

1. I/O:

C 中的 Stream 是 FILE*。缓冲发生在一个数据结构中，而在 C 中，流就是一个指向此数据结构的指针，每个流都带一个描述符上

C 中的黑人认识流：stdin (描述符 0)，stdout (描述符 1)，stderr (描述符 2)，可自定义：FILE* my_file;
 (类型均是 FILE*)

文件 I/O: 文件路径 (以谁是当前工作路径开始) 操作模式
 FILE* fopen (const char* path, const char* mode);
 "r", "rb" → read only
 "w", "wb" → write only
 "a", "ab" → append (write only, at end)
 "r+", "r+b", "rb+" → r/w, 不读从头写
 "w+", "wt", "wb+", "wb+" → r/w, 清空文件从头写
 "a+", "at", "ab+" → r/w, append

黑人认识文件描述符可在程序开始时被重载，例如可用 FILE* fopen (const char* path, const char* mode);
 等待关闭的流 → 转出文件关闭可能失败，转向文件本身
 关闭流：int fclose (FILE* stream);
 return 0 on success, EOF(-1) on failure
 没用完了一定记得关 ★ 程序能用的数量是有限的 打开/读取失败：①释放空间
 ②关闭流

五种 I/O: ① 一次 r/w 一个字符：
 成功：返回读入的字符 (2 EXT), 失败：return EOF (-1)
 int fgetc (FILE* stream); int getc (FILE* stream);
 成功：返回要写入的字符
 int putc (int c, FILE* stream); int pputc (int c, FILE* stream);
 成功：返回 character written (2 EXT from low 8 bits of c), 失败：返回 EOF (-1)

若 stream 为 stdin/stdout，可用简写：
 int getchar (void); int putchar (int c);
 返回值同上 返回值同上

② r/w 字符串：
 成功：return s, 失败：NULL
 读入数据应存到字符串数组 S 的大小
 输入流
 ① end of input (文件结尾)
 ② end of line (ASCII 0x0A 或 0x0D)
 ③ end of array s (会为字符串 NUL 留位置)
 int fgets (char* s, int size, FILE* stream); fgets 停止读入
 成功：return 非空串, 失败：EOF(-1)
 要输出的字符串
 要输出流
 → 不会自动在 S 结尾执行

若为 stdout 转写，简写：int puts (const char* s); 向 stdout 转出字符串，并在结尾加上换行符 0x0A，而 fputs 不会加换行符
 永远不要用 scanf for reading a string from stdin

③ formatted I/O: 对于 stdin/stdout：int scanf (const char* format, ...); → return 成功：值的个数
 int printf (const char* format, ...); → return 成功：返回写的字符串总长度
 失败：EOF(-1)
 成功：返回一个负数

int fscanf (FILE* stream, const char* format, ...);
 return 成功：number of conversions, 失败：(-1)
 &a, &b 佔地址位

int fprintf (FILE* stream, const char* format, ...);
 return 成功：打印总字符串，失败：-1 个数

④ Binary I/O:
 size_t fread (void* ptr, size_t size, size_t n_elt, FILE* stream);
 指向读入数据存储器的位置
 一个 "thing" 的大小
 "things" 的个数
 return 成功：返回读入 "things" 的个数，失败：返回 0

size_t fwrite (const void* ptr, size_t size, size_t n_elt, FILE* stream);
 指向待写入位置的地址
 一个 "thing" 的大小
 "things" 的个数
 return 成功：返回写的 "things" 的个数，失败：返回 0

⑤ 字符串 I/O:
 int sscanf (const char* s, const char* format, ...);
 从 s 读入内容, string from which to read
 return 成功：number of conversions, -1 on failure
 &a, &b 佔地址位

int snprintf (char* s, size_t size, const char* format, ...);
 转出到的字符串, the array which to write
 return 成功：打印的总字符串，失败：返回一个负数

2. 可变参数列表 #include <stdarg.h>

va_list args; args 为 va_list 类型的参数指针

va_start (args, 最后一个有名参数的名称); 将第一个可变参数地址传入 args

va_arg (args, 数据类型); 返回 args 31 用的参数值，并将 args 移向下一参数
 ↑ 当前被 args 31 用参数的数据类型

3. 命令行参数

int argc → 命令行参数的数量 (包含程序名)，不计算 argv 结尾的 NULL
 const char* const argv[] → array of arguments [length = argc, 开头元素
 为程序名, NULL 空指针结束]
 const char* const envp[] → NULL-terminated array of 环境变量

Examples of Arguments Passed to main

What does main receive if you type ...

```
./word_split ?  
argc is 1, argv is ("./word_split", NULL)  
./word_split word_split.c ?  
argc is 2, argv is ("./word_split", "word_split.c", NULL)  
./word_split a b c d ?  
argc is 5, argv is ("./word_split", "a", "b", "c", "d", NULL)
```

argc 7, it's length is 5

ECE 220: Computer Systems & Programming

© 2018 Steven S. Lumetta. All rights reserved.

slide 13

Graphical Interfaces Produce Arguments to main

What about graphical interfaces?

Double-clicking on an application produces an argc of 1.

mainfile ⇒ 1

Double-clicking on an associated file produces an argc of 2 (the program and the file).

Main file + Associated file ⇒ 2

Dropping N files into an application icon produces an argc of N + 1.

mainfile + main program ⇒ N + 1

ECE 220: Computer Systems & Programming

© 2018 Steven S. Lumetta. All rights reserved.

slide 14

调用系统函数失败可使用 void perror (const char* prefix); 输出错误信息 (prefix是在错误信息输出前先输出的东西)

在 Shell 中, program1 | program2 ⇒ 1 的输出作为 2 的输入

```
./line-sort < list-of-word ⇒ < 将 stdin 重定向为一个输入文件  
ls *grade.txt > grade-files ⇒ > 将 stdout 重定向为一个输入文件, 从头写入, 若想 append stdout, 用 "">>"  
program1 |& program2, program1 > output, & 将 stderror 重定向输出  
program1 >> output 2>> 1, 2>> 1 代表 stderr(2) 复制到 stdout(1) 前字 1, 2 代表描述符
```

4. 函数指针

int32_t func (double d, char* s); &func 的类型为 int32_t (*)(double, char*), 由于历史原因, 表达式 func 也是一样的, 但类型为 ↑

② i) Call back : int32_t isort (void* base, int32_t n_elt, size_t size, int32_t (*is_smaller) (void* t1, void* t2));
使用: isort (d_array, 8, sizeof (d_array[0]), &double_is_smaller); → 使用时 (*is_smaller)(a, b) → 表达式为 (*is_smaller) (void* t1, void* t2);

5. 类与对象

① static 变量在类内的声明并不会为其开辟存储空间, 自然也不能对其初始化

static 变量必须在类外再声明一次, 可在类外初始化

② 派生类的析构函数可直接为基类指针, 反之不行 ✗

③ virtual 限制符不会返回传播到父类 ✗ 如果只在子类 MyClass 添加, 通过 ParentClass* 类型的 ptr 调用的仍是父类的 aFunc()

④ 访问控制: private: 只限①类成员与类成员函数 ②类定义内的代码 ③友元类与友元函数
(默认为 private)

protected = private + 派生类函数可访问

public: 谁都可以访问

⑤ 构造函数: 调用时机
Automatic: 声明时调用, static: 运行 main 之前调用, Dynamic: 在 allocation 时调用
(无返回值)

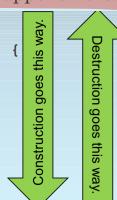
对于需从以参数构造函数, 一个类的实例若作为另一个类的字段, 则这个字段是由父类构造函数初始化的, 这种情况下, 父类构造函数必须有 copy (非类实例的字段如 int 类型的字段会 left as bits) 对于默认的复制构造函数, 父类的字段 are copied using the copy constructor, 这种情况下, 此字段的构造函数必须有。

⑥ 析构函数:

Destructors do the following...

1. Execute the body of the destructor.
2. Call destructors for all fields that are instances, in reverse order of declaration in the class definition.
3. Call destructors for base classes, if any.
4. (For dynamically-allocated instances, the deallocation happens here.)

```
class MyClass :  
    public ParentClass {  
        int x;  
        AnotherClass y;  
        double z;  
        // ...  
    }  
Code from constructor/destructor.)
```



调用时机

Destructor Bodies Must Perform Deallocation

Do not call destructors for fields that are instances; such calls are made automatically.

Fields that are pointers to instances are not implicitly destroyed.

If a field

- points to an instance
- that should be destroyed
- when the instance containing the field is destroyed,

the destructor body (the code) must perform that deallocation explicitly.



pointer 指向的内容不会被隐式地析构, 需在析构函数的 Body 中自己写 ✗

Automatic: 作用域结束, static: main 执行完了之后, Dynamic: 在 deallocation 时

时

6. new 和 delete

对于 new 若想使用无参构造函数却初始化字段，则不应带扩号(√)

Use `delete` to Deallocate Instances, `delete[]` for Arrays

Given `MyClass* m,`
◦ `delete m;` // deletes an instance
◦ `delete[] m;` // deletes an array

Before the memory is freed, destructors (with no arguments) are called on all instances.

As with modern C,
◦ deleting NULL has no effect, but
◦ deleting a “pointer” of uninitialized bits is problematic.

是 `delete [] m;` ✎
注意呢。

在 modern C 中
可以 `delete NULL`
但不能 `delete a "pointer" of uninitialized bits`

Initialization Rules Can Be Convoluted

Did you notice that I said that parentheses had to be omitted to get the constructor with no arguments?

In certain cases, C++ applies “value-initialization”:

```
int32_t i();  
int32_t i = int32_t (); // avoid  
MyClass* m = new MyClass ();  
// iff default no args constructor  
// is available; user-def'd is called
```

Value-initialization zeroes all non-instance fields, then calls constructors for base classes and instance fields.

若想使用无参构造函数却初始化字段，则不应带扩号(√)

若想让编译器将非常类实例的字段先置零，调用 Base Class 和类的对象的构造函数，就带上括号(不推荐)

7. 重载 运算符的参数应为常引用(避免拷贝，且若没有 Const，引起类型转换会失败)

2) 如果某函数有一个参数是类 A 的对象，那么该函数被调用时，类A的复制构造函数将被调用。

3) 如果函数的返回值是类A的对象时，则函数返回时，A的复制构造函数被调用：

进 main 前： 全局变量构造

出 main 前： 局部变量析构

出 main 后： 全局变量析构

变量的初始化顺序就应该是：

1 基类的静态变量或全局变量

2 派生类的静态变量或全局变量

3 基类的成员变量

4 派生类的成员变量

① 对象数组生命周期结束时，对象数组的每个元素的析构函数都会被调用。

② delete 运算导致析构函数调用。

③ 析构函数在对象作为函数返回值返回后被调用

☆ 注意：① 任何函数都有可能失败，都要判断 ✎

② `fclose (in)` 不会失败，但 `fclose (out)` 可能会失败

③ 负责从右向左结合

```
while (2 == fscanf (in_file, "%d%d", &a, &b)) {  
    if (0 > fprintf (out_file, "%d\n", (* (func_arr [func_index])) (a, b))) {  
        fclose (in_file);  
        fclose (out_file);  
        return -1;  
    }  
}
```

```
if (NULL == (in = fopen (fname, "r")) ||  
NULL == (out = fopen ("out.txt", "w"))) {  
    if (NULL != in) {  
        fclose (in);  
    }  
    return 0;  
}
```

int last = EOF, char;

```
while (EOF != (char = fgetc (in))) {  
    if (last != char) {  
        fputc (char, out);  
        last = char;  
    }  
}
```

```
fclose (in);  
return (0 == fclose (out) ? 1 : 0);
```

尝试可用注意数据大小 ✎ | 有 overflow 记得写 mod 2^n | 失败产生的垃圾要及时 free 掉

realloc 成功，指针数组不要更新 | 对递归的边界条件一定要仔细考虑，代入各种边界情况看会不会出错 ✎

Addition in Theory Implies Three Constructors

`complex b = a + a;`

What should happen?

In theory, an `instance`

- is constructed within operator+, then
- copy constructed as the return value,
- then copy constructed again as b.

函数返回新对象的值
理论上会用到了三次构造函数
① 构造
② 复制构造
③ 复制构造

Addition in Practice Usually Calls One Constructor

`complex b = a + a;`

What should happen?

In practice,

- variable b resides in the caller's stack frame,
- so a pointer to b is passed to `operator+`,
- and `operator+ constructs b`.

No Copy Constructor!!!

函数返回存入空间不需要用到复制构造函数