VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

## Cấu Trúc Dữ Liệu và Giải Thuật (DSA)

**DSA1 - HK241**

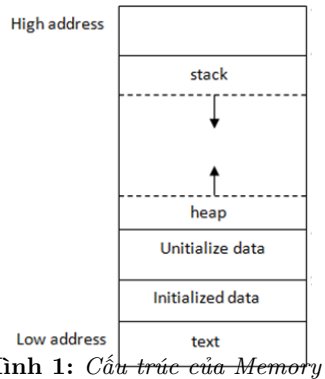
# Thiết kế quản lý bộ nhớ cho máy tính

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 6/2024

# Mục lục

# 1 Introduction




**Hình 1:** *Cấu trúc của Memory Layout*

1. **Text Segment (Phân đoạn Mã)**:
   - Nằm ở các địa chỉ thấp, **Text Segment** lưu trữ mã máy của chương trình đã được biên dịch. Đây là nơi chứa các lệnh thực thi của chương trình.
   - Nội dung trong phân đoạn này thường là *read-only* (chỉ đọc) để ngăn chương trình vô tình sửa đổi các lệnh của nó.
2. **Initialized Data Segment (Phân đoạn Dữ liệu Đã khởi tạo)**:
   - Phân đoạn này chứa các biến toàn cục và biến tĩnh được khởi tạo với một giá trị trước khi chương trình bắt đầu. Ví dụ, nếu bạn khai báo một biến toàn cục như `int x = 5;`, nó sẽ được lưu trữ ở đây.
   - Nó là một phần của phân đoạn dữ liệu của chương trình.
3. **Uninitialized Data Segment (Phân đoạn Dữ liệu Chưa khởi tạo - BSS)**:
   - Phân đoạn này chứa các biến toàn cục và biến tĩnh được khai báo nhưng chưa được khởi tạo bởi lập trình viên. Ví dụ, `int x;` mà không có giá trị khởi tạo sẽ được lưu trữ ở đây.
   - Hệ điều hành sẽ khởi tạo phân đoạn này với giá trị 0 trước khi chương trình bắt đầu.
4. **Heap**:
   - Phân đoạn **Heap** là nơi diễn ra việc cấp phát bộ nhớ động, được quản lý bởi lập trình viên sử dụng các hàm như `malloc` hoặc `new` trong các ngôn ngữ như C/C++.
   - **Heap** phát triển theo hướng lên trên (hướng tới các địa chỉ bộ nhớ cao hơn) khi bộ nhớ động được cấp phát trong thời gian chạy.
5. **Stack**:
   - Phân đoạn **Stack** được sử dụng để quản lý các lời gọi hàm, bao gồm các biến cục bộ, địa chỉ trả về, và tham số hàm.
   - **Stack** phát triển theo hướng xuống dưới (hướng tới các địa chỉ bộ nhớ thấp hơn) khi các hàm được gọi và các biến cục bộ được tạo ra.
   - Khi một hàm được gọi, một khung **stack** mới được đẩy vào **stack**, và khi hàm trả về, khung **stack** sẽ được bật ra.
6. **High and Low Addresses (Địa chỉ Cao và Thấp)**:
   - Hình ảnh này chỉ ra vị trí tương đối của các phân đoạn trong bộ nhớ, với các địa chỉ thấp bắt đầu từ dưới và các địa chỉ cao ở trên.
   - **Stack** thường bắt đầu từ một địa chỉ cao và phát triển xuống dưới, trong khi **Heap** bắt đầu từ một địa chỉ thấp và phát triển lên trên.

```cpp
#include <iostream>

// Text Segment (Phân đoạn Mã): Phần này chứa mã chương trình đã được biên dịch

int global_var = 42; // Initialized Data Segment
int uninit_global_var; // Uninitialized Data Segment
void function(int param) {
    int local_var = param; // Stack Segment
    std::cout << "Local variable: " << local_var << std::endl;
}

int main() {
    // Heap Segment: Cấp phát bộ nhớ động
    int* heap_var = new int[10]; // Cấp phát mảng 10 phần tử trên heap
    heap_var[0] = 99; // Gán giá trị cho phần tử đầu tiên của mảng heap

    std::cout << "Global variable: " << global_var << std::endl;
    std::cout << "Uninitialized global variable (default to 0): " <<
    ↪  uninit_global_var << std::endl;
    std::cout << "Heap variable: " << heap_var[0] << std::endl;

    function(10); // Gọi hàm để minh họa stack

    delete[] heap_var; // Giải phóng bộ nhớ trên heap

    return 0; // Kết thúc chương trình
}
```

## 1.1 Stack Segment (Phân đoạn Stack)

task sau

## 1.2 Heap Segment (Phân đoạn Heap)

**Heap** là một vùng bộ nhớ lớn, được sử dụng để cấp phát động cho các đối tượng hoặc biến có kích thước không cố định trong thời gian chạy. Không giống như **stack**, bộ nhớ được cấp phát trên heap không tự động được giải phóng khi một hàm kết thúc. Lập trình viên phải quản lý việc cấp phát và giải phóng bộ nhớ trên heap một cách thủ công.

1. **Cấp phát Bộ nhớ trên Heap:**

   Trong C++, bộ nhớ trên heap thường được cấp phát bằng cách sử dụng toán tử `new` (hoặc `malloc` trong C). Ví dụ, khi bạn cần cấp phát một mảng hoặc một đối tượng có kích thước không xác định trước trong thời gian biên dịch, bạn sẽ sử dụng heap.

   ```cpp
   int* arr = new int[100];  // Cấp phát mảng 100 phần tử trên heap
   ```

   Bộ nhớ cấp phát trên heap có thể tồn tại cho đến khi nó được giải phóng bằng cách sử dụng toán tử `delete` (hoặc `free` trong C).

2. **Giải phóng Bộ nhớ trên Heap:**

   Khi không còn cần thiết sử dụng bộ nhớ đã cấp phát trên heap, lập trình viên phải giải phóng bộ nhớ đó để tránh rò rỉ bộ nhớ (*memory leak*).

   ```cpp
   delete[] arr;  // Giải phóng bộ nhớ cấp phát cho mảng trên heap
   ```

3. **Ưu điểm của Heap:**

- **Linh hoạt:** Heap cho phép cấp phát bộ nhớ cho các đối tượng hoặc mảng có kích thước không cố định.
- **Thời gian sống dài:** Các đối tượng trên heap có thể tồn tại trong suốt vòng đời của chương trình nếu cần, không phụ thuộc vào phạm vi của hàm.

4. **Nhược điểm của Heap:**
   - **Tốc độ:** Việc cấp phát và giải phóng bộ nhớ trên heap thường chậm hơn so với stack do yêu cầu quản lý phức tạp.
   - **Rò rỉ bộ nhớ (Memory Leak):** Nếu lập trình viên không giải phóng bộ nhớ trên heap sau khi sử dụng, chương trình sẽ tiếp tục chiếm dụng bộ nhớ, dẫn đến rò rỉ bộ nhớ.
   - **Fragmentation (Phân mảnh):** Sau nhiều lần cấp phát và giải phóng, heap có thể bị phân mảnh, khiến không gian bộ nhớ trống không liền kề, gây khó khăn trong việc cấp phát bộ nhớ lớn.

# 2 Chuyển đổi text thành các đối tượng

## 2.1 Các khai báo được dùng

- `Type ID`
  Định nghĩa một kiểu dữ liệu và gán cho định danh.
- `Call ID`
  Gọi một định danh đã được định nghĩa trước đó.
- `Delete ID`
  Xóa một định danh đã được định nghĩa.
- `Type: Integer | Long | Character | Integer[List Int] | Long[List Int] | Character[List Int];`

  Trong đó:

  - `Integer, Long, Character`: Các kiểu dữ liệu cơ bản.
  - `Integer[List Int], Long[List Int], Character[List Int]`: Kiểu dữ liệu dạng mảng với số phần tử được xác định bởi một giá trị nguyên (`Int`).

**Ví Dụ**

- `Integer x;`
  Khai báo biến `x` có kiểu `Integer`.
- `Integer x[5];`
  Khai báo biến `x` là một mảng kiểu `Integer` có 5 phần tử.
- `Integer x[5, 6];`
  Khai báo biến `x` là một mảng hai chiều có kích thước `5x6`.
- `Call x;`
  Gọi đến biến `x` đã được khai báo trước đó.
- `Delete x;`
  Xóa biến `x` đã được khai báo.
- `Character c;`
  Khai báo biến `c` có kiểu `Character`.

## 2.2 Đối tượng Type và Declaration

### 2.2.1 Đối tương Type và các class Con của nó

1. **Lớp** `Type`:
   Đây là lớp trừu tượng cơ sở chứa hai phương thức thuần ảo:
   - `nameType()` const: Phương thức này trả về tên của kiểu dữ liệu dưới dạng chuỗi.
   - `sizeType()` const: Phương thức này trả về kích thước của kiểu dữ liệu dưới dạng số nguyên.
2. **Lớp** `IntegerType`:
   Lớp này kế thừa từ `Type` và biểu diễn kiểu số nguyên (`Integer`) với kích thước 4 byte.
   - `nameType()` const: Trả về chuỗi `"IntegerType()"`.
   - `sizeType()` const: Trả về `4`.
3. **Lớp** `LongType`:
   Lớp này kế thừa từ `Type` và biểu diễn kiểu số nguyên dài (`Long`) với kích thước 8 byte.
   - `nameType()` const: Trả về chuỗi `"LongType()"`.
   - `sizeType()` const: Trả về `8`.

4. **Lớp** `CharacterType`:
   Lớp này kế thừa từ `Type` và biểu diễn kiểu ký tự (`Character`) với kích thước 1 byte.
   - `nameType()` const: Trả về chuỗi `"CharacterType()"`.
   - `sizeType()` const: Trả về `1`.
5. **Lớp** `ArrayType`:
   Lớp này kế thừa từ `Type` và biểu diễn kiểu mảng.
   - Thuộc tính `sizeArray`: Kích thước của mảng, biểu thị số lượng phần tử trong mảng.
   - Thuộc tính `typeArray`: Con trỏ đến kiểu dữ liệu của các phần tử trong mảng.
   - `nameType()` const: Trả về chuỗi biểu diễn mảng, ví dụ: `"Array(5, IntegerType())"`.
   - `sizeType()` const: Trả về kích thước tổng của mảng bằng cách nhân số phần tử với kích thước của từng phần tử.

### 2.2.2 Đối tương Declaration và các class Con của nó

1. **Lớp Declaration (Lớp cơ sở trừu tượng)**
   Lớp `Declaration` là lớp cơ sở trừu tượng đại diện cho các loại khai báo trong hệ thống.
   - **Thuộc tính**:
     - `name`: Tên của biến hoặc đối tượng được khai báo.
   - **Phương thức**:
     - `str()`: Phương thức thuần ảo (pure virtual), yêu cầu các lớp con cài đặt. Phương thức này trả về chuỗi mô tả khai báo.
   - **Phương thức khởi tạo**: Nhận tham số là một chuỗi `name` và lưu trữ vào thuộc tính `name`.
2. **Lớp NewDeclaration (Khai báo biến mới)**
   Lớp `NewDeclaration` kế thừa từ `Declaration`, đại diện cho việc khai báo biến mới với một kiểu dữ liệu cụ thể.
   - **Thuộc tính**:
     - `type`: Con trỏ tới một đối tượng của lớp `Type` biểu diễn kiểu dữ liệu của biến.
   - **Phương thức**:
     - `str()`: Trả về chuỗi mô tả khai báo biến, ví dụ `"InitDeclaration(x, IntegerType())"`.
   - **Phương thức khởi tạo**: Nhận tham số `name` và `type` để khởi tạo cho biến khai báo.
   - **Phương thức hủy**: Giải phóng bộ nhớ cấp phát cho thuộc tính `type`.

   Ví dụ

   ```
   Integer var;
   => CallDeclaration("var", new IntegerType())

   Long Votien[2,3];
   => CallDeclaration("Votien", new ArrayType(2, new ArrayType
      (3, new LongType())))
   ```

3. **Lớp CallDeclaration (Gọi biến)**
   Lớp `CallDeclaration` kế thừa từ `Declaration`, đại diện cho việc gọi một biến đã được khai báo trước đó.
   - **Phương thức**:

  - `str()`: Trả về chuỗi mô tả lời gọi biến, ví dụ `"CallDeclaration(x)"`.
- **Phương thức khởi tạo**: Nhận tham số `name` là tên của biến cần gọi.

Ví dụ

```
Call var;
=> CallDeclaration("var")
```

4. **Lớp DeleteDeclaration (Xóa biến)**
Lớp `DeleteDeclaration` kế thừa từ `Declaration`, đại diện cho việc xóa một biến đã được khai báo.
   - **Phương thức**:
     - `str()`: Trả về chuỗi mô tả việc xóa biến, ví dụ `"DeleteDeclaration(x)"`.
   - **Phương thức khởi tạo**: Nhận tham số `name` là tên của biến cần xóa.

Ví dụ

```
Delete var;
=> DeleteDeclaration("var")
```

## 2.3 Đối tượng ListDelarations

1. **Hàm khởi tạo ListDelarations(int sizeMax)**
Hàm khởi tạo này dùng để tạo một danh sách các khai báo.
   - Khởi tạo biến `sizeMax` để lưu kích thước tối đa của danh sách.
   - Đặt biến `size` bằng 0, biểu thị số lượng khai báo hiện tại.
   - Khởi tạo mảng `listDeclarations` với kích thước `sizeMax`, mỗi phần tử là một con trỏ tới một đối tượng `Declaration`.
2. **Hàm void push(string declaration)**
Hàm `push()` thêm một khai báo mới từ chuỗi `declaration` vào danh sách.
   - Kiểm tra xem danh sách đã đầy chưa (nếu `size == sizeMax`), nếu đúng thì không thêm gì cả.
   - Phân tích cú pháp chuỗi `declaration`:
     - Nếu chuỗi bắt đầu bằng `"Integer"` hoặc `"Character"` và theo sau là dấu ngoặc vuông (ví dụ: `"Integer x[5]"`), khởi tạo đối tượng `NewDeclaration` với kiểu `ArrayType`.
     - Nếu chuỗi bắt đầu bằng `"Call"`, khởi tạo đối tượng `CallDeclaration`.
     - Nếu chuỗi bắt đầu bằng `"Delete"`, khởi tạo đối tượng `DeleteDeclaration`.
   - Thêm đối tượng khai báo mới vào mảng `listDeclarations`.
   - Tăng biến `size` lên 1.
3. **Hàm string str() const**
Hàm `str()` trả về chuỗi mô tả toàn bộ các khai báo trong danh sách.
   - Tạo một chuỗi rỗng để lưu trữ kết quả.
   - Duyệt qua các đối tượng trong `listDeclarations` từ 0 đến `size - 1`.
   - Gọi phương thức `str()` của mỗi đối tượng `Declaration` và nối kết quả vào chuỗi tổng.

- Thêm dấu phân cách giữa các khai báo nếu cần.
- Trả về chuỗi tổng chứa tất cả các khai báo.

**Ví dụ về việc phân tích cú pháp các chuỗi khai báo:**

- Chuỗi `"Integer x[5];"` sẽ được chuyển thành một đối tượng `NewDeclaration` với kiểu dữ liệu `ArrayType(5, IntegerType)`.
- Chuỗi `"Integer x[5, 6];"` sẽ được chuyển thành một đối tượng `NewDeclaration` với kiểu dữ liệu `ArrayType(5, ArrayType(6, IntegerType))`.
- Chuỗi `"Call x;"` sẽ được chuyển thành một đối tượng `CallDeclaration`.
- Chuỗi `"Delete x;"` sẽ được chuyển thành một đối tượng `DeleteDeclaration`.

# 3 Xây dựng cấu trúc dữ liệu cho bộ nhớ heap in mermory (đối tượng Heap)

1. **Constructor Heap()**
   Hàm khởi tạo dùng để khởi tạo một đối tượng `Heap`, bắt đầu với toàn bộ bộ nhớ trống.
   - Khởi tạo biến `free_memory` trỏ tới một đối tượng `Node`, đại diện cho một khối bộ nhớ trống có địa chỉ bắt đầu là 0 và kích thước là `MAX_SIZE`.
2. **Hàm int allocation(int size)**
   Hàm này cấp phát vùng nhớ có kích thước `size` từ bộ nhớ trống.
   - Duyệt qua danh sách `free_memory` để tìm một khối bộ nhớ trống có kích thước đủ lớn đầu tiên tính từ địa chỉ 0.
   - Nếu tìm thấy:
     - Nếu khối trống có kích thước đúng bằng `size`, xóa khối đó khỏi danh sách.
     - Nếu khối trống lớn hơn, cập nhật kích thước và địa chỉ bắt đầu của khối đó.
   - Trả về địa chỉ đầu của khối bộ nhớ được cấp phát.
   - Nếu không tìm thấy khối nào phù hợp, nén ra lỗi `throw std::runtime_error(No matching memory block found.)`

   Ví dụ

   ```
   [0;1000] // free_memory
   Integer var1; // allocation 4 byte
   Result: [4:1000]

   [0;40] -> [100:1000] // free_memory
   Integer[10] var2; // allocation 40 byte
   Result: [100:1000]
   ```
3. **Hàm void deallocation(int address, int size)**
   Hàm này giải phóng vùng nhớ tại địa chỉ `address` với kích thước `size`, trả lại khối đó vào danh sách bộ nhớ trống.
   - Tạo một đối tượng `Node` mới với địa chỉ `address` và kích thước `size`, và chèn nó vào danh sách `free_memory`.
   - Sau khi chèn, hợp nhất các khối liền kề với khối vừa được giải phóng (nếu có) để tránh phân mảnh.

   Ví dụ

   ```
   [4:1000] // free_memory
   Delete var1; // allocation 4 byte
   Result: [0:1000]

   [0;40] -> [100:1000] // free_memory
   Delete var2; // allocation 40 byte
   Result: [0:1000]
   ```
4. **Destructor ~Heap()**
   Hàm hủy giải phóng toàn bộ bộ nhớ đã được cấp phát cho các đối tượng `Node` trong danh sách `free_memory`.

- Duyệt qua toàn bộ danh sách `free_memory`, giải phóng từng đối tượng `Node`.

# 4 Quản lí memory (đối tượng MemoryManager)

1. **Hàm khởi tạo** `MemoryManager(string fileName)`:
   - **Mô tả:** Hàm khởi tạo sẽ nhận vào tên tệp `fileName` và sử dụng nó để khởi tạo danh sách các khai báo `listDelarations`. Cấu trúc này sẽ giúp quản lý các khai báo được đọc từ file hoặc tạo từ chương trình.
   - **Hành động cụ thể:**
     - Khởi tạo đối tượng `ListDelarations` cho danh sách các khai báo.
     - File có thể chứa các khai báo kiểu như: `"Integer x[5];"`, `"Call x;"`, `"Delete x;"`...
     - Chuyển đổi thông tin từ file thành các đối tượng `NewDeclaration`, `CallDeclaration`, và `DeleteDeclaration`.
   - **Cấu trúc của file** hàng đầu là số lượng các khai báo ở sau

     Ví dụ file

     ```
     3
     Integer var1;
     Call var1;
     Delete var1;
     ```

2. **Hàm** `run()`:
   - **Mô tả:** Hàm này sẽ tính toán kích thước tối đa có thể cấp phát trong khoảng từ 0 đến một giá trị `size`. Dựa trên việc cấp phát và giải phóng bộ nhớ trong `Heap`, hàm này sẽ tính toán và lưu trữ bộ nhớ lớn nhất có thể sử dụng được.
   - **Hành động cụ thể:**
     - Sử dụng đối tượng `Heap` để quản lý việc cấp phát và giải phóng bộ nhớ.
     - Lần lượt duyệt qua các khai báo trong `listDelarations`, thực hiện việc cấp phát hoặc giải phóng bộ nhớ dựa trên loại khai báo.
     - Theo dõi và ghi nhận kích thước lớn nhất đã được cấp phát trong quá trình.
3. **Hàm hủy** `MemoryManager()`:
   - **Mô tả:** Giải phóng các tài nguyên đã sử dụng bởi `MemoryManager`, bao gồm việc giải phóng đối tượng `listDelarations`.
4. **Hàm** `toString_ListDeclarations()`:
   - **Mô tả:** Trả về chuỗi mô tả toàn bộ danh sách các khai báo trong `listDelarations`. Hữu ích cho việc kiểm tra và ghi log.

# DSA1 - Layout Memory

Võ Tiến

08.09.2024

### Author Information


- Author: Võ Tiến
- Facebook Profile: https://www.facebook.com/Shiba.Vo.Tien
- Facebook Group: https://www.facebook.com/groups/khmt.ktmt.cse.bku


### Notification

- Google Drive: https://drive.google.com/drive/folders/1-cd7V9Ak-d9QWUcyd0JZ_q811TXQjd9Q
- Discord:
  https://discordapp.com/channels/1277639705189683292/1282337962482466931
  https://discord.com/channels/1277639705189683292/1282337921441206303

### Implementation

- heap.cpp
- declaration.cpp
- memory.cpp

### Create Test Cases

- unit_test.cpp
- random_test.cpp

## Build Code

```
g++ -o main main.cpp memory_layout/memory_layout.cpp
                memory_layout/text/declaration.cpp unit_test/unit_test.cpp
                random_test/random_test.cpp memory_layout/heap/heap.cpp
```

## Run Code

### Terminal Unit Test

```
./main test_unit
./main test_unit all
./main test_unit nameFunctionUnitTest
```

### Terminal Auto Test

```
./main test_random number_1 number_2
./main test_random number
```