Kỹ Thuật Lập Trình

(Ngôn Ngữ Lập Trình C)

Cấp phát động

Bộ Nhớ Động

- Chương trình không thể biết trước bao nhiêu ô nhớ người dùng sẽ cần.
- Giải pháp đơn giản là định ra kích thước xác định trước

  char buffer[SIZE];

- Nhưng giải pháp này không phải lúc nào cũng phù hợp.
- Cấp phát bộ nhớ động cung cấp vùng nhớ theo yêu cầu tại thời điểm chạy chương trình
  - Điều này cho phép lập trình viên hoàn toàn kiểm soát đối tượng điều khiển trong phần mềm

Các Vùng Nhớ Khác Nhau

- Bốn vùng nhớ cơ bản: *vùng nhớ các hằng*, *vùng dữ liệu tĩnh*, *ngăn xếp*, *vùng nhớ động*.
- Vùng nhớ các hằng
  - Dùng cho các xâu hằng và kiểu hằng đã được định nghĩa
  - Các giá trị chỉ đọc
  - Quản lý bởi chương trình dịch
- Vùng dữ liệu tĩnh
  - Vùng dữ liệu xuất hiện khi chương trình chạy
  - Có thể đọc và viết
  - Quản lý bởi chương trình dịch

Các Vùng Nhớ Khác Nhau

- Ngăn xếp
  - Cung cấp vùng nhớ tạm thời cho các biến cục bộ (automatic extent)
  - Dữ liệu được sắp xếp theo trình tự FILO
  - Dữ liệu được thêm vào (push in) khi gọi hàm và được giải phóng khi kết thúc hàm (pop out)
  - Quản lý bởi chương trình dịch
- Vùng nhớ động (the heap)
  - Vùng nhớ quản lý động, quản lý bởi lập trình viên chứ không phải chương trình dịch
  - Điểu khiển bới các hàm chuẩn của C

Những Hiểm Họa Phía Trước

- Lập trình viên phải hoàn toàn chịu trách nhiệm trong việc quản lý bộ nhớ này.
- Việc quản lý vùng nhớ không có một sự trợ giúp nào từ chương trình dịch
  - Có thể gây ra lỗi thời gian thực
- Ngăn xếp có tốc độ lưu trữ nhanh hơn rất nhiều so với vùng nhớ động
  - Ngăn xếp đơn giản quản lý con trỏ, trong khi vùng nhớ động liên quan tới quản lý bộ nhớ trong
  - Nên sử dụng ngăn xếp khi có thể

Các Hàm Quản Lý Bộ Nhớ Động

- Cung cấp bởi thư viện chuẩn, các hàm này cung cấp phương thức duy nhất để truy cập vùng nhớ này.
- Cấp phát bộ nhớ

```
void *malloc(size_t size)
```

```
int *p = malloc(10 * sizeof(int));
```

```
int *p = (int *) malloc(10 * sizeof(int));
```

malloc()

- Nếu **malloc()** thành công, nó sẽ trả về con trỏ tới địa chỉ của vùng nhớ.
- Nếu không, nó sẽ trả về **NULL**.
- Thực tế, các máy tính có *bộ nhớ ảo* (virtual memory) và có thể coi vùng nhớ là không giới hạn.
- Các bạn nên có thói quen kiểm tra giá trị trả về của các hàm này.

free()

- **free()** giải phóng vùng nhớ được cấp phát bởi **malloc()**.
- Vùng nhớ được giải phóng và sẵn sàng cho việc sử dụng của các hàm khác

void free(void *p)

- Không cần ép kiểu **(cast) xxx *** về **void ***

```
int *p = (int *) malloc(10 * sizeof(int));
...
free(p);
```

calloc()

- **calloc()** giống như **malloc()** nhưng vùng nhớ được khởi tạo với giá trị zero (clear allocate).
- Giao diện hơi khác so với **malloc()**

```
void *calloc(size_t n, size_t size)
```

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

realloc()

- **realloc()** sử dụng để thay đổi vùng nhớ được cấp phát bởi **malloc()**, **calloc()**, hoặc bởi chính hàm **realloc()** .

  void *realloc(void *p, size_t size)

- Hàm đa năng
  - Khi **p** là **NULL**, tương tự như **malloc()**
  - Nếu hàm không thành công, trả về **NULL**, và vùng nhớ cũ được bảo toàn
  - Nếu **size = 0**, tương tự như hàm **free()**, và hàm trả về **NULL**

Thông Tin Thêm Về `realloc()`

- **realloc()** được sử dụng để tăng hoặc giảm vùng nhớ động.
- Nếu tăng, các phần tử cũ không đổi và các phần tử mới thêm vào không có giá trị khởi đầu.
- Nếu giảm, các phần tử cũ không đổi.
- Tuy nhiên, khi không có đủ vùng nhớ, **realloc()** sẽ cấp thêm khối nhớ mới và copy toàn bộ vùng nhớ cũ sang vùng nhớ mới, sau đó xóa vùng nhớ cũ
  - Thao tác này làm cho con trỏ trỏ tới vùng nhớ cũ trở nên không có giá trị

Quản Lý Bộ Nhớ Động

- Điều khiển bởi lập trình viên.
- Bao gồm
  - Con trỏ tới mỗi vùng nhớ
  - Cấp phát và giải phóng vùng nhớ
  - Độ dài vùng nhớ
- Lỗi xảy ra ở hầu hết các phần mềm viết bằng C.

Ví Dụ Về Sử Dụng Sai

```
char *string_duplicate(char *s)
/* Dynamically allocate a copy of a string. User
   must remember to free this memory. */
{
   /* +1 for '\0' */
   char *p = malloc(strlen(s)+1);
   return strcpy(p, s);
}

char *s1;
s1 = string_duplicate("this is a string");
...
free(s1);
```

Các Lỗi Bộ Nhớ Thường Gặp

- Con trỏ đang trỏ tới giá trị vô nghĩa
  - "memory corruption"
- Con trỏ đã trỏ về **NULL**
  - Chương trình sẽ dừng ngay
- Giải phóng con trỏ đang trỏ đến vùng nhớ không phải vùng nhớ động (stack, constant data).
- Quên không giải phóng bộ nhớ (memory leak).
- Truy cập phần tử ngoài khoảng của mảng được cấp phát.

Thói Quen Tốt

- Mỗi hàm **malloc()** đi kèm với một hàm hàm **free()**
  - Luôn tránh memory corruption và rò rỉ bộ nhớ (memory leaks)
  - Luôn sử dụng **malloc()** và **free()** trong cùng một hàm
  - Luôn tạo hàm **create()** và sau đó hàm **destroy()** đối với các đối tượng phức tạp
- Con trỏ phải khai báo khi bắt đầu
  - Hoặc bằng **NULL** hoặc bằng giá trị có trước
  - **NULL** có nghĩa là "trỏ tới không đâu"
- Con trỏ nên đặt về **NULL** sau khi được giải phóng
  - **free(NULL)** không có tác dụng

Ví Dụ Về Ma Trận

- Ma trận có kích thước cố định (3x3), kích thước ma trận được xác định tại thời điểm biên dịch (compiler time).
- Bạn có thể thấy ví dụ cho dưới đây tạo ma trận theo yêu cầu của người dùng

```
double **matrix = create_matrix(2,3);

matrix[0][2] = 5.4;

...

destroy_matrix(matrix);
```

Ví Dụ Về Ma Trận

```
double **create_matrix1(int m, int n) {
  double **p;
  int i;
  /* Allocate pointer array. */
  p = (double **) malloc(m * sizeof(double*));
  /* Allocate rows. */
  for (i = 0; i < m; ++i)
    p[i] = (double *) malloc(n * sizeof(double));
  return p;
}
```

Ví Dụ Về Ma Trận

```
double **create_matrix2(int m, int n) {
   double **p; int i;
   assert(m>0 && n>0);
   p = (double **) malloc(m * sizeof(double*));
   if (p == NULL)
      return p;
   for (i = 0; i < m; ++i) {  /* Allocate rows. */
      p[i] = (double *) malloc(n * sizeof(double));
      if (p[i] == NULL)
         goto failed;        /* Allocation failed */
   }
   return p;
failed:
   for (--i; i >= 0; --i)
      free(p[i]);    /* delete allocated memory.*/
   free(p);
   return NULL;
}
```

Ví Dụ Về Ma Trận

```
/* Destroy an (m x n) matrix. Notice, the n
   variable is not used, it is just there to
   assist using the function. */
void destroy_matrix1(double **p, int m, int n) {
  int i;
  assert(m>0 && n>0);
  for (i = 0; i < m; ++i)
    free(p[i]);
  free(p);
}
```

Ví Dụ Về Ma Trận

```
double **create_matrix(int m, int n) {
  double **p, *q;
  int i;
  assert(m>0 && n>0);
  p = (double **) malloc(m * sizeof(double*));
  if (p == NULL)
    return p;
  q = (double *) malloc(m * n * sizeof(double));
  if (q == NULL) {
    free(p);
    return NULL;
  }
  for (i = 0; i < m; ++i, q += n)
    p[i] = q;
  return p;
}
```

Ví Dụ Về Ma Trận

```
void destroy_matrix(double **p)
/* Destroy a matrix. Notice, due to the
   method by which this matrix was created,
   the size of the matrix is not required.*/
{
   free(p[0]);
   free(p);
}
```

Mảng Có Thể Mở Rộng?

- Mở rộng mảng để khắc phục nhược điểm của mảng trong C.
- Mảng có kích thước tùy theo yêu cầu. Các phần tử sẽ được thêm vào cuối mảng và mảng sẽ tự động cập nhật chỉ số.
- Nhược điểm: mảng luôn chiếm vùng nhớ liên tục, việc thêm phần tử mới sẽ dẫn tới việc sao chép toàn bộ phần tử cũ của mảng sang vị trí mới.
- Chúng ta sẽ cấp phát một vùng bộ nhớ, thay vì từng phần tử bộ nhớ.

Mảng Mở Rộng

- Phương pháp cấp phát

  Mỗi vùng nhớ :

  newsize = K * oldsize;

  – Thông thường chọn K = 2

- Demo cách sử dụng **realloc()**

Mảng Mở Rộng

```
/* Vector access operations. */
int push_back(int item);
int pop_back(void);
int* get_element(int index);

/* Manual resizing operations. */
int get_size(void);
int set_size(int size);
int get_capacity(void);
int set_capacity(int size);
```

Mảng Mở Rộng

```
#include "vector.h"
#include <stdlib.h>
#include <assert.h>

/* Private interface */
/* initial vector capacity */
static const int StartSize = 1;
/* geometric growth of vector capacity */
static const float GrowthRate = 1.5;

/* pointer to vector elements */
static int *data = NULL;
/* current size of vector */
static int vectorsize = 0;
/* current reserved memory for vector */
static int capacity = 0;
```

Mảng Mở Rộng

```
/* Add element to back of vector. Return index of new
   element if successful, and -1 if fails. */
int push_back(int item) {
  /* If out-of-space, allocate more. */
  if (vectorsize == capacity) {
    int newsize = (capacity == 0) ? StartSize :
                  (int)(capacity*GrowthRate + 1.0);
  int *p = (int *)realloc(data, newsize*sizeof(int));
  if (p == NULL)
    return -1;
  capacity = newsize; /* update data-structure */
  data = p;
  }
  data[vectorsize] = item; /* We have enough room. */
  return vectorsize++;
}
```

Mảng Mở Rộng

```
/* Return element from back of vector, and remove it
   from the vector. */
int pop_back(void) {
    assert(vectorsize > 0);
    return data[--vectorsize];
}

/* Return pointer to the element at the specified
   index. */
int* get_element(int index) {
    assert(index >= 0 && index < vectorsize);
    return data + index;
 }
```

Mảng Mở Rộng

```
/* Manual size operations. */
int get_size(void)      {return vectorsize;}
int get_capacity(void) {return capacity;}

/* Set vector size.
   Return 0 if successful, -1 if fails. */
int set_size(int size) {
  if (size > capacity) {
    int *p = (int *) realloc(data, size*sizeof(int));
    if (p == NULL)
      return -1;
    /* allocate succeeds, update data-structure */
    capacity = size;
    data = p;
  }
  vectorsize = size;
  return 0;
}
```

Mảng Mở Rộng

```
/* Shrink or grow allocated memory reserve for array.
   A size of 0 deletes the array. Return 0 if
   successful, -1 if fails. */
int set_capacity(int size) {
  if (size != capacity) {
    int *p = (int *) realloc(data, size*sizeof(int));
    if (p == NULL && size > 0)
      return -1;
    capacity = size;
    data = p;
  }
  if (size < vectorsize)
    vectorsize = size;
  return 0;
}
```