须不同（个数不同、类型不同、参数排列顺序不同等）。

函数的返回类型可以相同也可以不相同。

仅仅返回类型不同不足以成为函数的重载。

C++代码在编译时会根据参数列表对函数进行重命名（不同的编译器有不同的重命名方式，这里仅仅举例说明，实际情况可能并非如此），例如 `void print(int i)` 会被重命名为 `_print_int`，`void print(float f)` 会被重命名为 `_print_float`。当发生函数调用时，编译器会根据传入的实参去逐个匹配，以选择对应的函数，如果匹配失败，编译器就会报错。

因此，函数重载只是在用户层面看到的函数名称是一致的，本质上它们还是不同的函数，占用不同的内存，函数地址也不一样。

5.5 代码经验分享

执行一个程序时，等待另一个任务执行完成的模式叫做同步；

执行一个程序时，调用另一个函数执行另一个任务，不用等待这个任务执行完成继续执行主调函数的模式叫做异步。

函数指针，其本质是一个指针，指向的是一个函数的地址。

//返回 int 类型的函数指针

```
int * (*fun) (int);
```

指针函数，即返回指针的函数，其本质是一个函数，而该函数的返回值是一个指针。

//返回 int 类型指针的指针函数

```
int* fun(int x,int y);
```

回调函数就是将函数以参数的形式传递给另外一个函数,其实就是一个通过函数指针调用的函数。

为什么需要回调函数（callback）？回调函数会要用在异步执行中，因为被调函数执行耗时太久，不想阻塞主调函数线程就需要通过回调的方式。

回调包括同步回调和异步回调。

Linux 端代码：

```
Go
#include <iostream>
#include <unistd.h>    //Linux 端头文件
```

```
using namespace std;

typedef void (*callback)();

void funcA()
{
    sleep(2);
    cout << "in function A!" << endl;
}

void funcB(callback func)
{
    func();
    while(1)
    {
        cout << "in function B!" << endl;
    }
}

int main()
{
    funcB(funcA);
}
```

第六章

6.1 using 与命名空间

命名空间的定义使用关键字 `namespace`。

常用的标准库命名空间为 `std`。

使用 `using` 声明命名空间，该声明告诉编译器，后续的代码将使用指定的命名空间中的名称。

6.2 string 的定义与初始化

`string` 是 C++ 标准库的一个重要的部分，主要用于字符串处理。可以使用输入输出流方式直接进行操作，也可以通过文件等手段进行操作。同时 C++ 的算法库对 `string` 也有着很好的支持，而且 `string` 还和 C 语言的字符串之间有着良好的接口。

使用 `string` 首先要在头文件当中加入 `< string >`：

```
#include <string>
```

string 在命名空间 std 中定义，使用时需要声明命名空间 std。

string s;//默认初始化，一个空字符串

string s1("cpp");//s1 是面值“cpp”的副本

string s2(s1);//s2 是 s1 的副本

string s3=s2;//s3 是 s2 的副本

string s4(10,'c');//使用 10 个字符 c 把 s4 初始化

string s5="helloworld";//拷贝初始化

string s6=string(10,'c');//拷贝初始化，生成一个初始化好的对象，拷贝给 s6

char cs[]="cplusplus";

string s7(cs,3);//复制字符串 cs 的前 3 个字符到 s 当中

string s8="cplusplus";

string s9(s8,2);//从 s8 的第二个字符开始拷贝，不能超过 s8 的 size

string s10(s8,3,4);//s10 是 s8 从下标 3 开始 4 个字符的拷贝

6.3 string 的常用操作

迭代器相关

begin 指向开始的迭代器

end 指向结尾的迭代器

反向迭代器：rbegin/rend

常量迭代器：cbegin/cend

常量反向迭代器：crbegin/crend

返回字符串长度：size/length

max_size：返回最大尺寸（size）

resize：调整字符串尺寸

capacity：返回字符串容量

reserve：调整字符串容量

clear：字符串清空

empty：字符串判空

operator[] 字符串访问运算符

at：获取字符串中的字符

back: 获取尾字符

front: 获取首字符

append: 向字符串添加内容

push_back: 在字符串结尾增加字符

insert: 字符串插入

erase: 从字符串删除内容

replace: 字符串内容替换

swap: 字符串交换

pop_back: 尾字符删除

c_str: 获取 C 风格字符串

data: 获取字符串数据

find: 字符串查找

find_first_of: 查找字符在字符串中出现的第一个位置

find_last_of: 查找字符在字符串中出现的最后一个位置

substr: 获取子字符串

compare: 字符串比较

6.4 代码经验分享--正则表达式

正则表达式(Regular Expression)使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。正则表达式是繁琐的, 但它是强大的, 可以很好的提高效率。许多程序设计语言都支持利用正则表达式进行字符串操作。

std::regex 是 C++用来表示正则表达式的库, 于 C++11 加入, 它是 class

std::basic_regex<>针对 char 类型的一个特化, 还有一个针对 wchar_t 类型的特化为 std::wregex。

std::regex 中正则表达式的语法规则:

符号 意义

^ 匹配行的开头

\$ 匹配行的结尾

.

[...] 匹配[]中的任意一个字符
(...) 设定分组
\ 转义字符
\d 匹配数字[0-9]
\D \d 取反
\w 匹配字母[a-z], 数字, 下划线
\W \w 取反
\s 匹配空格
\S \s 取反
+ 前面的元素重复 1 次或多次
* 前面的元素重复任意次
? 前面的元素重复 0 次或 1 次
{n} 前面的元素重复 n 次
{n,} 前面的元素重复至少 n 次
{n,m} 前面的元素重复至少 n 次, 至多 m 次
| 逻辑或

示例:

数字: `^[0-9]*$`

n 位的数字: `^d{n}$`

至少 n 位的数字: `^d{n,}$`

m-n 位的数字: `^d{m,n}$`

零和非零开头的数字: `^(0|[1-9][0-9]*)$`

汉字: `^\u4e00-\u9fa5]{0,}$`

英文和数字: `^[A-Za-z0-9]+$` 或 `^[A-Za-z0-9]{4,40}$`

长度为 3-20 的所有字符: `^. {3,20}$`

由 26 个英文字母组成的字符串: `^[A-Za-z]+$`

Email 地址: `^\w+([-+.] \w+)*@ \w+([-.] \w+)*\ . \w+([-.] \w+)*$`

域名: `[a-zA-Z0-9] [-a-zA-Z0-9]{0,62} (\ . [a-zA-Z0-9] [-a-zA-Z0-9]{0,62})+ \ . ?`

InternetURL: [a-zA-z]+://[^\s]* 或 ^http://([w-]+\.)+[w-]+(/[w-./?%&=]*)?\$

正则表达式测试网站:

<https://c.runoob.com/front-end/854/>

第七章 顺序容器

顺序容器按照插入元素的顺序存储，不会对元素进行排序。

vector 是最常用的顺序容器，容器的行为类似于数组，但可以根据要求自动增长。它可以随机访问、连续存储，长度也非常灵活。

array 容器类似数组，但是长度并不灵活，在 C++11 之后有定义。

deque（双端队列）容器支持在容器的起点和终点进行快速插入和删除，提供随机访问和灵活长度，但不连续。

list 容器是双向链表，可在容器中的任何位置实现双向访问、快速插入和快速删除，但无法随机访问容器 **list** 中的元素。

forward_list 容器是前向列表，是单一方向的 **list**。

7.1 vector 的定义与初始化

vector (vector) 是一个封装了动态大小数组的顺序容器。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，**vector** 是一个能够存放任意类型的动态数组。

vector 特点:

1.顺序存储

顺序容器中的元素按照严格的线性顺序排序。可以通过元素在序列中的位置访问对应的元素。

2.动态数组

支持对序列中的任意元素进行快速直接访问，可以在末尾相对快速地添加/删除元素的操作。

定义

需包含头文件与命名空间

```
#include < vector>
```

```
using namespace std;
```

`vector<类型>` 标识符;

初始化方法（即构造函数的调用）

`vector()`: 创建一个空 `vector`

`vector(int nSize)`: 创建一个 `vector`, 元素个数为 `nSize`

`vector(int nSize, const T& v)`: 创建一个 `vector`, 元素个数为 `nSize`, 且值均为 `v`

`vector(const vector&)`: 复制构造函数

`vector(begin,end)`: 复制`[begin,end)`区间内另一个数组的元素到 `vector` 中

7.2 迭代器的使用

迭代器是一个对象，可以循环访问 C++ 标准库容器中的元素，并提供对各个元素的访问。C++ 标准库容器全都提供迭代器，以便算法可以采用标准方式访问其元素，而不必考虑用于存储元素的容器类型。

可以使用成员和全局函数（例如 `begin()`以及 `end()`运算符）显式使用迭代器，`++/--`向前或向后移动。

在 C++ 标准库中，序列或范围的开头是第一个元素（`begin`），序列或范围的末尾始终定义为最后一个元素的下一个位置（`end`）。

迭代器按照定义方式分成以下四种。

1) 正向迭代器，定义方法如下：

容器类名::`iterator` 迭代器名;

2) 常量正向迭代器，定义方法如下：

容器类名::`const_iterator` 迭代器名;

3) 反向迭代器，定义方法如下：

容器类名::`reverse_iterator` 迭代器名;

4) 常量反向迭代器，定义方法如下：

容器类名::`const_reverse_iterator` 迭代器名;

通过迭代器可以读取它指向的元素，*迭代器名就表示迭代器指向的元素。通过非常量迭代器还能修改其指向的元素。

迭代器都可以进行++操作。反向迭代器和正向迭代器的区别在于：
对正向迭代器进行++操作时，迭代器会指向容器中的后一个元素；
而对反向迭代器进行++操作时，迭代器会指向容器中的前一个元素。

7.3 vector 的常用操作

元素增加

void push_back(const T& x):vector 尾部增加一个元素 X

iterator insert(iterator it,const T& x):vector 中迭代器指向元素前增加一个元素 x

iterator insert(iterator it,int n,const T& x):vector 中迭代器指向元素前增加 n 个相同的元素 x

iterator insert(iterator it,const_iterator first,const_iterator last):vector 中迭代器指向元素前插入另一个相同类型 vector 的[first,last)间的数据

元素删除

iterator erase(iterator it):删除 vector 中迭代器指向元素

iterator erase(iterator first,iterator last):删除 vector 中[first,last)中元素

void pop_back():删除 vector 中最后一个元素

void clear():清空 vector 中所有元素

元素遍历

reference at(int pos):返回 pos 位置元素的引用

reference front():返回首元素的引用

reference back():返回尾元素的引用

iterator begin():返回 vector 头指针，指向第一个元素

iterator end():返回 vector 尾指针，指向 vector 最后一个元素的下一个位置

reverse_iterator rbegin():反向迭代器，指向最后一个元素

reverse_iterator rend():反向迭代器，指向第一个元素之前的位置

是否为空判断

bool empty() const:判断 vector 是否为空，若为空，则 vector 中无元素

获取大小

`int size() const`:返回 `vector` 中元素的个数

`int capacity() const`:返回当前 `vector` 所能容纳的最大元素值

`int max_size() const`:返回最大可允许的 `vector` 元素数量值

其他操作

`void swap(vector&)`:交换两个同类型 `vector` 的数据

`void assign(int n,const T& x)`:设置 `vector` 中前 `n` 个元素的值为 `x`

`void assign(const_iterator first,const_iterator last)`:`vector` 中`[first,last)`中元素设置成当前 `vector` 元素

常用操作总结

- 1.`push_back` 在数组的最后添加一个数据
- 2.`pop_back` 去掉数组的最后一个数据
- 3.`at` 得到编号位置的数据
- 4.`begin` 得到数组头的指针
- 5.`end` 得到数组的最后一个单元+1 的指针
- 6.`front` 得到数组头的引用
- 7.`back` 得到数组的最后一个单元的引用
- 8.`max_size` 得到 `vector` 最大可以是多大
- 9.`capacity` 当前 `vector` 分配的大小
- 10.`size` 当前使用数据的大小
- 11.`resize` 改变当前使用数据的大小, 如果它比当前使用的大, 则填充默认值
- 12.`reserve` 改变当前 `vector` 所分配空间的大小
- 13.`erase` 删除指针指向的数据项
- 14.`clear` 清空当前的 `vector`
- 15.`rbegin` 将 `vector` 反转后的开始指针返回(其实就是原来的 `end-1`)
- 16.`rend` 将 `vector` 反转构的结束指针返回(其实就是原来的 `begin-1`)

- 17.empty 判断 vector 是否为空
- 18.swap 与另一个 vector 交换数据

7.4 list 与 deque 的使用

list 容器

list 容器，又称双向链表容器，即该容器的底层是以双向链表的形式实现的。这意味着，list 容器中的元素可以分散存储在内存空间里，而不是必须存储在一整块连续的内存空间中。所以 list 的插入和删除比较快，访问则比较慢。

5 种创建 list 容器的方式：

1) 创建一个没有任何元素的空 list 容器：

```
std::list<int> values;
```

2) 创建一个包含 n 个元素的 list 容器：

```
std::list<int> values(10);
```

3) 创建一个包含 n 个元素的 list 容器，并为每个元素指定初始值。例如：

```
std::list<int> values(10, 5);
```

4) 在已有 list 容器的情况下，通过拷贝该容器可以创建新的 list 容器。例如：

```
std::list<int> value1(10);  
std::list<int> value2(value1);
```

注意，采用此方式，必须保证新旧容器存储的元素类型一致。

5) 通过拷贝其他类型容器（或者普通数组）中指定区域内的元素，可以创建新的 list 容器。例如：

```
int a[] = { 1,2,3,4,5 };  
std::list<int> values(a, a+5);  
//拷贝其它类型的容器，创建 list 容器  
std::vector<int>vec{ 11,12,13,14,15 };  
std::list<int>values(vec.begin()+2, vec.end());//拷贝 vec 中的{13,14,15}
```

list 成员函数与功能

成员函数	功能
begin()	返回指向容器中第一个元素的双向迭代

	器。
end()	返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。
rbegin()	返回指向最后一个元素的反向双向迭代器。
rend()	返回指向第一个元素所在位置前一个位置的反向双向迭代器。
cbegin()	常量迭代器的 begin
cend()	常量迭代器的 end
crbegin()	常量反向迭代器的 begin
crend()	常量反向迭代器的 end。
empty()	判断容器中是否有元素，为空返回 true；否则返回 false。
size()	返回当前容器实际包含的元素个数。
max_size()	返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 $2^{32}-1$ ，所以我们很少会用到这个函数。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换容器中原有内容。
emplace_front()	在容器头部生成一个元素，和 push_front() 的功能相同，但效率更高。
push_front()	在容器头部插入一个元素。

<code>pop_front()</code>	删除容器头部的一个元素。
<code>emplace_back()</code>	在容器尾部生成一个元素，和 <code>push_back()</code> 的功能相同，但效率更高。
<code>push_back()</code>	在容器尾部插入一个元素。
<code>pop_back()</code>	删除容器尾部的一个元素。
<code>emplace()</code>	在容器中的指定位置插入元素，和 <code>insert()</code> 功能相同，但效率更高。
<code>insert()</code>	在容器中的指定位置插入元素。
<code>erase()</code>	删除容器中一个或某区域内的元素。
<code>swap()</code>	交换两个容器中的元素
<code>resize()</code>	调整容器的大小。
<code>clear()</code>	删除容器存储的所有元素。
<code>splice()</code>	将一个 <code>list</code> 容器中的元素插入到另一个容器的指定位置。
<code>remove(val)</code>	删除容器中所有等于 <code>val</code> 的元素。
<code>remove_if()</code>	删除容器中满足条件的元素。
<code>unique()</code>	删除容器中相邻的重复元素，只保留一个。
<code>merge()</code>	合并两个已排序的 <code>list</code> 容器，合并之后的 <code>list</code> 容器依然是有序的。
<code>sort()</code>	通过更改容器中元素的位置，将它们进行排序。
<code>reverse()</code>	反转容器中元素的顺序。

deque 容器

deque 是双端队列容器。

创建 deque 容器的几种方式（和 list 类似）

1) 创建一个没有任何元素的空 deque 容器：

```
std::deque<int> d;
```

2) 创建一个具有 n 个元素的 deque 容器，其中每个元素都采用对应类型的默认值：

```
std::deque<int> d(10);
```

3) 创建一个具有 n 个元素的 deque 容器，并为每个元素都指定初始值：

```
std::deque<int> d(10, 10);
```

4) 通过拷贝 deque 容器创建一个新的 deque 容器；

5) 通过拷贝其他类型容器中指定区域内的元素创建一个新容器。

函数成员	函数功能
begin()	返回指向容器中第一个元素的迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的迭代器。
rend()	返回指向第一个元素所在位置前一个位置的迭代器。
cbegin()	常量迭代器的 begin
cend()	常量迭代器的 end
crbegin()	常量反向迭代器的 begin
crend()	常量反向迭代器的 end
size()	返回实际元素个数。
max_size()	返回容器所能容纳元素个数的最大值。

resize()	改变实际元素的个数。
empty()	判断容器中是否有元素，为空返回 <code>true</code> ；否则返回 <code>false</code> 。
shrink_to_fit()	将内存减少到等于当前元素实际所使用的大小。
at()	使用经过边界检查的索引访问元素。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换原有内容。
push_back()	在序列的尾部添加一个元素。
push_front()	在序列的头部添加一个元素。
pop_back()	移除容器尾部的元素。
pop_front()	移除容器头部的元素。
insert()	在指定的位置插入一个或多个元素。
erase()	移除一个元素或一段元素。
clear()	移出所有的元素，容器大小变为 0。
swap()	交换两个容器的所有元素。
emplace()	在指定的位置直接生成一个元素。
emplace_front()	在容器头部生成一个元素。和 <code>push_front()</code> 的区别是，该函数直接在容器头部构造元素，省去了复制移动元素的过程。

`emplace_back()`

在容器尾部生成一个元素。和 `push_back()` 的区别是，该函数直接在容器尾部构造元素，省去了复制移动元素的过程。

对比与总结：

(1) vector

连续存储结构，每个元素在内存上是连续的；支持高效的随机访问和在尾端插入/删除操作，但其他位置的插入/删除操作效率低下；相当于一个数组，但是与数组的区别为：内存空间的扩展。`vector` 支持不指定 `vector` 大小的存储，但是数组的扩展需要程序员自己写。

(2) deque

连续存储结构，即其每个元素在内存上也是连续的，类似于 `vector`，不同之处在于，`deque` 提供了两级数组结构，第一级完全类似于 `vector`，代表实际容器；另一级维护容器的首位地址。这样，`deque` 除了具有 `vector` 的所有功能外，还支持高效的首/尾端插入/删除操作。

优点：(1) 随机访问方便，即支持 `[]` 操作符和 `vector.at()`

(2) 在内部方便的进行插入和删除操作

(3) 可在两端进行 `push`、`pop`

缺点：占用内存多

(3) list

非连续存储结构，具有双链表结构，每个元素维护一对前向和后向指针，因此支持前向/后向遍历。支持高效的随机插入/删除操作，但随机访问效率低下，且由于需要额外维护指针，开销也比较大。每一个结点都包括一个信息块 `Info`、一个前驱指针 `Pre`、一个后驱指针 `Post`。可以不分配必须的内存大小方便的进行添加和删除操作。使用的是非连续的内存空间进行存储。

优点：(1) 不使用连续内存完成动态操作。

(2) 在内部方便的进行插入和删除操作

(3) 可在两端进行 `push`、`pop`

缺点：(1) 不能进行内部的随机访问，即不支持 `[]` 操作符和 `vector.at()`

(2) 相对于 `verctor` 占用内存多

(4) vector、list、deque 容器选择原则：

a、若需要随机访问操作，则选择 `vector`；

- b、若需要随机插入/删除（不仅仅在两端），则选择 `list`
- c、只有需要在首端进行插入/删除操作的时候，还要兼顾随机访问效率，才选择 `deque`，否则都选择 `vector`。
- d、若既需要随机插入/删除，又需要随机访问，则需要在 `vector` 与 `list` 间做个折中-`deque`。
- e、当要存储大型类对象时（每个对象都占用内存较大），`list` 要优于 `vector`。

7.5 代码经验分享

(1) `size` 与 `capacity` 比较

`capacity()` 成员函数获取 `vector` 的容量；`size()` 成员函数获取 `vector` 的大小。

`vector` 容器的容量（用 `capacity` 表示），指的是在不分配更多内存的情况下，容器可以保存的最多元素个数；而 `vector` 容器的大小（用 `size` 表示），指的是它实际所包含的元素个数。

(2) 为提升性能 `vector` 最好预先分配好空间

`vector` 最常用的顺序容器，其底层使用一段连续的线性内存空间进行数据存储。

当 `vector` 的大小和容量相等，即 `size()` 和 `capacity()` 结果相同时，如果再向其添加元素，那么 `vector` 就需要扩容。`vector` 容器扩容需要三步：

- (1) 重新申请一段更大的内存空间；
- (2) 将旧内存空间中的数据按原有顺序移动到新的内存空间中；
- (3) 旧的内存空间释放。

这一过程会比较耗时，所以在使用时如果频繁对 `vector` 进行扩容，而程序对性能要求较高，则可以使用 `reserve` 成员函数预先分配空间。

(3) 尽量不要使用 `vector<bool>`

为了节省空间，`vector<bool>` 底层在存储各个 `bool` 类型值时，每个 `bool` 值都只使用一个比特位（二进制位）来存储。也就是说在 `vector<bool>` 底层，一个字节可以存储 8 个 `bool` 类型值。在这种存储机制的影响下，`operator[]` 势必就需要返回一个指向单个比特位的引用，但显然这样的引用是不存在的。

(4) CMake 软件编译项目

CMake 是一种跨平台编译工具，主要是编写 `CMakeLists.txt` 文件，用 `cmake` 命令将 `CMakeLists.txt` 文件转化为 `make` 所需要的 `makefile` 文件（linux 端）或者生成 VS 使

用的项目文件（Windows 端），最后用 `make` 命令或者 VS 软件编译源码生成可执行程序或共享库。

使用前需要先安装 CMake 软件。

一般把 `CMakeLists.txt` 文件放在工程目录下，`cmake` 命令指向 `CMakeLists.txt` 所在的目录，例如 `cmake ..` 表示 `CMakeLists.txt` 在当前目录的上一级目录。

CMakeLists.txt 文件示例

1. 声明要求的 cmake 最低版本

```
cmake_minimum_required( VERSION 3.15 )
```

2. 添加 c++11 标准支持

```
#set( CMAKE_CXX_FLAGS "-std=c++11" )
```

3. 声明一个 cmake 工程

```
PROJECT(test)
```

4. 头文件

```
include_directories(  
${PROJECT_SOURCE_DIR}/include/  
)
```

5. 通过设定 SRC 变量，将源代码路径都给 SRC，如果有多个，可以直接在后面继续添加

```
set(SRC  
${PROJECT_SOURCE_DIR}/src/test.cpp  
)
```

6. 创建共享库/静态库

设置生成共享库的路径，如下表示即生成的共享库在工程文件夹下的 lib 文件夹中

```
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/lib)  
set(LIB_NAME test)
```

生成库文件：SHARED 表示生成动态库，STATIC 表示生成静态库

```
add_library(${LIB_NAME} STATIC ${SRC})
```

7. 链接其他库文件

把刚刚生成的\${LIB_NAME}库和所需的其它库链接起来

如果需要链接其他的动态库，-l后接去除 lib 前缀和.so 后缀的名称，以链接

libpthread.so 为例,-lpthread

```
target_link_libraries(${LIB_NAME} pthread)
```

8. 编译主函数，生成可执行文件

先设置路径

```
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/bin)
```

生成

```
add_executable(${PROJECT_NAME} ${SRC})
```

可执行文件链接其他的库

```
target_link_libraries(${PROJECT_NAME} pthread)
```

第八章 关联容器

顺序容器中无序的存储了基本类型或者自定义类型的数据；

关联容器不仅存储了数据，还为数据配备了一个键值，组成了“键值对”（key-value），可以实现高效的查找和访问。

关联容器有 8 种：

按 key 有序保存元素（底层红黑树）

map 键值对保存数据

set 键值相同，所以只保留 key

multimap 可以有多个相同 key 的 map

multiset 可以有多个相同 key 的 set

key 无序（底层哈希表）

unordered_map 键值对保存数据，key 不排序

`unordered_set` 键值相同，所以只保留 key，key 不排序

`unordered_multimap` 可以有多个相同 key 的 map，key 不排序

`unordered_multiset` 可以有多个相同 key 的 set，key 不排序

8.1 map 的使用

pair 数据类型

pair 保存两个数据成员，用于生成特定类型的模板，定义在头文件 `utility` 中，定义方式如下：

```
pair<string, string> p1;
```

数据成员分别为 `first` 和 `second`。

C++ 中 `map` 提供的是一种“键值对”容器，里面的数据都是成对出现的，每一对中的第一个值称之为键(key)，`map` 中每个 key 只能出现一次，第二个值为该 key 的对应值 (value)。

```
#include <map>
```

```
map<int, string> map1;
```

C++ `map` 是一种关联式容器，包含“关键字/值”对。

常用操作如下：

<code>begin()</code>	返回指向 <code>map</code> 头部的迭代器
<code>rbegin()</code>	返回一个指向 <code>map</code> 尾部的逆向迭代器
<code>rend()</code>	返回一个指向 <code>map</code> 头部的逆向迭代器
<code>clear()</code>	删除所有元素
<code>count()</code>	返回指定元素出现的次数
<code>empty()</code>	如果 <code>map</code> 为空则返回 <code>true</code>
<code>end()</code>	返回指向 <code>map</code> 末尾的迭代器
<code>equal_range()</code>	返回特殊条目的迭代器对
<code>erase()</code>	删除一个元素
<code>find()</code>	查找一个元素
<code>get_allocator()</code>	返回 <code>map</code> 的配置器

<code>insert()</code>	插入元素
<code>key_comp()</code>	返回比较元素 <code>key</code> 的函数
<code>lower_bound()</code>	返回键值 \geq 给定元素的第一个位置
<code>max_size()</code>	返回可以容纳的最大元素个数
<code>size()</code>	返回 <code>map</code> 中元素的个数
<code>swap()</code>	交换两个 <code>map</code>
<code>upper_bound()</code>	返回键值给定元素的第一个位置
<code>value_comp()</code>	返回比较元素 <code>value</code> 的函数

8.2 set 的使用

(1)创建空的 set

```
std::set<std::string> set1;
```

(2)创建 set 的同时对其进行初始化

```
std::set<std::string> set2{"java", "cpp", "python"};
```

(3)将已有 set 容器中存储的所有元素赋值给新 set 容器

```
std::set<std::string> set3(set2);
```

(4)将已有 set 容器中的部分元素初始化新 set 容器

```
std::set<std::string> set4(set3.begin(), set3.end());
```

(5)set 容器创建默认的使用 `std::less<T>` 规则（升序排列），可以使用 set 类模板定义中第 2 个参数，手动修改 set 容器中的排序规则。

```
std::set<std::string, std::greater<string> > set5{"java", "C++", "python"};
```

常用操作（成员函数）

<code>begin()/end()</code>	正向迭代器
<code>rbegin()/rend()</code>	反向迭代器
<code>cbegin()/cend()</code>	常量迭代器
<code>crbegin()/crend()</code>	常量反向迭代器
<code>find(val)</code>	查找值为 <code>val</code> 的元素，如果成功找到，则返回指向该元素的迭代器
<code>lower_bound(val)</code>	返回一个指向当前 set 容器中第一个大于或等于 <code>val</code> 的元素的双向迭代器

`upper_bound(val)` 返回一个指向当前 `set` 容器中第一个大于 `val` 的元素的迭代器。

`equal_range(val)` 该方法返回一个 `pair` 对象（包含 2 个双向迭代器）

`lower_bound()` 方法的返回值等价，`pair.second` 和 `upper_bound()` 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的值为 `val` 的元素（`set` 容器中各个元素是唯一的，因此该范围最多包含一个元素）。

`empty()` 若为空则返回 `true`；否则 `false`。

`size()` 返回容器中存有元素的个数。

`max_size()` 返回 `set` 容器所能容纳元素的最大个数

`insert()` 向容器中插入元素。

`erase()` 删除容器中存储的元素。

`swap()` 交换 2 个 `set` 容器中存储的所有元素

`clear()` 清空 `set` 容器中所有的元素

`emplace()` 在当前 `set` 容器中的指定位置直接构造新元素。其效果和 `insert()` 一样，但效率更高。

`count(val)` 在当前 `set` 容器中，查找值为 `val` 的元素的个数，并返回，因为 `set` 容器中各元素的值是唯一的，因此该函数的返回值最大为 1。

8.3 其他关联式容器

其他的关联式容器：`multimap`、`multiset`、`unordered_map`、`unordered_set`、`unordered_multimap`、`unordered_multiset`

`multimap/multiset`：相比于 `map/set`，可以存储多个相同 `key` 值的元素。

`unordered_map`：也是存储键值对类型的元素，各个键值对键的值不允许重复，但是容器中存储的键值对是无序的。

`unordered_multimap`：相比于 `unordered_map`，该容器允许存储多个键相同的键值对。

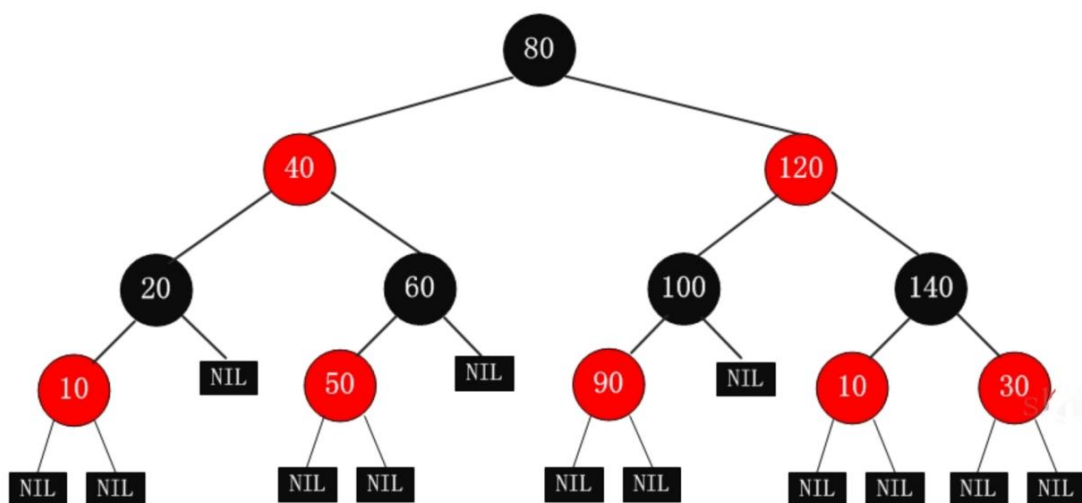
`unordered_set`：和 `set` 一样，`unordered_set` 存储的元素不能重复，但是容器内部存储的元素是无序的。

`unordered_multiset`：相比于 `unordered_set`，该容器允许存储值相同的元素。

8.4 代码经验分享

1、红黑树规则（有序关联容器底层红黑树）

- 节点不是黑色，就是红色（非黑即红）；
- 根节点为黑色；
- 叶节点为黑色（叶节点是指末梢的空节点 Nil 或 Null）；
- 一个节点为红色，则其两个子节点必须是黑色的（根到叶子的所有路径，不可能存在两个连续的红色节点）；
- 每个节点到叶子节点的所有路径，都包含相同数目的黑色节点（相同的黑色高度）。



2、哈希表（无序关联容器底层哈希表）

哈希表也叫散列表，提供了快速的插入操作和查找操作，无论哈希表总中有多少条数据，插入和查找的时间复杂度都是为 $O(1)$ 。

哈希表是通过哈希函数实现的，选取一个哈希函数，对于 key 不同的值，哈希函数计算得到的下标是不同的，对于相同的 key 得到的下标是相同的，所以选择一个好的哈希函数很关键。

但是哈希冲突不可避免，可以通过开放地址法（线性探测、二次探测、再哈希法）和链表法等方法解决哈希冲突。

3、如何看懂开源项目代码

- (1) 需要有语言基础；
- (2) 需要先了解项目中的算法的基本原理，结合代码能更清晰的了解实现过程；
- (3) 找到这个项目使用时对外暴露的接口文件；
- (4) 从编译脚本了解代码结构。

第九章 泛型算法

多用已有算法，少造轮子。

```
#include <algorithm>
```

9.1 只读算法

`for_each`: 应用函数于范围中

`find`: 范围中查找值

`find_if`: 范围中按照条件查找

`find_if_not`: 范围中不满足条件的查找

`find_end`: 在范围中查找子序列

`find_first_of`: 函数用于在第一个范围内查找和第二个范围中任何元素相匹配的第一个元素。

`adjacent_find`: 用于在指定范围内查找多个连续相等的元素

`count`: 统计范围内值的出现次数

`count_if`: 范围内满足条件的元素的个数

`mismatch`: 两个范围中的元素第一个不相同的位置

`equal`: 测试两个范围中的元素是否都相等

`is_permutation`: 用来检查一个序列是不是另一个序列的排列，如果是，会返回 `true`

`search`: 范围中查找子序列

`search_n`: 用于在指定区域内查找第一个符合要求的子序列

9.2 容器修改算法

`copy`: 拷贝一段元素

`copy_n`: 从源容器复制指定个数的元素到目的容器中

`copy_if`: 按条件拷贝

`copy_backward`: 从后向前拷贝

`move`: 元素移动

`move_backward`: 从后向前移动元素

`swap`: 交换两个对象的值

`swap_ranges`: 交换两个范围内的值

`iter_swap`: 交换指向两个迭代器对象的值

transform: 根据函数对范围内的元素做变换

replace: 用新的值来替换和范围中给定值相匹配的元素

replace_if: 替换满足条件的值

fill: 用给定值填充范围

fill_n: 元素填充, 以给定的迭代器为起始位置, 为指定个数的元素设置值

generate: 根据生成器函数生成数字, 然后将这些值分配给容器中范围内的元素。

generate_n: 使用函数为序列生成值

remove: 从范围中移除元素

remove_if: 根据条件删除元素

remove_copy: 将前两个正向迭代器参数指定的序列中的元素复制到第三个参数指定的目的序列中, 并忽略和第 4 个参数相等的元素

remove_copy_if: 将前两个正向迭代器参数指定的序列中, 能够使作为第 4 个参数的函数返回 `true` 的元素, 复制到第三个参数指定的目的序列中。

unique: 在序列中原地移除重复的元素

unique_copy: 删除元素并将结果拷贝到另一个容器中

reverse: 序列反转

reverse_copy: 序列反转并将结果拷贝到另一个容器中

rotate: 序列旋转

rotate_copy: 序列旋转并将结果拷贝到另一个容器中

random_shuffle: 随机打乱范围内元素顺序

shuffle: 使用生成器随机打乱元素顺序

9.3 其他算法

partition: 根据第三个参数定义的筛选规则, 重新排列指定区域内存储的数据, 使其分为 2 组, 第一组为符合筛选条件的数据, 另一组为不符合筛选条件的数据

sort: 对范围中的元素进行排序

lower_bound: 用于在指定区域内查找不小于目标值的第一个元素, 若没有与目标值相等的元素, 可能返回比目标值大的元素, 与之对应的是 **upper_bound**。

equal_range: 用于在指定范围的有序序列内查找等于目标值的所有元素, 注意序列需要排好序。

binary_search: 用于在已排序的序列中查找是否包含某个目标元素。

merge: 合并两个有序序列。

min: 返回最小值

max: 返回最大值

minmax: 返回最小和最大值

9.4 匿名函数

匿名函数，即没有名字的函数。使用匿名函数，可以免去函数的声明和定义，函数仅在调用时才会创建函数对象，调用结束后立即释放，所以匿名函数比非匿名函数更节省空间。

语法：

`[capture](parameters)->return-type{body}`

若 parameters 为空()可以省去，当 body 只有“return”或者返回为 void，那么“->return-type”可以被省去。

捕获内容说明：

`[]` // 不捕获任何变量

`[x, &y]` // x 按值捕获, y 按引用捕获

`[&]` // 用到的任何外部变量都按引用捕获

`[=]` // 用到的任何外部变量都按值捕获

`[&, x]` // x 按值捕获. 其它变量按引用捕获

`[=, &z]` // z 按引用捕获. 其它变量按值捕获

9.5 实战经验分享

C++程序的编译与链接过程

1、c++程序编译相关的文件

.cpp/.cc/.c : C/C++源文件

.h/.hpp: C/C++头文件

.exe : 可执行文件，点击即可运行

.obj : 目标文件，obj 文件与 cpp 文件名字一一对应

.pch : prcompiled-header,预编译头文件

.pdb：Program Database，即程序数据库文件，用来记录调试信息

.dll：Windows 动态库文件

.lib：Windows 静态库文件

.so：Linux 端动态库文件

.a：Linux 端静态库文件

2、编译与链接

编译：把文本形式的源代码翻译成机器语言，并形成目标文件

链接：把目标文件、操作系统的启动代码和库文件组织起来形成可执行程序

编译过程：

(1) 预处理：也称为预编译，执行代码文本替换工作。编译器执行预处理指令（以#开头，例如#include），这个过程会拷贝#include 包含的文件代码，进行#define 宏定义的替换，处理条件编译指令（#ifndef #ifdef #endif）等；

(2) 编译优化

本过程进行语法分析和词法分析，确定所有指令是否符合规则，然后翻译成汇编代码。

(3) 汇编

把汇编语言翻译成目标机器指令，生成目标文件（.obj/.o 等）。目标文件中存放的是与源程序等效的目标的机器语言代码。

链接过程：

由汇编程序生成的目标文件并不能立即就执行，还要通过链接过程生成可执行文件。

需要链接是因为程序调用了依赖库（第三方库或者标准库），文件之间存在互相调用，需要将相关文件的目标文件进行链接。

3、跨平台

跨平台概念是软件开发中一个重要的概念，即不依赖于操作系统，也不依赖硬件环境。一个操作系统下开发的应用，放到另一个操作系统下依然可以运行。相对而言如果某种计算机语言不用修改代码即可做到高度跨平台，那么此语言就越抽象，硬件控制力就越低，只适合开发高度抽象的模型系统。

跨平台开发的注意事项：

1、路径分隔符的差异

在 Windows 中，正斜杠和反斜杠都可以，但是在 Linux 中，只能是/。

2、数据长度的差异问题

不同平台直接声明 char，会导致 signed 或者 unsigned 的不确定性；

在 Windows 中，wchar_t 占两个字节，Linux 中占四个字节...

3、与平台相关的调用使用平台宏判断

4、头文件包含和文件路径不要写绝对路径

5、注意机器大小端的区别

6、尽量使用标准 C 和 C++ 的函数以及 STL，使用 C 语言中定义的类型。

7、#pragma once 防止头文件重复包含，这种写法不能跨平台

第十章 类的定义与使用

10.1 类的基本介绍

类（class）也是一种用户自定义的数据类型，它是创建对象的模板，一个类可以创建多个对象，每个对象都是这个类类型的一个变量；创建对象的过程也叫类的实例化。每个对象都是类的一个实例（Instance）。

```
Go
// 类的定义语法
class Student{
public:
    // 成员变量
    string name;
    int age;
    string address;
    // 成员函数
    void print(){
        cout<< "姓名: " << name << ", 年龄: " << age << ", 住址" <<
address << endl;
    }
};

// 创建对象
Student s;
```

```
// 创建指针
Student *p = new Student();
```

10.2 类的构造与析构

构造函数（Constructor）是一种特殊的成员函数，函数的名字和类名相同，没有返回值，它不需要用户显式调用，而是在创建对象时自动执行。

构造函数重载：与所有函数一样，构造函数，构造函数也可重载，一个类可以有多个重载的构造函数，创建对象时根据传递的实参来判断调用哪一个构造函数。

构造函数的作用主要是进行初始化。

如果自己没定义构造函数，编译器会自动生成一个默认的构造函数，这个构造函数的函数体是空的，没有形参，不执行任何操作。

一个类一定有构造函数，可以自己定义，也可以编译器默认生成。如果自己定义了构造函数，编译器就不会自动生成。

调用没有参数的构造函数也可以省略括号。

析构函数用于释放对象的内存，在类的操作过程中分配的内存记得在这里做释放，否则会内存泄漏。

this 指针

在 C++ 中，每一个对象都能通过 this 指针来访问自己的地址。this 指针是所有成员函数（除了静态成员函数）的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

10.3 访问控制

数据封装是一种把数据和执行函数捆绑在一起的机制。

数据抽象是一种仅向用户暴露接口而把具体的实现细节隐藏起来的机制。

类包含私有成员（private）、保护成员（protected）和公有成员（public）成员。默认情况下，在类（class）中定义的所有成员都是私有的。

（注意） struct 中的所有成员默认都是公有的（public）。

设计原则：

如非必要，成员变量都设计的是 `private` 的。

对外暴露的接口设计为 `public` 的，需要子类访问的成员设计为 `protected`，设置为 `protected` 权限的成员在子类的成员函数中可以访问。

10.4 友元

friend

C++中访问控制实现了数据的隐藏与封装，但是有时需要定义一些函数，这些函数不是类的成员函数，但却需要频繁的访问类的数据成员，这时我们可以将这些函数定义为该类的友元函数。

如果是类需要访问类的数据成员，还可以定义友元类，友元函数与友元类统称为友元 (friend)。

友元提高了程序的运行效率(即减少了类型和安全性检查及调用的时间开销)，但它破坏了类的封装，使得非成员函数可以访问类的私有成员。

成员函数中有隐参 `this` 指针，可以直接访问成员，而友元中则没有，必须得通过对象来访问，友元打破了外部访问中的 `private` 权限，声明为友元就可以通过对象进行访问。

10.5 类的作用域

在类外定义成员函数或者静态变量，需要使用类的作用域。

在有重名变量的时候，可以使用类的作用域访问类成员。

10.6 类的静态成员

在 C++中，可以使用静态变量来实现多个对象共享数据的目标，静态变量由关键字 `static` 修饰定义。

在类中，每个类的不同对象都有自己独立的存储空间，所以对象的成员变量之间互不干扰，但是静态成员变量是一种特殊的成员变量，所有的对象都共享这个变量的数据。

`static` 成员变量属于类，而不属于某个具体的对象，因此 `static` 对象只有一份存储，所有对象共用这份内存中的数据，如果某个对象修改了静态成员的数据，其他对象使用的数据就是被修改了的。

因为 `static` 成员变量不属于某个对象，因此必须在类声明的外部进行初始化，初始化

了才可以使用这个静态成员。静态成员变量数据存储在全局存储区，可以通过类的作用域方式访问，不一定需要通过对象来访问，访问时遵循访问控制权限的限制。

静态成员变量特点：

- 所有对象共享同一份数据
- 在编译阶段分配内存
- 在类内声明，在类外初始化

静态成员函数特点：

- 所有对象共享同一个函数
- 静态成员函数只能访问静态成员变量

10.7 代码实战经验分享

1、拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。

2、C++中拷贝构造函数调用时机

- (1) 使用一个已经创建完毕的对象来初始化一个新对象
- (2) 值传递的方式给函数参数传值
- (3) 以值方式返回局部对象

3、深拷贝与浅拷贝

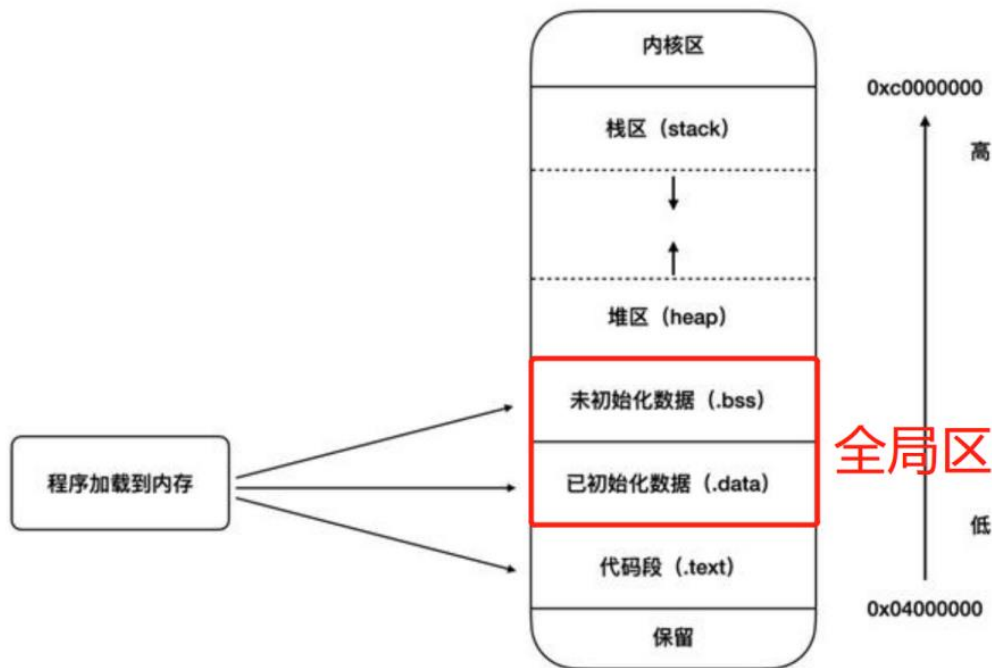
浅拷贝：简单的赋值拷贝操作

深拷贝：在堆区重新申请空间，进行拷贝操作

如果属性有在堆区开辟的，一定要自己提供拷贝构造函数，防止浅拷贝带来的问题。

第十一章 内存管理

11.1 内存分布介绍



C++程序在执行时，将内存划分为 4 个区域：

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量、静态变量、常量

该区域数据在程序执行结束后由操作系统释放

- 栈区：由编译器自动分配与释放，存放函数的参数、局部变量等

注意：不要返回局部变量的地址

- 堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统释放

不主动释放申请的内存，会造成内存泄漏。

C++中主要利用 new 在堆区开辟内存。

11.2 动态内存

C++中利用 new 操作符在堆区开辟内存，由程序员手动开辟与释放，内存释放使用 delete 操作符。

- (1) new 与 delete
- (2) new[]与 delete[]

malloc/free 和 new/delete 差异:

共同点:

都是从堆上申请空间, 并且需要用户手动释放。

不同点:

- 1、 malloc 和 free 是函数, new 和 delete 是操作符;
- 2、 malloc 申请的空间不会初始化, new 可以初始化;
- 3、 malloc 申请空间时, 需要手动计算空间大小并传递, new 只需在其后跟上空间的类型即可;
- 4、 malloc 的返回值为 void*, 在使用时必须强转, new 不需要, 因为 new 后跟的是申请对象的类型;
- 5、 malloc 申请空间失败时, 返回的是 NULL, 使用时必须判空, new 失败会抛出异常;
6. 对于自定义类型对象, malloc/free 只会开辟空间, 不会调用构造函数与析构函数, 而 new 在申请空间后会调用构造函数完成对象的初始化, delete 在释放空间前会调用析构函数完成空间中资源的清理。

11.3 智能指针

为什么要智能指针: 为了便于堆内存管理, 防止内存泄漏/重复释放等问题。

头文件: <memory>。

智能指针本质上是模板类。

C++11 引入了 3 个智能指针类型:

std::unique_ptr<T> : 独占资源所有权的指针, 离开 unique_ptr 对象的作用域时, 会自动释放资源。

std::shared_ptr<T> : 共享资源所有权的指针, 当引用计数为 0 的时候, 自动释放资源。

std::weak_ptr<T> : 共享资源的观察者, 需要和 std::shared_ptr 一起使用。

weak_ptr 是为了配合 shared_ptr 而引入的一种智能指针, 因为它不具有普通指针的行为, 没有重载 operator* 和 ->, 它的最大作用在于协助 shared_ptr 工作, 像旁观者那样观测资源的使用情况。weak_ptr 可以从一个 shared_ptr 或者另一个 weak_ptr 对象构造, 获得资源的观测权。但 weak_ptr 没有共享资源, 它的构造不会引起指针引用计数的增加。使用 weak_ptr 的成员函数 use_count() 可以观测资源的引用计数, 另一个成员函数 expired() 的功能等价于 use_count() == 0, 但更快, 表示被观测的资源(也就是 shared_ptr 管理的资源)已经不复存在。weak_ptr 可以使用一个非常重要的成员

函数 `lock()` 从被观测的 `shared_ptr` 获得一个可用的 `shared_ptr` 对象，从而操作资源。但当 `expired()==true` 的时候，`lock()` 函数将返回一个存储空指针的 `shared_ptr`。

`std::auto_ptr` 已被废弃。

11.4 生命周期

常见的四种对象：全局、静态、局部、动态创建对象四种。

全局对象在 `main` 函数执行前被创建，`main` 退出后被销毁。

静态对象在第一次进入作用域时被创建，在 `main` 退出后被销毁（若程序不进入其作用域，则不会被创建）。

局部对象在进入作用域时被创建，在退出作用域时被销毁。

动态创建对象（`new/malloc` 创建）由程序员创建与销毁，如果不销毁则在程序退出时由计算机回收，不是“优雅”的退出（不会调用析构函数），程序员不释放容易造成内存泄漏。

11.5 代码实战经验分享

多线程

进程是正在运行的程序的实例，而线程是进程中的实际运作单位。一个程序有且只有一个进程，但可以拥有至少一个的线程。不同进程拥有不同的地址空间，互不相关，而不同线程共同拥有相同进程的地址空间。

C++ 11 之后添加了新的标准线程库 `std::thread`，`std::thread` 在 `<thread>` 头文件中声明。

```
#include<thread>
```

```
std::thread thread(callable)
```

`callable` 为一个可调用对象，可以是以下三个中的任何一个：

函数指针、函数对象、`lambda` 表达式

`thread` 常用成员函数

`join()` 等待线程结束并清理资源（会阻塞）

`joinable()` 返回线程是否可以执行 `join` 函数

`detach()` 将线程与调用其的线程分离，彼此独立执行（此函数必须在线程创建时立即调用，且调用此函数会使其不能被 `join`）

get_id()获取线程 id

std::mutex 是 C++11 中最常用的互斥量，调用 mutex 的 lock 上锁后，其它的线程就不能操作 mutex，直到这个线程将 mutex 解锁 unlock。

mutex 的函数

lock()将 mutex 上锁，如果 mutex 已经被其它线程上锁，那么会阻塞，直到解锁；如果 mutex 已经被同一个线程锁住，那么会产生死锁。

unlock()解锁 mutex，释放其所有权，如果有线程因为调用 lock()不能上锁而被阻塞，则调用此函数会将 mutex 的主动权随机交给其中一个线程；如果 mutex 不是被此线程上锁，那么会引发未定义的异常。

try_lock()尝试将 mutex 上锁，如果 mutex 未被上锁，则将其上锁并返回 true；

如果 mutex 已被锁则返回 false。

linux 多线程的使用

```
#include <pthread.h>
```

```
pthread_create (thread, attr, start_routine, arg) // 创建线程
```

```
pthread_exit (status) // 退出线程
```

代码编译：g++ ./test_thread.cpp -lpthread

注意链接 pthread 库

代码执行结果展示：

未加锁：

```
this is run in thread lambda function!  
this is run in thread lthis is run in thread lambda function!  
this is run in thread lambda function!  
  
in thread func_thread !
```

加锁：

```
in thread func_thread !
in thread func_thread !
in thread func_thread !
in thread func_thread !
in thread func_thread !
in thread func_thread !
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
this is run in thread lambda function!
in thread func_thread !
in thread func_thread !
in thread func_thread !
in thread func_thread !
this is run in thread lambda function!
this is run in thread lambda function!
in thread func_thread !
in thread func_thread !
in thread func_thread !
in thread func_thread !
in thread func_thread !
```

linux 线程 pthread 结果:

```
in thread 2 !
in thread 1 !in thread 2 !
in thread 1 !
in thread 1 !
in thread 1 !
in thread 1 !
in thread 1 !
in thread 1 !
in thread 1 !

in thread 2 !
in thread 2 !
in thread 2 !
in thread 2 !
in thread 2 !
in thread 2 !
in thread 2 !
in thread 2 !in thread 1 !
in thread 2 !
in thread 2 !

in thread 1 !
```

第十二章 面向对象程序设计

C++面向对象的三大特性为：封装、继承、多态。

12.1 基类与派生类介绍

继承机制可以利用已有的数据类型来定义新的数据类型。所定义的新的数据类型不仅拥有新定义的成员，而且还同时拥有旧的成员。

称已存在的用来派生新类的类为**基类**，又称为**父类**。

由已存在的类派生出的新类称为**派生类**，又称为**子类**。

一个派生类可以从一个基类派生，也可以从多个基类派生。从一个基类派生的继承称为**单继承**；从多个基类派生的继承称为**多继承**。

三种不同的继承方式的基类特性和派生类特性：

继承方式	基类特性	派生类特性
公有继承	public	public
	protected private	protected 不可访问
私有继承	public	private
	protected private	private 不可访问
保护继承	public	protected
	protected private	protected 不可访问

基类是对派生类的抽象，而派生类是基类的具体化。基类抽取了它的派生类的公共特征，而派生类通过增加行为将抽象类变为某种有用的类型。

12.2 虚函数

多态意指相同的消息给予不同的对象会引发不同的动作。

C++的两种多态：

- (1) 编译时多态：重载实现；
- (2) 运行时多态：虚函数实现。

C++中，在用父类指针调用函数时，实际调用的是指针指向的实际类型（子类）的成员函数。

C++的运行时多态性使得程序调用的函数是在运行时动态确定的，而不是在编译时静态确定的，运行时多态是通过虚函数实现的，虚函数是加了 **virtual** 修饰词的类的成员函数。

虚函数的实现是由两个部分组成的，虚函数指针与虚函数表。(代码经验分享详细讲解)

12.3 纯虚函数与抽象类

对于普通函数，如果仅仅给出它的声明而没有实现它的函数体，这是编译不过的。

纯虚函数在基类中仅仅给出声明，不对虚函数实现定义，而是在派生类中实现，因而纯虚函数没有函数体。

纯虚函数是在虚函数声明之后加个=0。

含有纯虚函数的类被称为抽象类。

抽象类只能作为派生类的基类，不能定义对象，但可以定义指针。在派生类实现该纯虚函数后，定义抽象类对象的指针，并指向或引用派生类对象。

- 1) 在定义纯虚函数时，不能定义虚函数的实现部分；
- 2) 在没有在子类中实现纯虚函数之前，是不能调用这种函数的。

抽象类的唯一用途是为派生类提供基类，纯虚函数的作用是作为派生类中的成员函数的基础，并实现动态多态性。

继承于抽象类的派生类如果不能实现基类中所有的纯虚函数，那么这个派生类也还是抽象类。

12.4 多继承与虚继承

派生类只有一个基类称为**单继承**，但是如果派生类有两个或多个基类则称为**多继承**。

多继承的语法是将多个基类用逗号隔开，每个基类前面都需要加上继承方式 (public/private/protected)，不写则默认为 private。

为了解决多继承时的命名冲突和冗余数据问题，C++提出了**虚继承**，使得在派生类中只保留一份间接基类的成员，虚继承的语法是在继承的基类前面加上 virtual，如 class A : virtual public B。

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。其中，这个被共享的基类就称为虚基类 (Virtual Base Class)。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含一份虚基类的成员。

12.5 代码经验分享

1、多继承的注意点

(1) 命名冲突

当两个或多个基类中有同名的成员时，如果直接访问该成员，就会产生命名冲突，编译器不知道使用哪个基类的成员。这个时候需要在成员名字前面加上类名和域解析符::，以显式地指明到底使用哪个类的成员，消除二义性。

(2) 菱形继承

菱形继承指的是一个子类多继承的俩个父类又同时继承于同一个父类。

所以在菱形继承中会存在一个问题：D 会继承 B/C 成员，但 B/C 同时继承了 A，所以在 D 中会有俩组 A 的成员，这就是二义性与数据冗余问题。

解决方法是虚继承。

虽然通过 `virtual` 解决了菱形继承的数据冗余与二义性问题，但是在内存的存放与寻找中特别麻烦，同时也伴随着空间布局与存取时间的额外成本。这样会降低程序的效率，带来性能上的损耗，所以不到万不得已不要使用菱形继承。

2、虚函数指针与虚函数表

C++中虚函数 (Virtual Function) 是通过一张虚函数表 (Virtual Table) 来实现的。简称为 V-Table。表中存储虚函数的地址表，这张表解决了继承、覆盖的问题，有虚函数的类的实例中这个表被分配在了这个实例的内存中，而 C++的编译器保证了虚函数表的指针存在于对象实例最前面的位置 (这是为了保证取到虚函数表有最高的性能)，因此可以通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

虚函数指针指向虚函数表。

第十三章 模板与泛型编程

13.1 C++模板的定义与使用

1、函数模板

函数模板是建立一个通用函数，它所用到的数据的类型 (包括返回值类型、形参类型、局部变量类型) 可以不具体指定，而是用一个虚拟的类型来代替 (相当于用一个标识符来占位)，等函数调用时再根据传入参数的类型推断出数据类型。这个通用函数就称为函数模板 (Function Template)。

语法：

```
template<typename T> void get_max(vector<T> vec, T& max){}
```

类型定义可以使用 `typename`，也可以使用 `class`。

2、类模板

类模板的定义和函数模板类似，如下：

```
template <class T1, class T2> class Line{};
```

标准库的容器都是模板类实现的。

3、模板中 `typename` 和 `class` 关键字说明

功能相同，`typename` 是为了模板定义新引进的关键字。`class` 可以定义模板数据类型，还是类创建的关键字。

13.2 模板实参推断

对于函数模板和类模板，编译器利用调用中的实参类型来确定其模板参数类型，这个过程称为模板实参推断。

13.3 函数模板重载

函数模板可以重载，只要它们的形参表不同即可。

13.4 运算符重载

运算符重载是为了让自定义的数据类型可以使用 C++ 内置的运算符进行运算，让代码更美观易懂。

重载的运算符是带有特殊名称的函数，函数名是由关键字 `operator` 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

可重载运算符/不可重载运算符

下面是可重载的运算符列表：

双目算术运算符	+ (加), -(减), *(乘), /(除), %(取模)
关系运算符	==(等于), != (不等于), < (小于), > (大于), <=(小于等于), >=(大于等于)
逻辑运算符	(逻辑或), &&(逻辑与), !(逻辑非)
单目运算符	+ (正), -(负), *(指针), &(取地址)
自增自减运算符	++(自增), --(自减)
位运算符	(按位或), & (按位与), ~(按位取反), ^(按位异或), << (左移), >>(右移)
赋值运算符	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=
空间申请与释放	new, delete, new[], delete[]
其他运算符	() (函数调用), -> (成员访问), , (逗号), [] (下标)

下面是不可重载的运算符列表：

- `.`：成员访问运算符
- `.*`, `->*`：成员指针访问运算符
- `::`：域运算符
- `sizeof`：长度运算符
- `?:`：条件运算符
- `#`：预处理符号

13.5 代码实战经验分享

1、虚析构

让一个基类指针指向用 `new` 运算符动态生成的派生类对象，在析构函数中用 `new` 运算符动态生成的对象需要通过 `delete` 释放指向对象的指针来释放，否则会造成内存泄露。如果一个基类指针指向用 `new` 运算符动态生成的派生类对象，而释放该对象时是通过释放该基类指针来完成的，那么 `delete` 指针对象只会调用基类的析构函数，不会调用派生类的析构函数，派生类对象无法释放，造成内存泄露。

2、=default 创建默认构造函数

3、=delete 可以禁止不想要的成员函数

4、explicit

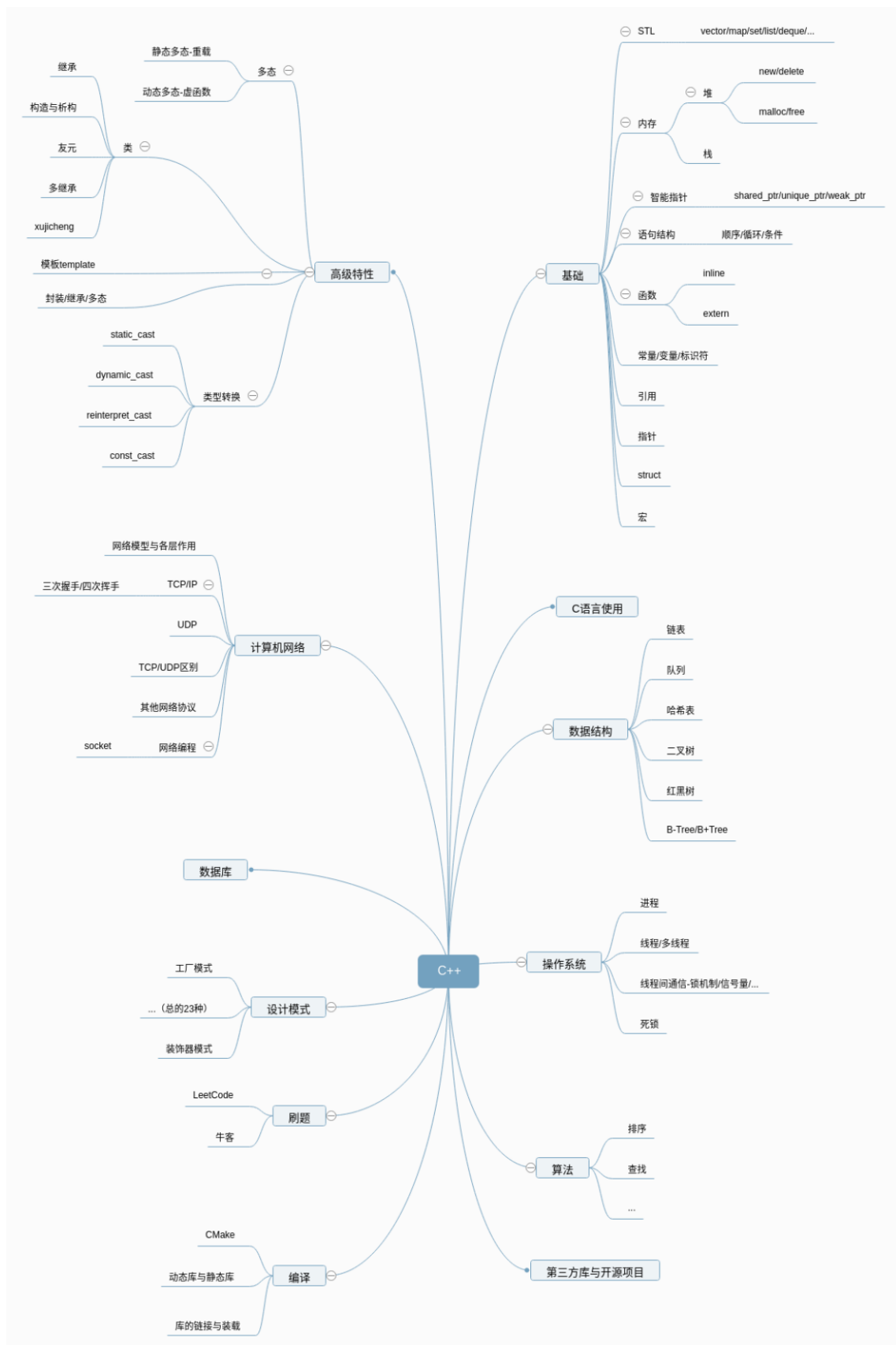
禁止编译器执行非预期 (往往也不被期望) 的类型转换。

第 14 章 C++学习总结

14.1 C++常见知识点总结

- C++语言基础：常量、变量、标识符、循环/条件/跳转语句
- 指针、引用、数组
- 引用与指针区别
- C 和 C++ 的一些区别，比如 new、delete 和 malloc、free 的区别
- 一些关键字的作用：static、const、volatile、extern
- 四种类型转换：static_cast, dynamic_cast, const_cast, reinterpret_cast
- STL 部分容器的实现原理，如 vector、deque、map
- 类的封装，构造和析构、静态成员、对象管理
- 类的构造(有参构造函数、无参构造、拷贝构造、默认构造函数)和析构
- 友元函数、友元类、运算符重载
- 模板使用
- 继承、虚继承、菱形继承等
- 多态：动态绑定，静态多态
- 虚机制：虚函数、虚函数表、纯虚函数
- 重写、重载
- 智能指针原理：引用计数、RAII（资源获取即初始化）思想
- 智能指针使用：shared_ptr、weak_ptr、unique_ptr 等
- 编译链接机制、内存布局（memory layout）、对象模型
- C++11 部分新特性

14.2 C++的考核范畴



14.3 SLAM 算法中 C++语法解读

看不懂源码的原因：

1、语法层面的问题

(1) 写法比较奇怪

```
49 class Atlas
50 {
51     friend class boost::serialization::access;
52
53     template<class Archive>
54     void serialize(Archive &ar, const unsigned int version)
55     {
56         ar.template register_type<Pinhole>();
57         ar.template register_type<KannalaBrandt8>();
58
59         // Save/load a set structure, the set structure is broken in libboost 1.58 for ubuntu 16.
60         //ar & mspMaps;
61         ar &.mvpBackupMaps;
62         ar &.mvpCameras;
63         // Need to save/load the static Id from Frame, KeyFrame, MapPoint and Map
64         ar & Map::nNextId;
65         ar & Frame::nNextId;
66         ar & KeyFrame::nNextId;
67         ar & MapPoint::nNextId;
68         ar & GeometricCamera::nNextId;
69         ar & mnLastInitKFidMap;
70     }
```

(2) 宏

```
72 public:
73     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
```

可以逐层跳转进去看最终的实现

也可以不用纠结，看名字猜含义，不重要的就略过。

(3) 基础弄懂，看不懂的代码先往熟悉的实现方式上去靠

& 常用的含义就是：按位与，引用（左值引用与右值引用）...

(4) 看不明白的实现，可以自己参考该代码写 demo 进行测试，推断功能

2、语义层面

(1) 代码逐行/逐函数/逐类/逐文件分解式阅读与理解；

(2) 开源一般会有很多的博客介绍，可以参考别人的文章理解；

(3) 开源中的算法，可以先找到算法介绍进行理解，然后再看代码。