C++自学——基础部分

第一个C++程序

```
# include <iostream>
using namespace std;
int main()
{
    cout << "C++ program" << endl;
    printf("hello world!");
    return 0;
}
```

虽然这个程序看起来十分的简单, 但是麻雀虽小五脏俱全

首先来谈谈main()函数:

第一行的int main()叫做函数头, 花括号中间包含的部分叫做函数体, 而在C++中每条完整的指令称为语句

main()中最后一行语句叫做返回语句, 这还是太繁琐了, 于是系统中, 如果main()运行到最后没有遇到返回语句, 则默认最后以return 0结尾, 但是这种默认**只在*main()*函数里面有**.

那么前面的int是什么? 函数头描述了函数与调用它的函数之间的接口, 而函数是有返回值的; 因此, 如果是int main(), 那么就代表: main()函数可以给调用他的函数返回一个整数值

那么这个括号又是什么东西? 括号部分叫做形参列表, 描述的是从调用函数传递给被调用函数的信息. 但是在main()这个函数中, 一般来说是空括号, 因为main()一般用作程序与操作系统之间的桥梁. 换而言之, main()函数头描述的是main()和操作系统之间的接口.

空括号就是代表不接受人员任何信息, 或者说不接受任何参数. 值得一提的是, 在C++中, 如果括号里面加上void, 那么这就相当于空括号, 代表不会接受任何参数. 但是在C中, 这种两种做法其实是有区别的, C中的空括号代表对是否接受参数保持沉默.

那么对于这个函数来说, 他的名字可以换吗? 答案是不行, 一个程序中必须包含一个名为main()的函数, 而运行程序的时候, 一般从main()开始运行.

那么接下来, 我们来看# include

这代表什么东西? 首先我们要了解, 一个C/C++程序是需要一个预处理器的, 改程序进行主编译之前对源文件的处理. 而# include 就是相当于把iostream文件添加到程序之中, 或者说, iostream文件的内容代替了这一行. 值得注意的是, 这种取代并不是"一次性的", 或者说"取代"一次其实并不合适, 而是在原有文件不变的情况下, 产出一份源代码文件和iostream组合成的一个复合文件

io代表input output, 因此, 使用了cin和cout的程序必须都包含iostream

这种文件称为包含文件(include file)或者说头文件(heading file). C/C++各自的头文件名是有些不一样的

1. C++ 旧风格: .h结尾 例如iostream.h C++可以使用
2. C旧风格: .h结尾, 例如math.h, C/C++都可以使用
3. C++新风格: 没有扩展名, 例如iostream C++可以使用, 但是要加上 namespace std
4. C 转换后的风格: 前缀加上c, 没有扩展名 例如cmath C++ 可以使用, 但是不是C的特性

接下来我们来看看, 什么是using namespace std

如果使用的是iostream而不是iostream.h, 则应该使用下面的名称空间编译指令来使iostream中的定义对程序可用

名称空间支持是C++的一个特性. 两个文件可能包含一个相同名称的函数, 那么编译器运行的时候遇上了调用这个函数, 它怎么是知道调用的是谁的函数? 例如有一个函数叫做damn(), 而Franklin 和 Van都有这个函数.

要调用谁的呢? 于是会用Franklin::damn(); Van::damn(), 以做区分

类, 函数, 变量是C++编译器的标准组件, 它们都放置在名称空间std中, 因此事实上, iostream中用于输出的cout其实是std::out. 那么这里的using namespace std就代表着: std名称空间的所有名称都可用, 是一种偷懒的做法. 更好的方法其实是: (假如说, 我只用到了cout 和 cin)
using std::cout;

using std::cin;

最后看看cout << endl;

endl是C++中特殊的符号, 表示一个为重要的概念: 重启一行. 一个endl会使得光标移到下一行

endl对于cout来说具有特殊的意义, 因此它被称为控制符

当然, 换行还有一种旧方法: \n; 因此, 如果想要一行空着, 有两种方法:

cout << "\n" ;

cout << endl;

那么涉及到输入输出流, 那么有个不得不提的那便是: 缓冲区(buffer). 缓冲区是一个存储区域, 用于保存数据, IO设施通常将输入或输出保存在一个缓冲区中, 读写缓冲区的动作与程序中的动作无关. 我们可以显式地刷新输出缓冲, 以便于强制将缓冲区中的数据写入输出设备

当然, 输出还能有另一套方法:

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

int main() {
    // Complete the code.
    int a;
    long b;
    char c;
    float f;
    double d;
    scanf("%d %ld %c %f %lf", &a, &b, &c, &f, &d); // 一次性读取五个参数
    // 把输入的当做是一个字符串, 每个元素是不同数据类型
    printf("%d\n%ld\n%c\n%.3f\n%.9lf\n",a, b, c, f, d);
    // \n代表换行, 然后.3和.9代表保留的小数点位数
    return 0;
}
// 这里是C++的一道题目, 但是可以改过来
```

注意到, 这里调用的是, 说明是<stdio.h>的转换后的风格!

C++源代码风格

1. 每条语句占一行
2. 每个函数都有一个开始的花括号和一个结束的花括号, 两个花括号各占一行
3. 函数中的语句都相对于花括号进行缩进
4. 与函数名称相关的圆括号周围没有空白

注释

两种注释格式:

单行注释: //　　多行注释: /*　*/(注意, 这种多行注释的语法在C中也可以用)

```
# include <iostream>
using namespace std;
/*
main是一个程序的入口, 每个程序都必须有这么一个函数
而且一个名字有且只能有一个
*/
int main()
{
    cout << "C++ program" << endl; // 输出字符串
    return 0;
}
```

变量

变量的创建

变量存在的意义: 方便我们管理内存空间

变量创建的语法: 数据类型 变量名 = 变量初始值

例如: `int a = 10;`, 创建变量a的同时, 赋予其一个初始值, 这称之为初始化

但是注意! 初始化不等于赋值! **后者是把对象当前的值擦除, 而以一个新值来代替**

当然同时也可以给多个相同类型的变量进行创建:

`int a = 10, b, c, d = 154;`

```cpp
# include <iostream>
using namespace std;
int main()
{
    int a = 10;
    cout << "a = " << a << endl;
    return 0;
}
// output:  a = 10
```

为什么变量必须声明? **为了防止因为变量名称拼写错误而导致的错误**

常量

用于记录程序中不可更改的数据

```cpp
/*
C++定义常量两种方式:
#define 常量名 常量值, 这通常在文件上方进行定义
const 数据类型 常量名 = 常量值
*/
```

注意: 一般来说, 不同文件之间可能有const定义的相同名字的变量, 但是默认就是const对象仅在各自文件内有效. 假如说文件1和2都有const int a, 但是其实两个文件中的两个a是独立的

如果想要一个const的常量被多个文件共享, 不管是声明还是定义都添加extern关键词

附: const的实质:

```cpp
double dval = 3.14;
const int &ri = dval;
// 实际上发生了什么呢?
const int temp = dval;
const int &ri = temp;
// 这种情况下, ri绑定在了一个临时量对象
```

关键字

C++中预先保留的单词(标识符)

在给变量或者常量起名称的时候, 不能够使用关键词, 否则会产生歧义

标识符命名规则:

给标识符(变量 常量)命名的时候, 有一套自己的规则

1. 标识符不能是关键字
2. 标识符只能由字母、数字、下划线构成（没错，空格不可以）
3. 第一个字符必须是字母或者是下划线(不可以是数字)
4. 标识符中是区分大小写的

初始化

一般来说: int a = 5; 这种初始化语句是来自于C语言. 而C++也有C中没有的初始化语句:

例如: int a(5); // alternative C++ syntax

还有一种初始化方式, 这种方式用于数组和结构, 但是在C++98中, 也可以用于单值变量:

int a = {5}; 而这种情况在C++11中更为常见, 而且登号可以不使用: int a{5};

这有助于防范类型转换的错误!! C++11使得大括号初始化器可以用于任何类型!!

此外, 可以使用undesigned语法来扩大储存范围: 例如int原本的范围是-32768 到32767, 但是如果使用了:

undesigned int ...; 那么这个储存的范围可以变成0-65535; 当然, 前提是这个变量将不能变成负数

那么顺嘴提到了储存的上限, 那么假如说int a = 32767; 那么 a+1 = ?

事实上, 这个数据将会溢出, 而反过来取中区间另一端的值, 即为-32768

同理可得, 如果是undesigned int a = 65535, then a+1 = 0; undesigned int a = 0; a-1 = 65535

那么, 假如说我只想声明一个变量(declaration), 但是并不想初始化它, 那么就加上关键词extern.

`extern int a;`

注意: 变量能且仅能被定义一次, 但是可以被多次声明

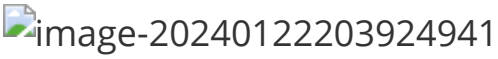
数据类型

整型

C++规定在创建一个变量的时候, 必须之处它的数据类型, 否则不能分配内存

而数据类型的存在意义: 给变量分配合适的内存空间

整型的类型也用很多: 区别在于所占内存的空间不同



记住指数的方式:15 * 2 + 1 = 31　31 * 2 + 1 = 32

注意: short范围: -32768 ~ 32767

请注意: *short*占用内存比*int*少, **但是两者的长度是一样的! 因此如果节省内存很重要, 那么一定要是用*short*而不是*int*!**

整型字面值

C++使用前一两位来标识数字常量的基数(进制), 如果开头是1-9, 那么基数就是10

如果第一位是0, 第二位是1-7, 那么就是八进制

如果前两位是0x或者0X, 那么就是十六进制: a-f或者A-F代表了10-15

在默认情况下, cout都是以十进制格式显示整数; 但是提供了切换输出格式的控制符: dec(十进制) hex(十六进制) oct(八进制)

```
# include <iostream>
using namespace std;
int main()
{
    short a = 114;
    short b = 156;
    cout << hex;
```

```
    cout << a << endl << endl;// 两个endl代表额外空出一行!
    cout << b << endl << endl;
    cout << dec;
    cout << a << endl;
    return 0;
}
/*输出
72    (114的十六进制)

9c    (156的十六进制)

114
*/
```

注意: 单独一个cout << hex; 并不会显示任何东西, 或者是换行, 但是会将输出格式变为十六进制, 而且直到下一次修改, 输出一直是十六进制!!

sizeof 关键字

利用sizeof可以统计数据类型所占内存大小

```
# include <iostream>
using namespace std;
int main()
{
    short a = 32765;
    int b = 114514;
    long long c = 1919810;
    cout << "The size of " << "a" << " is " << sizeof(a) <<
endl;
    cout << "The size of " << "b" << " is " << sizeof(b) <<
endl;
    cout << "The size of " << "c" << " is " << sizeof(c) <<
endl;
    return 0;  // 注意字符串必须使用双引号, 而不能是的单引号
}
/*
The size of a is 2
The size of b is 4
The size of c is 8
*/
```

实型(浮点型)

用于表示小数

浮点型变量分为两种: 单精度 float　双精度 double

 image-20240122205145591

注意: 这里说的是有效数字! 3.14有效数字是3位!

字符型

char 变量名 = ' '; 字符串变量用于显示**单个字符**

字符串变量只占用一个字节! 字符型变量并不是把字符本身放在内存之中, 而是把对应的ASCII码放到存储单元之中

而且注意! 这个时候, **只能使用单引号**!!!!

我们可以使用(int)变量名 来查看这个变量名所对应的单字符的ASCII码

```
# include <iostream>
using namespace std;
int main()
{
    char ch1 = 'a';
    char ch2 = 'A';
    cout << "The size of ch1 is " << sizeof(ch1) << endl;
    cout << "The ASCII of ch1 is " << (int)ch1 << endl;
    cout << "The ASCII of ch2 is " << (int)ch2 << endl;
    return 0;
}
/*
The size of ch1 is 1
The ASCII of ch1 is 97
The ASCII of ch2 is 65
*/
```

注意: 如果cin >> ch1, 如果输入的是A, 那么其实储存的是它的ASCII码! 那么为什么输入的是A, 输出的也是A, 但是储存的却是其ASCII码? 那是因为cin和cout帮忙完成了转化, 这两个的行为都是由变量类型引导的!!!

另外, 额外介绍一个函数: cout.put(), 这个可以显示一个字符. 这个函数长得和python的风格很像, 没错, 它与OOP离不开关系

字符串型

C++中, 有两种风格:

沿用C的: char 变量名[] = "字符串"

C++风格: string 变量名 = "字符串"

注意, 字符串的话, 一定要用双引号; 单字符型的才用单引号

但是注意!!!!!

如果是使用string的话, 一定要在程序的最前面加上:

```
# include <string>
```

这个是头文件, 一定要加上这句话才能用string语法

布尔类型

bool类型只有两个值: true = 1 flase = 0

语法: bool 变量名 = true/flase

它们占用的类型只有一个字节

占位符

`scanf` / `printf`

Refer to the table in this page.

type	format specifier
`short`	`%hd`
`int`	`%d`
`long`	`%ld`
`long long`	`%lld`

type	format specifier
`unsigned short`	`%hu`
`unsigned`	`%u`
`unsigned long`	`%lu`
`unsigned long long`	`%llu`

- `%f` for `float`, `%lf` for `double`, and `%Lf` for `long double`.

占用多少内存?

下表列出了关于标准整数类型的存储大小和值范围的细节：

类型	存储大小	值范围
char	1 字节	-128 到 127 或 0 到 255
unsigned char	1 字节	0 到 255
signed char	1 字节	-128 到 127
int	2 或 4 字节	-32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
unsigned int	2 或 4 字节	0 到 65,535 或 0 到 4,294,967,295
short	2 字节	-32,768 到 32,767
unsigned short	2 字节	0 到 65,535
long	4 字节	-2,147,483,648 到 2,147,483,647
unsigned long	4 字节	0 到 4,294,967,295

注意，各种类型的存储大小与系统位数有关，但目前通用的以64位系统为主。

以下列出了32位系统与64位系统的存储大小的差别（windows 相同）：

Windows vc12		Linux gcc-5.3.1		Compiler
win32	x64	i686	x86_64	Target
1		1	1	char
1		1	1	unsigned char
2		2	2	short
2		2	2	unsigned short
4		4	4	int
4		4	4	unsigned int
4		4	8	long
4		4	8	unsigned long
4		4	4	float
8		8	8	double
4		4	8	long int
8		8	8	long long
8		12	16	long double

根据上图，可见（unsigned）int, long和long double占用的内存和系统版本有关

同时，char究竟是默认signed还是unsigned是**implementation-defined**!

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

类型	存储大小	值范围	精度
float	4 字节	1.2E-38 到 3.4E+38	6 位有效位
double	8 字节	2.3E-308 到 1.7E+308	15 位有效位
long double	16 字节	3.4E-4932 到 1.1E+4932	19 位有效位

- It is also guaranteed that `sizeof(short)` ⩽ `sizeof(int)` ⩽ `sizeof(long)` ⩽ `sizeof(long long)`.
  - `sizeof(T)` is the number of **bytes** that `T` holds.

如何选择类型

1. 如果明确知道不可能是负数, 使用unsigned
2. **使用*int*执行整数运算!** (后面好像理解错了, 说short更好, 但是其实int更好), short常常显得太小而long一般和int有一样的尺寸, 如果超过了int, 选用long long
3. 执行浮点数运算用double, 因为float经常精度不够, 而且两者计算代价相差无几. 事实上, 某些机器上, double运行更快; long double提供的精度是没有必要的, 而且消耗很大

`int` is the most optimal integer type for the platform.

- Use `int` for integer arithmetic by default.
- Use `long long` if the range of `int` is not large enough.
- Use smaller types (`short`, or even `unsigned char`) for memory-saving or other special purposes.
- Use `unsigned` types for special purposes. We will see some in later lectures.

类型转换

- 当我们把一个非布尔类型的算数值赋为布尔类型, 1就是true, 0就是false
- 把一个布尔类型赋给非布尔类型, true就是1, false就是0
- 把一个浮点数赋给整数类型, 做近似处理, **只保留小数点之前的部分**

- 把一个整数值赋给浮点数类型, 小数部分记为0
- 给undesigned类型一个超出范围的值, 结果是初始值对无符号类型表示值数值取模后的余数. 例如8字节大小的undesigned char范围是0-255, 表示总数为256, 因此undesigned char i = -1, 代表的就是255
- 如果是赋给signed类型一个超出范围的值, 那么结果是未定义的(undefined), 可能产生垃圾数据

还有一些情况会发生类型的转换:

1. Binary `+`, `-` and `*`, `/` If any one operand is of floating-point type and the other is an integer, **the integer will be implicitly converted to that floating-point type**.

   ```
   double pi = 3.14;
   int diameter = 20;
   // WhatType c = pi * diameter;  ??
   double c = pi * diameter; // 62.8
   ```

   二元计算中有浮点数， 那么结果一定是浮点数

2. 除法：a/b

Assume `a` and `b` are of the same type `T` (after conversions as mentioned above).

- Then, the result type is also `T`.

Two cases:

- If `T` is a floating-point type, this is a floating-point division.

因此如果希望两个整数相除得到double, 可以前面乘以一个1.0出发隐形转换

- If `T` is an integer type, this is an integer division， 结果**保留**整数

3/-2 = **-1**（***truncated towards zero***） a/2: int a/2.0: double

3. signed integer overflow: 数据溢出

```
int ival = 100000; long long llval = ival;
int result1 = ival * ival;                  // (1) overflow
long long result2 = ival * ival;            // (2) overflow
long long result3 = llval * ival;           // (3) not overflow
long long result4 = llval * ival * ival; // (4) not overflow
// RHS 有long long, 那么结果类型会“臣服于”long long
```

复合类型

compound type是指基于其他类型定义的类型, 有几种复合类型, 我们现在介绍两种: 引用和指针

引用: 函数中传参, 一般是右值引用(rvalue reference) , 因此对这个参数看似做了很多, 一点用也没有:

```
int a = 10; // 全局变量
void plusone(int a) // 实质: 临时再创建一个名字是a的变量, 把原来a的值赋给了它
{
    a += 1; // 一点用也没有!
    return;
} // 这个函数一旦调用结束, 临时创建的a直接释放
```

因此如果使用了&关键字符, 来左值引用(lvalue reference), 这样的话就会对这个变量进行实质性变化

```
int a = 10; // 全局变量
void plusone(int &a)
{
    a += 1; // 有用!
    return;
}
```

注意: 引用并非对象, 因此不能给引用"引用"; 引用就是给存在的对象其别名, 避免混乱

指针: 与引用类似, 指针也实现了对其他对象的间接访问, 但是两者又有不同的地方:

1. 指针本身是个对象(因此有指向指针的指针, 想要访问数据, **两次解引用; 也有指针的引用(见下))

```
int i = 42, *p;
int *&r = p; // r是一个对p的引用, 左边有个*是因为p指针数据类型视为int *
r = &i; // 通过r操控p指向i
*p = 0; // 通过p修改内存数据为0
```

2. 指针在定义的时候无需赋值(NULL)

```
# include <iostream>
using namespace std;
int a = 10;
void plusone(int *p)
{
    *p += 1; // 解引用, 直接获取到了对应内存的数据
}
int main()
{
    int * p = nullptr; // 创建一个空指针, 注意啫喱的正确语法
    p = &a; // 获取它的地址, 指向它
    plusone(p);
    p = nullptr; // 置为空指针
    cout << a << endl; // 11
    return 0;
}
```

建议: 初始化所有指针(nullptr), 并且在可能的情况下, 尽量等定义了对象之后再定义指向它的指针. 如果实在是不知道指针指向了哪里, 那么初始化为nullptr或者NULL

tips: `int i = 1024; *p = &i; &r = i;`完全合法! 基本数据类型和类型修饰符的关系: 后者不过是声明符的一部分罢了. 注意, *是修饰p的, 而不是int * 是一个整体:

`int *p1, p2;  // p1是int类型指针, p2是没有初始化的int变量`

auto声明

可以使用auto关键词, 来将变量的类型设置成与初始值相同. 这看似很方便, 但是有的时候也会带来麻烦

例如： auto a = 0; 但是我想把a的类型设置成double呢? 对不起, 这句话只能把a的类型设置为int

使用auto也能在一条语句中声明多个变量, 因为一条声明语句只能有一个基本数据类型, 所以语句所有变量的 初始基本数据类型都必须一样:

```
auto i = 0, *p = i; // 合法, i是整型, p是整型指针
auto sz = 0, pi - 3.14 // 不合法
```

数据的输入

语法: cin >> 变量

```
# include <iostream>
# include <string>
using namespace std;
int main()
{
    string a = "";
    cout << "Do you like what you see?" << endl;
    cin >> a;
    cout << "Your answer is: " << a << endl;
    int b = 0;
    cout << "How old are you" << endl;
    cin >> b;
    cout << "You are " << b << " years old" << endl;
    return 0;
}
/*
Do you like what you see?
Yes
Your answer is: Yes
How old are you
18
You are 18 years old
*/
```

注意: 甚至可以给布尔类型的数据进行输入赋值

数据的输出

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

int main() {
    // Complete the code.
    int a;
    long b;
    char c;
    float f;
    double d;
    scanf("%d %ld %c %f %lf", &a, &b, &c, &f, &d);
    printf("%d\n%ld\n%c\n%.3f\n%.9lf\n",a, b, c, f, d);
    return 0;
}
```

Static variables & Const

```
void start_game(Player *p1, Player *p2, int difficulty,
GameWorld *world) {
  bool called = false; // Note that we can use a shorter name,
                       // because it is local to this function.
  if (called)
    report_an_error("You cannot start the game twice!");
  called = true;
  // ...
}// not gonna work
```

```
void start_game(Player *p1, Player *p2, int difficulty,
GameWorld *world) {
  static bool called = false;
  if (called)
    report_an_error("You cannot start the game twice!");
  called = true;
  // ...
} // gonna work
```

想尝试设计一种只能调用一次的函数

第一个程序：

- Every time `start_game` is called, `called` is created and initialized to zero.
- Every time `start_game` returns, `called` is destroyed.
- We need a "local" variable whose lifetime is longer!

第二个程序：

The lifetime of a local `static` variable is **as long as** that of a global variable. *(They both have static storage duration.)*

- A local `static` variable is initialized **during program startup**, and is destroyed **on program termination**.

It behaves just like a global variable, but its name is inside a function, which does not pollute the global name space.

总而言之，static充当了global的功能，但是这个静态变量的初始化又没有污染全局变量空间。

Const:

A `const` variable cannot be modified after initialization.

Therefore, an uninitialized `const` local non-`static` variable is almost a non-stop ticket to undefined behavior.

```
// in some function
const int n; // `n` has indeterminate values
n = 42; // Error: cannot modify a const variable.
scanf("%d", &n); // Error: cannot modify a const variable.
```

In C++, `const` variables of built-in types must be initialized.

读入整数

使用声明在标准库头文件中的函数：`<ctype.h>`

```
// isupper(c) 判断 c 是不是大写字母
// islower(c) 判断 c 是不是小写字母
// isdigit(c) 判断 c 是不是数字字符
// isspace(c) 判断 c 是不是空白字符
```

```
int read(void) {
int x = 0;
char c = getchar();
while (isspace(c)) // 跳过空白
c = getchar();
while (isdigit(c)) {
x = x * 10 + c - '0';
c = getchar();
}
return x;
}
```

作用域

不论在程序的什么位置, 使用的每一个名字都会指向一个特定的实体: 变量, 函数, 类型等. 然而, 同一个名字如果出现在程序的不同位置, 也可能指向的是不同的实体

作用域是程序的一部分, 在其中名字有其特定的含义, C++中大多作用于以花括号分隔

同一个名字在不同的作用域中可能指向不同的实体, 名字的有效区域始于名字的声明语句, 以声明语句所在的作用域末端为结束; 如果定义在main和函数体之外, 就拥有全局作用域(global scope)

```
int i = 42;
int main()
{
    int i = 0;
    cout << i << endl; // 0
    cout << ::i << endl; // 42
    return 0;
}
```

转义字符

作用: 用于显示一些不能显示出来的ASCII

\n 换行, 将当前位置换到下一行开头

\\ 代表一个反斜线字符

\t 水平制表, 跳转到下一个tab的位置

```
# include <iostream>
using namespace std;
int main()
{
    cout << "Hello world";
    cout << "kiss my ass";
    return 0;
}
// 输出: Hello worldkiss my ass
# include <iostream>
using namespace std;
int main()
{
    cout << "Hello world/n";
    cout << "kiss my ass";
    return 0;
}
//输出: Hello world
//kiss my ass
# include <iostream>
using namespace std;
int main()
{
    cout << "Hello world\n" << endl;
    cout << "kiss my ass";
    return 0;
}
/*
Hello world

kiss my ass
*/
```

可见, cout代表你将开始输出(而且每一行都要有), 但是你要输出的这一行为是随着endl进行的, 并且每一次终止了输出行动, 自动换行; 而\n进行换行, 如何如第三个例子中的代码, 还完行之后endl进行结束, 那么再换到下一行, 接着进行输出内容

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

```cpp
int main()
{
    cout << "aaaa\tHello World" << endl;
    cout << "aa\tHello World" << endl;
    cout << "aaaaaa\tHello World\\" << endl;
    return 0;
}
/*
aaaa    Hello World
aa      Hello World
aaaaaa  Hello World\
*/
```

可见, 一个tab是8个space, 这样的话可以自动进行对齐

运算符

算数运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

注意, 两个整数相除, 依然是整数, 只不过是将小数点去除掉

当然, 两个小数也是可以相除的; 两个小数是不可以进行取模运算的

++a & a++ 均等效于 a+1

那么前置和后置的区别是什么呢?

前置, 先让变量+1, 然后进行表达式运算; 后置就是先进行运算, 然后变量+1

逻辑运算符

逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 **A** 的值为 1，变量 **B** 的值为 0，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

要记忆运算符所对应的术语!6

特殊：逗号运算符，是运算优先级最低之一的一种运算符，最后一个逗号后面的值作为整个表达式的值：

```
int a=3, b=5, c;
c = a>b, a+b; // c = 0, 因为优先运算a>b是false(0)
c = (a>b, a+b) // c = 8
```

程序流程结构

支持三种运行结构:

顺序结构 选择结构 循环结构

优先级，结合性和求值顺序

以下的内容十分重要！！！

```
优先级： f() + g() * h() 被解析为 f() + (g() * h()) 而非 (f() + g())
* h()
结合性： f() - g() + h() 被解析为 (f() - g()) + h() 而非 f() - (g()
+ h())
但是 f() , g() , h() 三者的调用顺序是 unspecified 的!
类似的还有： func(f(), g(), h()) 这里的 f() , g() , h() 三个函数的调用
顺序是
unspecified 的。
如果两个表达式 A 和 B 的求值顺序是 unspecified 的，而它们
都修改了一个变量的值，或者一个修改了某个变量的值，另一个读取了那个变量的值，
```

那么这就是 undefined behavior。
例： i = i++ + 2 的结果是什么？
一旦你开始分析它是 +1 还是 +2 ，你就掉进了 undefined behavior 的陷阱里。
++ 对 i 进行了修改，赋值也对 i 进行了修改，两个修改的顺序 unspecified，所以就是 undefined behavior。

求值顺序确定的运算符：
常见的运算符中，其运算对象的求值顺序确定的只有四个： && ， || ， ?: ， ，
&& 和 || ：短路求值，先求左边，非必要不求右边。
cond ? t : f ：先求 cond ，根据它的真假性选择求 t 还是求 f 。
， ：一种存在感极低的运算符。

位运算符
~a ：返回 a 的每个二进制位都取反后的结果。
a & b 的第i位是 1 当且仅当 a 和 b 的第i位都是 1 。
a | b 的第i位是 1 当且仅当 a 和 b 的第i位至少有一个是 1 。
a ^ b 的第i位是 1 当且仅当 a 和 b 的第i位不同。
假设 a 是无符号整数。
a << i 返回将 a 的二进制位集体左移 i 位的结果。
例如， a << 1 就是 a * 2 ， a << i 就是 a 乘以 2的i次方 。
左边超出的部分丢弃。
a >> i 返回将 a 的二进制位集体右移 i 位的结果。
例如， a >> 1 就是 a / 2 。右边超出的部分被丢弃。

如何获得一个无符号整数 x 的第 i 位？
我们约定第 0 位是最右边的位
(least significant bit)。
```
unsigned test_bit(unsigned x, unsigned i) {
return (x >> i) & 1u; // u代表unsigned (int)
}
```
或者：
```
unsigned test_bit(unsigned x, unsigned i) {
return (x & (1u << i)) >> i;
}
```

如何翻转一个无符号整数 x 的第 i 位？
```
unsigned bit_flip(unsigned x, unsigned i) {
return x ^ (1u << i);
}
```

如何截取一个无符号整数 x 的第i位？

```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {
return (x >> low) & ((1u << (high - low)) - 1);
}
```
或者
```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {
return (x & ((1u << high) - 1)) >> low;
}
```

if语句

单行

```
int main()
{
    if ( ... )
    {
        ...;
    }
}
```

注意, 这个if后面反而没有分号

多行

```
int main()
{
    if (...)
    {
        ...;
    }
    else
    {
        ...;
    }
}
```

多条件

```
int main()
{
```

```
    if (...)
    {
        ...;
    }
    else if ()
    {
        ...;
    }
    else
    {
        ...;
    }
}
```

嵌套If

在一个if或者else if 或者else里面的语句中再加入if模块

注意首行缩进!

三目运算符

表达式1 ? 表达式2 : 表达式3

如果1是真, 那么执行2, 并且返回2的结果

如果1是假, 那么执行3, 并且返回3的结果

```
# include <iostream>
# include <string>
using namespace std;
int main()
{
    int a = 10;
    int b = 20;
    int c = 0;
    c = ( a > b ? a:b);  // 将a和b进行比较, 将变量大的值赋值给变量c
    (a < b ? a:b) = 114514; //三目运算符返回的是变量, 可以继续赋值
    cout << a << endl;
    cout << c << endl;
    return 0;
}
```

```
/*
114514
20
*/
```

```
# include <iostream>
# include <string>
using namespace std;
int main()
{
    int a = 10;
    int b = 20;
    string c = "That's good";
    ( a > b ? cout<<"damn!"<<endl:cout<<c<<endl);// 三目运算符还可以执行表达式
    return 0;
}
// 返回值: That's good!
```

三目运算符能够很好简化语法:

```
int abs_int(int x) {
  if (x < 0)
    return -x;
  else if (x == 0)
    return 0;
  else if (x > 0)
    return x;
}
```

```
int abs_int(int x) {
  return x < 0 ? -x : x;
}333333333333333333333333333333333333333333333333333333333333333333333333
```

switch

语法: (注意: case后面有冒号!!!)

```
switch(表达式)
{
    case 结果1:
        执行语句;
        break;
    case 结果2:
        执行语句;
        break;
    ...
    default:
        执行语句;
        break;
}
```

```
# include <iostream>
# include <string>
using namespace std;
int main()
{
    int a = 0;
    cin >> a;
    switch(a)
    {
        case 0:
            cout << "good" << endl;
            break;
        default:
            cout << "bad" << endl;
            break;
    }
    return 0;
}
```

swtich也有自己的作用域问题：

```
switch (expr) {
case 1: { // 用 {} 将 `x` 限定在内层作用域中。
int x = 42;
do_something(x, expr);
}
case 2:
```

```
printf("%d\n", x); // Error: `x` was not declared in this scope.
}
/*
如果 expr == 2 ，控制流根本就没有经过 int x = 42; 这条语句，但是根据名字查找的
规则却能找到 x 。
为了解决这个问题，如果在某个 case 内部声明了变量，这个变量必须存在于一个内层作用域中。
简单来说就是要加 {} */
```

switch的缺点: 判断的时候只能是整型或者字符型, 不可以是一个区间

switch的优点: 结构清晰, 执行效率高

注意, case里面的break一定是要加上去的! 不然的话程序会一直执行下去!

- Starting from the selected label, **all subsequent statements are executed until a `break;` or the end of the `switch` statement is reached.**
- Note that `break;` here has a special meaning.
- If no `case` label is selected and `default:` is present, the control goes to the `default:` label.
- `default:` is optional, and often appears in the end, though not necessarily.
- `break;` is often needed. Modern compilers often warn against a missing `break;`
- The expression in a `case` label must be an integer *constant expression*, whose value is known at compile-time

```
int n; scanf("%d", &n);
int x = 42;
switch (value) {
  case 3.14: // Error: It must have an integer type.
    printf("It is pi.\n");
  case n:    // Error: It must be a constant expression (known
at compile-time)
    printf("It is equal to n.\n");
  case 42:   // OK.
    printf("It is equal to 42.\n");
  case x:    // Error: `x` is a variable, not treated as
"constant expression".
    printf("It is equal to x.\n");
}
```

案例

统计输入中每个值连续出现了多少次

```
# include <iostream>
using namespace std;
int main()
{
    int currval = 0, val = 0;
    if (cin >> currval)
    {
        int cnt = 1; // 保存我们正在处理的当前值的个数
        while(cin >> val)
        {
            if (val == currval)
            {
                cnt ++;
            }
            else
            {
                cout << currval << " occurs" << cnt << " times"
<< endl;
                currval = val;
                cnt = 1;
            }
        }
```

```
        // 记得打印文件中最后一个值的个数
        cout << currval << " occurs " << cnt << " times" <<
endl;
    }
    return 0;
}
```

两个亮点:

1. 输入流的灵活使用: 输入一次cin, 然后判断只会用一个数据, 然后while会灵活读取后面的数据
2. 最后一个数字可能只出现一次, 这种情况如何灵活处理

while循环

基本while

```
# include <iostream>
using namespace std;
int main()
{
    int num = 1;
    while (num < 10)
    {
        cout << num << endl;
        num++;
    }
    // print numbers 1 - 9
    return 0;
}
```

小例子: 假如说我想读取数量不定的输入数据呢?

`while(cin >> )`这个用法可以帮忙读取数量不定的数据

```
# include <iostream>
using namespace std;
int main()
{
    int sum = 0, value = 0;
    while (cin >> value)
    {
        sum += value;
    }
    cout << "The sum is: " << sum << endl;
    return 0;
}
```

假如说输入了 1 2 3 4, 那么这会是正确的, 因为我输入的其实是一个"流", 流里面的空格不是错误的

如果说输入1 2a3 4, 那么立刻输出3, 因为遇到了无效字符

案例: 猜数字

案例综合练习:

系统随机生成一个1-100的数字, 玩家进行猜测, 如果猜错的话, 提醒玩家数字过大或者是过小, 如果猜对的话玩家胜利, 并且退出游戏

```
# include <iostream>
using namespace std;
int main()
{
    int num = rand()%100+1;  //rand()%100生成的范围是0-99, 很像
python
    int guess = 0;
    int times = 0;
    while (guess != num)
    {
        times++;
        cout << "Please input a number" << endl;
        cin >> guess;
        if (guess > num)
        {
            cout << "Your number is too big! Try again!" <<
endl;
```

```
        }
        else if (guess < num)
        {
            cout << "Your number is too small! Try again!" <<
endl;
        }
        else
        {
            cout << "Your number is correct! Congrats!" << endl;
        }
    }
    cout << "The game is over, you used " << times <<" rounds to
finish the game" << endl;
    return 0;
}
```

do while

do{循环语句}while(判断条件);

首先要注意的是, 最后while括号后面是有分号的!

其次, 这个语法的特殊之处在于: 先执行一次循环语句, 再进行判断; 判断TRUE则循环执行

do while 也有自己的作用域!

```
do - while 的循环体是一个内层的作用域，它以 { 开始、 } 结束，不包含 while
(cond) 的部分。
在 do - while 循环体内声明的变量，无法在 cond 部分使用。
do {
int x;
scanf("%d", &x);
do_something(x);
} while (x != 0); // Error: `x` undeclared.
```

案例: 水仙花数

水仙花数是指一个三位数, 它的每个位上的数字的三次幂之和等于本身

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

```
int main()
{
    int num = 99;
    int num1 = 0;
    int num2 = 0;
    int num3 = 0;
    do
    {
        num++;
        num1 = num / 100;
        num2 = num / 10 % 10;
        num3 = num % 10;
        if (num1*num1*num1 + num2*num2*num2 + num3*num3*num3 ==
num)
        {
            cout << num << " is a number wanted" << endl;
        }
    }
    while (num < 999);
    return 0;
} // 找到 153 370 371 407
```

for循环语句

for(启示表达式; 条件表达式; 末尾循环体){循环语句}

末尾循环体: 循环语句结束一次之后运行的表达式

```
# include <iostream>
using namespace std;
int main()
{
    for (int i = 0; i < 10; i++)
    {
        cout << i << endl;
    }
    return 0;
}
```

当然, 上面这个代码完全等效于:

```
# include <iostream>
using namespace std;
int main()
{
    int i = 0;
    for (;;)
    {
        if (i >= 10)
        {
            break; // like python, 用于退出for循环
        }
        cout << i << endl;
        i++;
    }
    return 0;
}
```

当然, 要避免死循环, 例如:

`for(undesigned u = 10; u >= 0; u--)`,这个循环永远成立, 因为undesigned不可能是非正数

案例: 敲桌子

```
# include <iostream>
using namespace std;
int main()
{
    for (int i = 0; i < 100 ; i++)
    {
        if (i%7 == 0 || i%10 == 7 || i/10 == 7)
        {
            cout << "Knock the table" << endl;
        }
        else
        {
            cout << i << endl;
        }
    }
    return 0;
}
```

嵌套循环 案例: 乘法口诀表的打印

```
# include <iostream>
using namespace std;
int main()
{
    for (int i = 1; i < 10 ; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            cout << i << " * " << j << " = " << i*j <<  endl;
        }
    }
    return 0;
}
```

跳转语句

break使用的时机:

1. 出现在switch条件语句中, 作用是终止case并且跳出switch
2. 出现在循环语句中, 作用是跳出当前的循环语句
3. 出现在嵌套循环中, 跳出最近的内层循环语句

continue:

跳过本次循环中余下的未执行的语句, 继续执行下一次循环

goto语句:

语法: goto标记(注意冒号的使用)

如果标记的名称存在, 执行到goto语句时, 会跳转到标记的位置

```
goto flag:
...
flag:
...
```

但是程序中不建议使用goto, 避免造成混乱.

数组

概述

所谓数组, 就是一个集合, 淋面放了相同类型的数据元素

特点1: 数组中的每个元素都是相同的数据类型

特点2: 数组是由连续的内存位置组成的

一维数组

一维数组定义的三种方式:

数据类型 数组名[数组长度];

数据类型 数组名[数组长度] = { 值1, 值2..... };

数据类型 数组名[ ] = { 值1, 值2 };(可以不用说长度)

```
int main()
{
    int arr[5];
    arr[0] = 10;//给数组中的元素进行赋值
    cout << arr[0] << endl;//访问数据类型
    //注意: 如果访问没有赋值的数据, 会自动输出0
}
```

一维数组数组名的用途:

可以统计整个数组在内存中的长度; 可以获取数组在内存中的首地址

那么如何获得这两个数据呢?

第一个是sizeof(arr); 第二个是cout << (int)arr << endl;

另外, sizeof(ar/r)/sizeof(arr[0])就可以获得数组数据的个数

另外, 想看第一个元素的内存地址: cout << (int)&arr[0] << endl;

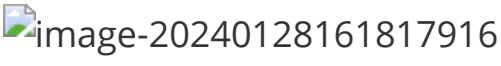
另外: 数组名是常量, 不可以进行复制操作

注意: 创建数组的时候, 数组成员数量不能通过传入变量来确定!

```
int n = 5;
int arr[n] = {1,2,3,4,5};
// 不可以，不能用变量
```

一位数组案例

```
# include <iostream>
using namespace std;
int main()
{
    int arr[5] = {300, 350, 200, 400, 250};
    int max = 0;
    for(int i = 0; i < 5; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
        else // 这一部分其实可以不要
        {
            continue;
        }
    }
    cout << max << endl;
    return 0;
}
```



```
# include <iostream>
using namespace std;
int main()
{
    int arr[5] = {1,3,2,5,4};
    int start = 0;
    int end = 0;
    int temp = 0;
```

```
    for(; start < sizeof(arr)/2+1; start++)
    {
        end = sizeof(arr)/sizeof(arr[0])-1;
        temp = arr[end];
        arr[end] = arr[start];
        arr[start] = temp;
    }
    for(int i = 0; i<sizeof(arr); i++)
    {
        cout << arr[i] << endl;
    }
    return 0;
}
```

上面这个代码有点小问题, 而且非常不方便

start & end两个指针停止下来的判断能不能简单一些?

```
# include <iostream>
using namespace std;
int main()
{
    int arr[5] = {1,3,2,5,4};
    int start = 0;
    int end = sizeof(arr)/sizeof(arr[0]) - 1;
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
    for(int i=0; i<sizeof(arr)/sizeof(arr[0]); i++)
    {
        cout << arr[i] << endl;
    }
    return 0;
}
```

一维数组的冒泡排序

作用: 最常见的排序算法, 对数组内元素进行排序

1. 比较相邻的元素, 如果第一个比第二个大, 那么就进行交换
2. 对每一对相邻的元素做同样的工作, 执行完毕之后, 找到第一个最大值
3. 重复以上操作, 每次比较次数减一, 直到不需要比较

实例: 将数组{4,2,8,0,5,7,1,3,9}进行升序排序

```
# include <iostream>
using namespace std;
int main()
{
    int arr[9] = {4,2,8,0,5,7,1,3,9};
    for (int i=8; i>0; i--)
    {
        for(int j=0; j<i; j++)
        {
            if (arr[j] > arr[j+1])
            {
                int temp = arr[j+1];
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
        }
    }
    for (int k = 0; k<9; k++)
    {
        cout << arr[k] << endl;
    }
    return 0;
}
```

二维数组

二维数组的定义方式

```
/*
1. 数据类型 数组名[行数][列数];
2. 数据类型 数组名[行数][列数] =
{
    {1,2}
    {3,4}
};
3. 数据类型 数组名[行数][列数] = {1,2,3,4};
(花括号里面的数据个数应该正好是行数乘以列数)
4. 数据类型 数组名[ ][列数] = {1,2,3,4};
(为什么行数不用写? 因为程序会自己算!)
*/
```

二维数组组名

1. 查看二维数组所占内存　cout << sizeof(arr) <<endl;　　cout << sizeof(arr[0]) << endl;(第一行所占内存)
2. 获取二维数组首地址　　cout << (int)arr << endl; (第几行的地址, 哪一个元素的地址都可以看)

利用内存, 可以计算行数和列数

例如: sizeof(arr) / sizeof(arr[0])

二维数组指针

在C语言中，`int **arr` 表示一个指向指针的指针，通常用于表示指向二维数组的指针。

考虑一个二维数组 `int arr[3][4]`，我们知道数组名 `arr` 在大多数情况下会被解释为指向数组第一个元素的指针，即 `int (*arr)[4]`。如果我们想要通过指针访问这个二维数组，可以使用 `int (*p)[4] = arr;` 这样的方式。

但是，如果我们想要使用一个指针来管理这个二维数组，可以使用指向指针的指针。`int **arr` 就是这样的一个指针，它指向一个指针数组，每个指针指向一维数组的第一个元素

`int **(array+1)` 实际上是将指针 `array` 向后移动了一个单位（`sizeof(int*)`），然后取得该位置的指针的地址。在大多数情况下，`array` 是一个指向指针的指针，比如 `int **array`，那么 `array+1` 就是指向 `array` 下一个指针的位置。

如果 `array` 是一个指向指针的指针数组的话，`int **(array+1)` 就指向了数组中的第二个指针，即 `array[1]`。这种操作通常用于遍历指针数组或者二维数组的行。

The following declarations are equivalent: The parameter is of type `int (*)[N]`, which is a pointer to `int[N]`.

```
void fun(int (*a)[N]);
void fun(int a[][N]);
void fun(int a[2][N]);
void fun(int a[10][N]);
```

We can pass an array of type `int[K][N]` to `fun`, where `K` is arbitrary.

- The size for the second dimension must be N
    - `T[10]` and `T[20]` are different types, so the pointer types `T(*)[10]` and `T(*)[20]` are not compatible.

In each of the following declarations, what is the type of `a`? Does it accept an argument of type `int[N][M]`?

1. `void fun(int a[N][M])`: A pointer to `int[M]`. Yes.
2. `void fun(int (*a)[M])`: Same as 1.
3. `void fun(int (*a)[N])`: A pointer to `int[N]`. **Yes iff** `N == M`**.**
4. `void fun(int **a)`: A pointer to `int *`. **No.**
5. `void fun(int *a[])`: Same as 4.
6. `void fun(int *a[N])`: Same as 4.
7. `void fun(int a[100][M])`: Same as 1.
8. `void fun(int a[N][100])`: A pointer to `int[100]`. Yes iff `M == 100`.

```c
void print(int (*a)[5], int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 5; ++j)
      printf("%d ", a[i][j]);
    printf("\n");
  }
}
int main(void) {
  int a[2][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
  int b[3][5] = {0};
  print(a, 2); // OK
  print(b, 3); // OK
}
```

数组与指针

A pointer to an array of `N` `int`s:

```c
int (*parr)[N];
```

An array of `N` pointers (pointing to `int`):

```c
int *arrp[N];
```

Too confusing! How can I remember them?

- `int (*parr)[N]`

  has a pair of parentheses around * and parr, so

  - `parr` is a pointer (`*`), and
  - points to something of type `int[N]`.
- Then the other one is different:
  - `arrp` is an array, and
  - stores `N` pointers, with pointee type `int`.

函数

作用: 讲一段经常使用的代码封装起来, 减少重复代码

一个较大的程序, 一般分为若干个程序块, 每个模块实现特定的功能

函数的定义与调用

一般有五个步骤

1. 返回值类型
2. 函数名
3. 参数列表
4. 函数体语句
5. return 表达式

```
返回值类型 函数名(参数列表)
{
    函数体语句
    return 表达式
}
```

例如: 实现一个加法函数, 传入两个整型数据, 计算相加的数据结果, 并且返回它

```
# include <iostream>
using namespace std;
int add(short num1, short num2) // 建议多用short, 更何况short和int是等效的, 但是short更能节省内存
{  // 括号里面的是形式参数
    int sum = num1 + num2;
    return sum;
}
int main()
{
    short a = 5;
    short b = 10;
    cout << add(a, b) <<endl;     // 这里的a, b是实参(实际参数)
    return 0;
}
```

在上面的代码中, add(a ,b)就是实现了函数的调用

注意: 如果函数不需要返回值的话, 那么声明返回值类型的地方可以写一个void, 而return后面直接加分号

```
# include <iostream>
using namespace std;
void tell_add(short num1, short num2)
{
    int sum = num1 + num2;
    cout << "The sum of these two numbers is " << sum <<endl;
    return;
}
int main()
{
    short a = 5;
    short b = 10;
    tell_add(a, b);
    return 0;
}
```

值得注意的是: 当我们做值传递的时候, 函数的形参会发生改变, 但是并不会影响实参!!

注意：编译器没有那么聪明，不能”满足一定条件才返回值“，因为编译器总会认为”有不满足条件的时候“

```
int abs_int(int x) {
  if (x < 0)
    return -x;
  else if (x == 0)
    return 0;
  else if (x > 0)
    return x;
}
```

```
a.c: In function abs_int:
a.c:8:1: warning: control reaches end of non-void function [-
Wreturn-type]
    8 | }
```

函数的常见样式

常见的函数样式有四种:

无参无返 有参无返 无参有反 有参有返

无返回值就是返回值类型改成void并且return 直接加上分号

无参数就是括号里面什么也不加

函数的声明

作用: 告诉编译器函数名称及如何调用函数, 函数的实际主题可以单独定义

函数的声明可以有多次, 但是函数的定义只能有一次!

```
# include <iostream>
using namespace std;
// 接下来函数的定义是什么
int max(short a, short b)
{
    return (a > b ? a : b);
} // 注意灵活使用三目运算符
int main()
{
    cout << max(15, 16) <<endl;
    return 0;
}
```

上面这个函数没有声明. 而且注意到, 我们总是把自定义的函数写在main函数前面

那么如果max函数定义在main的后面呢? 这将是不合法的. 为什么呢? 编译器会直接告诉你: 我不知道max是什么, 因为**代码是一行一行执行的!**

但是有了声明, 就可以提前告诉程序函数的存在

```cpp
# include <iostream>
using namespace std;
// 接下来函数的定义是什么
int max(short a, short b);// 注意分号
int main()
{
    cout << max(15, 16) <<endl;
    return 0;
}
int max(short a, short b)
{
    return (a > b ? a : b);
} // 注意灵活使用三目运算符
```

上面这个程序之中, max的函数定义在了后面. 当然, 如果max依然是在前面进行了定义, 有了声明也是没关系的

函数的分文件编写

作用: 让代码更加清晰

函数分文件编写一般有四个步骤

1. 创建一个后缀名为.h的头文件
2. 创建后缀名为.cpp的源文件
3. 在头文件中写函数的声明
4. 在源文件中写函数的定义

```cpp
# include <iostream>
# include "test_func.h"
using namespace std;
int main()
{
    cout << max(15, 16) <<endl;
    return 0;
}  // main_program.cpp
```

```cpp
// test_func.h
# include <iostream>
using namespace std;
int max(short a, short b);// 注意分号, 这里是函数声明
```

```cpp
// test_func.cpp
# include "test_func.h"
int max(short a, short b)
{
    return (a > b ? a : b);
} // 注意灵活使用三目运算符
```

注意每个文件配套的东西!

头文件统领对应的cpp文件, 因此头文件里面要有, using namespace std等等配套东西

而对应的cpp文件, include "头文件名"即可

而在调用这个函数的主文件中, 要include头文件; 注意使用的是双引号, 因为这个是自己定义的头文件

指针

指针基本概念与使用

指针的作用: 可以通过指针间接访问内存

1. 内存编号是从0开始记录的, 一般用16进制表示
2. 可以利用指针变量保存地址

假设 int a = 10, 编号为0x0000的内存记录了10这个数据

那么可不可以再开一个变量来保存这个地址呢? 当然是可以的, 而这个变量就叫做指针变量; 换而言之, 指针就是一个地址, 可以通过指针来保存一个地址

那么如何创建和使用呢?

```cpp
# include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int * p; // 语法: 数据类型 * 指针变量名
    p = &a; // 建立联系, 让指针记录a的地址
    cout << "The address of a is: " << p << endl;
    *p = 1000;// 这个称为解引用, *p将代表a
    cout << a << endl; // 将会输出1000
    return 0;
}
```

一定要注意的是: **指针类型必须与所指对象的类型一致!!**

```cpp
const double pi = 3.14;
double *ptr = &pi; // 错误, ptr只是一个普通的指针, 类型不一致
const double *cptr = &pi; // 正确, const使得类型一致
*cptr = 42; // 错误, 不能赋值
```

指针所占的内存空间

指针也是一种数据类型, 那么这种数据类型占用多少内存空间? 在32位系统中, 地址只占用4个字节; 在64位下, 占8个字节. 所以占用内存大小与操作系统有关, 而不与数据类型有关

空指针

空指针定义: 指针变量指向内存中编号为0的空间

用途: 初始化指针变量

注意: 空指针指向的内存是不可以访问的

(因为0-255的内存是系统占用的, 访问的话会出错)

```
int main(){
    int * p = NULL;
    *p = 100;//直接报错
    cout << *p << endl; //直接报错
}
```

```
int *ptr = NULL;
printf("%d\n", *ptr); // undefined behavior
*ptr = 42; // undefined behavior
```

那么有的时候我也不确定这个指针到底是不是空指针，但是我又想访问试一试看看是不是空指针，那么应该让如何实现呢?

```
if (ptr != NULL && *ptr == 42){}
// 这样的话就会先判断是不是空指针；如果是，那么这个判断直接就是False，不会触发后面的解引用
```

野指针

野指针: 指针变量指向的是非法的内存空间 （不指向任何目标，但是并不是空指针）

不是说什么内存空间我都能直接指针指向!!!

总结: 空指针和野指针都不是我们申请的空间, 因此不能访问

const修饰指针

const修饰指针有三种情况:

1. const修饰指针 --- 常量指针
2. const修饰常量 --- 指针常量
3. const即修饰指针, 又修饰常量

第一种情况: const修饰指针 --- 常量指针

```
int a = 10;
short b = 10;
const int * p = &a;    //这就是常量指针
//特点: 指针指向可以修改, 但是指针指向的值不可以改
*p = 20; // 非法
p = &b;  // 合法
```

第二种情况: const修饰常量 --- 指针常量

```
int a = 10;
short b = 10;
int * const p = &a;    //这就是常量指针
//特点: 指针指向不可以修改, 但是指针指向的值可以改
*p = 20; // 合法
p = &b;  // 非法
```

第三种情况: 修饰指针与常量

```
int a = 10;
short b = 10;
const int * const p = &a;    //这就是常量指针
//特点: 指针指向不可以修改, 指针指向的值不可以改
*p = 20; // 非法
p = &b;  // 非法
```

也称: 顶层const表示指针本身是个常量, 底层const表示指针所指的对象是一个常量

指针与数组

作用: 利用指针访问数组中的元素

```
# include <iostream>
using namespace std;
int main(){
    int arr[10] = {0,1,2,3,4,5,6,7,8,9};
    cout << "The first element is " << arr[0] << endl;
    int * p = arr; // 数组首地址
    cout << "The first element is " << *p << endl;
    for(int i = 0; i<10; i++)
    {
        cout << *p << endl;
```

```
        p++;// 因为p是int类型, 因此是四个字节
        // ++就自动代表内存地址往后面移动四个字节, 而arr中每个整型数据都正好是四个字节
    }
    return 0;
}
```

指针与函数

作用: 利用指针作函数参数, 可以修改实参的值

之前我们提到: 函数并不能对实参进行改变, 但是如果输入的参数是指针, 那么函数可以通过对地址的一系列参数, 从而操控参数背后所对应的数据, 从而实现对实参的改变

```
//实现两个数值的交换
# include <iostream>
using namespace std;
void swap(int * p1,int * p2) //地址作为传参
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
    return;
}
void swap(int * p1,int * p2);//声明
int main()
{
    int a = 10;
    int b = 20;
    swap(a, b);
    cout << a << endl;
    cout << b << endl;
    return 0;
}
```

总而言之, 只要形参的数据类型额外声称int *,那么传参就可以改变, 因为设置了会对内存进行改动, 而且函数里面照常像你不知道指针一样进行编写即可

综合案例: 指针 数组 函数

案例描述: 设计一个函数, 利用冒泡排序, 实现对整数型数组的升序排序

```
# include <iostream>
using namespace std;
void bubblesort(int * arr, int len)
{ // 设置: 可以对传参进行改变
    for (int i = 0; i < len; i++)
    {
        for (int j = 0; j < len-i-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
} // int * arr发生了指针退化，直接就传入了一个可以改变的数组
void printArray(int * arr, int len)
{
    for(int i = 0; i<len; i++)
    {
        cout << arr[i] << endl;
    }
}
int main()
{
    int arr[10] = {4,3,9,6,1,2,10,8,7,5};
    int len = sizeof(arr) / sizeof(arr[0]);
    bubblesort(arr, len);
    printArray(arr, len);
    return 0;
}
```

MALLOC AND FREE AND CALLOC

Declared in `<stdlib.h>`.

```
void *malloc(size_t size);
```

Allocates `size` bytes of uninitialized storage on heap.

If allocation succeeds, returns the starting address of the allocated memory block.

If allocation fails, a null pointer is returned.

- `size_t`

  : A type that can hold the size (number of bytes) of any object. It is

  - declared in `<stddef.h>`, and
  - is an **unsigned** integer type,
  - whose size is implementation-defined. For example, it may be 64-bit on a 64-bit machine, and 32-bit on a 32-bit machine.

```
T *ptr = malloc(sizeof(T) * n); // sizeof(T) * n bytes
for (int i = 0; i != n; ++i)
  ptr[i] = /* ... */
// Now you can use `ptr` as if it points to an array of `n`
objects of type `T`
// ...
free(ptr);
```

To avoid **memory leaks**, the starting address of that block memory must be passed to `free` when the memory is not used anymore.

Declared in `<stdlib.h>`.

```
void free(void *ptr);
```

Deallocates the space previously allocated by an allocation function (such as `malloc`).

If `ptr` is a null pointer, this function does nothing.

- There is no need to do a null check before calling `free`!

The behavior is undefined if `ptr` is not equal to an address previously returned by an allocation function.

- In other words, "double `free`" is undefined behavior (and often causes severe runtime errors).

After `free(ptr)`, `ptr` no longer points to an existing object, so it is no longer dereferenceable.

- Often called a "dangling pointer".

We can also create one single object dynamically (on heap):

```
int *ptr = malloc(sizeof(int));
*ptr = 42;
printf("%d\n", *ptr);
// ...
free(ptr);
```

But why? Why not just create one normal variable like `int ival = 42;`这就和内存四区有关了

这种方法能够把数据放在堆区，然后手动释放：

Benefit: The lifetime of a dynamically allocated object goes beyond a local scope.

It is not destroyed until we `free` it.

```
int *create_array(void) {
  int a[N];
  return a; // Returns the address of the local object `a`.
            // When the function returns, `a` will be destroyed,
so that
            // the returned address becomes invalid.
            // Dereferencing the returned address is undefined
behavior.
}
int *create_dynamic_array(int n) {
  return malloc(sizeof(int) * n); // OK. The allocated memory is
valid until
                                  // we free it.
}
```

Create a "2-d array" on heap?

```
int **p = malloc(sizeof(int *) * n);
for (int i = 0; i < n; ++i)
  p[i] = malloc(sizeof(int) * m);
for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j)
    p[i][j] = /* ... */
// ...
for (int i = 0; i < n; ++i)
  free(p[i]);
free(p);
```

Create a "2-d array" on heap? - Another way: Create a 1-d array of length `n * m`.

```
int *p = malloc(sizeof(int) * n * m);
for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j)
    p[i * m + j] = /* ... */ // This is the (i, j)-th entry.
// ...
free(p);
```

calloc:

Declared in `<stdlib.h>`

```
void *calloc(size_t num, size_t each_size)
```

Allocates memory for an array of `num` objects (each of size `each_size`), and initializes all bytes in the allocated storage to zero 1.

Similar as `malloc(num * each_size)`. 2 Returns a null pointer on failure.

The behaviors of `malloc(0)`, `calloc(0, N)` and `calloc(N, 0)` are **implementation-defined**:

- They may or may not allocate memory.
- If no memory is allocated, a null pointer is returned.
- They may allocate some memory, for some reasons. In that case, the address of the allocated memory is returned.
  - You cannot dereference the returned pointer.
  - It still constitutes **memory leak** if such memory is not `free`d.

Arrays vs `malloc`

- An array has limited lifetime (unless it is global or `static`). It is destroyed when control reaches the end of its scope.
- Objects allocated by `malloc` are not destroyed until their address is passed to `free`.
- The program crashes if the size of an array is too large (running out of stack memory). There is no way of recovery.
- Attempt to `malloc` a block of memory that is too large results in a null pointer. We can know if there is no enough heap memory by doing a null check.

  ```
  int *ptr = malloc(1ull << 60); // unrealistic size
  if (!ptr)
    report_an_error("Out of memory.");
  ```

结构体

基本概念: 属于用户自定义的数据类型, 允许用户储存不同的数据类型

结构体的定义与使用

struct 结构体名 {结构体成员列表}

通过结构体创建变量一般有三种:

1. struct 结构体名 变量名
2. struct 结构体名 变量名 = {成员1, 成员2,.....}
3. 定义结构体时顺便创建变量

```
# include <iostream>
# include <string>
using namespace std;
// 创建的第一种方法
struct Student
{
    string name;
    int age;
    int score;
}; // 注意分号
int main()
{
    struct Student s1;
    s1.name = "Bear";
    s1.age = 18;
    s1.score = 100;
    cout << "name: " << s1.name << endl;
}
```

```
# include <iostream>
# include <string>
using namespace std;
// 创建的第二种方法
struct Student
{
    string name;
    int age;
    int score;
}; // 注意分号
int main()
{
```

```
    struct Student s2 = {"Bear", 18, 100}
}
```

```
# include <iostream>
# include <string>
using namespace std;
// 创建的第一种方法
struct Student
{
    string name;
    int age;
    int score;
}s3; // 顺便创建, 后面可以像第一种方法一样给它赋值
```

注意: C++中, 定义结构体必须要有struct, 但是在创建一个实际结构体变量的时候, 可以省略

结构体数组

作用: 将自定义的结构体放入到数组中方便维护

语法: struct 结构体名 数组名[元素个数] = {{},{},.....}

```
struct student
{
    string name;
    int age;
    long score;
};
int main()
{
    struct student stuArray[3] =
    {
    {"Van", 20, 114514}
    {"Bear", 18, 100}
    {"Dark", 20, 1919810}
    }
}
```

说明白了, 就是一个数组, 里面存放了几个结构体变量

结构体指针

作用: 通过指针访问结构体中的成员

利用操作符->可以通过结构体指针访问结构体属性

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    int score;
};
int main()
{
    struct student s = {"Bear", 18, 100};
    //通过指针指向结构体变量
    //int * p = &s;  这会直接报错!!不兼容
    struct student * p = &s;
    // 上述两个struct可以省略
    cout << p->name << endl;
    return 0;
}
```

结构体嵌套结构体

作用: 结构体的成员可以是另一个结构体

例如: 每个老师福岛一个学员, 一个老师的结构体重, 还有学生的结构体

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    int score;
};
```

```
struct teacher
{
    int id;
    string name;
    int age;
    struct student stu;
};

int main()
{
    student s =  {"Bear", 18, 100};
    struct teacher tea = {123, "Lily", 25, s};
    cout << tea.stu.score<< endl;
    return 0;
}
```

注意:

1. 不能直接cout tea.stu, 因为不能输出结构体
2. 注意: struct student一定要在struct teacher之前

结构体做函数参数

作用: 将结构体作为参数向函数中传递

传递的方式有两种: 值传递, 和地址传递

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    int score;
};
void printinfo(student s)
{
    cout << "The name of the student: ";
    cout << s.name << endl;
    cout << "The age of the student: ";
```

```
    cout << s.age << endl;
    cout << "The score of the student: " ;
    cout << s.score << endl;
    return;
}
int main()
{
    struct student s = {"Bear", 18, 100};
    printinfo(s);
    return 0;
}
```

上述为值传递

当然, 如果想改动传入的结构体中的参数的话, 可以使用地址传递

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    int score;
};
void printinfo(struct student * s)
{
    cout << "The name of the student: ";
    cout << s->name << endl;
    cout << "The age of the student: ";
    cout << s->age << endl;
    cout << "The score of the student: " ;
    cout << s->score << endl;
    return;
}
void lao_ren(struct student * s)
{
    s->score += 20;
    return;
}
int main()
{
```

```
    struct student s = {"Bear", 18, 100};
    struct student * p = &s; // 创建地址
    printinfo(p);//或者直接传入&s
    lao_ren(p);
    printinfo(p);
    return 0;
}
/*
The name of the student: Bear
The age of the student: 18
The score of the student: 100
The name of the student: Bear
The age of the student: 18
The score of the student: 120
*/
```

注意指针来访问数据, 需要->

结构体中const使用场景

作用: 利用const防止误操作

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    int score;
};
void printinfo(student s)
{
    cout << "The name of the student: ";
    cout << s.name << endl;
    cout << "The age of the student: ";
    cout << s.age << endl;
    cout << "The score of the student: " ;
    cout << s.score << endl;
    return;
}
```

```
int main()
{
    student s = {"Bear", 18, 100};

}
```

上面是一个值传递, 可以打印学生信息, 而且可以保证不会对数据进行修改. 这是一件挺好的事情, 但是问题在于: 传入的和所使用的的是同一份数据吗? 可想而知, 并不是. 因此如果成千上万个学生的信息都打印一遍, 复制出来的内容所占的空间越多! 但是如果传入的是指针, 只会占用四个内存! 你把地址传给我, 我就能进行操作! 但是如果我写代码有问题, 如果传入的是参数, 会误操作数据, 怎么办?

直接: `void printinfo(const student *s)// 地址传入, 而且是常量指针`

一旦加入了const, 如果我可能会对数据进行修改, 就会立马报错!!!

为什么是在前面加上const呢?因为这样使得指针变成了常量指针, 指针所指向的可以改变, 但是指针所对应的数据不能改变!

综合案例:

有一个结构体, 然后想要利用冒泡排序, 按照年龄进行升序排序, 最后打印排序之后的结果, 要求利用结构体数组

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    string sex;
};
void printinfo(struct student array[]);
void bubblesort_printinfo(struct student stuarray[])
{
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < 5-i-1; j++)
        {
            if (stuarray[j].age > stuarray[j+1].age)
```

```
            {
                struct student temp = stuarray[j+1];
                stuarray[j+1] = stuarray[j];
                stuarray[j] = temp;
            }
        }
    }
    printinfo(stuarray);
    return;
}
void printinfo(struct student array[])
{
    for(int i = 0; i < 4; i++)
    {
        cout << "The name of the student: " << array[i].name <<
endl;
        cout << "The age of the student: " << array[i].age <<
endl;
        cout << "The sex of the student: " << array[i].sex <<
endl;
    }
    return;
}
int main()
{
    struct student stuarray[5] = {
        {
            "Lily", 23, "girl"
        },
        {
            "Jack", 21, "boy"
        },
        {
            "John", 20, "boy"
        },
        {
            "Van", 21, "boy"
        },
        {
            "Lucy", 19, "girl"
        }
```

```
    };
    bubblesort_printinfo(stuarray);
    printinfo(stuarray);
    return 0;
}
```

```
# include <iostream>
# include <string>
using namespace std;
struct student
{
    string name;
    int age;
    string sex;
};
void printinfo(struct student array[]); // 声明
void bubblesort(struct student stuarray[]) // 注意这个语法，其实实质上就是地址传递!
{
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < 5-i-1; j++)
        {
            if (stuarray[j].age > stuarray[j+1].age)
            {
                struct student temp = stuarray[j+1];
                stuarray[j+1] = stuarray[j];
                stuarray[j] = temp;
            }
        }
    }
    return;
}
void printinfo(struct student array[]) // 注意这个语法，其实实质上就是地址传递!
{
    for(int i = 0; i < 5; i++)
    {
        cout << "The name of the student: " << array[i].name << endl;
```

```
        cout << "The age of the student: " << array[i].age <<
endl;
        cout << "The sex of the student: " << array[i].sex <<
endl;
    }
    return;
}
int main()
{
    struct student stuarray[5] = {
        {
            "Lily", 23, "girl"
        },
        {
            "Jack", 21, "boy"
        },
        {
            "John", 20, "boy"
        },
        {
            "Van", 22, "boy"
        },
        {
            "Lucy", 19, "girl"
        }
    };
    bubblesort(stuarray);
    printinfo(stuarray);
    return 0;
}
```

注意点:

1. 传入array的时候, 后面加入[], 代表你传入的是student stuarray**的地址**
2. 实际上，在 C++ 中，当你将数组传递给函数时，传递的是数组的地址。这是因为数组在函数调用中会退化为指针。

   当你声明一个函数时，形如 `void myFunction(int myArray[])` 或 `void myFunction(int* myArray)`，这两种形式在实际使用中是等价的，都是传递了数组的地址（指针）。

所以，没有专门定义数组地址传递的语法，而是使用数组名作为函数参数，实际上传递的是数组的地址。

3. 不知道为什么bubblesort是"值传递", 但是最后却改变了结构体数组. 为什么? 因为事实上, 在C++中, 把数组名传递给函数的实质就是把数组地址传给了函数, 因此函数的相关定义就是"地址传递", 因此本来就是可以改变数组的

C++自学——核心编程

本阶段, 主要针对C++面向对象编程技术做详细讲解, 探讨C++中的核心和精髓

内存四区

C++程序在执行的时候, 将内存大方向分为四个区域:

1. 代码区: 存放函数体的**二进制代码**, 由操作系统进行管理
2. 全局区: 存放**全局变量**和**静态变量**以及**常量**
3. 栈区: 由编译器自动分配释放, 存放函数的参数值, 局部变量等
4. 堆区:由程序员分配和释放, 若程序员不释放, 程序结束的时候由操作系统回收

存在的意义: **不同区域存放的数据, 赋予不同的生命周期, 给我们更大的灵活编程**

程序运行前

程序编译之后, 生成了exe可执行程序, 未执行该程序前分为两个区域:

代码区:

存放CPU执行的机器指令

代码区是共享的, 共享的目的是对于频繁被执行的程序, 只需要在内存中有一份即可

代码区是只读的, 使其只读的原因是防止程序意外地修改了它的指令

全局区:

全局变量和静态变量存放在此

全局区还包括了常量区, 字符串常量和其他常量(**const修饰的变量**)

该区域的数据在程序结束后由操作系统释放

附: 写在函数体中的变量都是局部变量; 在函数体外面的, 是全局变量

```
# include <iostream>
using namespace std;
// 创建全局变量,g stands for global
int g_a = 10;
// const修饰的全局变量
```

```cpp
const int c_g_a = 10;
int main()
{
    // 创建普通局部变量
    int a = 10;
    // 创建静态变量, s stands for static
    static int s_a = 10;
    cout << "The address of a is: " << (long long)&a << endl;
    cout << "The address of g_a is: " << (long long)&g_a <<
endl;
    cout << "The address of a is: " << (long long)&s_a << endl;
    // 字符串常量
    cout << "The address of the string is: " << (long
long)&"Hello World" << endl;
    // const修饰的变量
    cout << "The address of c_g_a is: " << (long long)&c_g_a <<
endl;
    // const修饰的局部变量
    const int c_l_a = 10; // c: const    l: local
    cout << "The address of c_l_a is: " << (long long)&c_l_a <<
endl;
    return 0;
}
//The address of a is: 6487580
//The address of g_a is: 4206608
//The address of s_a is: 4206612
//The address of the string is: 4210799
//The address of c_g_a is: 4210692
//The address of c_l_a is: 6487576
```

注意: 上面我使用了long long, 因为我的系统是64位的, 指针占用了八个字节, 而int占用四个字节, 所以会lose precision

可见: 在全局区的有: 全局变量, 静态变量, 字符串常量, const修饰的全局常量

程序运行之后

栈区:

由编译器自动分配和释放, 存放函数的参数值, 局部变量等

注意事项: **不要返回局部变量的地址**, 栈区开辟的数据由编译器自动释放

这句话是什么意思呢?

```
# include <iostream>
using namespace std;
int * func() // 注意: 传入的形参也会放在栈区里面, 返回它的地址是不合理的
{
    int a = 10; // 局部变量, 存放在栈区, 栈区的数据在函数执行完后自动释放
    return &a;// 返回局部变量地址
}
int main()
{
    int * p = func();
    cout << *p << endl; // 第一次还可能打印这个数字(可能IDE直接给出警告), 因为编译器做了保留
    cout << *p << endl; // 第二次这个数据不会再保留
    return 0;
}
```

注意: 函数人为返回的是int* , 为什么不用long long 了?

因为那个long long事实上是强行把int * 转化为了long long *, 一开始这个&a确实是int *

只有要打印地址的时候, 才会需要(long long)强行转化

堆区:

由程序员分配释放, 若程序员不释放, 程序结束的时候由操作系统回收

在C++中主要利用*new*在堆区开辟一块内存

利用new创建的数据, 会返回该数据对应的类型的指针

```
# include <iostream>
using namespace std;
int * func()
{
    // 利用new关键词, 可以将数据开辟到堆区
    int * p = new int(10);
    // 指针本质也是局部变量, 放在栈上, 指针保存的是堆区的数据
    // 注意: new int(10)这个语句是有返回值的, 我们可以用指针来接受它
    return p;
}
```

```
int main()
{
    //在堆区开辟内存
    int * p = func(); // 用指针接受返回值
    cout << (long long)p << endl; // 注意long long 强行转化, 而且不同次运行程序, 这个地址都是不一样的
    cout << *p << endl; // 解引用
    cout << *p << endl; // 第二次依然能正常显示
    return 0;
}
```

实质: 函数返回的是栈区的数据, 这个数据会在函数调用完成之后销毁掉; 但是如果我让这个返回的栈区的数据是一个地址, 地址指向的地方是数据不会在函数结束掉用后被释放的地方, 那么在main函数里面用指针接受这个地址, 随即这个存放在栈区的、记录了一个堆区地址的数据被销毁, 但是这个数据被main里面的指针接受, 存放在栈区的内存里面(局部变量放在这里面), 而且这个数据在func()调用结束后不会释放(func()里面的局部变量会在func()调用完成之后释放)

那么有用new关键词开辟, 那么就有人工释放, 那么关键词就是***delete***

```
# include <iostream>
using namespace std;
int * func()
{
    int * p = new int(10);
    return p;
}
void test_01()
{
    int * p = func();
    cout << *p << endl;
    cout << *p << endl;
    delete p;
    cout << *p << endl; //到这里直接会报错, 内存已经被释放, 再次访问就是非法操作
    return;
}
// 接下来, 尝试用new在堆区开辟一个数组
int * test_02()
```

```
{
    // 创建一个10整型数据的数组
    int * arr = new int[10]; //注意，这里的arr是地址，同时也能担当数组本身，这个问题以前发现过
    for(int i = 0; i < 10; i++)
    {
        arr[i] = i+100;
    }
    return arr;
}
int main()
{
    // test_01();
    int * arr = test_02();
    for (int i = 0; i < 10; i++)
    {
        cout << arr[i] << endl; // 再次强调，arr既是地址，也担当数组本身
    }
    delete [] arr; // 释放数组，注意要加上一个中括号，告诉编译器我要释放的是整个数组
    // 因为这个数组的地址其实是第一个元素的地址，如果不告诉的话，那么只会释放第一个元素
    return 0;
}
```

注意:

1. arr既能充当地址, 也能充当数组本身
2. delete释放的时候一定要加中括号, 告诉编译器释放的是整个数组

引用

引用的基本使用

作用: 给变量起别名, 这样的话, 操纵一块内存的方式就会多一种(通过操作这个新名字)

语法: `数据类型 &别名 = 原名`

注意事项

1. 引用必须初始化
2. 引用在初始化后, 不可以改变

```
int a = 1;
int c = 11;
int &b; // 非法! 必须直接告诉是谁的别名, i.e., 必须直接初始化
int &b = a;
int &b = c; // 非法! 已经说明是a的别名了, 就不能再说是c的别名了

int &b = a;
b = c; // 合法! 这一步是把c的值付给了b(也就是a), 复制操作是可以的, 并不是更改引用
```

引用做函数参数

作用: 函数传参的时候, 可以利用引用的技术让形参修改实参

优点: 可以简化指针修改实参

一般来说, 函数传参有值传递和地址传递, 但是值传递是不能修改实参的; 以前为了修改实参, 我们使用了地址传递, 非常的麻烦, 但是如果可以使用引用的话, 这会简单很多

```
# include <iostream>
using namespace std;
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
} // 这个函数并不能真正交换实参
void swap_address(int*a, int*b) // 地址传参
{
    int temp = *a;
    *a = *b;
    *b = temp;
    return;
}
```

```
void swap_reference(int &a, int &b) // 给传入的两个参数起别名
{
    int temp = a;
    a = b;
    b = temp;
    return;
}
int main()
{
    int a = 10;
    int b = 20;
    swap_address(&a, &b); // 注意: 可以原名"等于"别名
    cout << a << " " << b << endl; // 20 10
    swap_reference(a,b);
    cout << a << " " << b << endl; // 10 20
    return 0;
}
```

实质: 地址传递和引用传递实质上都是提供了一个直接操控到内存数据的方式

引用做函数返回值

作用: 引用是可以作为函数的返回值存在的

注意: 不要返回局部变量引用

用法: 函数调用作为左值

```
# include <iostream>
using namespace std;
int& test_1() // 这个语法: 以引用的方式返回值
{
    int a = 10; // 存放在栈区
    return a; // IDE直接警告
}
int& test_2()
{
    static int a = 18; // 创建一个静态变量, 这将会放在全局区
    return a;
}
int main()
{
```

```
    int %ref = test_1();
    cout << ref << endl; // 如果IDE不警告，这一句话还可能返回正常的10
    cout << ref << endl; // 直接乱码，没有意义
    int &refer = test_2();
    cout << refer << endl; //这句将会正常输出18，因为static意味着静态变量，将会放在全局区
    cout << a << endl; // 这句话非法，我们在main里面不知道函数里面的静态变量是什么，只知道它有一个别名
    test_2() = 1000; // 函数返回的是引用，因此左式也相当于是一个操控a所对应内存数据的方式
    cout << refer << endl; // 这里将会输出1000，因为a所对应的内存数据通过test_2()改变了
    return 0;
}
```

引用的实质:

引用的本质在C++内部实现是一个指针常量, 意味着: 指向只能有一个, 但是可以改变内存数据

结论: C ++推荐引用技术, 因为语法简单, 引用本质是指针常量, 但是所有的指针操作编译都是编译器帮我们做了

常量引用

作用: 常量引用主要修饰形参, 防止误操作

在函数形参列表中, 可以加上const修饰形参, 防止形参修改实参

```
# include <iostream>
using namespace std;
void test(const int & a)
{
    a += 10; // 立马报错
    cout << a << endl;
}
int main()
{
    int a = 10;
    const int & ref = a;
```

```
    ref = 114514; //error: assignment of read-only reference
'ref'
    return 0;
}
```

实质: 这个`const int & ref`相当于是常量指针常量, 指向和对应的值是不变的(不可以通过它修改值)

一般地: 可以吧引用绑定到const对象上, 我们称之为**对常量的引用**. 与普通引用不同的是, 对常量的引用不能被用作修改它所绑定的对象.

同时注意: 引用的类型必须与其引用的对象类型一致! 但是有两个例外: 其中一个就是初始化常量引用时允许用任意表达式作为初始值, 只要表达式结果能转换成引用的类型:

```
int i = 42;
const int &r1 = i; // 正确
const int &r2 = 42; // 正确, r2是一个常量引用
const int &r3 = r1*2; // 正确, r3是一个常量引用
int &r4 = r1*2; // 错误, 因为r4是一个普通的非常量引用
```

为什么会这样? 因为只有const引用会把右边的式子用于创建一个**临时量对象**, 然后引用再绑定这对象.

函数高级

函数默认参数

在C++中, 函数的形参列表中的形参是可以有默认值的

`返回值类型 函数名 (参数 = 默认值){}`这一点和python十分相似

```
# include <iostream>
using namespace std;
int add(int a = 0, int b = 5, int c = 0)
{
    return a+b+c;
}
int func(int a = 0, int b = 0);
int main()
{
```

```
    cout << add(1,2) <<endl; // 正常输出3
    cout << add(3) << endl; // 正常输出8
    return 0;
}
int func(int a = 0, int b = 0) // 没错，即使是这样，编译器直接报错，因为歧义
{
    return a + b;
}
```

注意事项:

1. 如果某个位置已经有了默认参数, **那么从这个位置往后, 从左到右都必须有默认值**, 不写直接报错
2. 如果函数声明有了默认参数, 那么函数实现(定义)就不能有默认参数了, i.e, 声明和实现只能有一个默认参数

函数占位参数

C++中给函数的形参列表里面可以有占位参数, 用来做占位, **调用函数时必须填补该位置** (这一点要十分注意!!)

语法: `返回值类型 函数名 (数据类型){}`

在现阶段函数的占位函数存在意义不大, 但是后面的课程中会用到该技术

当然值得一提的是, 占位参数还可以同时是默认参数, 那么这样的情况下, 调用函数的时候雀氏不用再补上位置

函数重载

作用: 函数名可以相同, 提高复用性, 这一点在类的构造函数中十分重要

函数重载满足条件:

1. 同一个作用域下
2. 函数名称相同
3. 函数参数类型不同, 或者是个数不同, 后者是顺序不同

```
# include <iostream>
using namespace std;
void func()
```

```
{
    cout << "func的调用" << endl;
}
void func(int a)
{
    cout << "func(int a)的调用" << endl;
}
int main()
{
    func(); // func的调用
    func(1); // func(int a)的调用
    return 0;
}
```

注意事项:

1. 引用作为重载条件
2. 函数重载碰到函数默认参数
3. 函数的返回值不可以作为函数重载的条件(只能在参数上面有不同才有效)

关于第一点

```
# include <iostream>
using namespace std;
void func(int &a)
{
    cout << "func(int &a)的调用" << endl;
}
void func(const int &a)
{
    cout << "func(const int &a)的调用" << endl;
}
int main()
{
    int a = 2;
    func(a); // func(int &a)的调用, why?
    func(2); // func(const int &a)的调用, why?
    return 0;
}
```

两个why: 第一个why: 因为a是局部变量, 存放在栈区, 因此int &a = a合法

但是const int & a = a, 相当于变成了一个只读状态, 而变量a应该是可以改变的, 因此走上面的func**更好**(其实这个语法合法)

第二个why: int &a = 10; 明显是不合法的, 因为指向的必须是合法的**空间**, 10是数据本身(或者说, 它是常量, 储存在全局区)

但是const int &a = 10, 编译器内部实质是: `temp = 10; const int &a = temp`**(人为搭建了一个合法空间)**

关于这一点, 可以见下面这个例子:

```
# include <iostream>
using namespace std;
void print(const int & a)
{
    cout << a << endl;
}
int main()
{
    print(10); // 合法, 输出10
    return 0; // 当然,如果int a = 10;  print(a); 也完全没有问题, 就是temp与a空间挂扣
}
```

关于点2

```
# include <iostream>
using namespace std;
void func(int a)
{
    return;
}
void func(int a, int b = 0)
{
    return;
}
int main()
{
    func(12); // 直接报错!! 因为二义性!! 只能尽量避免这种情况(遇到重载, 尽量别用默认参数导致二义性)
}
```

类和对象

C++面对对象的三大特性: 封装 继承 多态

C++认为: 万事万物皆为对象, 对象上都有其属性和行为

封装的意义

封装大致情况

封装是三大特性之一, 意义在于:

1. 将其属性和行为作为一个整体, 表现生活中的事物
2. 将属性和行为加以权限控制

意义一: 属性和行为写在一起, 表现行为: `class 类名{ 访问权限: 属性 / 行为};`

实例: 设计一个圆类, 并且圆的周长

```
# include <iostream>
using namespace std;
const double pi = 3.14; // 圆周率, const修饰常量
class Circle
{
public: // 公共权限
    int m_r; // 半径
    double calculate_zc()
    {
        return 2 * pi * m_r;
    } // 这个函数就是这个类的一个行为
};
int main()
{
    Circle c1; // 通过圆类, 创建具体的圆, 这是一个实例化
    c1.m_r = 10;
    cout << "The zc of c1 is: " << c1.calculate_zc() << endl;
    return 0;
}
```

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

```
class student
{
public:
    string m_name;
    int m_id;
    void tell_name()
    {
        cout << "The name of the student is: " << m_name <<
endl;
    }
    void tell_id()
    {
        cout << "The id of the student is: " << m_id << endl;
    }
};
int main()
{
    student stu;
    stu.m_name = "Van";
    stu.m_id = 114514;
    stu.tell_name();
    stu.tell_id();
    return 0;
}
```

封装意义二: 类在设计时, 可以把属性和行为放在不同的权限下, 加以控制

有三种权限:

public 公共权限 类内和类外可以访问

protected 保护权限 类内可以访问, 类外不可以访问 (儿子也可以访问父亲中的保护内容)

private 私有权限 类内可以访问, 类外不可以访问 (后两者的区别在继承中可以体现) (儿子无法访问父亲中的私有内容)

```
# include <iostream>
# include <string>
using namespace std;
class Person
{
```

```
public:
    string m_name;
    void func()
    {
        m_Name = "Van";
        m_car = "toyota";
        m_password = 114514;
    }
protected:
    string m_car;
private:
    int m_password;
};
int main()
{
    Person Van;
    Van.func();
    cout << Van.m_car << endl; // 无法访问
    cout << Van.m_password << endl; // 无法访问
}
```

struct 和 class的区别

在C++中struct(结构体)和class唯一的区别就是: 默认访问权限不同

struct 默认权限为公共, 但是class默认权限为私有

在struct结构体的学习中, 结构体的什么东西我们都能默认get到, 但是class却不是这样

当然, struct里面也可以硬性定义private 或 protected的内容, 那这样一来, struct 和 class真的没什么区别了

成员属性设置为私有

优点1: 将成员属性设置为私有, 可以自己控制读写权限

优点2: 对于读写程序, 我们可以监测数据的有效性

```
# include <iostream>
# include <string>
using namespace std;
// 我们希望名字可读可写, 年龄只读, 偶像只写
```

```
class Person
{
public:
    void setname(string name)
    {
        m_name = name; // 注意，内部可以访问任何权限的属性和行为
    }
    string tellname()
    {
        return m_name;
    }
    int tellage()
    {
        return m_age;
    }
    void writeidol(string name)
    {
        m_idol = name;
    }
    void setheight() // num必须是0-200cm之间
    {
        int temp;
        while (true)
        {
            cout << "Please input his height: " << endl;
            cin >> temp;
            if (temp > 0 && temp < 200)
            {
                m_height = temp;
                break;
            }
            else
            {
                cout << "The height is invalid. Please input
again!" << endl;
            }
        }
    } // 这里体现了第二个优点
    int tellheight()
    {
        return m_height;
```

```
    }
private:
    string m_name; // 直接访问不行，必须通过公共的行为函数tellname()
    int m_age = 18; // 只读，不能写入，只能通过行为知道年龄
    string m_idol;
    int m_height;
};
int main()
{
    Person p;
    p.setname("Van");
    cout << "The name of this person is: " << p.tellname() <<
endl;
    cout << "The age of this person is: " << p.tellage() <<
endl;
    p.writeidol("Billy Harrington");
    // 以上充分体现了第一条优点
    p.setheight();
    cout << "The height of this person is: " << p.tellheight()<<
endl;
    return 0;
}
```

两个实例化相挂钩

比较两个正方体是否相同

```
# include <iostream>
using namespace std;
class Cube
{
public:
    void setlength(int length)
    {
        m_length = length;
    }
    int getlength()
    {
        return m_length;
    }
```

```
    bool is_same(Cube &c1 , Cube &c2)
    {
        if (c1.getlength() == c2.getlength())
        {
            return true;
        }
        else
        {
            return false;
        }
    }
private:
    int m_length;
};
int main()
{
    Cube c1;
    Cube c2;
    Cube c3;
    c1.setlength(4);
    c2.setlength(5);
    c3.setlength(5);
    cout << c1.is_same(c1, c2) << endl; // 0
    cout << c2.is_same(c2, c3) << endl; // 1
    return 0;
}
```

对象的初始化和清理

C++中的面向对象来源于生活，每个对象也都会有自己的初始设置以及对象销毁前的清理数据的设置

构造函数和析构函数

C++利用了构造函数和析构函数解决上述问题，这两个函数将会被编译器自动调用，完成对象初始化和清理工作

对象的初始化和清理工作是编译器强制我们要实现的事情，**如果我们不提供构造和析构，那么编译器会提供**

但是编译器提供的构造函数和析构函数是空实现

- 构造函数: 主要作用于创建对象时为对象的成员属性赋值, 构造函数你会由编译器自动调用, 无需手动调用
- 析构函数: 主要作用在于对象销毁前系统自动调用, 执行一些清理工作

构造函数语法: `类名(){}`

1. 构造函数, 没有返回值也不写void
2. 函数名称和类名相同
3. **构造函数可以有参数, 因此可以发生重载**
4. 程序在调用对象时会自动调用构造, 无须手动调用, 而且只会调用一次

析构函数语法: `~类名 (){}`

1. 析构函数, 没有返回值也不加void
2. 函数名称与类名相同, 在名称前加上符号~
3. **析构函数不可以有参数, 因此不可以发生重载**
4. 程序在对象销毁前会自动调用析构, 无需手动调用, 而且只会调用一次

例子:

```
# include <iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        cout << "I'm here!" << endl;
    }
    ~Person()
    {
        cout << "Hello! I'm here!" << endl;
    }
};
int test()
{
    Person p;
}
int main()
{
    test();// 同时输出: I'm here      Hello! I'm here
    return 0;
```

```
}
```

注意:

1. 为什么构造和析构函数都被调用了呢? 因为p是存放在栈区的, test函数一旦调用完毕, 就会销毁
2. 这是不是很像python魔法方法呢?

构造函数的分类和调用

两种分类方式:

- 按参数分为: 有参构造和无参构造(别名: 默认构造使)
- 按类型分为: 普通构造和拷贝构造

三种调用方式:

- 括号法　显示法　隐式转换法

```
# include <iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        cout << "Without parameter!" <<endl;
    }
    Person(int a)
    {
        cout << "With parameter!" <<endl;
        age = a;
    }
    Person (const Person &p) // 拷贝构造, 这样的语法意味着这个对象将会复制传入的对象的一些属性
     // 注意这里传入的是引用, 方便不必内存再复制一份
    {
        age = p.age;
        cout << "Copied!" << endl;
    }
    ~Person()
    {
```

```
        cout << "I'm here again" << endl;
    }
    int age;
};
int main()
{
    // Person p0(); 这句话虽然代表我想调用无参/默认构造，但是这样的语法是不合法的!
    // 想要无参/默认构造，不加括号!!!! 因为这行代码会被误认为是函数的声明(返回的是Person类的一个函数)
    Person p1(18); // 括号法
    Person p2(p1); // 括号法
    Person p3 = Person(22); // 显示法
    Person p4 = Person(p3); // 显示法
    Person p5 = 10; // 编译器视角下，它等价于: Person p5 = Person(10);
    Person p6 = p5; // Person p6 = Person(p5);
    cout << p1.age << endl;
    cout << p2.age << endl;
    cout << p3.age << endl;
    cout << p4.age << endl;
    cout << p5.age << endl;
    cout << p6.age << endl;
    return 0;
} /*
With parameter!
Copied!
With parameter!
Copied!
With parameter!
Copied!
18
18
22
22
10
10
I'm here again
I'm here again
I'm here again
I'm here again
```

```
I'm here again
I'm here again */
```

注意:

1. 想要无参/默认构造, 不加括号!
2. `Person(22);`单纯依据这样的话, 代表是匿名对象; `Person p3 = Person(22);` 代表着给这个匿名对象起了个名叫p3
3. 匿名对象的特点: 当前行执行结束后, 系统会立即回收掉匿名对象, 因为一点用也没有, 但是构造和析构函数仍然会被调用
4. **不要利用拷贝构造函数来初始化匿名对象**, 因为编译器会认为: `Person (p3);` 等价于`Person p3;`, 右边是一个默认构造, 相当于是对p3这个对象构造(初始化)了两次
5. 一般来说, 推荐使用括号法

拷贝构造函数调用时机

一般来说, 有三种时机

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

```
# include <iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        cout << "default" << endl;
    }
    Person(int age)
    {
        cout << "with parameter" << endl;
        m_age = age;
    }
    Person(const Person &p)
    {
        m_age = p.m_age;
        cout << "copy" << endl;
```

```
    }
    ~Person()
    {
        cout << "deleted!" << endl;
    }
    int m_age;
};
void test1()
{
    Person p1(20);
    Person p2(p1);
}
void doWork1(Person p)
{
    cout << "doWork1" << endl;
}
Person doWork2()
{
    Person p1;
    return p1;
}
void test2()
{
    Person p;
    doWork1(p);
}
void test3()
{
    Person p = doWork2();
}
int main()
{
    test1();
    // 这是所述的第一种情况
    // with parameter
    // copy
    // deleted!
    // deleted!
    test2();
    // default
    // copy
```

```
    // doWork1
    // deleted!
    // deleted!
    test3();
    // default
    // copy     但是这一行和下面这一行可能不会显示，因为编译器的原因
    // deleted!
    // deleted!
    return 0;
}
```

注意: 在test2中, 为什么会出现default? 因为值传递的时候, 事实上是copy了一份, 因此拷贝构造函数被调用了, 而且从顺序上来说, 是先发生了copy, 再发生了doWork1函数体的运行

构造函数的调用规则

默认情况下, C++编译器至少给一个类添加三个函数:

1. 默认构造函数(无参, 且函数体为空)
2. 默认构析函数(无参, 且函数体为空)
3. **默认拷贝构造函数, 对属性进行值拷贝**

构造函数调用规则如下:

- 如果用户定义有参构造函数, C++不会再提供默认无参构造, **但是会提供默认拷贝构造**
- 如果用户定义拷贝构造函数, **C++不会提供其他构造函数**

```
# include <iostream>
using namespace std;
class test
{
public:
    test(const test &t)
    {
        cout << "copy function of test" << endl;
    }
};
class Person
{ // 不提供拷贝构造函数
public:
```

```
    Person(int age)
    {
        cout << "with parameter" << endl;
        m_age = age;
    }
    ~Person()
    {
        cout << "deleted!" << endl;
    }
    int m_age;
};
void test1()
{
    Person p1(18); // 我们提供了有参构造函数
    Person p(p1); // 注意: Person类里面没有拷贝构造函数
    cout << "The age of p is: " << p.m_age << endl;
}
int main()
{
    test1();
    // with parameter
    // The age of p is: 18
    // deleted!
    // deleted!
    Person p; // 直接报错: 不存在默认构造函数
    test t; // 直接报错, 没有默认构造函数
    test t0; // 直接报错, 没有有参构造函数
    return 0;
}
```

可见, 拷贝依然是可以进行的, 虽然我们没有提供拷贝构造函数, 但是编译器提供了, 把p1的m_age属性传给了p

如果只定义了拷贝构造函数, 那么这是不是很奇怪呢? 奇怪就对了(...)

深拷贝和浅拷贝

是一个非常经典的问题, 也是一个常见的坑

浅拷贝: 简单的复制拷贝操作 (例如平常的等号赋值)

深拷贝: 在堆区重新申请空间, 进行拷贝操作

```
# include <iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        cout << "default" << endl;
    }
    Person(int age, int height) // 后面的这个是一个int, 不是指针
    {
        m_age = age;
        m_height = new int(height);
        cout << "with parameter" << endl;
    }
    ~Person()
    {
        if (m_height != NULL) // 如果这个指针是有指向的
        // 注意, 这个语法不是在判断m_height对应的堆区的数据是不是0, 而是判断它有没有指向
        {
            delete m_height; // 删除堆区的数据
            m_height = NULL; // 指针置空
        }
        cout << "deleted!" << endl;
    }
    int m_age;
    int *m_height; // 创建一个指向含有身高数据的指针, m_height就是一个指针
};
void test01()
{
    Person p1(18, 160);
    cout << "The age of p1 is: " << p1.m_age << endl;
    cout << "The height of p1 is: " << *p1.m_height << endl; // 注意解引用
    Person p2(p1);
    cout << "The age of p2 is: " << p2.m_age << endl;
    cout << "The height of p2 is: " << *p2.m_height << endl; // 注意解引用
}
```

```
int main()
{
    test01();
    // with parameter
    // The age of p1 is: 18
    // The height of p1 is: 160
    // The age of p2 is: 18
    // The height of p2 is: 160
    // deleted!
    /* 注意这里，本来应该有两个deleted，不是吗，但是为什么却只有一个?
    其实事实上，程序到这里已经开始有点问题了，但是由于编译器版本比较新，所以看起来还能跑
    在test01()结束的时候，p2,p1依次释放(没错，先进后出)，p2的m_height指向的堆区中的数据删除掉了
    轮到p1的m_height时候，它指向的还是堆区的那个地址，但是那个地址里面早就没有内存了
    因此就发生堆区内存的重复释放，这是非法的!!
    为什么会发生这样的问题呢? 因为我们没有提供拷贝构造函数，编译器给的默认的拷贝构造函数是浅拷贝
    因此使得p1p2各自两个指针指向的堆区的地址是一样的!
    浅拷贝的问题，要用深拷贝解决*/
    return 0;
}
```

```
# include <iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        cout << "default" << endl;
    }
    Person(int age, int height) // 后面的这个是一个int，不是指针
    {
        m_age = age;
        m_height = new int(height);
        cout << "with parameter" << endl;
    }
    Person (const Person &p)
    {
```

```
        m_height = new int(*p.m_height); // 注意解引用
        m_age = p.m_age;
        cout << "deep copy! " << endl;
    }
    ~Person()
    {
        if (m_height != NULL) // 如果这个指针是有指向的
        // 注意，这个语法不是在判断m_height对应的堆区的数据是不是0，而是判断它有没有指向
        {
            delete m_height; // 删除堆区的数据
            m_height = NULL; // 指针置空，不指向任何地方，非常规范!
        }
        cout << "deleted!" << endl;
    }
    int m_age;
    int *m_height; // 创建一个指向含有身高数据的指针，m_height就是一个指针
};
void test01()
{
    Person p1(18, 160);
    cout << "The age of p1 is: " << p1.m_age << endl;
    cout << "The height of p1 is: " << *p1.m_height << endl; //注意解引用
    cout << "The address of the p1_height is: "<< p1.m_height <<endl;
    Person p2(p1);
    cout << "The age of p2 is: " << p2.m_age << endl;
    cout << "The height of p2 is: " << *p2.m_height << endl; //注意解引用
    cout << "The address of the p2_height is: "<< p2.m_height <<endl;
}
int main()
{
    test01();
/*
with parameter
The age of p1 is: 18
The height of p1 is: 160
```

```
The address of the p1_height is: 0x792510
deep copy!
The age of p2 is: 18
The height of p2 is: 160
The address of the p2_height is: 0x792530
deleted!
deleted!
*/
    return 0;
}
```

初始化列表

作用: C++提供了初始化列表语法, 用来初始化属性

语法: `构造函数(): 属性1 (值1), 属性2 (值2) ...{函数体}`

```
# include <iostream>
using namespace std;
class Person{
public:
    // 传统初始化操作
    Person(int a, int b, int c)
    {
        m_a = a;
        m_b = b;
        m_c = c;
    }
    // 初始化列表的方式实现初始化
    Person():m_a(1), m_b(2), m_c(3)
    {
    }
    int m_a;
    int m_b;
    int m_c;
};
void test1()
{
    Person p(1,2,3);
    cout << p.m_a << endl;
    cout << p.m_b << endl;
    cout << p.m_c << endl;
```

```
}
void test2()
{
    Person p;
    cout << p.m_a << endl;
    cout << p.m_b << endl;
    cout << p.m_c << endl;
}
int main()
{
    test1();
    test2();
    return 0;
}
```

但是发现了吗, 初始化列表只能把固定的值赋给它们, 这是我们不希望看到的, 这个初始化列表像是无参构造函数

因此真正方便的使用方式见下

```
# include <iostream>
using namespace std;
class Person{
public:
    // 初始化列表的方式实现初始化
    Person(int a, int b, int c):m_a(a), m_b(b), m_c(c)
    {
    }
    int m_a;
    int m_b;
    int m_c;
};
void test1()
{
    Person p(1,2,3);
    cout << p.m_a << endl;
    cout << p.m_b << endl;
    cout << p.m_c << endl;
}
int main()
{
    test1();
```

```
    return 0;
}
```

上面这个初始化列表就是采用了有参构造函数形式, 就方便我们灵活传参了

类对象作为类成员

C++类中的成员可以是另一个类的对象, 我们称该成员为对象成员

例如:

```
class A{}
class B
{
    A a;
}
```

B类中有对象A作为成员, A为对象成员

那么问题来了: 创建B的时候, A与B的构造和析构顺序是谁先谁后?

事实上是: `A构造 >> B构造 >> B析构 >> A析构`

```
# include <iostream>
# include <string>
using namespace std;
class Phone{
public:
    Phone(string name)
    {
        phone_name = name;
        cout << "construction of Phone" << endl;
    }
    ~Phone()
    {
        cout << "destruction of Phone" << endl;
    }
    string phone_name;
};
class Person
{
public:
```

```
    Person(int num, string phonename): age(num),
phone(phonename)
    // 注意, 这里不是phone.phone_name(phonename), 我是给phone传参从而
实例化
    {
        cout << "construction of Person" << endl;
    }
    ~Person()
    {
        cout << "destruction of Person" << endl;
    }
    int age;
    Phone phone;
};
void test()
{
    Person p(18, "iphone");
}
int main()
{
    test();
// construction of Phone
// construction of Person
// destruction of Person
// destruction of Phone
    return 0;
}
```

上面这个实验代表着: 先是Phone完成了对象实例化(构造), 然后Person的构造才算正式完成; 在析构的时候, 先是Person析构, 然后Phone的对象成员析构

1. 当其他类对象作为本类成员, 构造的时候先构造对象成员, 再构造自身
2. 当其他类对象作为本类成员, 析构的时候先析构自身, 再析构对象成员

静态成员

静态成员就是在成员变量和成员函数前加上关键词static, 称为静态成员

静态成员分为:

- 静态成员变量:
    1. 所有类对象共享同一份数据

    2. 在编译阶段分配内存 (分配至全局区)
    3. 类内声明, 类外初始化
- 静态成员函数
    1. 所用对象共享同一个函数
    2. 静态成员函数只能访问静态成员变量

静态成员变量

访问公共静态成员变量的方式有两种: 通过对象; 通过类名

但是访问私有静态成员变量类外访问不到 (除非通过公共函数赋值给私有静态成员变量)

```
# include <iostream>
using namespace std;
class Person{
public:

    // 2. 在编译阶段分配内存 (分配至全局区)
    static int a;
private:
    static int b;
};
// 3. 类内声明, 类外初始化; 注意对应的语法
// 通过类名访问静态成员
int Person::a = 100;
// int Person::b = 810; 这是非法的
void test()
{
    Person p1;
    cout << p1.a << endl;
    Person p2;
    // 通过对象访问静态成员
    p2.a = 200;
    cout << p1.a << endl;
    // 1. 所有类对象共享同一份数据
    cout << p2.a << endl;
    // cout << p2.b << endl;    报错, 不知道这个值是谁
}
int main ()
{
```

```
    test(); // 依次输出: 100 200 200
    return 0;
}
```

静态成员函数

访问公共静态成员函数有两种方法: 通过对象; 通过类名

```
# include <iostream>
using namespace std;
class Person
{
public:
    static void func()
    {
        cout << "The static int a = " << a << endl;
     // cout << "The int b = " << b << endl;  如果这个语句加上, 程序报错, 因为b不是静态成员变量
    }
    static int a;
    int b;
};
int Person::a = 100; // 通过类名访问静态成员变量
void test()
{
    Person p;
    p.b = 100;
    p.func(); // 通过对象访问
    Person::func(); // 通过类名访问
}
int main()
{
    test();
    Person::func(); // 通过类名访问
    return 0;
}
```

其实关于静态成员函数只能访问静态成员变量是十分容易认同的, 因为不同的对象都能使用这个函数, 如果还能访问非静态成员变量, 那么不同对象的该函数的结果就会不一样, 那么这个"静态成员函数"有什么用?

同时注意: 类外访问不了私有静态成员函数(除非一个公共的函数里面会调用这个函数)

C++对象模型和this指针

成员变量和成员函数分开储存

在C++中, 类内的成员变量和成员函数分开储存, 只有非静态成员才属于类的对象

换而言之, 静态成员变量和静态成员函数都不是称为属于类的对象

```
# include <iostream>
using namespace std;
class Person{
};
class Test1{
    int m_a; // 非静态成员变量
};
class Test2{
    static int m_a; // 静态成员变量
    static void func1()
    {
    }
    void func2()
    {
    }
};
int main()
{
    Person p;
    Test1 test1;
    Test2 test2;
    cout << sizeof(p) << endl; // 输出的是1
    cout << sizeof(test1) << endl; // 输出的是4
    cout << sizeof(test2) << endl; // 输出的是1
    return 0;
}
```

第一个输出的是1. 为什么呢? C++编译器会给每个空对象也分配一个字节空间, 是为了区分空对象占内存的位置

换而言之, 假如我创建了两个空对象, 我要确保这两个空对象不能再同一个地址里面, 如何保证呢? 给它们分配一个字节空间就可以了, 只不过人为规定只要给一个字节的字节空间就可以了

同时注意: **静态成员变量, 静态成员函数, 非静态成员函数都是不属于类对象上的**

this指针概念

我们注意到非静态成员函数不属于类对象, 是分开存储的. 每一个非静态成员函数只会诞生一份函数实例, 也就是说多个同类型的对象会共用同一块代码, 那么问题来了: 这一块代码是如何区分那个对象调用自己的呢?

C++通过提供特殊的对象指针, this指针, 解决上述问题, this指针指向被调用的成员函数所述的对象

this指针是隐含每一个非静态成员函数内的一种指针, this指针不需要定义, 直接使用就可以了

this指针的用途:

- 当形参和成员变量同名时, 可用this指针进行区分
- 在类的非静态成员函数中返回对象本身, 可使用return *this

```
# include <iostream>
using namespace std;
class Person{
public:
    Person(int age)
    {
        // 解决名称冲突
        this->age = age;
    }
    // 返回对象本身用this
    Person& PersonAddAge(Person *p) // 引用传递,
    // 注意, 同时也要用引用的方式返回
    // 为什么呢? 为了确保返回的是本体!! 否则会复制一份新的出来
    {
        this->age += p.age;
        return *this;
    }
    int age;
};
```

```
int main()
{
    Person p(18);
    cout << p.age << endl; // 18
    Person p2(10);
    p2.PersonAddAge(p1); // 28
    // 链式编程思想
    p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1); // 合法了
    return 0;
}
```

事实证明, 这一节的内容在实战中十分重要

空指针访问成员函数

C++中空指针也是可以调用成员函数的, 但是也要注意有没有用到this指针

如果用到了this指针, 需要加以判断 保证代码的健壮性

```
# include <iostream>
using namespace std;
class Person{
public:
    void showClassName()
    {
        cout << "This is Person class" << endl;
    }
    void showAge()
    {
        cout << "age = " << m_age << endl;
    }
    int m_age;
};
void test()
{
    Person *p = NULL;
    p->showClassName(); // 这里还是正常的
    // p->showAge();  如果这句话加上的话, 立马报错
}
int main()
{
```

```
    test();
    return 0;
}
```

为什么p->showAge就会出问题呢? 因为, 第11行其实事实上是:

`cout << "age = " << this->m_age << endl;`

但是一个空指针去用这个函数, 我连实体都没有, 拿来的对象? 都不知道这个m_age在哪里

所以为了方式这种事情发生, 一般这个类方法会这么写:

```
public:
    void showClassName()
    {
        cout << "This is Person class" << endl;
    }
    void showAge()
    {
        if (this == NULL)
        {
            return;
        }
        cout << "age = " << m_age << endl;
    }
    int m_age;
};
```

先判断是不是空指针访问这个方法(因为这个方法既可以用对象访问, 也可能会被空指针访问), 如果是, 直接提前return

const修饰成员函数

常函数:

- 成员函数后加const后我们称这个函数为常函数
- 常函数内不可以修改成员属性
- 成员属性声明时加上关键词mutable后, 在常函数中依然可以修改

常对象:

- 声明对象前加const称该对象为常对象

- 常对象只能调用常函数
- 常对象只能修改有了mutable的常变量

```
# include <iostream>
using namespace std;
class Person{
public:
    void showPerson() const // 这是一个常函数
    {
        m_b = 100; // 合法
        // m_a = 100;    不合法, 提示左边的值必须是可以修改的
    }
    void func()
    {
    }
    int m_a;
    mutable int m_b; // 这样一来, 这个值在常函数依然可以修改
};
void test()
{
    const Person p; // 创建一个常对象
    p.m_b = 114; // 合法, 因为修改的是加了mutable的变量
    // p.m_a = 514;   直接报错
    // p.func();    直接报错
}
```

为什么加了const能达到这个效果? 其实跟this指针有关

因为this指针的本质, 是指针常量, 指针的指向是不可以修改的, 就是相当于一个`Person * const this`

但是加上了const, 相当于修饰了指针, 变成了`const Person * const this`

因此值不可以进行改变了

友元

在程序里, 有些私有属性也想让类外特殊的一些函数或者类进行访问, 就需要用到友元的技术

友元的目的就是让一个函数或者类, 访问另一个类中的私有成员, 关键词: friend

三种实现方式: 全局函数做友元, 类做友元, 成员函数做友元

全局函数做友元

```
# include <iostream>
# include <string>
using namespace std;
class Building
{
    friend void goodGay(BUilding *building); // 友元函数声明
public:
    Building()
    {
        m_SittingRoom = "Sitting Room";
        m_Bedroom = "Bedroom";
    }
    string m_SittingRoom;
private:
    string m_Bedroom;
};
void goodGay(BUilding *building) // 引用传递, 好基友全局函数
{
    cout << building.m_SittingRoom << endl;
    cout << building.m_Bedroom << endl; // private也可以访问!
}
```

类做友元

让一个类能访问另一个类的私有成员

```
# include <iostream>
# include <string>
using namespace std;
class Building
{
    friend class GoodGay; // GoodGay是Building的朋友, 可以访问
Building中的私有成员
public:
    Building(); // 声明, 在类外写该函数
    string m_SittingRoom;
private:
```

```
    string m_Bedroom;
};
class GoodGay
{
public:
    GoodGay();
    void visit(); // 参观函数，可以访问Building中的属性
    Building *building; // 注意，这是一个指针，可以指向一个Building类
成员
};
// 拓展：在类外写类函数
Building::Building()
{
    m_SittingRoom = "Sitting Room";
    m_Bedroom = "Bedroom";
}
GoodGay::GoodGay()
{
    building = new Building; // 创建建筑物对象，指针接受它
}
void GoodGay::visit()
{
    cout << building->m_Bedroom << endl;
}
int main()
{
    GoodGay gg;
    gg.visit();
    return 0;
}
```

成员函数做友元

```
# include <iostream>
# include <string>
using namespace std;
class Building;
class GoodGay
{
public:
    GoodGay();
```

```
    void visit(); // 让它可以访问Building中的私有成员
    Building * building;
};
class Building
{
    friend void GoodGay::visit();
public:
    Building()
    {
        m_SittingRoom = "Sitting Room";
        m_Bedroom = "Bedroom";
    }
    string m_SittingRoom;
private:
    string m_Bedroom;
};
GoodGay::GoodGay()
{
    building = new Building;
}
void GoodGay::visit()
{
    cout << building->m_Bedroom <<endl;
}
int main()
{
    GoodGay gg;
    gg.visit();
    return 0;
}
```

注意:要先写GoodGay, 再写Building; 而且在GoodGay前面要声明class Building;

i.e. 顺序是一个要十分注意的点!!! **将要被friend修饰的类成员函数所对应的类的定义必须在friend声明所在类的前面**

运算符重载

概念: 对已有的运算符重新进行定义, 赋予其另一种功能, 以适应不同的数据类型

加号运算符重载

通过成员函数重载+号

```
# include <iostream>
using namespace std;
class Person{
public:
    Person()
    {
    }
    Person(int age)
    {
        this->m_age = age;
    }
    Person operator+ (Person &p) // 引用传递
    { // 通过成员函数重载+号
        Person temp;
        temp.m_age = this->m_age + p.m_age;
        return temp;
    }
    int m_age;
};
int main()
{
    Person p1(18);
    Person p2(20);
    Person p3 = p1 + p2; // 实际上是: Person p3 = p1.operator+(p2)
    cout << p3.m_age << endl;
    return 0;
}
```

通过全局函数重载

```
# include <iostream>
using namespace std;
class Person{
public:
    Person()
    {
    }
```

```
    Person(int age)
    {
        this->m_age = age;
    }
    int m_age;
};
Person operator+ (Person &p1, Person &p2)
{
    Person temp;
    temp.m_age = p1.m_age + p2.m_age;
    return temp;
}
int main()
{
    Person p1(18);
    Person p2(20);
    Person p3 = p1 + p2; // 实际上是: Person p3 = operator+(p1,
p2)
    cout << p3.m_age << endl;
    return 0;
}
```

注意:

1. 对于内置的数据类型的表达式的运算符是不可以改变的
2. 不要滥用运算符重载

左移运算符重载

作用: 可以输出自定义数据的类型

```
# include <iostream>
using namespace std;
class Person
{
public:
    int m_a;
};
ostream& operator<<(ostream &cout, Person &p) // 本质:
operator(cout, p) == cout << p, 注意点3
{ // 注意点1, 2
    cout << "m_a of p is: " << p.m_a;  // 注意点4
```

```
}
void test()
{
    Person p;
    p.m_a = 10;
    cout << p << endl;
}
int main()
{
    test();
    return 0;
}
```

注意: 不可以利用成员函数重载<<运算符, 因为无论如何, 简化后, cout在左侧

因此, 只能用全局函数重载左移运算符

注意点:

1. 这里为什么前面的数据类型是ostream? 因为后面的`cout << p << endl;`实际上是一个链式法则, 前面一个输出的是ostream, 才能作为后面一个<<的传参
2. 实际上, 这里面`ostream &cout`的cout可以随便改名字(函数体内的也对应改), 因为这里是引用, 起别名
3. 注意这里要返回引用!! 不然的话, 事实上是另外新开了一个ostream!
4. 这里不要有endl

当然, 如果这输出的东西里面有私有成员呢?

```
# include <iostream>
using namespace std;
class Person
{
    friend ostream& operator<<(ostream &cout, Person &p);
public:
    Person(int a)
    {
        this->m_a = a;
    }
private:
    int m_a;
};
```

```
ostream& operator<<(ostream &cout, Person &p) // 本质:
operator(cout, p) == cout << p
{
    cout << "m_a of p is: " << p.m_a;
    return cout;
}
void test()
{
    Person p(18);
    cout << p << endl;
}
int main()
{
    test();
    return 0;
}
```

注意全局函数写在了类的后面, **意在先让类知道, 这个函数是友元**

递增运算符重载

作用: 通过重载递增运算符, 实现自己的整型数据

```
# include <iostream>
using namespace std;
class MyInteger
{
    friend ostream& operator<<(ostream &cout, const MyInteger
&p) ;
public:
    MyInteger()
    {
        m_num = 0;
    }
    MyInteger& operator++()
    {
        this->m_num++;
        return *this; // 返回自己
    }
    MyInteger operator++(int) // 占位参数, 可以用类区分前置和后置
    // 注意点2
    { // 这里不用引用了, 返回的是值! 注意前置和后置的本质区别
```

```
        MyInteger temp = *this; // 记录当前结果
        this->m_num++;
        return temp; // 将记录结果返回
    }
private:
    int m_num ;
};
ostream& operator<<(ostream &cout, const MyInteger &p) // 注意点1
{ // 返回引用的目的是为了对同一个数据进行操作(引用十分常见的用法!!!)
    cout << p.m_num;
    return cout;
}
void test1()
{
    MyInteger m;
    cout << m << endl; // 0
    m++; // 后置递增
    cout << m << endl; // 1
    ++m; // 前置递增
    cout << m << endl; // 2
}
void test2()
{
    MyInteger m;
    cout << ++(++m) << endl; // 2
    cout << m++ << endl; // 2
    cout << m << endl; // 3
}
int main()
{
    test1();
    test2();
    return 0;
}
```

注意点:

1. 为什么后面要加const? 因为return temp是一个临时变量, 即它默认为const MyInteger, 而operator<<后面的定义中没说可以是const MyInteger; 而事实上, 非const可以引用至const, 但是const不能引用至非const, 所以为了完美解决

问题(兼顾后置和前置两种const和非const返回), 直接定义里面加上const, 使得该左移运算符能够接受const和非const

2. 为什么这里面不能返回引用? 否则直接返回改变后的自身了! 核心思想是创建临时的temp接受当前状态, 然后把这个const MyInteger(记录的是原来的状态)返回

赋值运算符重载

C++编译器至少给一个类添加四个函数:

1. 默认构造函数(无参, 函数体为空)
2. 默认析构函数(无参, 函数体为空)
3. 默认拷贝构造函数, 对属性进行值拷贝
4. 赋值运算符operator=, 对属性进行值拷贝

如果类中有属性指向堆区, 做复制操作也会出现深浅拷贝问题 (堆取数据重复释放)

```
# include <iostream>
using namespace std;
class Person{
public:
    Person(int age)
    {
        m_age = new int(age); // 指针指向堆区的一个数据
    }
    ~Person()
    {
        if (m_age != NULL)
        {
            delete m_age; // 人工释放堆区内存, 如果不是空指针, 那么指针对应堆区数据清零
            m_age = NULL; // 指针重新置回空指针
        }
    }
    int *m_age; // 创建一个指针
};
void test()
{
    Person p1(18);
    Person p2(20);
    p2 = p1; // 赋值操作
```

```
    cout << "The age of p2 is: "<< *p2.m_age << endl;
}
int main()
{
    test();
    return 0;
}
```

上面这个代码会直接崩掉, 但是p2年龄还是会显示出来, 而且是18; 那么为什么会崩掉呢? 问题在析构上

`p2 = p1`是复制操作, 由于是浅拷贝, p1 p2两个指针都是指向了同一个堆区中重复的数据, 造成了堆取数据的重复释放

如何解决呢? 我们要深拷贝, 因此我们所用的"等号"不能是程序提供的Operator, 要赋值运算符重载

```
    void operator=(Person &p1)
    {
        if (this->m_age != NULL)
        {
            delete m_age;
            m_age = NULL; // 如果原来这个对象在堆区有指向了, 那么要重置为空指针
        }
        this->m_age = new int(*p1.m_age);
    }
```

上面这个当然在`p2 = p1`的情况下可以正常运行, 但是我们还是注重链式编程

```
    Person& operator=(Person &p1) // 返回的是自身的解引用! 而不是Person的一个对象!
    // 如果是Person打头, 那么返回的是按照*this拷贝构造的对象
    {
        if (this->m_age != NULL)
        {
            delete m_age;
            m_age = NULL; // 如果原来这个对象在堆区有指向了, 那么要重置为空指针
        }
        this->m_age = new int(*p1.m_age);
        return *this; // 返回自身
    }
```

注意 Person& 和 Person 的区别!

关系运算符重载

作用: 关系运算符重载器, **可以让两个自定义类型对象进行对比操作**(很像*python*魔法方法)

```
# include <iostream>
# include <string>
using namespace std;
class Person{
public:
    Person(string str, int num)
    {
        name = str;
        age = num;
    }
    bool operator==(Person &p)
    {
        if (this->name == p.name && this->age == p.age)
        {
            return true;
        }
        return false;
    }
    bool operator !=(Person &p)
    {
```

```
        if (this->name == p.name && this->age == p.age)
        {
            return false;
        }
        return true;
    }
    string name;
    int age;
};
int main(){
    Person p1("Lily", 18);
    Person p2("Jack", 20);
    Person p3("Jack", 20);
    cout << (p1 == p2) << endl; // 0
    cout << (p2 == p3) << endl; // 1
    cout << (p1 != p3) << endl; // 1
    return 0;
}
```

函数调用运算符重载

由于重载后使用的方式非常像函数的调用, 因此称为仿函数; 仿函数没有固定写法, 非常灵活

```
# include <iostream>
# include <string>
using namespace std;
class Myprint{
public:
    void operator()(string test)
    {
        cout << test << endl;
    }
};
class Myadd
{
public:
    int operator()(int a, int b)
    {
        return a+b ;
    }
```

```
};
int main()
{
    Myprint print;
    print("Hello World!"); // 这看起来是不是非常像函数调用?
    Myadd add;
    cout << add(1, 2) << endl;
    cout << Myadd()(3, 4) << endl; // 注意: Myadd()代表创建一个匿名函数对象
    // 一旦该句执行完毕, 匿名对象立马释放
    return 0;
}
```

继承

继承是面向对象的三大特性之一; 有些类与类之间存在特殊的关系

我们发现, 定义这些类的时候, 下级别的成员除了拥有上一级的共性之外, 还有一些自己的特性

这个时候我们就可以考虑利用继承技术, 减少重复的代码

继承的基本语法

```
# include <iostream>
using namespace std;
class BasePage{
// 设想一个教JAVA, PYTHON的页面
public:
    void header()
    {
        cout << "Public header" << endl;
    }
    void footer()
    {
        cout << "Public bottom" << endl;
    }
    void left()
    {
        cout << "Catagories" << endl;
    }
```

```
    // 以上都是JAVA, PYTHON两个页面的公共部分
};
class Java : public BasePage // 注意是冒号, 后面没有分号
{
public:
    void content()
    {
        cout << "Java teaching courses" << endl;
    }
};
class Python : public BasePage
{
public:
    void content()
    {
        cout << "Python teaching courses" << endl;
    }
};
int main()
{
    Java().header();
    Python().footer();
    Java().content();
    Python().content(); // 创建匿名对象, 方便调用函数
    return 0;
}
```

`class A : public B`, 其中A是子类或派生类, B是父类或基类

派生类中的成员, 包含两个部分:

一类是从基类继承过来的, 一类是自己增加的成员; 从基类继承过来的表现其共性, 而新增的成员体现了其个性

继承方式

`class 子类 : 继承方式 父类`

继承方式一共有三种: 公共继承, 保护继承, 私有继承

```
class A{
public:
    int a;
protected:
    int b;
private:
    int c;
};
```

```
class B : public A // a:public    b:protected    c:无法访问
class B : protected A // a&b : protected     c:无法访问
class B : private A // a&b : private    c:无法访问
```

可见, private c无论哪种继承, 一定都是无法访问

继承中的对象模型

问题: 从父类继承过来的成员, 哪些属于子类对象中?

```
# include <iostream>
using namespace std;
class A{
public:
    int a;
protected:
    int b;
private:
    int c;
};
class Son : public Base
{
public:
    int d;
};
int main()
{
    cout << sizeof(Son) << endl;
}
```

结果输出的是16, 相当于父类无论任何属性的成员都会继承, 并且子类自己还有额外的成员属性

虽然说私有成员"访问不到", 但是父类中的私有成员属性是被编译器给隐藏了, 因此访问不到, 但是确实被继承下去了

继承中构造和析构顺序

子类继承父类之后, 当创建子类对象, 也会调用父类的构造函数

问题: **父类和子类构造和析构的顺序?**

```
# include <iostream>
using namespace std;
class Base{
public:
    Base()
    {
        cout << "construction of Base" << endl;
    }
    ~Base()
    {
        cout << "destruction of Base" << endl;
    }
};
class Son : public Base
{
public:
    Son()
    {
        cout << "construction of Son" << endl;
    }
    ~Son()
    {
        cout << "destruction of Son" << endl;
    }
};
void test()
{
    Son();
}
int main()
{
    test();
    return 0;
```

```
    // construction of Base
    // construction of Son
    // destruction of Son
    // destruction of Base
}
```

可见顺序是: 父类构造-子类构造-子类析构-父类析构

继承同名成员处理方式

问题: 当子类与父类出现了同名成员的时候, 如何通过子类对象, 访问到子类或父类中同名的数据呢?

- 访问子类同名成员的时候, 直接访问即可
- 访问父类同名成员的时候, 需要加作用域

示例:

```
# include <iostream>
using namespace std;
class Base{
public:
    Base()
    {
        m_a = 100;
    }
    void func()
    {
        cout << "m3" << endl;
    }
    int m_a;
};
class Son : public Base
{
public:
    Son()
    {
        m_a = 200;
    }
    void func()
    {
        cout << "chipi chipi" << endl;
```

```
    }
    int m_a = 200;
};
int main()
{
    // 利用创建匿名函数, 方便我们直接访问成员
    cout << Son().m_a << endl; // 200
    cout << Son().Base::m_a << endl; // 100
    Son().func(); // chipichipi
    Son().Base::func(); // m3
    return 0;
}
```

这里的`Son().Base::m_a`, 就是利用了作用域进行访问, 函数同理

继承同名静态成员处理方式

问题: 继承中同名的静态成员在子类对象上如何进行访问

回顾: 静态成员变量特点:

1. 所有对象共享同一份数据
2. 编译阶段分配内存(放在全局区)
3. 类内声明, 类外初始化

静态成员函数特点:

1. 只能访问静态成员变量
2. 所有对象共享同一份函数实例

静态成员和非静态成员出现同名, **处理方式一致**

- 访问子类同名成员的时候, 直接访问即可
- 访问父类同名成员的时候, 需要加作用域

多继承语法

C++中允许一个类继承多个类

`class 子类 : 继承方式 父类1, 继承方式 父类2...`

但是多继承可能会引发父类中有同名成员出现, 那么需要加**作用域**即可(但是在实际开发中, **不建议多继承**)

菱形继承

概念:

- 两个派生类(B,C)继承同一个基类(A)
- 又有某个类(D)同时继承两个派生类(B,C)

这种继承称为菱形继承(或者钻石继承)

菱形继承问题:

1. D使用数据的时候, 可能产生二义性
2. D继承了两份A的数据, 但是其实我们清楚, 这份数据我们只要一份

接下来我们来看这两个问题是如何解决的

```
# include <iostream>
using namespace std;
class A
{ // A 称为虚基类
public:
    int a;
};
class B : virtual public A // 虚继承
{
};
class C : virtual public A // 虚继承
{
};
class D : public B, public C
{

};
int main()
{
    D d;
    d.B::a = 18;
    d.C::a = 20;
    cout << d.B::a << endl; // 20(如果不是虚继承, 那就是18)
    cout << d.C::a << endl; // 20    用作用域就能轻松解决这个问题(如果不是虚继承)
    cout << d.a << endl;  // 20!(说明两个age都是一个数了)
    // 虚继承之后, a这个数据只有一个了
```

```
}
```

不是虚继承, 就使用作用域, 这一点十分好理解

但是为什么虚继承了之后, a都变成了一个数呢? 事实上, BC两个继承下来的, 其实是vbptr(虚基类指针)

这个指针指向的是vbtable(虚基类表格), 所以21, 22两行其实都是通过指针对vbtable中的数据进行了修改

所以最后显示的是20, 因为最后一次修改是22行

注意: B,C里面不能再对a进行赋值, 一旦如此, 性质就变了, 继承的就不是vbptr

多态

基本概念

多态是C++面向对象的三大特性天之一, 分为两类:

- 静态多态: 函数重载和运算符重载属于静态多态, 复用函数名
- 动态多态: 派生类和虚函数实现运行时多态

区别:

- 静态多态的函数地址早绑定 - 编译阶段确定函数地址
- 动态多态的函数地址晚绑定 - 运行阶段确定函数地址

上面的总结还是太抽象了, 下面举个例子:

```
# include <iostream>
using namespace std;
class Animal{
public:
    // virtual void speak(), 这就是解决办法!!
    void speak()
    {
        cout << "Animal is speaking" << endl;
    }
};
class Cat : public Animal
{
public:
```

```
    void speak()
    {
        cout << "Cat is speaking" << endl;
    }
};
void doSpeak(Animal &animal)
// 注意! 这里是Animal类型的引用传递, 但是Cat派生类也能传进来, 这一点我们需要知道!
{
    animal.speak();
}
int main()
{
    Cat cat;
    doSpeak(cat); // Animal is speaking
    return 0;
}
```

很明显26行的输出并不是我们所想见到的, 但是为什么呢? 因为**地址早绑定**, 在编译阶段就确定了函数的地址

更详细一点来说, doSpeak函数里面的animal.speak()一定走的时候Animal里面的speak函数, 虽然说**派生类允许传进来**, **所以一定调用的是父类Animal**里面的speak()函数

为了解决这样的事情, 我们需要地址晚绑定, 即在运行阶段中, 收到了派生类, 再绑定派生类里面"刷新过的"speak()函数, 在Animal里面的speak函数前面加上关键词virtual, 就可以解决问题了

实质: doSpeak函数里面的animal.speak()在编译阶段就实现了地址绑定, 从而造成即使**传入派生类合法**, 但是调用的永远是Animal里面的speak, 因此给这个函数加上virtual, **让它在运行至这里之前都不绑定地址**, 这样一来, **传入派生类之后, 这个函数见机行事, 就绑定了Cat.speak()函数的地址**, 从而解决了问题

当virtual加上了之后, Cat里面再对speak进行的修改就叫**重写**, 其中派生类重写的时候, 函数头前面的virtual可写可不写

总结: 动态多态满足条件:

1. **有继承关系**
2. **子类重写父类的虚函数**

动态多态使用: 父类的指针或者引用, 执行子类对象

原理剖析

重写: 函数返回值类型 函数名 参数列表 完全一致, 称为重写

底层原理是什么呢?

```
# include <iostream>
using namespace std;
class Animal{
public:
    void speak()
    {
        cout << "Animal is speaking" << endl;
    }
    virtual void sleep()
    {
        cout << "Animal is sleeping" << endl;
    }
};
class Cat : public Animal
{
public:
    void speak()
    {
        cout << "Cat is speaking" << endl;
    }
    virtual void sleeping()
    {
        cout << "Cat is sleeping" << endl;
};
void doSpeak(Animal &animal)
{
    animal.speak();
    animal.sleep();
}
int test()
{
    cout << "sizeof Animal = " << sizeof(Animal) << endl;
}
int main()
```

```
{
    test(); //1+4 = 5, 因为里面的speak函数是非静态成员, 是和类对象分开绑定的, 因此只有函数头占了一个字节以做区分      // 这里的4就是sleep函数了, 那么这里的4个字节就代表: 指针!!
    return 0;
}
```

Animal类内部, 虚函数都是以**vfptr**(虚函数指针)存储, 这些指针指向的是**vftable**, 这个table里面记录着**虚函数的地址**

`Animal::$vftable@`里面: `&Animal :: sleep()`

当子类重写父类的虚函数, **子类中的**虚函数表以及内部会替换成子类的虚函数地址(父类的table不会变化)

`Cat::$vftable@`里面: `&Animal :: sleep()`

当父类的指针或者引用指向子类对象时候, 发生多态; 传cat时, 相当于发生了: Animal &animal = cat, 因此调用的时候走Cat虚函数表, 而不会走父类的

多态优点

C++开发中提倡利用多态设计程序框架, 因为多态优点很多

多态的优点:

- 代码结构组织清晰
- 可读性强
- 利用前期和后期的扩展以及维护

纯虚函数和抽象类

在多态中, 通常父类中虚函数的实现是毫无意义的, 主要都是调用子类重写的内容

如果是这种情况, 那么就可以将虚函数改为纯虚函数

语法: `virtual 返回值类型 函数名 (参数列表) = 0;` 当类中有了纯虚函数, 我们称之为抽象类

我们可以看到这个函数直接没有函数体, 就是因为我根本用不到它, 我只是想说明, 这个抽象的东西是子类共有的

抽象类特点:

1. **无法实例化对象**
2. 子类**必须重写**抽象类中的纯虚函数, **否则也属于抽象类**

虚析构和纯虚析构

多态使用时, 如果子类中有属性开辟到堆区, 那么父类指针在释放时无法调用到子类的析构代码

解决方式: 将父类中的析构函数改为**虚析构**或者**纯虚析构**

虚析构和纯虚析构共性:

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

区别: 如果是纯虚析构, 那么该类属于抽象类, 无法实例化对象

```
# include <iostream>
# include <string>
using namespace std;
class Animal{
public:
    Animal()
    {
        cout << "The construction of Animal" << endl;
    }
    // 如果是 virtual ~Animal(), 那么Cat就可以正常析构了
    ~Animal()
    {
        cout << "The destruction of Animal" << endl;
    }
    virtual void speak() = 0; // 纯虚函数
};
class Cat : public Animal
{
public:
    Cat (string name)
    {
        cout << "The construction of Cat" << endl;
        m_name = new string(name)
    }
     virtual void speak()
```

```
    {
        cout << *m_name << "Cat is speaking" << endl;
    }
    string m_name;
    ~Cat()
    {
        if (m_name != NULL)
        {
            cout << "destruction of Cat" << endl;
            delete m_name;
            m_name = NULL;
        }
    }
};
void test()
{
    Animal * animal = new Cat("Tom"); // 用父类指针在堆区创建Cat对象
    animal->speak(); // 通过指针调用
    delete animal; // 堆取数据手动释放
}
int main()
{
    test();
    // The construction of Animal
    // The construction of Cat
    // TomCat is speaking
    // The destruction of Animal
    return 0;
}
```

发现没有destruction of Cat! 说明堆取数据根本没有释放!

问题在于: 父类指针在析构的时候(44行), 不会调用子类中的析构函数, 因此我们要把父类中的析构改为虚析构(10行)

但是注意: 不能直接仅用`virtual ~Animal() = 0;` **因为父类中也可能有需要释放的堆取数据, 而这句话压根儿没有函数体**(需要有具体的函数实现); 因此我们把它视为一种声明, 除了上面这句话之外, 在**类外**还要进行定义:

```
Animal :: ~Animal()
{
    cout << "Another kind of destruction of Animal" << endl;
}
// 附: 有了纯虚析构, 那么就会使得这个类变成抽象类, 无法实例化对象
```

总结:

1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象
2. **如果子类中没有堆取数据**, 可以不写为虚析构或纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类

C++自学——提高编程

模板

模板就是建立通用的模具, 大大提高复用性

特点:

- 模板不可以直接使用, 它只是一个框架
- 模板的通用并不是万能的

函数模板

C++另一种编程思想称为**泛型编程**, 主要利用的技术就是**模板**

C++提供两种模板机制: 函数模板和类模板

函数模板概念

函数模板作用: 建立一个通用函数, 其函数返回值和形参类型可以不具体制定, 用一个虚拟的类型来代表

```
// 函数模板语法
template<typename T>
函数声明或定义
```

其中: template 声明创建模板

typename 表明其**后面的符号是一种数据类型,** 可以用class代替

T 通用的数据类型, 名称可以替换, **通常为大写字母(别的大写字母也可以, 但是T更为常用)**

```
# include <iostream>
using namespace std;
// 实现两个整形,浮点型的函数模板
template<typename T>
void mySwap(T &a, T &b) // 注意是引用传递
{
    T temp = b;
    b = a;
```

```
    a = temp;
}
void test()
{
    int a = 10;
    int b = 20;
    double c = 1.1;
    double d = 2.2;
    // 有两种方式使用函数模板
    mySwap(a, b); // 自动类型推导
    mySwap<double>(c, d); // 显示是指定类型
    cout << a << " " << b << endl;
    cout << c << " " << d << endl;
}
int main()
{
    test();
    return 0;
}
```

函数模板注意事项

注意事项:

- 自动类型推导, **必须推导出一致的数据类型T**, 才可以使用
- 模板必须确定出T的数据类型, 才可以使用

第一点十分容易认同, 第二点举例如下:

```
template <class T> // typename可以换成class
void func()
{
    cout << "func()" << endl;
}
void test()
{
    func(); // 直接报错, 编译器不知道T是什么
    func<int>(); // 合法, 告诉了编译器, T是int
}
```

案例

案例描述:

- 利用函数模板封装一个排序函数, 可以对**不同数据类型数组**进行排序
- 排序规则从大到小, 排序算法为**选择排序**
- 分别利用char数组和int数组进行测试

```
# include <iostream>
using namespace std;
// 交换模板
template<class T>
void mySwap(T &a, T &b)
{
    T temp = b;
    b = a;
    a = temp;
}
// 排序算法模板
template <class T>
void mySort(T arr[], int len) // 两种数组
{
    for(int i = 0; i < len; i++)
    {
        int max = i;
        for (int j = i+1 ; j < len ; j++)
        {
            if(arr[max] < arr[j]) // 认定的最大值比遍历的最大值小, 说明j下标元素才是真正的最大值
            {
                max = j;
            }
        }
        if (max != i)
        {
            mySwap(arr[max], arr[i]);
        }
    }
}
template <class T>
void printArr(T Arr, int len_Arr)
```

```
{
    for (int i = 0; i < len_Arr; i++)
    {
        cout << Arr[i] << " ";
    }
}
void test1()
{
    char charArr[] = "badcfe";
    int len_charArr = sizeof(charArr) / sizeof(char);
    mySort(charArr, len_charArr);
    printArr(charArr, len_charArr);
}
void test2()
{
    int intArr[] = {7,4,5,3,9,0,1,8};
    int len_intArr = sizeof(intArr) / sizeof(int);
    mySort(intArr, len_intArr);
    printArr(intArr, len_intArr);
}
int main()
{
    cout << ('c' < 'd') << endl; // 1, 说明比较运算符能比较字符间的大小
    test1();
    cout << endl;
    test2();
    return 0;
}
```

总结: 普通函数与函数模板的区别

1. 普通函数调用可以发生隐式类型转换
2. 函数模板: 用自动类型推导, 不可以发生隐式类型转换
3. 函数模板: 用显示指定类型, 可以发生隐式类型转换

隐式类型转换例子: char c = 'c'; 在int加减中, c会转化为99(ASCII码)

普通函数与函数模板的调用规则

调用规则如下:

1. 如果函数模板和普通函数都可以实现, 那么优先调用普通函数
2. 可以通过空模板参数列表来强调函数模板
3. 函数模板也可以发生重载
4. 如果函数模板可以产生更好的匹配, 优先调用函数模板
5. 允许普通函数和模板中的函数重名

上面说的都很玄学, 下面举例:

```
# include <iostream>
using namespace std;
void Myprint(int a, int b)
{
    cout << "ordinary Myprint" << endl;
}
template <class T>
void Myprint(T a, T b)
{
    cout << "Template Myprint" << endl;
}
template <class T>
void Myprint(T a, T b, T c)
{
    cout << "Re-Template Myprint" << endl;
}
void test1
{
    int a = 10;
    int b = 20;
    float c = 1.1;
    float d = 11.4514;
    char c1 = 'x';
    char c2 = 'y';
    Myprint(a, b);
    Myprint<>(a, b);
    Myprint(c, d);
    Myprint(a, b, 100);
    Myprint(c1, c2);
```

```
}
int main()
{
    test1();
    // ordinary Myprint     说明如果函数模板和普通函数都可以实现，那么优先调用普通函数
    // Template Myprint     说明可以通过空模板参数列表来强调函数模板
    // Template Myprint     这是因为只有模板中的能和它匹配
    // Template Myprint     说明函数模板也可以发生重载
    // Template Myprint     虽然传入两个字符，普通函数也可以调用(隐式类型转换)，但是明显模板更好
    // 说明如果函数模板可以产生更好的匹配，优先调用函数模板
    return 0;
}
```

模板的局限性

模板通用性并不是万能的, 例如:

```
template <typename T>
void f(T &a, T b) // 给a赋值
{
    a = b;
}
```

在上述的代码中, 如果a, b都是数组的话, 那就无法实现了

或者说:

```
template <class T>
void same(T &a, T &b)
{
    if (a == b)
    {
        cout << "They are the same" << endl;
    }
    else
    {
        cout << "They are not the same" << endl;
    }
}
```

上面如果传入的是Person自定义数据类型, 那就GG了

```
class Person
{
public:
    Person(int num)
    {
        age = num;
    }
    int age;
};
template <class T>
void same(T &a, T &b)
{
    if (a == b)
    {
        cout << "They are the same" << endl;
    }
    else
    {
        cout << "They are not the same" << endl;
    }
}
template<> void same(Person &a, Person &b)
{
    if (a.age == b.age)
    {
        cout << "They are the same" << endl;
    }
    else
    {
        cout << "They are not the same" << endl;
    }
}
```

相当于告诉了模板: 如果数据类型是Person, 编译器应该怎么操作

总结:

- 利用具体化的模板, 可以解决自定义类型的通用化
- 学习模板并不是为了写模板, 而是在STL能够运用系统提供的模板

类模板

这个暂时不接触, 先继续推进之后的内容

STL初识

一些基本的东西

STL的诞生:

- 长久以来, 软件界一直想建立一种可重复利用的东西
- C++的面向对象和泛型编程思想, 目的就是复用性的提升
- 大多情况下, 数据结构和算法都未能有一套标准, 导致被迫从事大量重复工作
- 为了建立数据结构和算法的一套标准, 诞生了STL

基本概念:

- Standard Template Library 标准模版库
- STL广义上分为: 容器(container), 算法(algorithm), 迭代器(iterator)
- 容器和算法之间通过迭代器进行无缝连接
- STL几乎所有的代码都采用了模板类或者模板函数

六大组件:

1. 容器: 各种数据结构: 如vector, list, deque, set, map
2. 算法: 各种常用的算法: sort, find, copy, for_each
3. 迭代器: 扮演了容器和算法之间的胶合剂
4. 仿函数: 行为类似函数, 可作为算法的某种策略
5. 适配器: 一种用来修饰容器或者仿函数或迭代器接口的东西
6. 空间适配器: 负责空间的配置与管理

其中, 常见的数据结构: 数组, 链表, 树, 栈, 队列, 集合, 映射表

这些容器分为两种:

- 序列式容器: 强调值的排序, 序列式容器中的每个元素均有固定的位置
- 关联是容器: 二叉树结构, 各元素之间没有严格的物理上的顺序关系

算法分为:

- 质变算法: 指运算过程中会更改区间内的元素的内容, 例如拷贝, 替换, 删除
- 非质变算法: 指运算过程中不会更改区间内的元素的内容, 例如查找, 计数, 遍历, 寻找极值

迭代器: 提供一种方法, 使之能过依序寻访某个容器中所含的各个元素, 而又无需暴露该容器的内部表示方式

每个容器都有自己专属的迭代器, 这种迭代器非常类似于指针, 初始阶段我们可以理解为指针

STL内容很多, 此处只学习string和vector容器

string

基本概念

本质: string是C++风格的字符串, 而string本质上是一个类

string和char*的区别:

- char* 是一个指针
- **string 是一个类**, 类内部封装了char, 管理这个字符串, 是一个*char*型的容器

特点: string类内部封装了很多成员方法

string构造函数

构造函数原型:

`string(); // 创建一个空的字符串`

`string(const char* s); // 使用字符串s初始化`

`string(const string& str) // 使用一个string对象初始化另一个string对象`

`string(int n, char c) // 使用n个字符c初始化`

```
# include <iostream>
# include <string>
using namespace std;
void test()
{
    string s1; // 默认构造, 相当于string s1();
    const char* str = "Hello World";
```

```
    string s2(str);
    cout << s2 << endl;
    string(s2); // 类似于拷贝构造
    cout << s3 << endl;
    string s4(4, 'a');
    cout << s4 << endl;
}
int main()
{
    test();
    return 0;
}
```

string赋值操作

功能描述: 给string字符串进行赋值

```
string& operator=(const char* s); //char*类型字符串赋值给当前字符串
string& operator=(const string &s); // 把s赋给当前的字符串
string& operator=(char c); // 把当前字符赋给当前的字符串
string& assign(const char* s); // 把字符串s赋值给当前字符串
string& assign(const char* s, int n); // 把当前字符串的前n个赋值给字符串
string& assign(const string &s); // 把字符串s赋给当前字符串
string& assign(int n, char c); // 把n个字符赋给当前的字符串
```

```
string s1, s2, s3, s4, s5, s6, s7;
s1 = "Hello World";
s2 = s1;
s3 = 'a';
s4.assign("Hello World");
s5.assign(s1, 5);
s6.assign(s1);
s7.assign(5, "Hello World")
```

这里的operator都强调了是等号, 因为其他的符合有不同的含义!

字符串的拼接

```
string s1 = "I";
s1 += " am handsome";
cout << s1 << endl; // I am handsome
s1 += "!";
string s2 = " I'm sure.";
s2 += s1;
cout << s2 << endl; // I am handsome. I'm sure
string s3 = "Long";
string s4 = "Long"
s3.append(" time no see.");
s4.append(" time no see.", 5); // Long time
s3.append(s2); // Long time no see. I'm sure
s5 = "I";
s5.append(s1, 0, 2); // I am
// 代表引索从0到2的字母截进去
```

string查找和替换

功能描述:

- 查找: 查找指定字符串是否存在
- 替换: 在指定的位置替换字符串

```
# include <string>
void test()
{
    s1 = "abcde";
    cout << find("de") << endl; // 3,因为d首次出现的位置的引索是3
    cout << find("df") << endl; // -1, 代表没找到
    // 这个-1值可以用于帮助设置判断条件: if(find(...) == -1)
    // rfind & find区别: rfind从右往左查找, find从左往右
    // 影响的是最先发现的位置! 但是返回的引索依然还是"排头位置"
    cout << s1.replace(1,3,"111") << endl; // a111e
    // replace在替换的时候, 要指定从哪个位置起, 多少个字符, 替换成什么样的
字符串
}
```

string字符串比较

比较方式:

- 字符串比较是按照字符的ASCII码进行对比的

= 返回0　>返回1　<返回-1

`str1.compare(st2)`

字符存取

```
void test()
{
    string str = "hello";
    for(int i = 0; i < str.size(); i++)
    {
        cout << str[i] << " ";
    }
    cout << endl;
    for(int i = 0; i < str.size(); i++)
    {
        cout << str.at(i) << " ";
    }
    cout << endl;
    str[1] = 'a';
    cout << str << endl; // hallo
}
```

image-20240212210628951

```
void test()
{
    string str = "hello";
    str.insert(1, "111");
    cout << str << endl; // h111ello
    str.erase(1, 3);
    cout << str << endl; // hello
}
```

子串获取

`str.substr(int pos = 0, int n = npos) // 后面一个代表截的字符个数`

```
void test()
{
    string str = "zhangsan@email.com";
    int pos = str.find('@')
    cout << str.substr(0, pos) << endl; // zhangsan
}
```

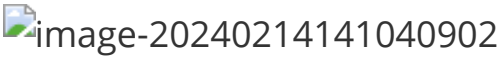
vector容器

基本概念

vector 数据结构和数组非常相似, 也称为单端数组; 但是有一个非常大的特点:

- 数组是静态空间, 但是vector可以**动态扩展**
- vector容器的迭代器是**支持随机访问的迭代器**

动态扩展: **并不是在原空间之后续接新空间, 而是找更大的内存空间, 然后将原数据拷贝新空间, 释放原空间**



vector构造函数

```
# include<vector>
void test()
{
    vector<int>v1; // 默认构造
    for(int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    vector<int>v2(v1.begin(), v1.end()); // 通过区间方式进行构造
    // begin代表首元引索, end代表尾元引索
    vector<int>v3(10, 100); // 10个100
    vector<int>v4(v3); // 拷贝构造
}
```

vector复制操作

功能描述:

- 给vector容器进行赋值

```
# include<vector>
void printVector(vector<int>& v) // 引用传递
{
    for(vector<int>::iterator it = v.begin(); it != v.end();
it++)
    // 这个迭代器多写写就熟练了, 且注意是end,不是rbegin
    {
        cout<< *it << " ";
    }
    cout << endl;
}
void test()
{
    vector<int>v1; // 默认构造
    for(int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);
    vector<int>v2;
    v2 = v1; // operator=构造
    vector<int>v3;
    v3.assign(v1.begin, v1.end()); // assign分配
    vector<int>v4;
    v4.assign(10, 100); // 10个100
}
```

vector容量和大小

功能描述:

- 对vector容器的容量和大小操作

`empty()` `capacity()` `size()` `resize(int num)` `resize(int num, elem)`

```
# include<vector>
void printVector(vector<int>& v)
{
    for(vector<int>::iterator it = v.begin(); it!=v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}
void test()
{
    vector<int>v1;
    {
        v1.push_back(i);
    }
    if(!v1.empty()) // 这个方法返回的是bool值, ! 代表非
    {
        printVector(v1);
    }
    cout << v1.capacity() << endl; // 这个数是随机的一个比10大的数字
    // 因为动态扩展, 使得v1接受了10个数据之后, 自动找到了一块比10大的内存空
间
    cout << v1.size() << endl; // 10, 因为返回的是"真正"的大小
    v1.resize(15);
    printVector(15); // 后面5个用0填充
    v1.resize(20, 9); // 指定填充的内容
    printVector(v1); // 后面再加5个9
}
```

vector插入和删除

```
# include<vector>
void test()
{
    vector<int>v1;
    v1.push_back(10); // 尾部插入10, 有点像python的append
    v1.pop_back(); // 尾部删除10
    for(int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
```

```
    v1.insert(v1.begin(), 100); // 在首元的前面插入100; 第一个参数是迭代器
    v1.insert(v1.rbegin(),2,1000); // 在尾元的前面插入两个1000, 注意不是屁股
    // 如果是屁股, 那么就是v1.end()
    v1.erase(v1.begin()); // 把首元删掉了
    v1.erase(v1.begin(), v1.rbegin()); // 除了最后一个元素保留, 其他全部删除
    // 当然如果是完全删除, 那么就v1.end(), 因为后面一个迭代器参数是开区间, 取不中!
    v1.erase(); // 完全删除
}
```

vector数据存取与互换容器

```
# include <vector>
void test()
{
    vector<int>v1;
    for(int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    for(int i = 0; i < v1.size(); i++)
    {
        cout << v1[i] << " " << v1.at(i) << endl; // 两种方法访问
    }
    cout << v1.front() << " " << v1.end() << endl;
}
```

而实现两个容器内元素进行互换, 需要使用:

`v1.swap(v2);` // 将v2与v1的元素进行互换

那么这个东西究竟有什么用呢? 实际用途: **收缩内存空间**

```
# include<vector>
void test()
{
    vector<int>v;
    for(int i = 0; i<100000; i++)
    {
```

```
        v.push_back(i);
    }
    cout << v.capacity() << endl; // 十万多，多出来了很多预留内存
    cout << v.size() << endl; // 100000
    v.resize(3); // 重新指定大小，我只想用前三个数据！
    cout << v.capacity() << endl; // 和上面的capacity一样大，大量内存浪费掉了！
    cout << v.size() << endl; // 3
    vector<int>(v).swap(v);
}
```

那么发生了什么? `vector<int>(v)`代表的是用v拷贝构造一个匿名vector, 这个vector和v进行了交换, 然后占据了大量内存空间的匿名vector在这行代码结束之后被自动释放

vector的预留空间

之前提到, 如果进行vector插入或者删除, 内存会动态变化; 我不希望它动来动去, 直接声明我要求划出一块内存让我操作:

`vec.reserve(int len)`, **预留**len个元素长度, 预留位置不初始化, 元素不可访问

```
# include <iostream>
# include <vector>
using namespace std;
void test()
{
    vector<int>v;
    // v.reserve(100000); 如果加了这句话，下面输出就是1
    int *p = NULL;
    int num = 0;
    for(int i = 0; i < 100000; i++)
    {
        v.push_back(i);
        if (p != &v[0])
        {
            p = &v[0];
            num ++;
        }
    }
```

```
    cout << "Times of changes: " << num << endl; // 一般来说是两位数; 如果有了reserve, 那么就是1
}
int main()
{
    test();
    return 0;
}
```

因此总结: 如果数据量较大, 那么一开始就可以利用reserve预留空间