1、I/O：

C中的stream是FILE*。缓冲会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，每个流构建在个描述符上

C中的默认流：stdin（描述符0），stdout（描述符1），stderr（描述符2），可自定义：FILE* my_file;
（类型均为FILE*）keyboard　display (normal)　display (error)

文件I/O：
默认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);`
- path：文件路径（从程序当前工作路径开始）
- mode：操作模式
- ↳ 返回 new stream 则成功，失败返回 NULL

操作模式：
- "r", "rb" → read only ⎫ 该文件必须存在
- "w", "wb" → write only ⎭
- "a", "ab" → append (write only, at end)
- "r+", "r+b", "rb+" → r/w，不清空从头写
- "w+", "w+b", "wb+" → r/w，清空文件从头写
- "a+", "a+b", "ab+" → r/w, append　读取从头开始，写入为追加

关闭流：`int fclose (FILE* stream);`（stream：待关闭的流）
return 0 on success，EOF(-1) on failure
→ 输出文件关闭可能失败，在输入文件不会
流用完了一定记得关☆ 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：`int fgetc (FILE* stream);`　`int getc (FILE* stream);`
成功：返回读入的字符（ZEXT），失败：return EOF (-1)

`int putc (int c, FILE* stream);`　`int putc (int c, FILE* stream);`（c：要写入的字符）
成功：返回 character written (ZEXT from low 8 bits of c)，失败：返回 EOF (-1)

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

② r/w字符串：`char* fgets (char* s, int size, FILE* stream);`
- 成功：return s，失败：NULL
- s：读入数据应存到的字符数组
- size：s的大小
- stream：输入流

fgets停止读入：
- ① end of input（如文件结尾）
- ② end of line (ASCII 0x0A 或 0x0D)
- ③ end of array s（会为空字符 NUL 留位置）

or 读取n-1个字符，并在最后加个null字符

`int fputs (const char* s, FILE* stream);` → 不会自动在s结尾换行
成功：return 非负数，失败：EOF(-1)　s：要输出的字符串　stream：输出流

若为stdout输出，简化：`int puts (const char* s);` 向stdout输出字符串，并在结尾加上换行符 0x0A，而fputs不会加换行符
↳ 会自动在s结尾换行☆
永远不要用 shortcut 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（成功匹配并已赋值的）；文件结尾/失败：EOF (-1) }
`int printf (const char* format, ...);` → return { 成功：返回写入的字符总数；失败：返回一个负数 }

`int fscanf (FILE* stream, const char* format, ...);` → &a, &b 传地址☆
↳ return 成功：number of conversions，失败：(-1)

`int fprintf (FILE* stream, const char* format, ...);`
↳ return 成功：打印总字符数，失败：一个负数

④ Binary I/O：`size_t fread (void* ptr, size_t size, size_t n_elt, FILE* stream);`
- ptr：指向读入数据存储的位置
- size：一个"thing"的大小
- n_elt："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)`
- ptr：指向待写入位置的地址
- size：一个"thing"的大小
- n_elt："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, ...);`
- s：从s读入内容，string from which to read
- ...：&a, &b 传地址☆
- ↳ return 成功：number of conversions，-1 on failure

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

2、可变长参数列表 #include <stdarg.h>

`va_list args;` args为va_list类型的参数指针

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

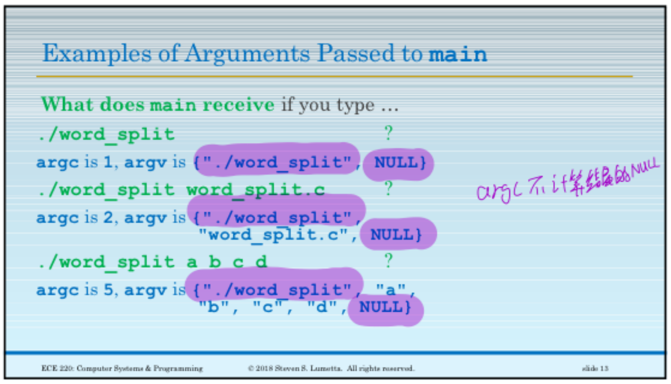
`va_arg (args, 数据类型);` 返回当前args引用的参数值，并将args移向下一参数
（数据类型：当前被args引用参数的数据类型）

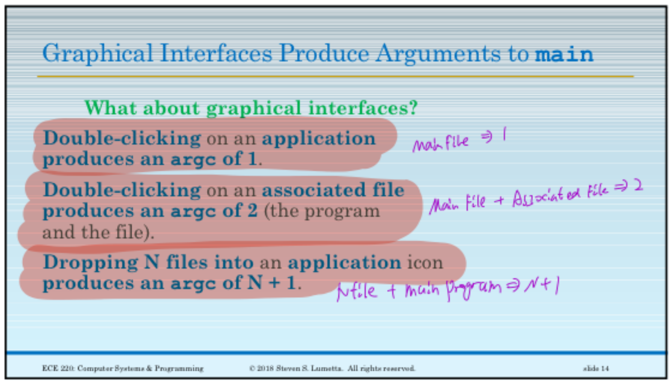
3、命令行参数

`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，开头是程序名，NULL空指针结尾)

`const char* const envp[]` → NULL-terminated array of 环境变量





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字面)

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　word 作为 sort 的输入

(该命令的输出) ls */grend.txt > grade_files ⇒ > 将 stdout 重定向为一个输入文件，从头写入，若想 append (追加) stdout，用 ">>"

program1 |& program2，program1 >& output，& 将 stderr 包含进输出

program1 >> output 2>&1，2>&1 代表 stderr (2) 复制到 (追加) stdout (1) 数字 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 int32_t (double char*) ↑

回调 Call back：int32_t isort (void* base, int32_t n_elts, size_t size, int32_t (*is_smaller) (void* t1, void* t2));
→ 使用时 (*is_smaller)(a, b) → 函数名为 (*is_smaller) 解引用加括号

使用：isort (d_array, 8, sizeof (d_array[0]), &double_is_smaller);

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 派生类指针可隐式转换为基类指针，反之不行☆

③ virtual 限定符不会逆向传播到父类☆ 如果只在子类 MyClass 添加，则通过 Parent Class* 类型的 ptr 调用的仍是父类的 aFunc( )

④ 访问控制：private：可被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

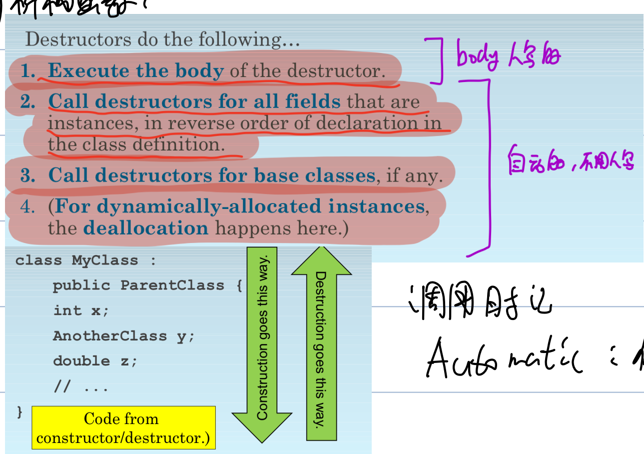
public：谁都可以访问

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

对于默认无参构造函数，一个类的实例若作为另一个类的字段，则这个字段是由无参构造函数初始化的，这种情况下，无参构造函数必须存在☆

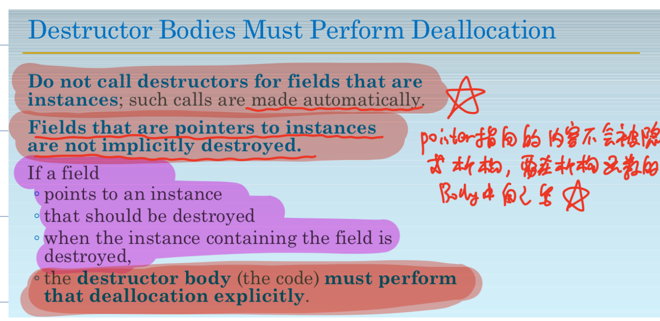
(非类实例的字段例如 int 类型的字段会 left as bits) | 对于默认的复制构造函数，类实例的字段 are copied using the copy constructor，这种情况下，此字段复制构造函数必须存在。

⑥ 析构函数：



body 人写的

自动的，不用人写



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

调用时机

Automatic：作用域结束，static：main 运行完了之后，Dynamic：在 deallocation 时调用

6. new 和 delete

对于 new 若想使用无参构造函数初始化字段，则不应带括号 (✓)

Use delete to Deallocate Instances, delete[] for Arrays

Given MyClass* m,

- delete m; // deletes an instance
- delete[] m; // deletes an array

是 delete [] m; ☆ 注意顺序.

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.

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

ECE 220: Computer Systems & Programming © 2018 Steven S. Lumetta. All rights reserved. slide 7

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初始化字段，则不应带括号 (✓)

若想让编译器将非类的实例的字段先置零，调用 Base Class 和实例对象的构造函数，就带上括号 (不推荐)

ECE 220: Computer Systems & Programming © 2018 Steven S. Lumetta. All rights reserved. slide 8

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

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 赋值是从右向左结合

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

```
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;
    }
}
```

```
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);
```

考试可用注意数据大小☆ | 有 overflow 记得写上，还要写 mod 2^n | 失败产生的垃圾要及时 free 掉

realloc成功，指针、数组大小要更新 | ☆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.

函数返回并初始化实例 理论上会调用3次构造函数

① 构造

② 复制构造

③ 复制构造

ECE 220: Computer Systems & Programming © 2018 Steven S. Lumetta. All rights reserved. slide 31

Addition in Practice Usually Calls One Constructor

```
complex b = a + a;
```

What should happen?

In practice,

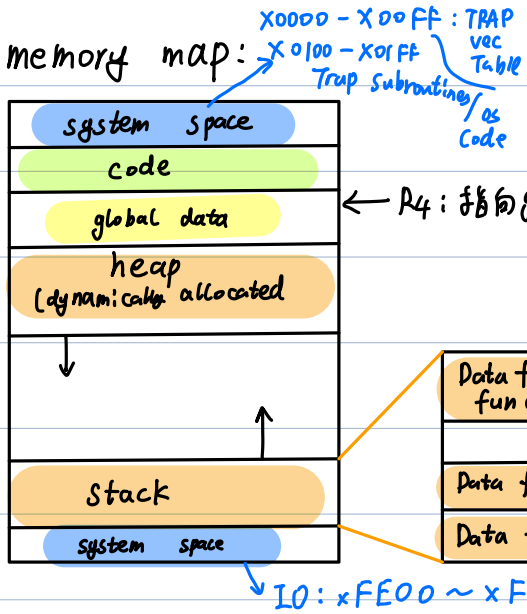
- variable b resides in the caller's stack frame,
- so a pointer to b is passed to operator+,
- and operator+ constructs b.

No Copy Constructor!!!

函数返回存入变量不需要调用复制构造函数

ECE 220: Computer Systems & Programming © 2018 Steven S. Lumetta. All rights reserved. slide 32

19. memory map:



21、全局变量可永限自动初始化为0，局部变量无法自动初始化

22、LC-3 Stack frame：（栈帧底部）
⑤ Local variable
④ Address of caller's stack frame
③ Return address (R7 in LC-3)
② Outputs (Return value)
① Inputs (parameters, arguments)（栈帧底部）

Caller负责将函数参数压入栈(①)，Callee负责②~⑤

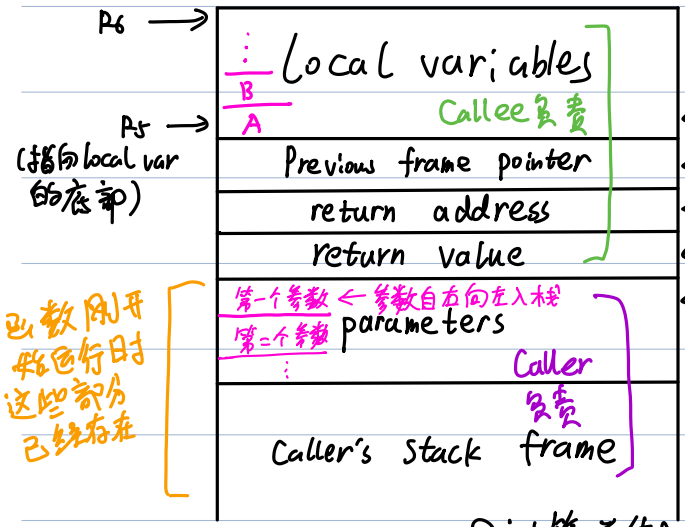
☆2

返回时，③~⑤ Pop出栈，

Return value 刚在栈顶

Caller可用R6直接访问返回值

23、Stack Map



返回前弹出，栈顶为返回值

☆

（参数从右向前push入栈）

Note：全局变量通过R4调用(R4+0, R4+1, …)
局部变量通过R5调用(R5-0, R5-1, …)

24、调用函数步骤：①计算并传入参数(将参数压入栈)②调用函数③从栈顶读取返回值④返回值和传入参数出栈

25、函数代码4部分：①设置栈帧②运行程序③拆除栈帧④返回return

栈帧设定方法：
```
ADD R6, R6, #-4 → make space for the remainder of the stack frame
STR R5, R6, #1  → save caller's frame pointer into stack frame
ADD R5, R6, #0  → set frame pointer
STR R7, R5, #2  → save return address into stack frame
```

返回值存储：
```
LDR R0, R5, #?
STR R0, R5, #3
```

返回步骤：
```
LDR R7, R5, #2 → restore return address from the stack frame
LDR R5, R5, #1 → restore caller's frame pointer from the stack frame
ADD R6, R6, #3 → Pop down to return value
RET
```

26、如果一个子程序本身还调用其它子程序，则必须要保存R7 ☆

27、Static变量仅初始化一次，之后保留上一次运算后的值 ☆

28.

A	B	C	D	E	F
10	11	12	13	14	15
1010	1011	1100	1101	1110	1111

别漏分号在语句结束！！！

☆

(NOT A = xFFFF - A , -A = x10000 - A = xFFFF - A + 1)

Final

Short Answer

LABEL may be **too far away** for a 9-bit offset (LD). Sequence 2 works for any memory location.

ALPHA* a cannot be cast safely to **BETA* -- *a** might not be a **BETA**!

```
void applyRotation (float t, float p, ALPHA* a)
{
    BETA* b = a; // problem is here
    b->rotate3D (t, p);
}
```

subroutine A executes **JSR** without **saving R7** —infinite loop!

does not wait for display to be ready

```
        .ORIG x3000
        LD R0,NUM5
        STI R0,DDR
        HALT
NUM5    .FILL x35     ; ASCII digit '5'
DDR     .FILL xFE06
        .END
```

C++ program crashes after **main** has returned

crashes in destructor (for variable in static storage)

```
#include <stdio.h>
int weird () {
  printf ("weird");
  return 0;
}
int run () {
  char buffer[10];
  scanf ("%s", buffer);
  return 0;
}
int main() {
    run ();
    printf ("main");
    return 0;
}
```

In response, the program prints out "weird" instead of "main", then terminates. Based on your knowledge of the LC-3 calling convention

Special InputTM overwrote return address on stack with address of weird

```
int player_sort_by_rank (const void* p1, const void* p2)
{
    int32_t r1 = player_get_rank (p1);
    int32_t r2 = player_get_rank (p2);

    if (r1 > r2) { return -1; }
    if (r2 > r1) { return  1; }
    return 0;
}
```

Calculate rank once for each player and store in a new field of player_t

```
class ALPHA {
    private:
        int val;
    public:
        ALPHA (int start) : val (start) { }
        void add (int amt) { val += amt; }
        void add (double amt) { add (ceil (amt)); }
        int value (void) { return val; }
};

int
main ()
{
    ALPHA a (40);

    a.add (1.5);

    printf ("%d\n", a.value ());

    return 0;
}
```

infinite recursion to ALPHA::add with double argument

```
typedef struct book_t book_t;
struct book_t {
   // some stuff
   book_t* next;              // for the library
};

typedef struct good_book_t good_book_t;
struct good_book_t {
   book_t base;
   // some other stuff
   void (*promote_book) (void);  // a function pointer for good books
};
```

Not all books are good books!
(Not safe to cast book_t* to good_book_t*.)

```
typedef struct 3D_point_t 3D_point_t;
struct 3D_point_t {
    int32_t      x, y, z;  // coordinates of point
    double_list_t dl;      // for list of points
};
```

double_list_t field must be first in 3D_point_t!_

LC-3 Stack

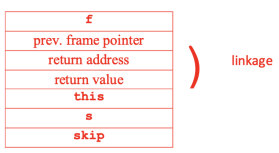
```
class ALPHA {

private:                          ALPHA::func
    char x;

public:
    ALPHA (char _x) : x (_x) { }

    char* func (const char* s, int16_t skip) {
        const char* f;

        for (f = s; '\0' != *f; ++f) {
            if (x == *f && 0 == --skip) {
                return f;
            }
        }
        return NULL;
    }
};
```



序号	函数和描述
1	void *calloc(int num, int size); 在内存中动态地分配 num 个长度为 size 的连续空间，并将每一个字节都初始化为 0。所以它的结果是分配了 num*size 个字节长度的内存空间，并且每个字节的值都是 0。
2	void free(void *address); 该函数释放 address 所指向的内存块，释放的是动态分配的内存空间。
3	void *malloc(int num); 在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的。
4	void *realloc(void *address, int newsize); 该函数重新分配内存，把内存扩展到 newsize。

```
/* 动态分配内存 */
description = (char *)malloc( 30 * sizeof(char) );
if( description == NULL )
{
   fprintf(stderr, "Error - unable to allocate required memory\n");
}
else
{
   strcpy( description, "Zara ali a DPS student.");
}
```

```
// If empty list or only one element, done!
if (NULL == head || NULL == head->next) {
    return head;
}
// Otherwise, divide the list into two sublists of equal length.
divide_list (head, &fst, &sec);

// Sort each half.

fst = slow_sort (fst);

sec = slow_sort (sec);

// If fst is larger than sec, swap them (you MUST use the swap function).
if (fst->value > sec->value) {

    swap (&fst->next, &sec->next); // as shown; not needed for correctness

    swap (&fst, &sec);
}
// Reconnect fst and sec into a single list.

for (last = fst; NULL != last->next; last = last->next) { }

last->next = sec;

// Sort the rest of the list.

fst->next = slow_sort (fst->next);
// Return the sorted list.
return fst;
```

```
bird_t* find_fastest_migratory_bird (dl_t* head) {

    bird_t*   rval = NULL; // return value
    double    max = -1;    // maximum speed seen
    animal_t* a;
    bird_t*   b;

    for (dl_t* elt = head->next; head != elt; elt = elt->next) {

        a = (animal_t*)elt;

        b = (bird_t*)elt;

        if (BIRD == a->type && b->migratory && b->speed > max) {

            rval = b;

            max = b->speed;

        }
```

```
typedef struct node_t Node;
struct node_t {
    int32_t data;
    Node*   next;
};

void remove_duplicates (Node* head)
{
    if (NULL == head || NULL == head->next) {
        return;
    }
    remove_duplicates (head->next);
    if (head->data == head->next->data) {
        Node* remove = head->next;
        head->next = remove->next;
        free (remove);
    }
```

注意位置

单/双链表 Doubly-linked list/ The Link List

```
typedef struct node_t Node;
struct node_t {
    int32_t X;    // metric one: smaller is better
    int32_t Y;    // metric two: smaller is better
    Node*   next;
};
```

A node is **Pareto-dominated** if another node in the list has smaller or equal values for both X and Y.

```
void remove_dominated (Node* head)
{
    if (NULL == head || NULL == head->next) {
        return;
    }
    // original problem had non-unique X values (blue text also necessary)
    int32_t head_dom = (head->X == head->next->X && head->Y > head->next->Y);
    if (head->Y <= head->next->Y || head_dom) {
        Node* remove = head->next;
        head->next = remove->next;
        if (head_dom) {
            head->Y = remove->Y; // Note: X value is already the same.
        }
        free (remove);
        remove_dominated (head);
    } else {
        remove_dominated (head->next);
    }
}
```

List variables 静态初始化

```
static double_list_t my_list =
    {&my_list, &my_list};
```

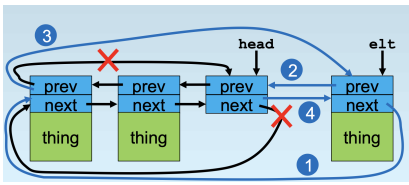
But **dynamically-allocated lists** must be **initialized at runtime.**

List variables 动态初始化

```
void dl_init (double_list_t* head)
{
    head->prev = head->next = head;
}
```

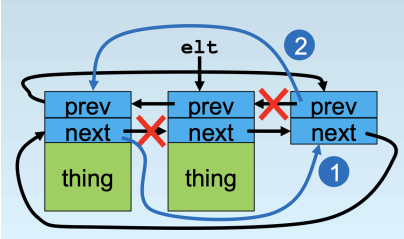
双向链表插入

```
void dl_inset (double_list_t* head, double_list_t* elt)
{
   elt->next = head->next;
   elt->prev = head;
   head->next->prev = elt;
   head->next = elt;
}
```



双向链表删除

```
elt->prev->next = elt->next; // Step 1
elt->next->prev = elt->prev; // Step 2
```



找到 List 里的第一个元素

```
void* dl_first (double_list_t* head)
{
  return (head == head->next ? NULL : head->next);
}
```

Callbacks 回调函数

```
int32_t isort (void* base, int32_t n_elts, size_t size,
               int32_t (*is_smaller) (void* t1, void* t2));
typedef struct horse_t horse_t;
struct horse_t {
    char*   name;   // dynamically allocated
    int32_t age;    // in years
    int32_t height; // in hands
};
```

int strcmp (const char* s1, const char* s2);

The **strcmp** function returns 0 iff the strings **s1** and **s2** are the same.

```
int32_t compare_horses (const void* elt1, const void* elt2)
{

    const horse_t* h1 = elt1;
    const horst_t* h2 = elt2;
    return (0 == strcmp (h1->name, h2->name) &&
            h1->age == h2->age &&
            h1->height == h2->height);

}       // horse_t structure and compare_horses signature
        //   replicated for your convenience.
typedef struct horse_t horse_t;
struct horse_t {
    char*   name;   // dynamically allocated
    int32_t age;    // in years
    int32_t height; // in hands
};
int32_t compare_horses (const void* elt1, const void* elt2);

void* find_element (void* array, int32_t n_elts, size_t size, void* elt_to_find,

                    int32_t (*cmp) (const void* elt1, const void* elt2) )

{
  char*   ar = array;
  int32_t i;

  for (i = 0; n_elts > i; i++) {
      if (cmp (elt_to_find, ar + i * size)) {
          return (ar + i * size);
      }
  }
  return NULL;
```

C++

```
class Base {
                int A;
protected:      int B;
private:        int C;
public:         int D;
};

class Derived: public Base {
private:        int E;
                static void aFunction (void);
public:         int F;
};

Derived instance;

void anotherFunction (void);
```

Derived::aFunction B D E F
anotherFunction. D F

```
#include <stdio.h>

class Mystery {
private:
    int x;
public:
    Mystery () { printf("M"); }
    Mystery (int xval) : x(xval + 1) { printf("Y"); }
    const Mystery& operator= (int xval) {
        xval = 1;
        printf("S");
        return *this;
    }
    Mystery (const Mystery& m) : Mystery(m.x + 10) { printf("T"); }
    ~Mystery() { printf("E"); }
};

Mystery c, d;

int main() {
    printf("---START---\n");
    c = d = 0;
    printf("\n");
    Mystery a = 42;
    printf("\n");
    Mystery b = a;
    printf("\n");
    c = a;
    printf("\n---END---");
    return 0;
}
```

Constructors, Destructors, and Operator Overloading

Line 1: MM---START---
Line 2: S
Line 3: Y
Line 4: YT
Line 5: blank
Line 6: ---END---EEEE

a.x = 43 b.x = 54 c.x = 43 d.x = bits

```
class Tricky {
private:
    int32_t a;
    int32_t b;
    Tricky (int32_t x, int32_t y) : a (x), b (y) { }
    friend Tricky operator& (const Tricky& t1, const Tricky& t2) {
        Tricky rval (t1.a * t2.b, t2.a * t1.b);
        return rval;
    }
    friend Tricky operator/ (const Tricky& t1, const Tricky& t2) {
        Tricky rval (t1.a / t2.a, t1.b / t2.b);
        return rval;
    }
public:
    Tricky (const Tricky& t) : a (t.a), b (t.a - 1) { }
    Tricky (double p) : a (15), b ((int32_t)round (p + 0.3)) { }
    Tricky (int32_t z) : a (z), b (z) { }

    void report (void);
};

int main ()
{
    Tricky one = 23.45;
    Tricky two = (5 & one);
    Tricky three = (one & (two / 10)) / two;

    one.report ();
    two.report ();
    three.report ();

    return 0;
}

void Tricky::report (void)
{
    printf ("%d\n", a - b);
}
```

The program's output is

____-9____
____45____
____-3____

```
MyClass* m = new MyClass
                 (arg1, arg2, ...);
```

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

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

Single-Argument Constructors Create Implicit Casts

```
class complex {
    // ...
public:
    complex (int32_t real_part);
    complex (double real_part);
    friend complex operator*
        (const complex& a,
         const complex& b);
};
```

Creates implicit cast from **int32_t** to **complex**.

Creates implicit cast from **double** to **complex**.

C++

Friday, May 24, 2024 18:59

1. C language extensions in C++

- void fcs; // function declared without parameters
- const 相当于 static，并必须初始化
- struct X {int x;}; // type X can be used without struct

2. Reference

```
double d
double &q = d; // create reference to d
q=1; // changes value of d
double *p = &q; // address of d
int d; double &q = d; // error, mismatched types
double &q = 5.2; // error, 5.2 is a constant
```

- references 不能被 reassign (创建后)
- reference to functions ✓
- pointers to ref..., ref..to ref... array of ref..., void& ✗

必须是变量

eg:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<string> vect{ "geeksforgeeks practice",
                         "geeksforgeeks write",
                         "geeksforgeeks ide" };
    // We avoid copy of the whole string
    // object by using reference.
    for (const auto& x : vect) {
        cout << x << '\n';
    }
    return 0;
}
```

static 静态 | 自动把变量类型 | 遍历 vect 中的元素 | 输出

区别 ① reference can't be NULL ② access members: reference: "." ; pointer: "->"

3. Compiles in C but not in C++

1) Calling a function before declaration
2) Using a normal pointer with const variable
   eg: int const j = 20;
   int* ptr = &j; ✗
3) Using typecasted pointers
   eg: void* vptr;
   int* iptr = vptr; ✗
4) Declaring constant values without initializing
   eg: const int a; ✗
5) Using specific keywords as variable names
   eg: int new = 0; (new is keyword in C++, not in C)
6) Strict type checking
7) Return type of main()
   int main() int main(void) = void main()
   C 可以传入参数 不可传入参数 (不推荐)
   C++ 不可传入参数
8) C++: fun() = fun(void)
   C: fun() = fun(...)

⚠ undefined behavior in C/C++:
modification of string literals: char* s = "geek"; s[0] = 'e'; ✗

4. 类 & 对象

1) 类

2) 定义对象 eg: Box Box1; // 声明 Box1，类型为 Box
3) 访问数据类型
   ① 类的对象的公共数据成员可以直接用 "." 访问

5. 继承

1) 基类 & 派生类

```
class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…
{
<派生类类体>
};
```

2) 访问控制和继承

派生类可以访问基类中所有的非私有成员 (public, protected) (~~privacy~~)

没继承：基类的 { 构造函数、析构函数和拷贝构造函数 / 重载运算符 / 友元函数

6. 重载运算符和重载函数 Overload

1) 函数重载

同一个作用域内，可以声明几个功能类似的同名函数

but 它们的形式参数（个数，类型，顺序…）必须不同

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

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

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

```
int main(void)
{
   printData pd;

   // 输出整数
   pd.print(5);
   // 输出浮点数
   pd.print(500.263);
   // 输出字符串
   char c[] = "Hello C++";
   pd.print(c);

   return 0;
}
```

2) 运算符重载

③ 非成员函数 Box operator+(const Box&, const Box&)

```
class Box
{
   public:

      double getVolume(void)
      {
         return length * breadth * height;
      }
      void setLength( double len )
      {
          length = len;
      }

      void setBreadth( double bre )
      {
          breadth = bre;
      }

      void setHeight( double hei )
      {
          height = hei;
      }
      // 重载 + 运算符，用于把两个 Box 对象相加
      Box operator+(const Box& b) ① 类成员函数
      {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }
   private:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
};
```

```
// 程序的主函数
int main( )
{
   Box Box1;                // 声明 Box1, 类型为 Box
   Box Box2;                // 声明 Box2, 类型为 Box
   Box Box3;                // 声明 Box3, 类型为 Box
   double volume = 0.0;     // 把体积存储在该变量中

   // Box1 详述
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // Box2 详述
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // Box1 的体积
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // Box2 的体积
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // 把两个对象相加，得到 Box3
   Box3 = Box1 + Box2;

   // Box3 的体积
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

(向下传参)

可重载的运算符

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

不可重载：?:(条件)，::(scope)，.(dot)，sizeof，typeid

*?: condition ? expression_if_true : expression_if_false

```
int max = (a > b) ? a : b;
```

const：声明常量

Volatile：声明一个变量可能会被修改

7. Dynamic allocation

```
// Example 1
try {
      int * ui = new int; // int * ui = new int(55)
} catch (bad_alloc) { error_occurred(); }
delete ui; // delete does not return any value
// Example 2
int * upi = new int[n];
for(int i=0; i<n; i++) upi[i]=0;
delete[] upi;
// Example 3
int (*a)[3] = new int[n][3];
for(i=0;i<n;i++) for(j=0;j<3;j++) { a[i][j]=0; }
```

- using new with class → call → constructor
- using delete with class → call → destructor

8. 虚函数 Virtual

```
#include <iostream>

// 基类 Animal
class Animal {
public:
    // 虚函数，表示各种动物都会发出声音
    virtual void makeSound() {
        std::cout << "An animal makes a sound.\n";
    }
};

// 派生类 Dog
class Dog : public Animal {
public:
    // 重写基类的虚函数
    void makeSound() override {
        std::cout << "The dog barks.\n";
    }
};

int main() {
    Animal* animalPtr;

    // 使用基类指针指向派生类对象
    animalPtr = new Dog();
    animalPtr->makeSound(); // 输出: The dog barks.

    // 使用基类指针指向另一个派生类对象
    animalPtr = new Cat();
    animalPtr->makeSound(); // 输出: The cat meows.

    delete animalPtr; // 释放内存
    return 0;
}
```

9. 类

1) 类成员函数

```
Box myBox;           // 创建一个对象

myBox.getVolume();   // 调用该对象的成员函数
```

可以在类的外部使用 :: 定义函数

2) 类访问修饰符 派生类也可 only 类，友元函数

继承方式	基类的public成员	基类的protected成员	基类的private成员
public继承	仍为public成员	仍为protected成员	不可见
protected继承	变为protected成员	变为protected成员	不可见
private继承	变为private成员	变为private成员	不可见

所有都为 | 默认为 private

Constructor, destructor

3) 构造，析构函数，拷贝构造函数

- 在每次创建类的新对象时执行
- 构造函数名称与类相同，不会返回任何类型（包括void）
- 可用于为某些成员变量设置初始值（默认不带参数，但是也可自己加）

```
#include <iostream>

using namespace std;

class Line
{
   public:
      int getLength( void );
      Line( int len );             // 简单的构造函数
      Line( const Line &obj);      // 拷贝构造函数
      ~Line();                     // 析构函数

   private:
      int *ptr;
};

// 成员函数定义，包括构造函数
Line::Line(int len)
{
    cout << "调用构造函数" << endl;
    // 为指针分配内存
    ptr = new int;
    *ptr = len;
}

Line::Line(const Line &obj)
{
    cout << "调用拷贝构造函数并为指针 ptr 分配内存" << endl;
    ptr = new int;
    *ptr = *obj.ptr; // 拷贝值
}

Line::~Line(void)
{
    cout << "释放内存" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
   cout << "line 大小 : " << obj.getLength() <<endl;
}
```

```
// 程序的主函数
int main( )
{
   Line line(10);

   display(line);

   return 0;
}
```

```
调用构造函数
调用拷贝构造函数并为指针 ptr 分配内存
line 大小 : 10
释放内存
释放内存
```

② 使用已有的同类型对象来初始化新的对象

```
// 程序的主函数
int main( )
{
   Line line1(10);

   Line line2 = line1; // 这里也调用了拷贝构造函数

   display(line1);
   display(line2);

   return 0;
}
```

```
调用构造函数
调用拷贝构造函数并为指针 ptr 分配内存
调用拷贝构造函数并为指针 ptr 分配内存
line 大小 : 10
释放内存
释放内存
释放内存
释放内存
```

4) 友元函数：可访问 all 成员 ~~成员函数~~

定义在类外部，友元 { 函数 / 类：整个类及其成员都是友元

```
class Box
{
   double width;
public:
   double length;
   friend void printWidth( Box box );
   void setWidth( double wid );
};
```

在类外定义

类

```
class ClassOne
{ public:
    friend class ClassTwo;
};
```

5) this 指针

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    void setValue(int value) {
        this->value = value;
    }

    void printValue() {
        std::cout << "Value: " << this->value << std::endl;
    }
};

int main() {
    MyClass obj;
    obj.setValue(42);
    obj.printValue();

    return 0;
}
```

每个对象都能通过 this 来访问自己的地址

可以在类的成员函数中用 (只有成员函数有)

6) 指向类的指针

```
#include <iostream>

class MyClass {
public:
    int data;

    void display() {
        std::cout << "Data: " << data << std::endl;
    }
};

int main() {
    // 动态分配内存创建类对象
    MyClass *ptr = new MyClass;
    ptr->data = 42;

    // 通过指针调用成员函数
    ptr->display();

    // 释放动态分配的内存
    delete ptr;

    return 0;
}
```

用于动态分配内存

```
#include <iostream>

class MyClass {
public:
    int data;

    void display() {
        std::cout << "Data: " << data << std::endl;
    }
};

// 函数接受指向类的指针作为参数
void processObject(MyClass *ptr) {
    ptr->display();
}

int main() {
    MyClass obj;
    obj.data = 42;

    // 将指向类的指针传递给函数
    processObject(&obj);

    return 0;
}
```

作为函数参数传递

7) 静态成员 static

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // 每次创建对象时增加 1
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // 长度
      double breadth;    // 宽度
      double height;     // 高度
};
// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main(void)
{

   // 在创建对象之前输出对象的总数
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // 声明 box1
   Box Box2(8.5, 6.0, 2.0);    // 声明 box2

   // 在创建对象之后输出对象的总数
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

没有 this 指针

只能访问静态成员（变量，函数）

必须在类外部初始化

10. 继承

1) 基类 & 派生类

```
class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…
{
<派生类类体>
};
```

2) 访问控制和继承

派生类可以访问基类中所有的非私有成员 (public, protected) (~~privacy~~)

没继承：基类的 { 构造函数、析构函数和拷贝构造函数 / 重载运算符 / 友元函数

I/O FILE

```
int32_t file_reduce (const char* fname)
{
    FILE* in;  // input stream
    FILE* out; // output stream
    // First, write code to prepare the streams for use.

    if (NULL == (in = fopen (fname, "r")) ||
        NULL == (out = fopen ("out.txt", "w"))) {
        if (NULL != in) {
            fclose (in);
        }
        return 0;
    }
// Read the input file and produce the output.

int last = EOF, char;

while (EOF != (char = fgetc (in))) {
    if (last != char) {
        fputc (char, out);
        last = char;
    }
}
    // Clean up and return.

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

}
```