目录

1. Dynamic memory allocation
   1. Dynamic memory allocation
      1. A variable-length array
   2. Malloc
      1. Variables and pointers and making space
   3. Freeing malloc'd memory
      1. Memory leaks
      2. Example: available memory
   4. Realloc

Dynamic memory allocation

When data is static:

- sizes of data is known to the compiler

Examples:

切换行号显示

```
int   x;           // 4 bytes: a 32-bit integer value
char *cp;          // 8 bytes: address of a char
typedef struct {float x; float y} Point;
Point p;           // 8 bytes: two 32-bit float values
char  s[20];       // array: 20 1-byte chars

```

In many applications, you do not know how much data you will need.

For example:

切换行号显示

```
char name[MAXNAME];  // how long is a name?
char line[MAXLINE];  // what's the longest line?
char words[MAXWORDS][MAXWORDLENGTH];
                     // how many words are there?
                     // how long is each word?

```

How do we know how big to make each array?

切换行号显示

```
#define MAXNAME ??
#define MAXLINE ??
#define MAXWORDS ??
#define MAXWORDLENGTH ??

```

The size must be fixed before the program starts, so we have to guess what the largest sizes will be.

- if we make them too small:
  - the program may seg-fault
  - continually need re-sizing and recompiling (is the user able to do this?)
- if we make them really big to be safe:
  - it is wasteful as possibly 99% of the time we'll never use it all
  - there is still no guarantee that it will be big enough

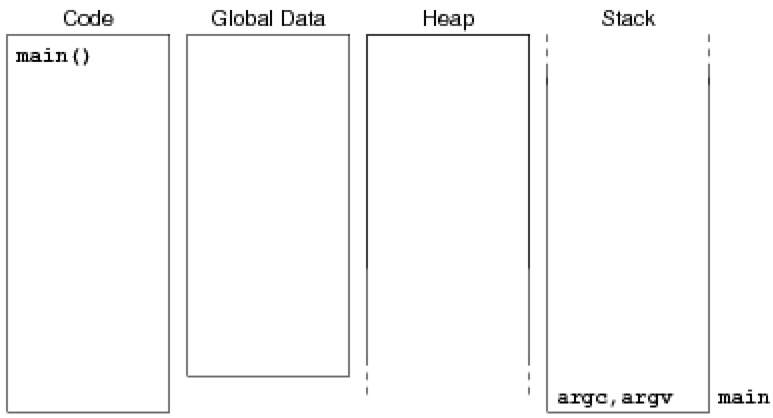
Dynamic memory allocation

- determine the size based on actual input
- allocate space that we actually need at run-time

The compiler uses dynamic memory allocation. Here is the memory model of a program:

- there are 'static segments' that are fixed at compile-time
  - *code*: fixed size, read-only segment
    - contains machine code instructions
  - *global data*: fixed size, read/write region
    - contains global variables and constant strings
- 'dynamic segments' that are 'empty' at compile time, and get filled at run-time
  - *Heap*: used to store data defined by the program, usually from *malloc()*
    - values of data change at runtime
    - in principle 'limitless'
  - *Stack*: used when functions are called
    - created/removed by the system during runtime
    - local variables in functions stored here
    - in principle 'limitless'

The *Heap* and *Stack* are drawn 'open-ended' to indicate they grow/shrink at run-time



The easiest way to use dynamic memory allocation is:

- read the size you need from the command line or *stdin*
- create the data structure of this size

This approach is:

- flexible
- no wastage
- guaranteed correct size (if you've calculated it correctly of course)

Consider the following problem:

- the first number on *stdin* indicates how many numbers follow
- rest of the numbers are read (into an array)

Examples of data could be:

- **6** 25 -1 999 42 -16 64
- **20** 34 76 -123 1 54 96 3 646 -432 -2 19 213 6667 90 6 4 99 0 101 12

A variable-length array

切换行号显示

```
// dynamicC99.c: declare a variable-length array
#include <stdio.h>
#include <stdlib.h>
int main() {
   int numberOfElems;
   if (scanf("%d", &numberOfElems) == 1) { // read the size
      int vector[numberOfElems];           // array of variable size
      for (int i = 0; i < numberOfElems; i++) {
              scanf("%d", &vector[i]);
      }
      printf("I have read: ");
      for (int i = 0; i < numberOfElems; i++) {
              printf("%d ", vector[i]);
      }
      putchar('\n');
   }
   return EXIT_SUCCESS;
}
```

- this program fragment generates a compile-time error in older versions of the C compiler (C90)

Malloc

The user can request memory space to store data.

- do this by calling the function **malloc()** (defined in *stdlib.h*)

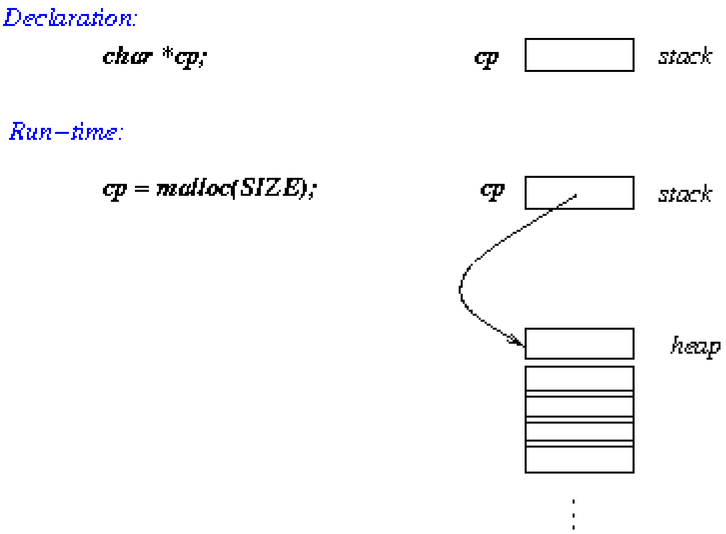
Example: if we declare:

切换行号显示

```
char *cp;          // give me 8 bytes to store an address
cp = malloc(SIZE); // give me SIZE bytes, and put the address in cp

```

Pictorially:



A full example.

切换行号显示

```
// dynamica.c: read ints from stdin into a dynamic data structure
#include <stdlib.h>
#include <stdio.h>

int main(void) {
   int number;
   if (scanf("%d", &number) == 1) { // read the size
       int *dynamica;               // this will point to an int
       dynamica = malloc(number * sizeof(int)); // alloc this many bytes
       if (dynamica == NULL) {
          fprintf(stderr, "Run out of memory .. must die\n");
          return EXIT_FAILURE;
       }
       printf("I'm now going to read %d numbers\n", number);
       int *d;
       for (d = dynamica; d < dynamica + number; d++) {
           if (scanf("%d", d) == 1) { // read an int and put at address d
                printf("yeh, I've read %d\n", *d);
           }
       }
       // do nothing with this data
       free(dynamica);
   }
   return EXIT_SUCCESS;
}
```

Note:

- if *malloc()* returns NULL, the heap is full, there is no memory left, and you need to handle this
  - this is serious: it is reported to *stderr*
- I do not change the pointer *dynamica*
  - *d* is a pointer that puts the input data into the heap
- this is pointer arithmetic
  - it is *quintessential C*
  - violates the style guide, acceptable in this course, but use with great caution

Compile and execute:

```
prompt$ dcc dynamica.c
prompt$ echo 3 123 456 789 | ./a.out
I'm now going to read 3 numbers
yeh, I've read 123
yeh, I've read 456
yeh, I've read 789
```

Variables and pointers and making space

When we declare a variable, memory space is created for the variable. For example:

- the following program creates a struct
- the size of this struct is 12 bytes (including padding)
- it defines a variable *mandy* with this struct as type

切换行号显示

```
//  mandy0.c: populate a struct and print it
#include <stdio.h>
#include <stdlib.h>

struct somedata {
   int age;       // 4 bytes
   float weight;  // 4 bytes
   char gender;   // 1 byte: either 'f' or 'm'
};                // aligns to 12 bytes

int main(void) {
    struct somedata mandy; // this is a struct 12 bytes long
    mandy.age    = 21;     // fill in the struct
    mandy.weight = 65.5;
    mandy.gender = 'f';
    printf("%d %.2f %c\n", mandy.age, mandy.weight, mandy.gender);
    return EXIT_SUCCESS;
}
```

- compiling and executing

```
prompt$ dcc mandy0.c
prompt$ ./a.out
21 65.50 f
```

The following program uses a pointer to fill in and print the struct

- 切换行号显示

```
//  mandy1.c: using a pointer, populate a struct and print it
#include <stdio.h>
#include <stdlib.h>

struct somedata {
   int age;
   float weight;
   char gender;
};

int main(void) {
    struct somedata mandy;
    struct somedata *p;
    p = &mandy;                      // p is assigned mandy's address
    p->age    = 21;                  // fill in the struct
```

```
    p->weight = 65.5;
    p->gender = 'f';
    printf("%d %.2f %c\n", p->age, p->weight, p->gender);
    return EXIT_SUCCESS;
}
```

- note that we again have a variable *mandy* in this program
- but we use *mandy's* address to populate the struct
- compiling and executing

```
prompt$ dcc mandy1.c
prompt$ ./a.out
21 65.50 f
```

Can we get rid of the variable completely?

切换行号显示

```
// mandybad.c: get rid of the variable, and just use a pointer
#include <stdio.h>
#include <stdlib.h>

struct somedata {
   int age;
   float weight;
   char gender;
};

int main(void) {
   struct somedata *p;

   p->age    = 21;
   p->weight = 65.5;
   p->gender = 'f';
   printf("%d %.2f %c\n", p->age, p->weight, p->gender); // print all
   return EXIT_SUCCESS;
}
```

- compiling and executing

```
prompt$ gcc mandybad.c
prompt$ ./a.out
Segmentation fault
```

- **Why?**
  - pointer *p* is declared but not defined
    - it is <u>not</u> initialised (it does not contain a heap address)
- (Note the use of *gcc* instead of *dcc*
  - *dcc* reports *variable 'p' is uninitialized*)

Can we avoid bringing variable *mandy* back?

切换行号显示

```
//  mandy2.c: using a malloc(), populate a struct and print it
#include <stdio.h>
#include <stdlib.h>

struct somedata {
```

```
    int age;
    float weight;
    char gender;  // either f or m
};

int main(void) {
    struct somedata *p;
    p = malloc (sizeof(struct somedata)); // allocates 12bytes to p
    if (p == NULL) {
       fprintf(stderr, "Out of memory\n");
       return EXIT_FAILURE;
    }
    p->age    = 21;                      // fill in the struct
    p->weight = 65.5;
    p->gender ='f';
    printf("%d %.2f %c\n", p->age, p->weight, p->gender); // print all
    free(p);                             // return memory
    p = NULL;                            // 'clean' the pointer
    return EXIT_SUCCESS;
}
```

Compile and execute

```
prompt$ dcc mandy2.c
prompt$ ./a.out
21 65.50 f
```

What the function *malloc()* does:

- it is defined in *stdlib.h*
- its argument is number of bytes required
- it will allocate memory in the heap
- it returns the address of the start of this allocated memory (if successful)
  - if unsuccessful, it returns NULL
- the heap space is *owned* by the program for the whole life of the program

AND

- variable *mandy* is not required!
- we use just one pointer

Freeing malloc'd memory

Dynamic memory should be **freed** when no longer required

- for every *malloc()* there should be a corresponding *free()*
- when free'd, that heap space may be re-used by the system
- after free'ing the pointer still contains the address ...
  - *...but of what?*
    - **the pointer is undefined after the memory it points to is free'd**
    - it is said to be a **dangling pointer**
    - it is said to **point to garbage**
  - the pointer can be **cleaned** ('undangled') by assigning NULL to it

The system will free all malloc'd memory when the program terminates of course.

C programmers must manage (allocate and free) memory used by their program

If you *malloc* a lot, and do not *free* the memory, eventually

- you will die because memory is exhausted, or
- the system will 'kill' you

Free in the right order if malloc'd data contains more malloc'd data

切换行号显示

```
node *nptr = malloc(sizeof(node));
if (nptr == NULL) {
    ...
}
else {
    nptr->field = malloc(SIZE); // this is a malloc inside a malloc
    if (nptr->field == NULL) {
        ...
    }
    else {
        ...
        free(nptr->name);    // first free the deepest malloc
    }
    free(nptr);              // then free the shallowest malloc
}
nptr = NULL                  // then make sure nothing is left dangling

```

Does the system really re-cycle free'd memory?

Examine the following program carefully.

切换行号显示

```
// free_check.c: print the pointer returned by successive calls to malloc
#include <stdio.h>
#include <stdlib.h>

#define MALLNUMBER 10
#define MALLSIZE   16

int main() {
   int *p;
   for (int i=0; i<MALLNUMBER; i++) {
      p = malloc(MALLSIZE);
      printf("Memory allocated at %p\n", (void *)p); // not checking for NULL
      free (p);                                  // because freeing it
   }
   p = NULL;                                     // undangle p
   return EXIT_SUCCESS;
}
```

Compile and execute with *dcc*

```
prompt$ dcc free_check.c
prompt$ ./a.out
Memory allocated at 0x602000000030
Memory allocated at 0x602000000050
Memory allocated at 0x602000000070
```

```
Memory allocated at 0x602000000090
Memory allocated at 0x6020000000b0
Memory allocated at 0x6020000000d0
Memory allocated at 0x6020000000f0
Memory allocated at 0x602000000110
Memory allocated at 0x602000000130
Memory allocated at 0x602000000150
```

The system has **NOT** recycled the free'd memory!

What about gcc?

```
prompt$ gcc free_check.c
prompt$ ./a.out
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
Memory allocated at 0x555ee152c260
```

The system **HAS** recycled the free'd memory!

Memory leaks

Memory needs careful management

- first rule is: ***don't lose your memory***
- if you overwrite a pointer returned by a *malloc()*, you've lost memory!
  - the memory is still allocated, but it is not accessible
    - you cannot even free it
      - because you've lost its address!!

This is called a ***memory leak***

- 'the memory has leaked away, forever'

In very large systems

- memory leaks can accumulate over time
- the program can eventually crash when memory is exhausted
- there is much software on the market that specialises in finding memory leaks in C

An example of a leak:

切换行号显示

```
char *getMemory (int n) { // returns a pointer to 'n' bytes
   return malloc(n);
}
void callingFunc() {
   getMemory(20);          // get me 20 bytes
}
```

- this leak is an example of very sloppy programming

The following program leaks 'elephant'

切换行号显示

```
// elephantleak.c: malloc 'elephant' and let it leak away
#include <stdlib.h>
#include <stdio.h>

#define NUM 10

int main(void) {
   char *reserve = malloc(NUM); // grab NUM bytes
   if (reserve == NULL) {
      fprintf(stderr, "Sorry, out of memory\n");
      return EXIT_FAILURE;
   }
   *reserve++ = 'e'; // changing this ptr is living dangerously
   *reserve++ = 'l';
   *reserve++ = 'e';
   *reserve++ = 'p';
   *reserve++ = 'h';
   *reserve++ = 'a';
   *reserve++ = 'n';
   *reserve++ = 't';
   *reserve++ = '\0';
   printf("%s\n", reserve-9);
   reserve = malloc(NUM);    // grab another NUM bytes
   // ...
   return EXIT_SUCCESS;
}
```

- this is a poorly written program that is also leaky
- the first address in *reserve* is lost when the second *malloc()* is done
  - there is no way of getting the previous *reserve* back once this is done
    - you've lost that *reserve*, and its 'elephant' of course

When a program terminates, all memory the program uses is automatically freed.

- it is not dangerous to forget to free memory at the end of a program, but ...
  - ... it is poor style

It **is dangerous** not to free memory within the program

- if programs execute non-stop for days, weeks or months, memory leaks may kill the program

Example: available memory

We know *malloc()* will return NULL if there is no more memory

- *can we use that fact to determine how much memory we can ask for?*

切换行号显示

```
// mandymal.c: mandy malicious, leak all of memory!
#include <stdio.h>
#include <stdlib.h>

typedef struct block { // 'block' is 1 kilobyte = 1024 bytes
   char dummy[1024];
} Block;

int exhaust(void) {
   int counter;
   Block *blockp;
```

```

    counter = 0;
    blockp  = malloc(sizeof(Block));      // grab first block
    while (blockp != NULL) {
        counter++;                        // keep count
        blockp  = malloc(sizeof(Block)); // get a new block
        printf("%d %p\n", counter, (void *)blockp);
        // DO NOT free(blockp);
    }
    // reach here when memory is exhausted
    return counter;
}

int main(void) {
    printf("Found %d blocks\n", exhaust());
    return EXIT_SUCCESS;
}
```

The program does not finish or exhaust memory: it is *killed*

Using *dcc*:

```
1 0x619000000580
2 0x619000000a80
...
1481344 0x61907104f880
Killed
```

Using *gcc*

```
1 0x55b596adc670
2 0x55b596adda90
...
2056374 0x55c424e019d0
Killed
```

Realloc

If the space returned by a *malloc()* is too little, you can create more by calling *realloc()*.

切换行号显示

```
char *p = malloc(SIZEINBYTES)
.
.
char *pext = realloc(p, BIGGERSIZEINBYTES);
.
.
```

Notice:

- the first argument of *realloc()* is the pointer returned by *malloc()*
    - the new pointer returned by *realloc()* *pext* may or may not be the same address as *p*
        - but 'do you care?'
    - the old and new pointers will normally have the same type, but do not have to
- *realloc()* will return NULL if memory is exhausted (just like *malloc()*)
- the data pointed to by the *malloc* pointer is copied to the *realloc* pointer
- the *malloc* pointer is free'd (so you should not free it yourself)

Consider the following program:

切换行号显示

```
// mandyextra.c: realloc a malloc
//               malloc too few bytes to store the string 'SPACE', so do a realloc

#include <stdio.h>
#include <stdlib.h>

#define TOOFEW 4
#define ENOUGH 6

int main(void) {
   char *p = malloc(TOOFEW);   // allocate TOOFEW bytes
   if (p==NULL) {
       fprintf(stderr, "malloc() failed\n");
       return EXIT_FAILURE;
   }

   *p     = 'S';               // assign all the letters
   *(p+1) = 'P';
   *(p+2) = 'A';
   *(p+3) = 'C';
   // *(p+4) = 'E';            // dcc generates a runtime error: why??
   // printf("%s\n", p);       // dcc generates a runtime error: why???

   char *pextra = realloc(p, ENOUGH); // re-allocate p to ENOUGH bytes  ...
                               // ... p must come from a malloc()
                               // the system copies *p  to pextra
                               // the system frees p (cannot be used)
   if (pextra==NULL) {
       fprintf(stderr, "realloc() failed\n");
       return EXIT_FAILURE;
   }
   *(pextra+4) = 'E';             // assign the rest of the string
   *(pextra+5) = '\0';            // assign a terminator
   printf("%s\n", pextra);
   free(pextra);
   return EXIT_SUCCESS;
}
```

Compile and execute

```
prompt$ dcc mandyextra.c
prompt$ ./a.out
SPACE
```

Notice there are just 2 variables in the program: *p* and *extra*, both pointers.

Dynamic (2019-06-24 17:56:37由AlbertNymeyer编辑)