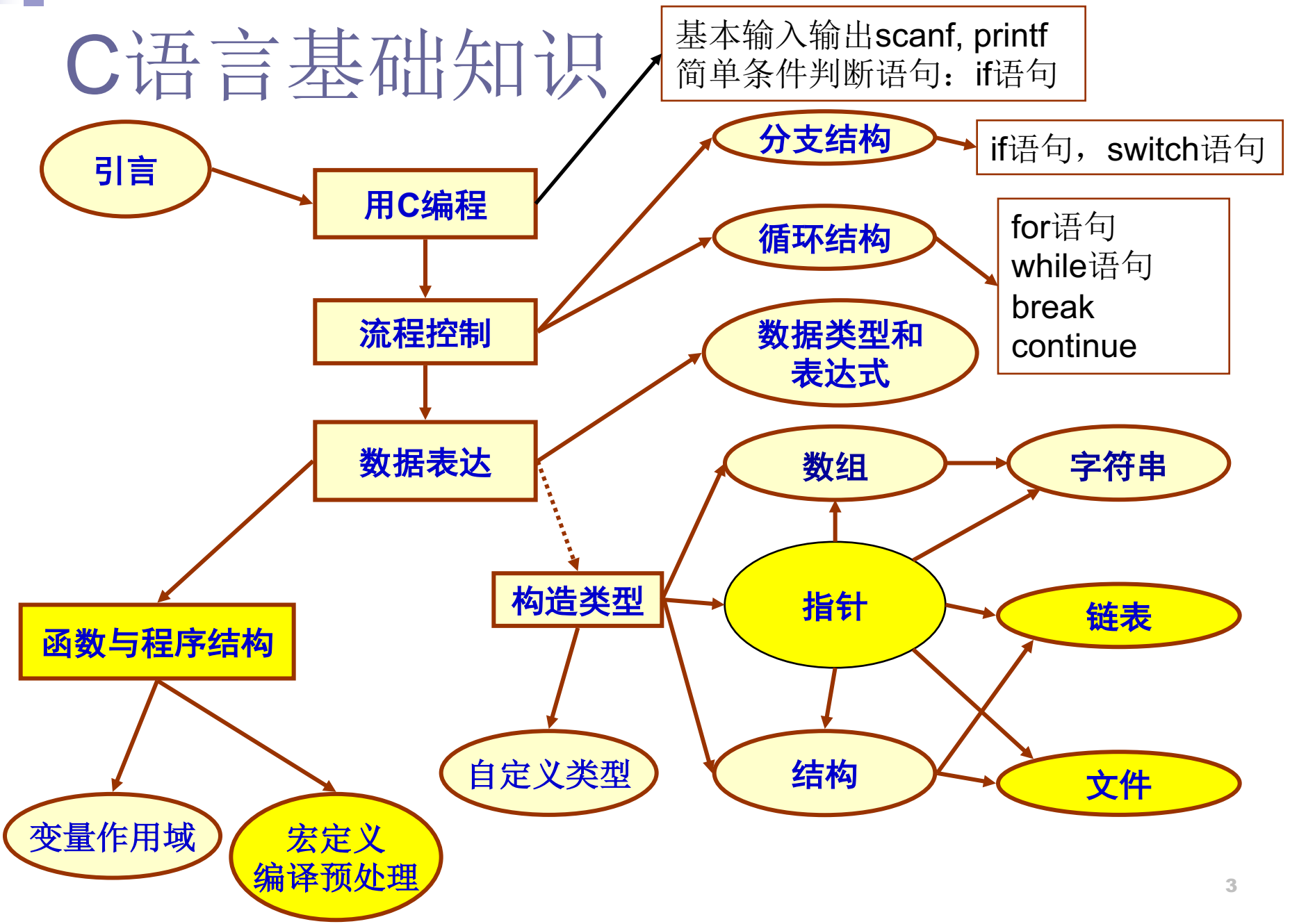
专题一 指针进阶

Lecture 1

专题一 指针进阶

- C语言基础知识回顾
- 有序表的操作 (10.1)
- 动态内存分配 (8.5)
- 指针数组、二级指针 、数组指针 (11.1)
- 函数指针 (11.2.3)

C语言基础知识

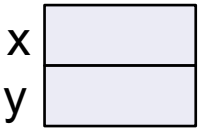

基本输入输出scanf, printf
简单条件判断语句：if语句
引言
用C编程
流程控制
数据表达
函数与程序结构
变量作用域
宏定义
编译预处理
分支结构
if语句，switch语句
循环结构
for语句
while语句
break
continue
数据类型和
表达式
数组
字符串
构造类型
指针
链表
自定义类型
结构
文件

C语言基础知识

- 数据类型
  - 基本数据类型 (scanf/prinf)
    - 整型 short, int, unsigned long, …
    - 浮点型 float, double
    - 字符型 char
    - 空类型 void (函数类型)
  - 构造数据类型
    - 数组、结构、联合、枚举
  - 指针类型

结构：struct data { int x; int y; }

struct data a;内存分布



联合 (P337)：

union data { int x; char s[4]; }

union data a;内存分布

a.x = 256; a.s = ?

枚举：enum 枚举名 { 枚举值1, 枚举值2, … };

enum color { red, green, blue = 3, black };

enum color c1 = red, c2 = black;

C语言基础知识

- 数据类型
- 变量
  - 变量类型：局部变量、全局变量、静态变量
  - 输入输出：整型、浮点型、字符型、字符串
- 表达式：优先级和结合性 (表A.2)
- 流程控制：顺序结构、分支结构、循环结构
  - 分支结构：if/else, switch/case/break
    - else找前面最近的未被匹配的if，与之配对
  - 循环结构：for, while, do-while, break, continue

C语言基础知识

- 数据类型
- 变量
- 表达式：优先级和结合性 (表A.2)
- 流程控制：顺序结构、分支结构、循环结构
- 函数
  - 函数参数都是传值，整体赋值，单向传递，多个参数计算顺序通常为从右往左
    - 修改被调函数的形参，不会影响主调函数的实参
  - 修改主调函数的变量，传递该变量的地址
    - 数组名作为指针常量，传递数组首地址
    - swap(int a, int b) vs. swap(int *pa, int *pb)

专题一 指针进阶

- C语言基础知识回顾
- 有序表的操作 (10.1)
  - 数组的常用操作回顾
  - 函数的顺序调用和嵌套调用
- 动态内存分配 (8.5)
- 指针数组、二级指针 、数组指针 (11.1)
- 函数指针 (11.2.3)

有序表的操作

- [例10-1] 首先输入一个无重复元素的、从小到大排列的有序表，并在屏幕上显示以下菜单(编号和选项)，用户可以反复对该有序表进行插入、删除和查找操作，也可以选择结束。当用户输入编号1～3和相关参数时，将分别对该有序表进行插入、删除和查找操作，输入其他编号，则结束操作

  [1] Insert

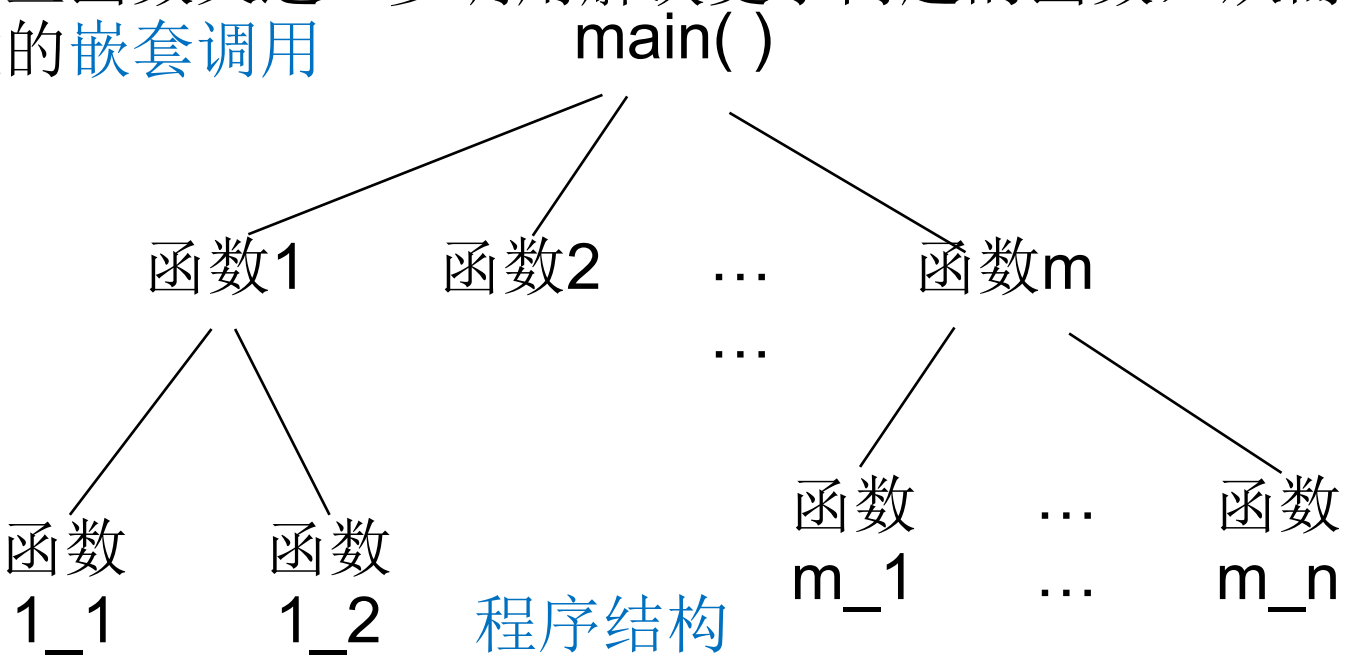
  [2] Delete

  [3] Query

  [Other option] End

结构化程序设计

- 使用结构化程序设计方法解决复杂的问题
  - 把大问题分解成若干小问题，小问题再进一步分解成若干更小的问题
  - 写程序时用main()解决整个问题，调用解决小问题的函数
  - 这些函数又进一步调用解决更小问题的函数，从而形成函数的嵌套调用

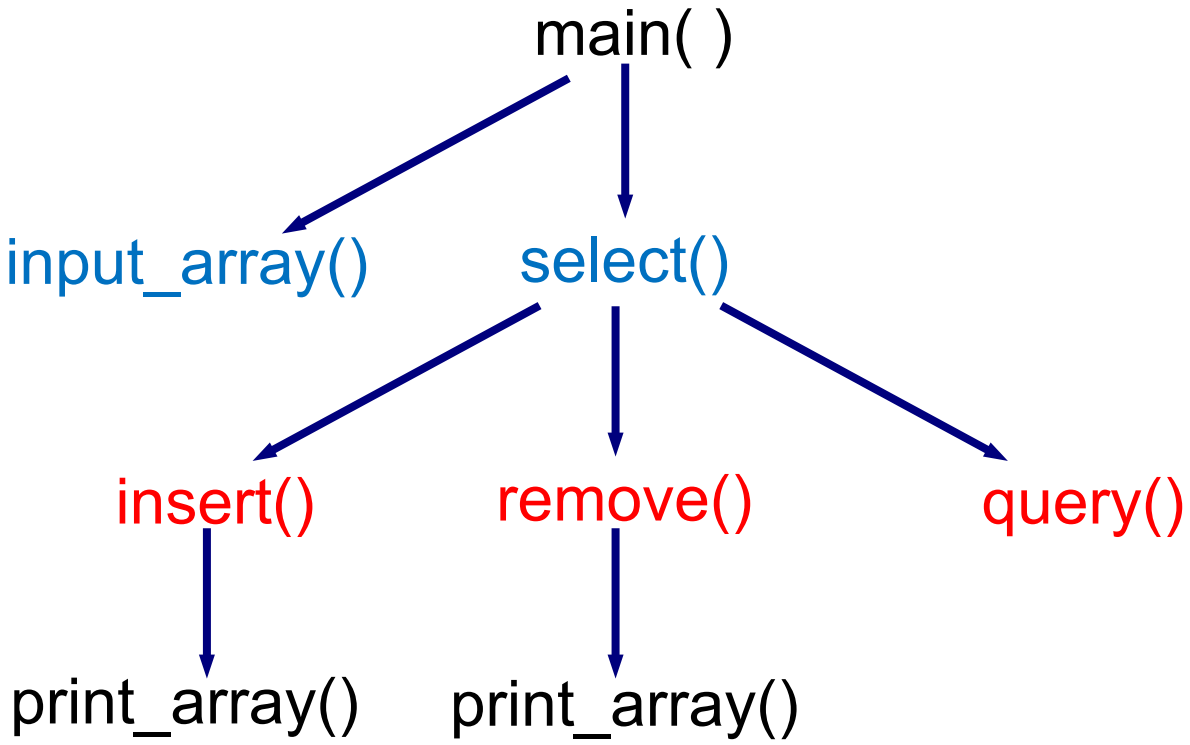


程序结构

例10-1分析

- 输入有序表
- 输入1、2、3选择插入、删除、查找操作，其他输入结束
  - 设计一个控制函数select()，经它辨别用户输入的编号后，调用相应的插入、删除和查找函数，再调用有序表输出函数显示结果
- 分别设计函数实现有序表的插入、删除、查找和输入、输出等常规操作

例10-1程序结构



4层结构，7个函数

降低程序的构思、编写、调试的复杂度，可读性好

例10-1 源程序

```
int Count = 0;  #define MAXN 100
void select(int a[], int option, int value);
void input_array(int a[ ]);
void print_array(int a[ ]);
void insert(int a[ ], int value);
void remove(int a[ ], int value);
void query(int a[ ], int value);
int main(void)
{   int option, value, a[MAXN];
    input_array(a);                    /* 调用函数输入有序数组 a */
    printf("[1] Insert\n");   printf("[2] Delete\n");
    printf("[3] Query\n");  printf("[Other option] End\n");
    while (1) {
        printf("Input option: ");   scanf("%d", &option);
        if (option < 1 || option > 3) { break; }
        printf("Input an element: ");   scanf("%d", &value);
        select(a, option, value);       /* 调用控制函数 */
        printf("\n");
    }
    printf("Thanks.");
    return 0;
}
```

例10-1 源程序

```
/* 控制函数 */
void select(int a[ ], int option, int value)
{
    switch (option) {
        case 1:
            insert(a, value);    /* 调用插入函数在有序数组a中插入元素value */
            break;
        case 2:
            remove(a, value);  /* 调用删除函数在有序数组a中删除元素value */
            break;
        case 3:
            query(a, value);     /* 调用查询函数在有序数组a中查找元素value */
            break;
    }
}
```

例10-1 源程序

```
/* 有序表输入函数 */
void input_array(int a[ ])
{
    printf("Input the number of array elements: ");
    scanf("%d", &Count);
    printf("Input an ordered array element: ");
    for (int i = 0; i < Count; i ++) {
        scanf("%d", &a[i]);
    }
}
/* 有序表输出函数 */
void print_array(int a[ ])
{
    printf("The ordered array a is: ");
    for (int i = 0; i < Count; i ++) {  /* 相邻数字间空格分隔，行末无空格 */
        if(i == 0) {
            printf("%d", a[i]);
        } else {
            printf(" %d", a[i]);
        }
    }
}
```

例10-1 源程序

```
/* 有序表插入函数 */
void insert(int a[ ], int value)
{
    int i, j;
    for (i = 0; i < Count; i++) {        /* 定位：待插入的位置即退出循环时i的值 */
        if(value < a[i]) {
            break;
        }
    }
    for (j = Count - 1; j >= i; j--) {   /* 腾位：将a[i]~a[Count-1]向后顺移一位 */
        a[j+1] = a[j];
    }
    a[i] = value;                        /* 插入：将value 的值赋给a[i] */
    Count++;                             /* 增1：数组a中待处理的元素数量增1 */
    print_array(a);                      /* 调用输出函数，输出插入后的有序数组a */
}
```

-2	3	7	9	13	40				
-2	0	3	7	9	13	40			

例10-1 源程序

```
/* 有序表删除函数 */
void remove(int a[ ], int value)
{
    int i, index = -1;                  /* index的值为-1表示没找到，否则表示找到 */
    for(i = 0; i < Count; i++) {        /* 定位：找到则用index记录其下标 */
        if(value == a[i]) {
            index = i;
            break;
        }
    }
    if(index == -1) {                   /* 没找到，则输出相应的信息 */
        printf("Failed to find the data, deletion failed.");
    } else {                            /* 找到，则删除a[index] */
        for(i = index; i < Count - 1; i++) {
            a[i] = a[i+1];              /* 将a[Count-1]~ a[index+1]向前顺移一位 */
        }
        Count--;                        /* 减1：数组a中待处理的元素数量减1 */
        print_array(a);                 /* 调用输出函数，输出删除后的有序数组a */
    }
}
```

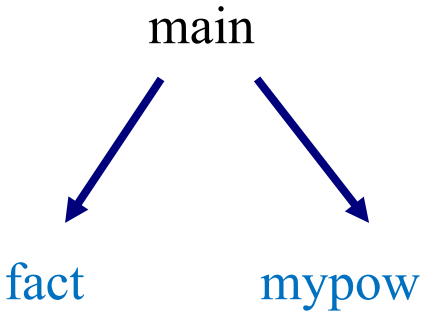
-2	3	7	9	13	40				
-2	7	9	13	40	40				

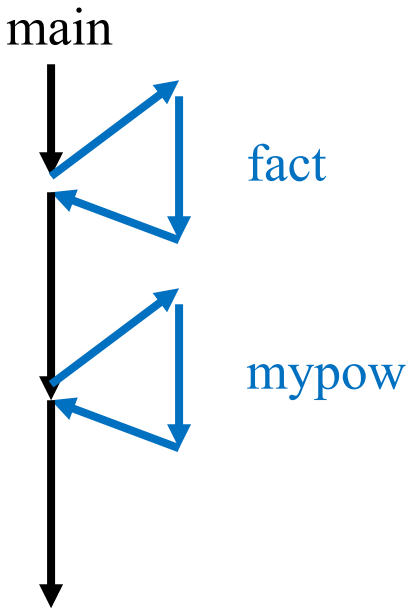
例10-1 源程序

```
/* 有序表二分法查询函数 */
void query(int a[ ], int value)
{
    int mid, left = 0, right = Count - 1;        /* 开始时查找区间为整个数组 */
    while (left <= right) {
        mid = (left + right) / 2;          /* 得到中间位置 */
        if (value == a[mid]) {             /* 查找成功，输出下标，函数返回 */
            printf("The index is: %d", mid);
            return;
        } else if (value < a[mid]) {       /* 缩小查找区间为前半段，right前移 */
            right = mid - 1;
        } else {                           /* 缩小查找区间为后半段，left后移 */
           left = mid + 1;
        }
    }
    printf( "This element does not exist.");       /* value不在数组a中 */
}
```

函数调用 - 顺序调用

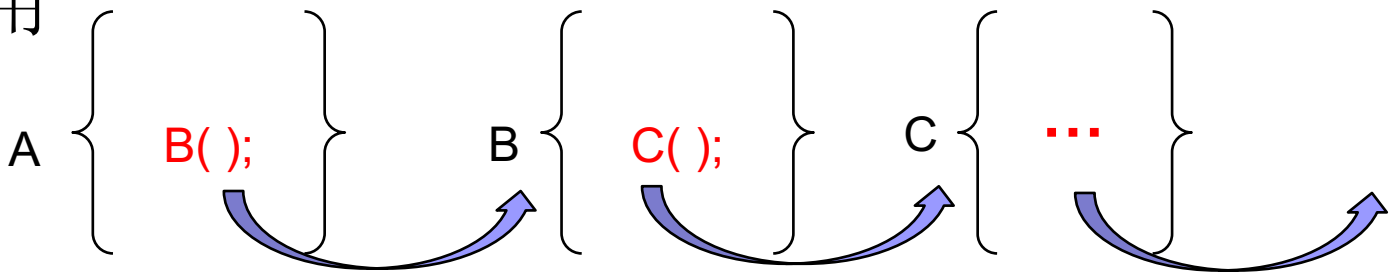
```
int main(void)
{
    ……
    y = fact(3);
    ……
    z = mypow(3.5, 2);
    ……
}
double fact(int n)
{
    …….
}
double mypow(double x, int n)
{
    ……
}
```





函数调用 - 嵌套调用

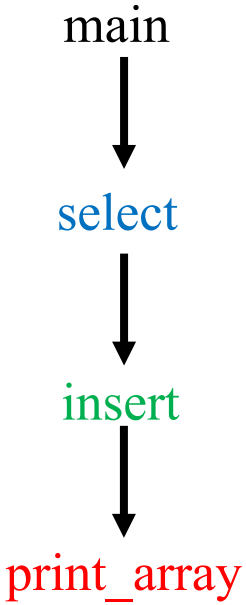
- 在一个函数中再调用其它函数的情况称为函数的嵌套调用
  - 如果函数A调用函数B，函数B再调用函数C，一个调用一个地嵌套下去，构成了函数的嵌套调用

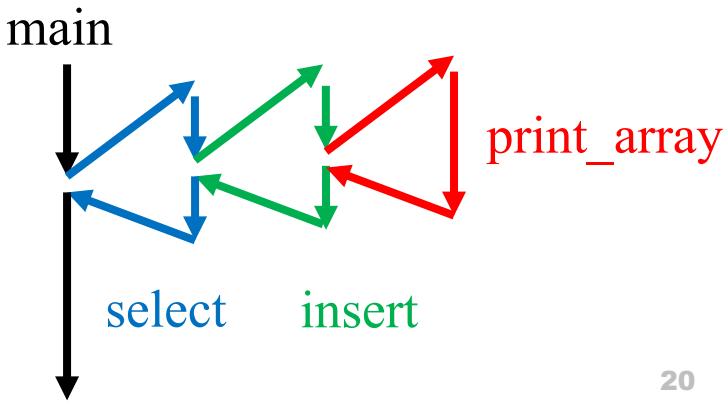


  - 具有嵌套调用函数的程序，需要分别定义多个不同的函数体，每个函数体完成不同的功能，它们合起来解决复杂的问题

函数调用 - 嵌套调用

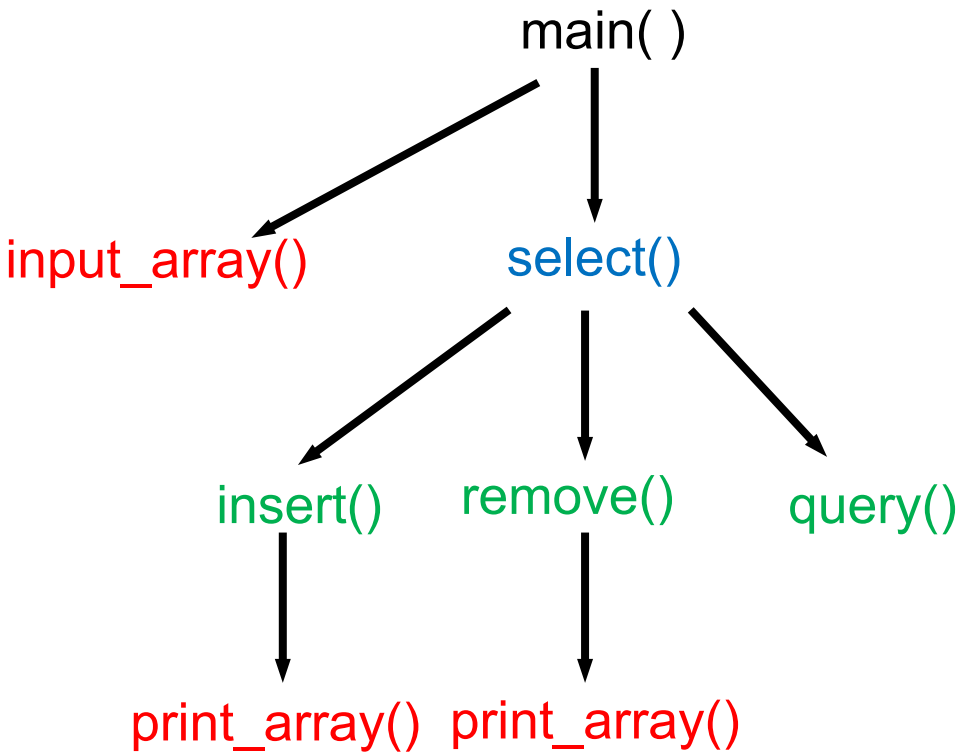
```
int main(void)
{   ……
    select(a, option, value);
    ……
}
void select(int a[ ], int option, int value)
{   ……
    insert(a, value);
    ……
}
void insert(int a[ ], int value)
{   ……
    print_array(a);
    ……
}
void print_array(int a[ ])
{   ……
}
```





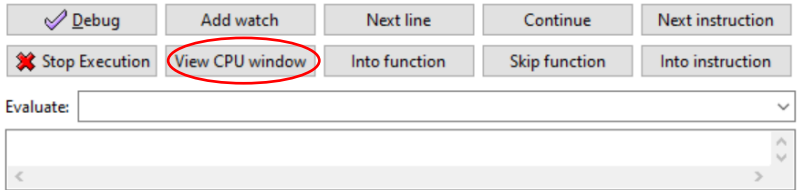
例10-1 分析

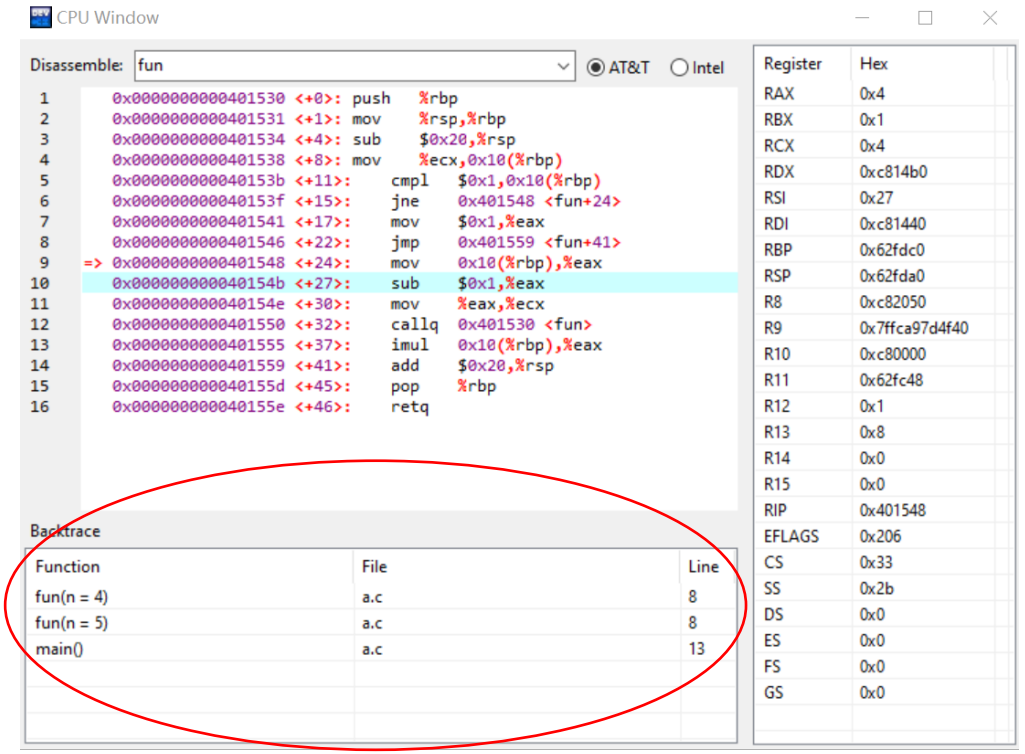
```
int main (void)
{   ……
}
void select(int a[ ], int option, int value)
{   ……
}
void input_array(int a[ ])
{   ……
}
void print_array(int a[ ])
{   ……
}
void insert(int a[ ], int value)
{   ……
}
void remove(int a[ ], int value)
{   ……
}
void query(int a[ ], int value)
{   ……
}
```



函数的调用堆栈

- Dev-C++中如何查看函数的调用堆栈？
  - 选择Debug模式，编译，设置断点，Debug运行
  - 点击调试控制面板的“View CPU Window/查看CPU窗口”





有序表的操作

- 扩展
  - 如果不使用全局变量Count，如何修改程序
  - 如何修改输入函数input_array()的定义，当输入数据无序或重复时，使数组a有序且无重复元素
  - 如何修改插入函数insert()的定义，使得重复元素不会被插入，且不会越界访问
  - 如何修改程序，使得有序表支持重复元素
  - 如何修改程序，使得有序表支持任意元素数量(动态内存管理)

专题一 指针进阶

- C语言基础知识回顾
- 有序表的操作 (10.1)
- 内存动态分配 (8.5)
  - 程序运行时内存模型
  - 动态内存分配
  - 动态内存分配应用
- 指针数组、二级指针 、数组指针 (11.1)
- 函数指针 (11.2.3)

程序运行时内存模型

- 每个程序运行时，都有独立的内存空间
- 程序的内存空间大小与位数有关
  - 32位
    - 最多4G内存，内存地址范围0 ~ 2^{32} - 1
  - 64位
    - 最多1024 * 1024 * 1024 * 16 G内存

内存模型

- Stack
  - 栈
- Heap
  - 堆
- Globals
  - 全局变量
  - 静态变量
- Constants
  - 常量
- Code
  - 代码

Memory memorizer

Stack

This is the section of memory used for **local variable storage**. Every time you call a function, all of the function's local variables get created on the stack. It's called the *stack* because it's like a stack of plates: variables get added to the stack when you enter a function, and get taken off the stack when you leave. Weird thing is, the stack actually works upside down. It starts at the top of memory and **grows downward**.

Heap

This is a section of memory we haven't really used yet. The heap is for **dynamic memory:** pieces of data that get created when the program is running and then hang around a long time. You'll see later in the book how you'll use the heap.

Globals

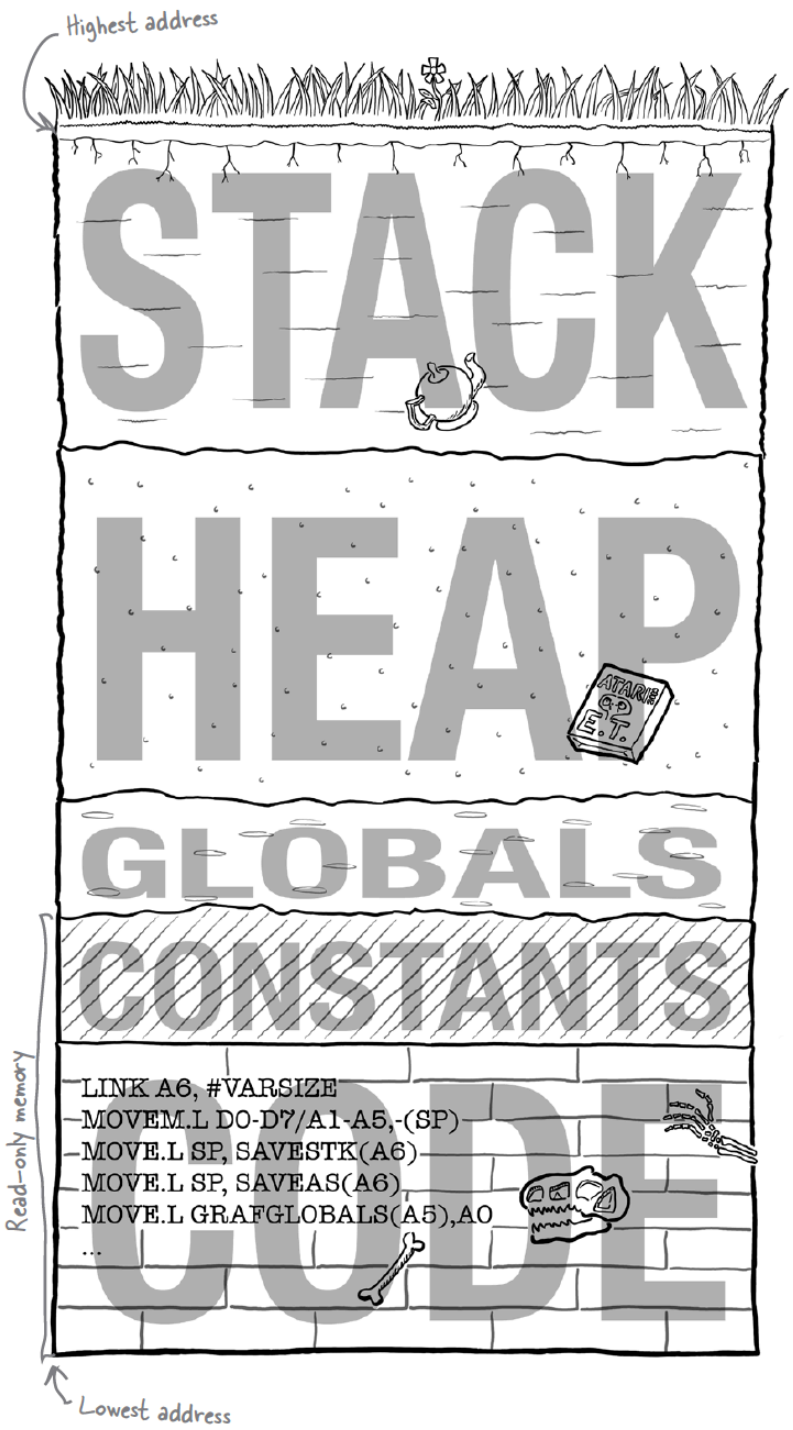
A global variable is a variable that lives outside all of the functions and is visible to all of them. Globals get created when the program first runs, and you can update them freely. But that's unlike…

Constants

Constants are *also* created when the program first runs, but they are stored in **read-only** memory. Constants are things like *string literals* that you will need when the program is running, but you'll never want them to change.

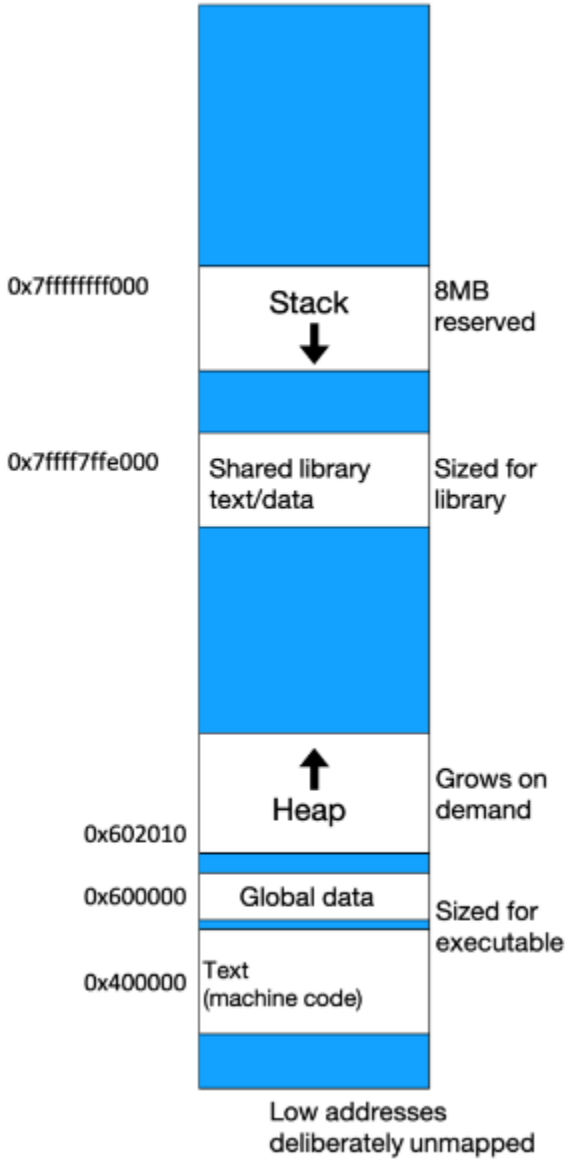
Code

Finally, the code segment. A lot of operating systems place the code right down in the lowest memory addresses. The code segment is also read-only. This is the part of the memory where the actual assembled code gets loaded.



程序运行时内存模型

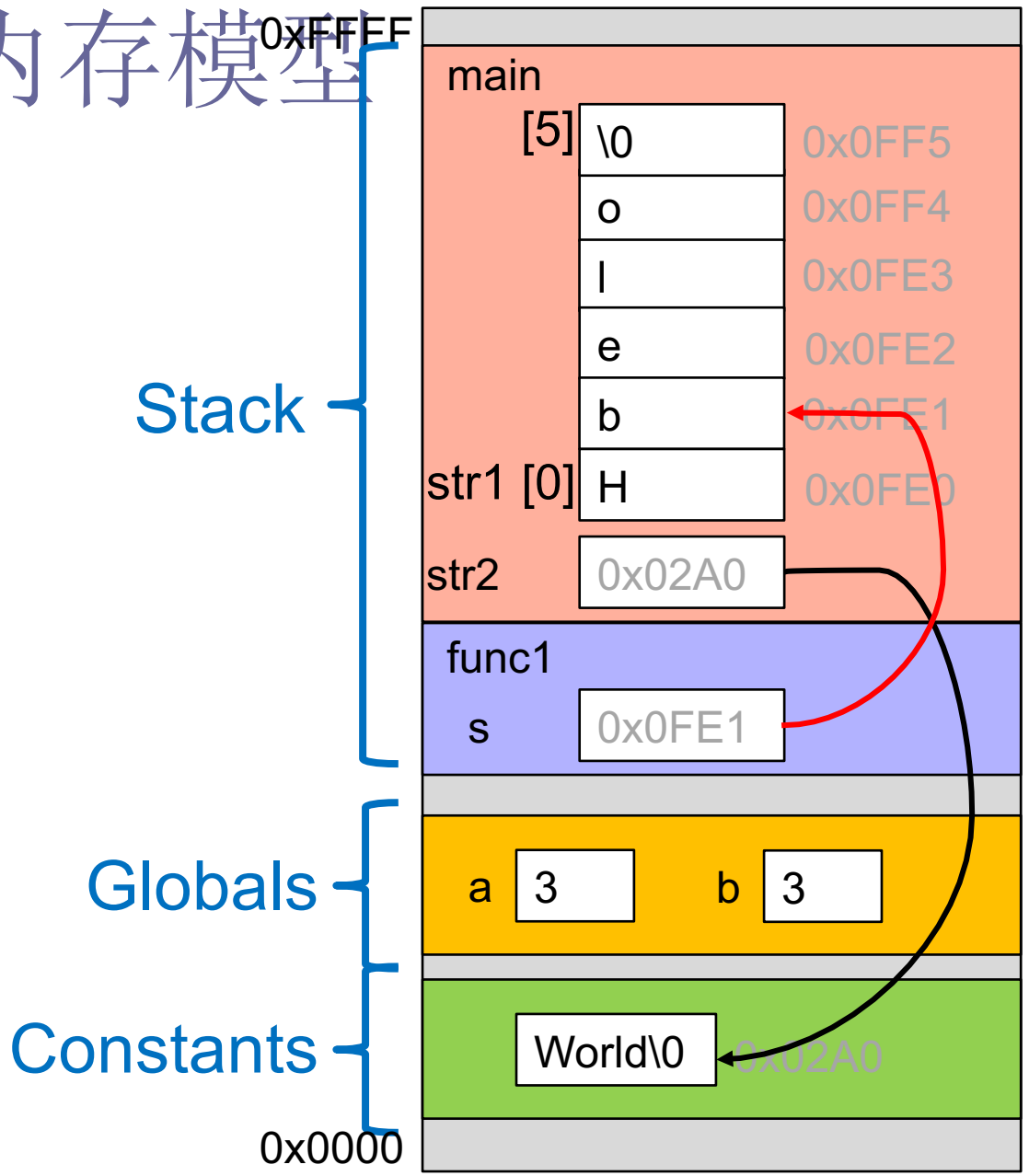
- Stack (栈，函数调用栈)
  - 函数的形参和变量
    - 每个函数都有独立的存储空间
  - 从高地址到低地址内存增长
  - 调用函数时，申请栈空间
    - 当栈用完时，stack overflow
  - 函数返回时，返还栈空间
    - 不会清理返还的内容空间
  - 通常较小，8M



程序运行时内存模型

```
int a = 1;
void func1(char *s)
{
    static int b = 2;
    s[1] = 'a' + a++;
    *(s+2) = 'b' + ++b;
}

int main(void)
{
    char str1[] = "Hello";
    char *str2 = "World";
    func1(str1);
    func1(str1+1);
    func1(str2+1);
    return 0;
}
```



动态内存分配

- 为什么需要内存动态分配？
  - 无法在程序运行时动态分配内存
    - 变量数量、数组长度在编译时确定，运行时无法修改
      - 变量在使用前必须被定义且安排好存储空间
      - 全局变量、静态局部变量的存储是在编译时确定，在程序开始执行前完成
      - 自动变量，在执行进入变量定义所在的复合语句时为它们分配存储，变量的大小也是静态确定的
      - 一般情况下，运行中的很多存储要求在写程序时无法确定

动态内存分配

- 为什么需要内存动态分配？
  - 无法在程序运行时动态分配内存
    - 变量数量、数组长度在编译时确定，运行时无法修改
    - 可变长数组 (Variable Length Array)
      - C99标准中加入，C11标准变成可选 (可能会降低代码效率)。具体看编译器是否支持，微软VC不支持，gcc/clang (Dev-C++)等支持
      - 建议使用定长数组，在编译前确定数组长度，使得代码具有更好的兼容性
      - C++标准也不支持，但g++支持，属于g++的扩展，不属于C++标准

```
int n;
scanf("%d", &n);
int a[n];
```

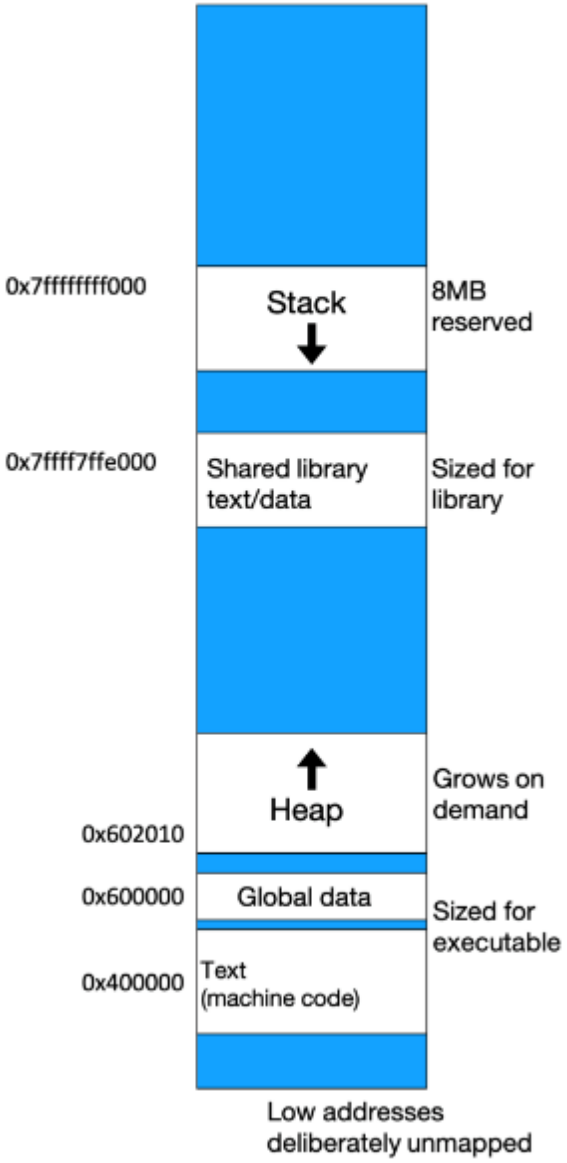
```
#define MAXN 100
int n, a[MAXN];
scanf("%d", &n);
```

动态内存分配

- 为什么需要内存动态分配？
  - 无法在程序运行时动态分配内存
    - 变量数量、数组长度在编译时确定，运行时无法修改
    - 可变长数组 (Variable Length Array)
      - 运行时，数组长度确定，首次分配存储空间后，数组长度无法再次修改
  - 自动变量分配在栈(Stack)区
    - 栈区内存容量有限 (8M)，无法处理大数据
    - 例如一张2048 * 2048图片，原始数据就有12M

动态内存分配

- 不是由编译系统分配的(Stack)，而是由用户在程序内存的堆(Heap)中通过动态分配获取
- 使用动态内存分配能有效地使用内存
  - 使用时申请
    - 大内存尽量使用堆中申请
  - 用完就释放
    - 不释放将导致内存泄漏
- 同一段内存可以有不同的用途



动态内存分配的步骤

- (1) 了解需要多少内存空间
- (2) 利用C语言提供的动态分配函数来分配所需要的存储空间
- (3) 使指针指向获得的内存空间，以便用指针在该空间内实施运算或操作
- (4) 当使用完毕内存后，释放这一空间

动态存储分配函数malloc()

void *malloc(unsigned size)

在内存的动态存储区(Heap)中分配一连续空间，其长度为size字节

- 若申请成功，则返回一个指向所分配内存空间的起始地址的指针
- 若申请内存空间不成功，则返回NULL (值为0)
  - 什么情况下会不成功？
- 返回值类型：void *
  - 通用指针的一个重要用途
  - 将malloc的返回值转换到特定指针类型，赋给一个指针

malloc()示例

```
/* 动态分配n个整数类型大小的空间 */
if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
    printf("Not able to allocate memory. \n");
    exit(1);
}
```

- 调用malloc时，用sizeof计算存储块大小
- 每次动态分配都要检查是否成功，考虑例外情况处理
- 虽然存储块是动态分配的，但它的大小在分配后也是确定的，不要越界使用

计数动态存储分配函数calloc()

void *calloc(unsigned n, unsigned size)

在内存的动态存储区(Heap)中分配n个连续空间，每一存储空间的长度为size字节，并且分配后还把存储块里全部初始化为0

- 若申请成功，则返回一个指向被分配内存空间的起始地址的指针
- 若申请内存空间不成功，则返回NULL

- malloc对所分配的存储块不做任何事情（未初始化）
  - 自动变量
- calloc对整个区域进行初始化（初始化为0）
  - 全局变量、静态变量

动态存储释放函数free()

void free(void *ptr)

释放由动态存储分配函数申请到的整块内存空间，ptr为指向要释放空间的首地址

当某个动态分配的存储块不再用时，要及时将它释放，否则就会产生内存泄漏，也不要重复释放，即重复调用free(ptr)

- 只能整块释放free(ptr)，不能只释放部分内存free(ptr+10)
- 释放后，建议将指针赋值为NULL，避免使用野指针(已释放的内存)和重复释放

分配调整函数realloc()

void *realloc(void *ptr, unsigned size)

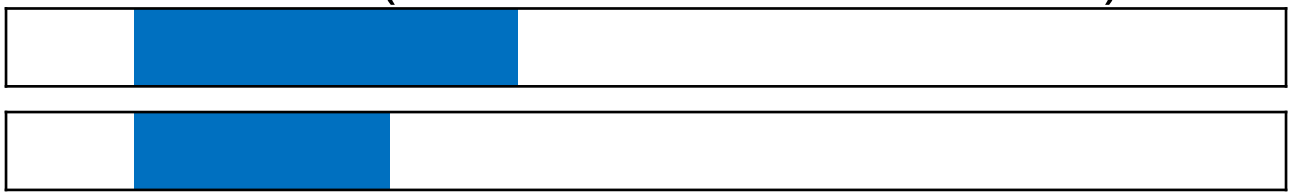
更改以前的存储分配

- ptr必须是以前通过动态存储(Heap)分配得到的指针
- 参数size为现在需要的空间大小

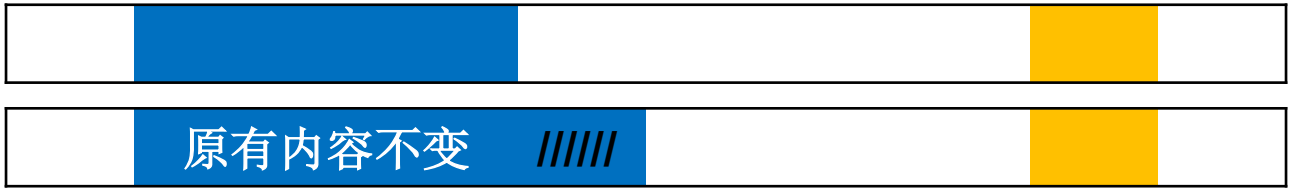
- 如果调整失败，返回NULL，同时原来ptr指向存储块的内容不变
- 如果调整成功，返回一片能存放大小为size的区块，并保证该块的内容与原块的一致。如果size小于原块的大小，则内容为原块前size范围内的数据；如果新块更大，则原有数据存在新块的前一部分
- 如果分配成功，原存储块的内容就可能改变了，因此不允许再通过ptr去使用它

分配调整函数realloc()

- size <= 原块大小 (等价实现部分内存的释放)

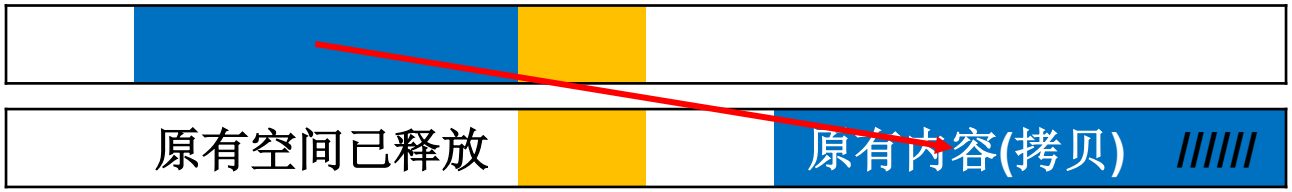


- size > 原块大小，且原块后有足够连续未使用空间



不同颜色表示不同动态内存申请

- size > 原块大小，且原块后未有足够连续未使用空间，但Heap区有连续size未使用空间，拷贝原块数据至新分配内存



不能通过原ptr使用

- size > 原块大小，但Heap区未有连续size未使用空间

动态内存分配函数总结

功能	函数	说明	与酒店类比
首次申请[成功返回内存起始地址(void *)，不成功返回NULL]	malloc(size)	在Heap区分配长度为size字节的连续空间，参数size为分配内存大小，未初始化	check in，申请连续房间，成功领取起始房间号，其他房间通过房间号+n(指针运算)获得，不要闯入他人房间(访问越界)，不成功告知无房 malloc 未打扫房间 calloc 需打扫房间
	calloc(n, size)	在Heap区分配长度为n * size字节的连续空间，参数size为单个元素的内存大小，分配完成后初始化为0	
分配调整	realloc(ptr, size)	调整在Heap区分配的内存空间，ptr为指向要调整空间的起始地址，size为新的内存大小。分配成功，ptr可能失效，使用返回的指针，分配失败，ptr不变，返回NULL	调整房间数量：若后续为空房间，直接分配，行李仍在原房间；若后续房间有人入住，在酒店其他楼层找连续房间，若找到，将原房间行李搬运至新房间，否则，告知无房
用完释放	free(ptr)	释放Heap区动态分配的整块内存空间，ptr为指向要释放空间的起始地址	check out，通过起始房间号退所有房，不能退出部分房间

动态内存分配应用 - 例1

- 申请1个整数

```
int *p = (int *) malloc(sizeof(int));
```

- 申请10个双精度浮点数，且初始化为0

```
double *p = (double *) calloc(10, sizeof(double));
```

- 申请10个Point结构

```
Point *s = (Point *) malloc(10 * sizeof(Point));
Point *s = (Point *) calloc(10, sizeof(Point));
```

- 调整Point结构为20个

```
if((s1 = realloc(s, 20 * sizeof(Point))) != NULL)
    s = s1;
```

动态内存分配应用 - 例2

- [例8-10] 先输入一个正整数n，再输入任意n个整数，计算并输出这n个整数的和
  - 要求使用动态内存分配方法为这n个整数分配空间

例8-10源程序-任意个整数求和

```
int main ( )
{
    int n, sum, i, *p;
    printf ("Enter n: ");
    scanf ("%d", &n);
    if ((p = (int *) calloc (n, sizeof(int))) == NULL) {
        printf ("Not able to allocate memory. \n");
        exit(1);
    }
    printf ("Enter %d integers: ", n);
    for (i = 0; i < n; i++) {
        scanf("%d", p+i);
    }
    sum = 0;
    for (i = 0; i < n; i++) {
        sum = sum + *(p+i);
    }
    printf ("The sum is %d \n",sum);
    free (p);
    return 0;
}
```

```
Enter n: 10
Enter 10 integers: 3 7 12 54 2 -19 8 -1 0 15
The sum is 81
```

动态内存分配应用 - 例3

- 除了数组外，函数参数和返回值都是传值
- 函数返回值的类型 (11.2.2)
  - 整型
  - 浮点型
  - 字符型
  - 结构类型
  - 指针
    - 返回全局数据对象或主调函数中数据对象的地址
  - 数组
    - 返回数组的首地址，所有的局部数据对象在函数返回时就会消亡，其值不再有效

动态内存分配应用 - 例3

- 函数可以返回哪类指针？
  - 全局变量/静态变量的指针 ✓
    - char *s = "Test"; char * fun() { return s + 2; }
  - 主调函数传递的指针 ✓
    - int * fun(int *a) { return a + 5; }
  - 被调函数申请的动态内存 ✓
    - int * fun() { return (int *)malloc(……); }
  - 函数局部变量 ✗
    - int * fun() { int a = 10; return &a; }
    - char * fun() { char s[10]; …… return s; }

动态内存分配应用 - 例4

- 下面函数能否释放动态分配的内存，并将指向该内存的指针赋值为NULL？

```
void free_and_null(void *ptr)
{
    free(ptr);
    ptr = NULL;
}
void func()
{
    int *p = (int *) malloc(10 * sizeof(int));
    ……
    free_and_null(p);
    if (p == NULL) {
        p = (int *) malloc(20 * sizeof(int));
    }
    ……
}
```

动态内存分配应用 - 例4

- 下面函数能否释放动态分配的内存，并将指向该内存的指针赋值为NULL？

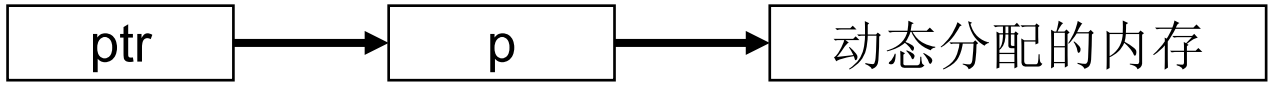
```
void free_and_null(void *ptr)
{
    free(ptr);
    ptr = NULL;
}
void func()
{
    int *p = (int *) malloc(10 * sizeof(int));
    ……
    free_and_null(p);
    if (p == NULL) {
        p = (int *) malloc(20 * sizeof(int));
    }
    ……
}
```

修改主调函数的变量，需要传递变量指针

```
void free_and_null(void **ptr)
{
    free(*ptr);
    *ptr = NULL;
}

int *p = ……
……
free_and_null((void **)&p);
```

void **ptr是指向指针的指针



动态内存分配应用 - 例4

- 下面函数能否释放动态分配的内存，并将指向该内存的指针赋值为NULL？

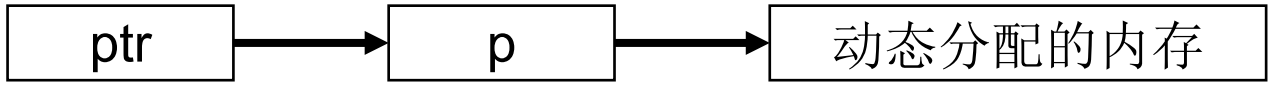
```
void free_and_null(void *ptr)
{
    free(ptr);
    ptr = NULL;
}
void func()
{
    int *p = (int *) malloc(10 * sizeof(int));
    ……
    //free_and_null(p);
    //if (p == NULL) {
        p = (int *) malloc(20 * sizeof(int));
    //}
    ……
}
```

修改主调函数的变量，需要传递变量指针

```
void free_and_null(void **ptr)
{
    free(*ptr);
    *ptr = NULL;
}

int *p = ……
……
free_and_null((void **)&p);
```

void **ptr是指向指针的指针



动态内存分配应用 - 例5

- [例10-1] 有序表使用动态内存分配方式实现
  - 如何设计input_array函数的形参?

```
int Count = 0;
int main(void)
{
    int option, value;
    int a[MAXN];

    intput_array(a);
    ……


    return 0;
}
```

```
/* 有序表输入函数 */
void input_array(int a[ ])
{
    printf("Input the number of array elements: ");
    scanf("%d", &Count);
    printf("Input an ordered array element: ");
    for (int i = 0; i < Count; i ++) {
        scanf("%d", &a[i]);
    }
}
```

动态内存分配应用 - 例5

- [例10-1] 有序表使用动态内存分配方式实现
  - 如何设计input_array函数的形参?

```
int Count = 0, Num = 0;
int main(void)
{
    int option, value;
    int *a = NULL;

    intput_array(&a);
    ……

    free(a);
    return 0;
}
```

```
/* 有序表输入函数 */
void input_array(int **pa)
{
    printf("Input the number of array elements: ");
    scanf("%d", &Count);
    Num = Count * 2;
    *pa = (int *) malloc(Num * sizeof(int));
    printf("Input an ordered array element: ");
    for (int i = 0; i < Count; i ++) {
        scanf("%d", *pa + i);
    }
}
```

注意：由于代码篇幅，此处省略了malloc返回值检查

动态内存分配应用 - 例5

```
void insert(int **pa, int value)
{   int i, j;
    if (Count == Num) {                  /* 申请空间用完，申请更大的空间 */
        int *pb = (int *) realloc(*pa, Num * 2 * sizeof(int));
        if (pb != NULL) { *pa = pb; Num = Num * 2; }
        else { printf("Memory allocation failed. \n"); return; }
    }
    /* 定位：待插入的位置即退出循环时i的值 */
    for (i = 0; i < Count && value > (*pa)[i]; i++) ;
    for (j = Count - 1; j >= i; j--)     /* 腾位：将a[i]~a[Count-1]向后顺移一位 */
        (*pa)[j+1] = (*pa)[j];
    (*pa)[i] = value;                    /* 插入：将value 的值赋给a[i] */
    Count++;                             /* 增1：数组a中待处理的元素数量增1 */
}
```

内容小结

- 静态内存有序表
  - 创建、插入、删除、查询
- 函数的顺序调用与嵌套调用
- 程序的内存模型
  - 栈、堆、全局区、常量区、代码区
- 动态内存分配函数
  - malloc, calloc, realloc, free (野指针，内存泄漏)
- 动态内存分配应用
  - 函数返回指针
  - 动态内存有序表