Computer Programming

Pointers. Dynamic memory allocation

Marius Minea

marius@cs.upt.ro

19 November 2013

When to use pointers ?

When the language *forces* us to:

arrays (memory blocks) cannot be passed / returned from functions
  only their *address* (array name is its address)

addresses carry *no size* information $\Rightarrow$ must pass size parameter

strings: a string (constant or not) is a `char *`
  need not pass size, since null-terminated

functions: a function name is its address

When a function needs to modify variable passed from outside
  pass *address* of variable

WARNING! Any address passed to a function needs to be valid (point to allocated memory)
  functions *use* their arguments $\Rightarrow$ must have valid values

Use pointers rather than indices when possible

A pointer is like an index, but points *directly* to object of interest

`a[i]` means `*(a + i)` $\Rightarrow$ must do addition first

$\Rightarrow$ loop with `++p` rather than `++i`, accesss `*p`

```
char *strcat_i(char *dest, const char *src) // NO
{
  int i = 0, j;
  while (dest[i]) ++i;
  for (j = 0; src[j]; ++j)
    dest[i+j] = src[j];
  dest[i+j] = '\0';
  return dest;
}
char *strcat_p(char *dest, const char *src) // YES
{
  char *d = dest; // need to save dest for return
  while (*d) ++d;
  while (*d++ = *src++);
  return dest;
}
```

Indices or pointers: use sensibly

Declare index in `for` loop header whenever possible (since C99)

enforces scope, visually clear, avoids affecting other loops

Do use indices if more suggestive, though combinations are possible

```
void matmul_i(unsigned m, unsigned n, unsigned p, double a[][n],
    double b[][p], double c[][p]) {
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < p; ++j) {
      c[i][j] = 0;
      for (int k = 0; k < n; ++k) c[i][j] += a[i][k]*b[k][j];
    }
}
void matmul_p(unsigned m, unsigned n, unsigned p, double a[][n],
    double b[][p], double c[][p]) {
  double *lastl = a[m];
  for (double *lp = a[0], *dp = c[0]; lp < lastl; ++lp)
    for (int j = 0; j < p; ++j, ++dp) {
      *dp = 0;
      for (int k = 0; k < n; ++k) *dp += lp[k]*b[k][j];
    }
} // could you use more pointers ? For b perhaps ?
```

Who's responsible for the memory ?

... or, when dynamic allocation becomes desirable

If function needs arrays only for *temporary storage*, can use *variable-length arrays* (since C99)

e.g. array of `n` elements, `n` passed as argument

But, if function has array result, array must be *allocated* and *passed from outside*
(including length, function has no way of knowing it!)

e.g. add two vectors, multiply two matrices

Burden on caller becomes higher the more flexible inputs are

multiply two bignums – caller needs to determine size of product

concatenate array of strings – caller needs to precompute length

and function is less natural (has address of *result* as *argument*)

$\Rightarrow$ would like to delegate this burden to the called function
should decide amount of memory needed, allocate, return result

Dynamic allocation

Dynamic memory allocation (functions from `stdlib.h`)
allows us to obtain *at runtime* a memory block of the desired size

```
void *malloc(size_t size);          allocates size bytes
void *calloc(size_t n, size_t size);    n*size bytes of 0
```

Return value: address of allocated memory or `NULL` on error
(insufficient memory) $\Rightarrow$ *must test result!*

Frequent use: dynamically allocate array of `n` objects of type `T`:

```
T *p = malloc(n * sizeof(T));
if (p) { /* non-null, successful: use p }
```

Reallocating and freeing memory

Changing the size of a memory zone allocated with `malloc/calloc`:

```
void *realloc(void *ptr, size_t size);
```
requests new `size`

Can only resize memory allocated *dynamically* (not static arrays)

May move memory contents and return address different from `ptr`

```
if (p1 = realloc(p, size)) { p = p1; /* now use p */ }
else { /* reallocation failed, but we still have p */ }
```

`realloc(NULL, len)` works like `malloc(len)`

$\Rightarrow$ loop can init `p = NULL` and trigger `realloc(p,...)` in first cycle

Allocated memory *must be freed* when no longer needed

```
void free(void *ptr);
```
frees memory block allocated with `c/malloc`

If forgotten, long-running programs (server, browser, etc.) may consume memory (*memory leaks*) until exhausted.

When and how to use dynamic allocation

NO when needed memory amount known in advance

YES, when needed memory amount not known at compile-time
(dynamically linked structures: lists, trees; arbitrarily large input)

YES, when we must return an object created in a function
(Can't return address of local variable, lifetime is function scope)

```
char *strdup(const char *s) {       // creates copy of s
  char *d = malloc(strlen(s) + 1); // enough for s and '\0'
  return d ? strcpy(d, s) : NULL; // copy and return dest
}
```

YES, to copy and keep an object read into a temporary variable

```
char *tab[10], buf[81];
int i = 0;
while (i < 10 && fgets(buf, 81, stdin))
  tab[i++] = strdup(buf); // save address of copy
```

Example: reading an arbitrarily long line

```
#include <stdio.h>
#include <stdlib.h>
#define BLOCK 64     // suitable size, not too small
char *getline(void) {
  char *tmp, *s = NULL;    // initialize for realloc
  int c, lim = -1, size = 0; // keep room for \0
  while ((c = getchar()) != EOF) {
    if (size >= lim)       // allocated block full
      if (!(tmp = realloc(s, (lim+=BLOCK)+1))) { // enlarge
        ungetc(c, stdin); break; // if no more room
      } else s = tmp;      // use new address
    s[size++] = c;         // add last char
    if (c == '\n') break;  // end on newline
  }              // end with \0, reallocate only size needed
  if (s) { s[size++] = '\0'; s = realloc(s, size); }
  return s;
}
```