SE02 - Con trỏ

A. Lý thuyết

- Con trỏ là một biến dùng để lưu địa chỉ của một biến khác hoặc một vùng nhớ. Ví dụ:
    - Con trỏ pointer lưu địa chỉ của biến value:

```
int value = 5;
int* pointer = &value;
```

    - Yêu cầu cấp phát một vùng nhớ để lưu giá trị kiểu số nguyên sau đó lưu địa chỉ vừa cấp phát vào con trỏ pointer để quản lý.

```
int pointer = new int;
```

- Kích thước con trỏ:
    - 8 byte trong hệ thống 64 bits
    - 4 byte trong hệ thống 32 bits
    - Không thể thay đổi kích thước con trỏ trên bộ nhớ

- Con trỏ cung cấp quyền truy cập bộ nhớ cấp thấp, và tạo điều kiện cấp phát bộ nhớ động.
- Lợi ích của việc sử dụng con trỏ:
    - Bằng cách sử dụng một con trỏ, có thể lấy hoặc thay đổi giá trị được lưu tại địa chỉ.
    - Bằng cách sử dụng chỉ một biến con trỏ trong mảng trong C, có thể nhận và thay đổi toàn bộ phần tử trong mảng.
    - Một biến con trỏ duy nhất cũng có thể nhận hoặc thay đổi giá trị của tất cả các phần tử của một struct (cấu trúc) trong C, giúp việc quản lý struct đơn giản hơn.

Ví dụ:

- Sử dụng con trỏ để xây dựng kiểu dữ liệu linked list

```
struct Node {
data;
struct Node* next;
};
```

- In dữ liệu của linked list bắt dầu từ node thứ n

```
void printList(struct Node* n)
{
   while (n != NULL) {
      printf(" %d ", n->data);
      n = n->next;                  //truy xuất đến node tiếp theo sử dụng toán tử ->
   }
}
```

- Với một biến con trỏ, có thể chọn hàm trong danh sách nhiều hàm để sử dụng, giúp việc thay đổi nội dung xử lý sau đó dễ dàng hơn.

Ví dụ:

- Ta có 2 hàm như sau

```
void add(int a, int b)
{
   printf("\nAddition is " << a + b;
}
```

```
void subtract(int a, int b)
{
   cout << "\nSubtraction is " <<  a - b;
}
```

- ta có thể sử dụng fucntion pointer để thay thế switch, cho phép người dùng lựa chọn hàm sẽ được thực thi.

```
int main()
{
   // fun_ptr_arr là mảng các function pointer
   void (*fun_ptr_arr[])(int, int) = {add, subtract};
   int ch, a = 15, b = 10;

   cout << "Enter Choice: 0 for add, 1 for subtract\n";
   cin >> ch

   if (ch > 2 || ch < 0)
return 0;

   (*fun_ptr_arr[ch])(a, b);

   return 0;
}
```

- So sánh tham chiếu và tham trị:

Tham chiếu	Tham trị
Địa chỉ của biến được truyền vào hàm	Một bản sao giá trị của biến được truyền vào hàm
Thay đổi biến bên trong hàm sẽ làm thay đổi biến truyền vào hàm	Thay đổi biến bên trong hàm không làm thay đổi biến truyền vào hàm
Tham số hình thức và tham số thực tế cùng địa chỉ trong bộ nhớ	Tham số hình thức và tham số thực tế khác địa chỉ trong bộ nhớ
Tham chiếu chỉ lưu địa chỉ trong vùng nhớ Stack còn giá trị biến nằm ở vùng nhớ Heap	Tham trị lưu trực tiếp dữ liệu trong bộ nhớ Stack

Ví dụ trong C:
`void change(int *num)`
`{`
`  *num = *num + 1;`
`}`
	Ví dụ trong C:
	`void change(int num)`
	`{`
	`  num = num + 1;`
	`}`

Ví dụ trong C++ (có thể sử dụng cách của C):
`void change(int &num)`
`{`
`  num = num + 1;`
`}`
	Ví dụ trong C:
	`void change(int num)`
	`{`
	`  num = num + 1;`
	`}`

- Cấp phát bộ nhớ động
  - Với ngôn ngữ C
    - *Dùng malloc()* để cấp phát động một vùng bộ nhớ với kích thước cho trước

Cú pháp: pointer-variable = *(cast-type*)malloc(byte-size)*

Ví dụ: Cấp phát một vùng nhớ động có kích thước 400 byte chứa dữ liệu kiểu int
int ptr = (int)malloc(100* sizeof(int));*

- Dùng *calloc()* để cấp phát động một số lượng vùng nhớ liên tục được chỉ định trước

Cú pháp: pointer-variable = (cast-type*)calloc(n, element-size);
Ví dụ: Cấp phát 25 vùng nhớ liên tục chứa dữ liệu kiểu float
float ptr = (float) calloc(25, sizeof(float));*

- Dùng *free()* để giải phóng vùng nhớ đã được cấp phát bởi *malloc()* hoặc *calloc().*

Cú pháp: *free(pointer-variable ;*

- Dùng realloc() để cấp phát động lại một vùng nhớ đã được cấp phát bởi *malloc()* hoặc *calloc().*

Cú pháp: pointer-variable = *realloc(ptr, newSize);*

- Với ngôn ngữ C++
  - Dùng toán tử new để cấp phát động

Cú pháp: *new data-type;*
Ví dụ:

- Cấp phát một vùng nhớ chứa dữ liệu kiểu int rồi lưu địa chỉ vùng nhớ vào con trỏ ptr để quản lý: *int* ptr = new int*
- Cấp phát 15 vùng nhớ liên tục chứa dữ liệu kiểu float rồi lưu địa chỉ vùng nhớ đầu tiên vào con trỏ ptr để quản lý: *float* ptr = new float[15];*

- Dùng toán tử delete để giải phóng vùng nhớ được cấp phát

Cú pháp: delete pointer-variable;
Ví dụ:

- Giải phóng vùng nhớ có địa chỉ lưu tại biến con trỏ ptr: *delete ptr;*
- Giải phóng các vùng nhớ liên tục có địa chỉ vùng nhớ đầu tiên lưu tại biến con trỏ ptr: *delete [] ptr;*

- Do sau khi giải phóng vùng nhớ, con trỏ vẫn còn lưu địa chỉ vủa vùng nhớ đã được giải phóng khiến việc sử dụng con trỏ đó có thể dẫn đến nguy hiểm. Nên gán cho con trỏ đó giá trị NULL sau khi giải phóng vùng nhớ lưu tại con trỏ đó.

B. Thực hành

Bài 0:

- Cách 1: sử dụng công cụ memcheck của valgrind.
  - Giả sử chương trình test.cpp như sau:

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *p;

  // Allocation #1 of 19 bytes
  p = new char[19];

  // Allocation #2 of 12 bytes
  p = new char[12];
  free(p);

  // Allocation #3 of 16 bytes
  p = new char[16];

  return 0;
}
```

- Sau khi build chương trình trên để tạo file thực thi test, Chạy lệnh sau để sử dụng công cụ memcheck của valgrind trên file thực thi.

  valgrind --tool=memcheck --leak-check=yes --show-reachable=yes –num-callers=20 --track-fds=yes ./test

```
home@home-vm:~/Documents$ valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-c
allers=20 --track-fds=yes ./test
==6101== Memcheck, a memory error detector
==6101== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6101== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6101== Command: ./test
==6101==
==6101== Mismatched free() / delete / delete []
==6101==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==6101==    by 0x1091DC: main (test.cpp:14)
==6101==  Address 0x4dd9ce0 is 0 bytes inside a block of size 12 alloc'd
==6101==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memc
heck-amd64-linux.so)
==6101==    by 0x1091CC: main (test.cpp:13)
==6101==
==6101==
==6101== FILE DESCRIPTORS: 3 open (3 std) at exit.
==6101==
==6101== HEAP SUMMARY:
==6101==     in use at exit: 35 bytes in 2 blocks
==6101==   total heap usage: 4 allocs, 2 frees, 72,751 bytes allocated
==6101==
==6101== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==6101==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memc
heck-amd64-linux.so)
==6101==    by 0x1091E6: main (test.cpp:17)
==6101==
==6101== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==6101==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memc
heck-amd64-linux.so)
==6101==    by 0x1091BE: main (test.cpp:10)
==6101==
==6101== LEAK SUMMARY:
==6101==    definitely lost: 35 bytes in 2 blocks
==6101==    indirectly lost: 0 bytes in 0 blocks
==6101==      possibly lost: 0 bytes in 0 blocks
==6101==    still reachable: 0 bytes in 0 blocks
==6101==         suppressed: 0 bytes in 0 blocks
==6101==
==6101== For lists of detected and suppressed errors, rerun with: -s
==6101== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

- Các memory leak mà valgrind phát hiện được:

```
==6101== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==6101==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memc
heck-amd64-linux.so)
==6101==    by 0x1091E6: main (test.cpp:17)
==6101==
==6101== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==6101==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memc
heck-amd64-linux.so)
==6101==    by 0x1091BE: main (test.cpp:10)
```

- Cách 2: Sử dụng công cụ heaptrack để kiểm tra memory leak
    - Chạy lệnh sau để chạy heaptrack lên file thực thi của chương trình test ở trên:

heaptrack ./test

```
home@home-vm:~/Documents$ heaptrack ./test
heaptrack output will be written to "/home/home/Documents/heaptrack.test.9873.zst"
/usr/lib/heaptrack/libheaptrack_preload.so
starting application, this might take some time...
heaptrack stats:
        allocations:            4
        leaked allocations:     2
        temporary allocations:  1
Heaptrack finished! Now run the following to investigate the data:

  heaptrack --analyze "/home/home/Documents/heaptrack.test.9873.zst"
```

    - Chạy lệnh sau để đọc output mà heaptrack trả về:

heaptrack --analyze "/home/home/Documents/heaptrack.test.9873.zst"

- Các memory leak mà heaptrack kiểm tra được:

```
PEAK MEMORY CONSUMERS
72.70K peak memory consumed over 1 calls from
0x7f0b24caa978
  in /lib/x86_64-linux-gnu/libstdc++.so.6
72.70K consumed over 1 calls from:
    call_init
      at ./elf/dl-init.c:70
      in /lib64/ld-linux-x86-64.so.2
    call_init
      at ./elf/dl-init.c:33
      in /lib64/ld-linux-x86-64.so.2
    _dl_init
      at ./elf/dl-init.c:117
    0x7f0b24e902e8
      at ./elf/rtld.c:0
      in /lib64/ld-linux-x86-64.so.2

19B peak memory consumed over 1 calls from
main
  at /home/home/Documents/test.cpp:10
  in /home/home/Documents/test
19B consumed over 1 calls from:

16B peak memory consumed over 1 calls from
main
  at /home/home/Documents/test.cpp:17
  in /home/home/Documents/test
16B consumed over 1 calls from:
```

Bài 1: Hàm hoán đổi giá trị 2 số sử dụng con trỏ:\

```
void swap (int* a, int* b)
{
  int temp = *a;
  *a = *b;
```

```
    *b = temp;
}
```

Bài 2: Chương trình tìm độ dài chuỗi do người dùng nhập vào

- Project pointer, file get_text_length.h

Bài 3: Chương trình cho người dùng nhập vào dãy số N phần tử là số nguyên. Thực hiện các thao tác như: tính tổng của dãy số, sắp xếp dãy số tăng dần hoặc giảm dần theo thuật toán Interchange Sort.

- Project pointer, file main.cpp

Bài 4: Chương trình cho người dùng nhập vào 02 chuỗi. In tất cả các vị trí xuất hiện của chuỗi thứ 2 trong chuỗi 1 (giả sử người dùng nhập vào chuỗi 1 dài hơn chuỗi 2)

- Project pointer, file find_text_in_text.h.