- Bộ nhớ trong chương trình C ++ được chia thành hai phần:
  - **stack**: Tất cả các biến được khai báo bên trong hàm sẽ chiếm bộ nhớ từ "**stack**".
  - **heap**: Được sử dụng để phân bổ bộ nhớ động khi chương trình chạy.
- Bạn có thể phân bổ bộ nhớ trong thời gian chạy trong **heap** cho biến của một loại đã cho bằng cách sử dụng toán tử đặc biệt trong C++, trả về địa chỉ của không gian được phân bổ. Toán tử này được gọi là "**new**".
  - Sử dụng toán tử "**delete**", để giải phóng bộ nhớ đã được cấp phát trước đó bởi toán tử "**new**".
- Có cú pháp chung sau để sử dụng toán tử "**new**" để phân bổ bộ nhớ động cho bất kỳ loại dữ liệu nào.

```
new data-type;
```

- Ở đây, kiểu dữ liệu có thể là bất kỳ kiểu dữ liệu "**built-in data type**" nào bao gồm mảng hoặc bất kỳ kiểu dữ liệu nào do người dùng định nghĩa bao gồm **class** hoặc **structure**.
- Toán tử "**new**" được sử dụng để phân bổ bộ nhớ trong thời gian chạy (**runtime**). Bộ nhớ được phân bổ theo byte.
- Chúng ta có thể khởi tạo một biến trong khi phân bổ động theo hai cách sau:

```
int *ptr = new int (4);
```

```
int *ptr = new int {4};
```

```
#include <iostream>

int main()
{
        int *ptr = new int;
        *ptr = 4;
        std::cout << *ptr << std::endl;
        return 0;
}
```

- Ví dụ, chúng ta có thể định nghĩa một con trỏ kiểu **double** và sau đó yêu cầu bộ nhớ được cấp phát tại thời điểm thực hiện. Chúng ta có thể làm điều này bằng cách sử dụng toán tử "**new**" với các câu lệnh sau:

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

- Bộ nhớ có thể **không được phân bổ thành công**, nếu "**free store**" đã được sử dụng hết. Vì vậy, nên kiểm tra xem toán tử "**new**" có trả về con trỏ **NULL** hay không và thực hiện hành động thích hợp như dưới đây:

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

- Hàm **malloc**() từ C, vẫn tồn tại trong C++, nhưng **nên tránh sử dụng hàm malloc**() in c++.
- Ưu điểm chính của **new** so với **malloc()** là new không chỉ phân bổ bộ nhớ, nó còn tạo dựng các đối tượng là mục đích chính của C++.
- Tại bất kỳ thời điểm nào, khi bạn cảm thấy một biến đã được phân bổ động không còn cần thiết nữa => chúng ta có thể giải phóng bộ nhớ mà nó chiếm trong (**free store**) với toán tử "**delete**" như sau:

```
delete pvalue;        // Release memory pointed to by pvalue
```

- Ví dụ sau đây để cho thấy cách thức hoạt động của "**new**" và "**delete**".

```
#include <iostream>
using namespace std;

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

malloc() vs new

Following are the differences between malloc() and operator new.:

1. **Calling Constructors:** new calls constructors, while malloc() does not. In fact primitive data types (char, int, float.. etc) can also be initialized with new. For example, below program prints 10.

```
#include<iostream>
using namespace std;
int main()
{
    // Initialization with new()
    int *n = new int(10);
    cout << *n;
    getchar();
    return 0;
}
```

Output:

```
10
```

2. **operator vs function:** new is an operator, while malloc() is a function.
3. **return type:** new returns exact data type, while malloc() returns void *.
4. **Failure Condition:** On failure, malloc() returns NULL where as new throws bad_alloc exception.
5. **Memory:** In case of new, memory is allocated from free store where as in malloc() memory allocation is done from heap.
6. **Overriding:** We are allowed to override new operator where as we can not override the malloc() function legally.
7. **Size:** Required size of memory is calculated by compiler for new, where as we have to manually calculate size for malloc().
8. **Buffer Size:** malloc() allows to change the size of buffer using realloc() while new doesn't

NEW	MALLOC
calls constructor	doesnot calls constructors
It is an operator	It is a function
Returns exact data type	Returns void *
on failure, Throws	On failure, returns NULL
Memory allocated from free store	Memory allocated from heap
can be overridden	cannot be overridden
size is calculated by compiler	size is calculated manually

new vs malloc()

1) `new` is an **operator**, while `malloc()` is a **function**.

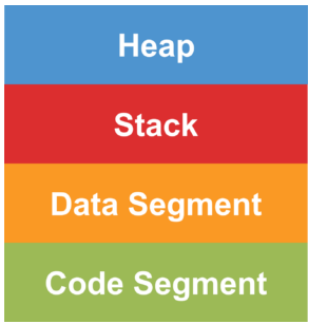
2) `new` calls **constructors**, while `malloc()` does not.

3) `new` returns **exact data type**, while `malloc()` returns **void** *.

4) `new` never returns a **NULL** (will throw on failure) while `malloc()` returns NULL

5) Reallocation of memory not handled by `new` while `malloc()` can

- Không sử dụng **malloc** trong C ++ mà không có lý do thực sự chính đáng để làm như vậy. **malloc** có một số thiếu sót khi được sử dụng với C ++, **New** được xác định để khắc phục vấn đề đó. **malloc** không phải là loại an toàn theo bất kỳ cách nào có ý nghĩa.
- **malloc** trả về **NULL** nếu phân bổ không thành công, **new** sẽ ném **bad_alloc** *(hành vi sau này khi sử dụng con trỏ NULL là không xác định).*
- Về cơ bản, **malloc** là một tính năng C và **new** là một tính năng C ++. Kết quả là **malloc** không hoạt động tốt với các constructors, nó chỉ xem xét việc phân bổ một đoạn **byte**.
- **new**/**delete** gọi hàm **constructor**/**destructor**;
- **new** là loại an toàn kiểu (type safe), **malloc** thì không;
- Luôn sử dụng **new** trong C++. Nếu bạn cần một khối bộ nhớ chưa định kiểu, bạn có thể sử dụng trực tiếp toán tử **new**.



- **Stack** được sử dụng để cấp phát bộ nhớ tĩnh *(static memory allocation)* và **Heap** cho cấp phát bộ nhớ động *(dynamic memory allocation)*, cả hai đều được lưu trữ trong RAM của máy tính.
- Các biến được phân bổ trên ngăn xếp **Stack** trong khi cấp phát bộ nhớ tĩnh được lưu trữ trực tiếp vào bộ nhớ và truy cập vào bộ nhớ này rất nhanh, phân bổ của nó được xử lý khi chương trình được biên dịch.
- **Stack** luôn được lưu trữ theo thứ tự **LIFO** *(last in first out)*.
- Các biến được phân bổ trên **Heap** có bộ nhớ được cấp phát ở thời điểm "**run time**".

- Truy cập bộ nhớ **heap** chậm hơn so với **stack**, nhưng kích thước của **heap** chỉ bị giới hạn bởi kích thước của **Ram**.
- Phần tử trong **heap** không có sự phụ thuộc lẫn nhau và luôn có thể được truy cập ngẫu nhiên bất cứ lúc nào. Có thể phân bổ một khối bất cứ lúc nào và giải phóng nó cũng vậy.
- Điều này sẽ khó khăn hơn cho việc theo dõi phần nào của **heap** được phân bổ hoặc giải quyết tại bất kỳ thời điểm nào so với **stack**.

1. Allocating memory?

- Có hai cách mà bộ nhớ được phân bổ để lưu trữ dữ liệu:
  - **Phân bổ thời gian biên dịch**
    - Bộ nhớ cho các biến được đặt tên được phân bổ bởi trình biên dịch.
    - Kích thước chính xác và loại lưu trữ phải được biết tại thời điểm biên dịch.
    - Đối với khai báo mảng tiêu chuẩn, đây là lý do tại sao kích thước phải không đổi
  - **Phân bổ bộ nhớ động**
    - Bộ nhớ được phân bổ "nhanh chóng" trong thời gian run time.
    - Không gian được phân bổ động thường được đặt trong một "**segment**" chương trình được gọi là **heap** hoặc **free store**.
    - Số lượng chính xác hoặc số lượng các phần tử không được trình biên dịch biết trước.
    - Để phân bổ bộ nhớ động, thì con trỏ đóng vai trò rất quan trọng.

2. Dynamic Memory Allocation

- Tự động phân bổ không gian lưu trữ trong khi chương trình đang chạy, nhưng lại không thể tạo tên biến mới.
- Vì lý do này, phân bổ động đòi hỏi hai bước:
  - Tạo không gian động (dynamic space).
  - Lưu địa chỉ của nó trong một con trỏ (để không gian có thể được tích lũy nhiều lần).
- Để tự động phân bổ bộ nhớ trong C++, sử dụng toán tử "**new**".
- Xóa phân bổ (De-allocation):
  - Thực hiện "**clean-up**" không gian đang được sử dụng cho các biến hoặc lưu trữ dữ liệu khác.
  - Các biến thời gian biên dịch được tự động xóa phân bổ dựa trên phạm vi đã biết của chúng.
  - Để xóa phân bổ bộ nhớ động sử dụng toán tử "**delete**" và có thể phân bổ lại.

3. Allocating space with new

- To allocate space dynamically, use the unary operator `new`, followed by the *type* being allocated.

```
new int;        // dynamically allocates an int
new double;     // dynamically allocates a double
```

- If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
new int[40];      // dynamically allocates an array of 40 ints
new double[size]; // dynamically allocates an array of size doubles
                  //  note that the size can be a variable
```

- These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the `new` operator returns the starting address of the allocated space, and this address can be stored in a pointer:

```
int * p;        // declare a pointer p
p = new int;    // dynamically allocate an int and load address into p

double * d;     // declare a pointer d
d = new double; // dynamically allocate a double and load address into d

// we can also do these in single line statements
int x = 40;
int * list = new int[x];
float * numbers = new float[x+10];
```

Notice that this is one more way of *initializing* a pointer to a valid target (and the most important one).

4. Accessing dynamically created space

- So once the space has been dynamically allocated, how do we use it?
- For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:

```
int * p = new int;    // dynamic integer, pointed to by p

*p = 10;              // assigns 10 to the dynamic integer
cout << *p;           // prints 10
```

- For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:

```
double * numList = new double[size];  // dynamic array

for (int i = 0; i < size; i++)
    numList[i] = 0;                   // initialize array elements to 0

numList[5] = 20;                      // bracket notation
*(numList + 7) = 15;                  // pointer-offset notation
                                      //   means same as numList[7]
```

5. Deallocation of dynamic memory

- To deallocate memory that was created with `new`, we use the unary operator `delete`. The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int;          // dynamically created int
// ...
delete ptr;                   // deletes the space that ptr points to
```

Note that the pointer `ptr` *still exists* in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10];            // point p to a brand new array
```

- To deallocate a dynamic array, use this form:

```
delete [] name_of_pointer;
```

Example:

```
int * list = new int[40];     // dynamic array

delete [] list;               // deallocates the array
list = 0;                     // reset list to null pointer
```

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

Application Example: Dynamically resizing an array

- Nếu chúng ta có một mảng và muốn làm cho nó lớn hơn *(có thể hiểu là muốn thêm các ô mảng vào nó)*.
- Không thể đơn giản là nối thêm các ô mới vào các ô cũ được.
- Hãy nhớ rằng các mảng được lưu trữ trong bộ nhớ liên tiếp và bạn không bao giờ biết liệu bộ nhớ có ngay lập tức hay không sau khi mảng đã được phân bổ cho một thứ khác.
- Vì lý do đó, quá trình này cần thêm một vài bước. Dưới đây là một ví dụ sử dụng một mảng số nguyên.

```
int * list = new int[size];
```

- Chúng ta muốn mở rộng danh sách thêm 5 số nữa.

1. Tạo một mảng hoàn toàn mới với loại thích hợp và kích thước mới *(cần một con trỏ khác cho việc này)*:

```
int * temp = new int[size + 5];
```

2. Sao chép dữ liệu từ mảng cũ vào mảng mới *(giữ chúng ở cùng vị trí, điều này thật dễ dàng với một vòng lặp for)*.

```
for (int i = 0; i < size; i++)
    temp[i] = list[i];
```

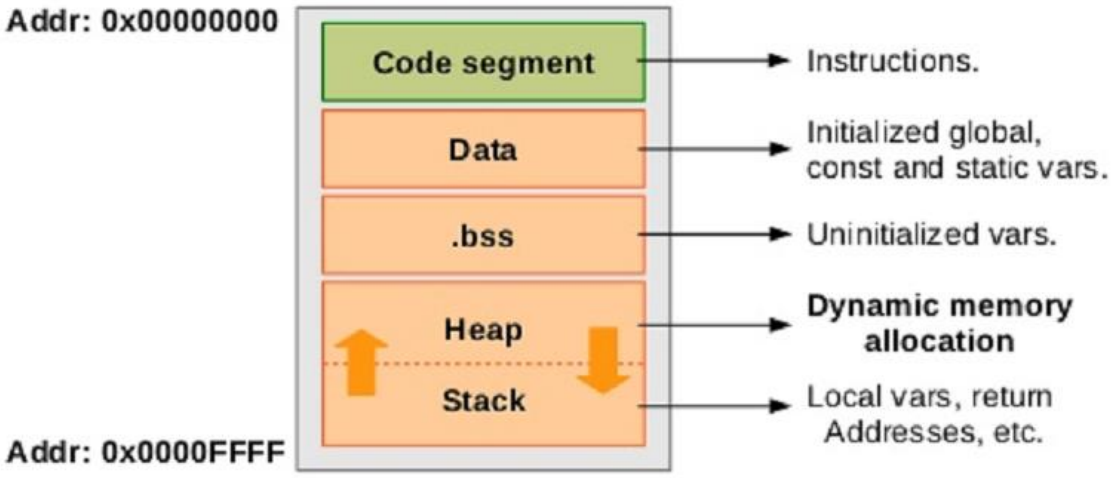
3. Xóa mảng cũ - chúng ta không cần nó nữa *(vứt rác đi thôi!)*

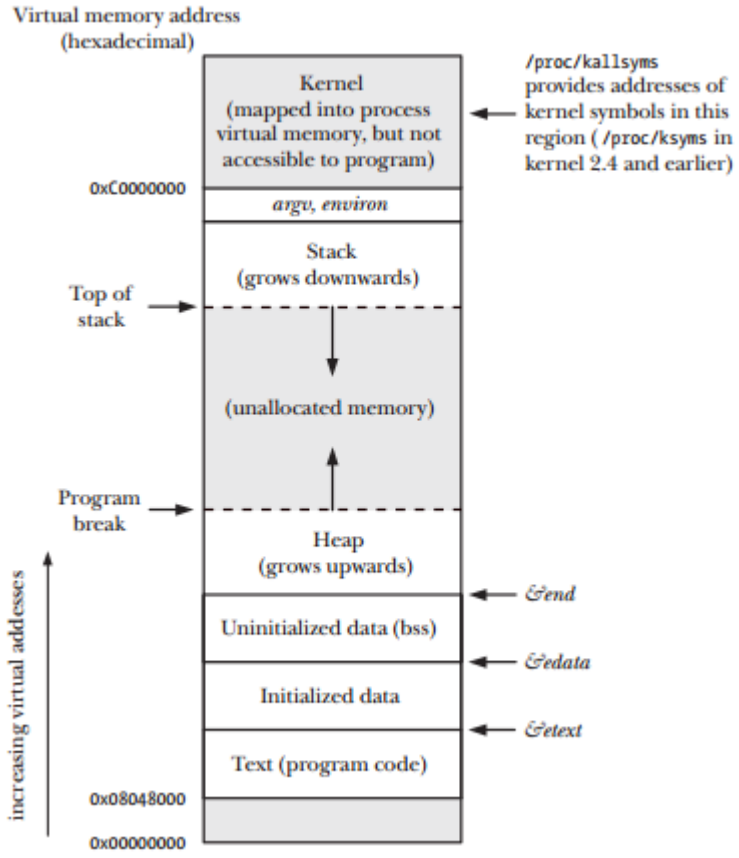
```
delete [] arr;  // this deletes the array pointed to by "arr"
```

4. Thay đổi con trỏ. Bạn vẫn muốn mảng được gọi là "**list**" *(tên gốc của nó)*, vì vậy hãy thay đổi con trỏ "**list**" với địa chỉ của "**temp**".

```
list = temp;
```

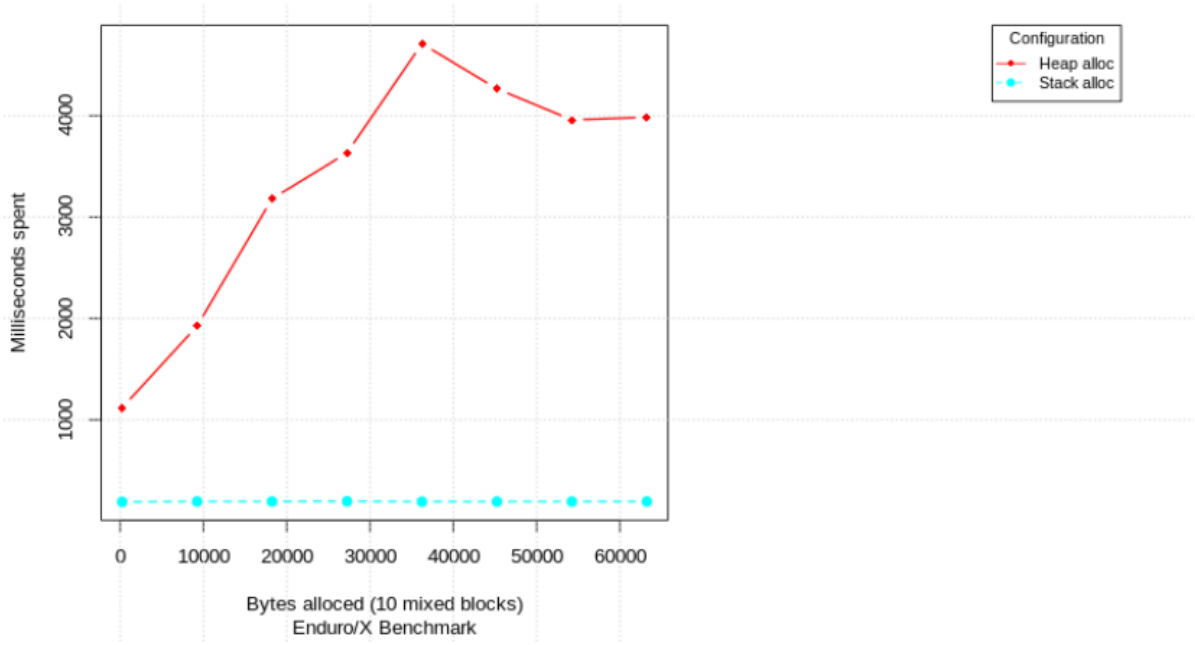
Kết Quả! Mảng list bây giờ đã lớn hơn mảng trước đó và nó có cùng dữ liệu với dữ liệu ban đầu. Bây giờ nó có đủ chỗ cho 5 mục nữa.


Addr: 0x00000000
Code segment
Data
.bss
Heap
Stack
Addr: 0x0000FFFF
Instructions.
Initialized global, const and static vars.
Uninitialized vars.
Dynamic memory allocation
Local vars, return Addresses, etc.

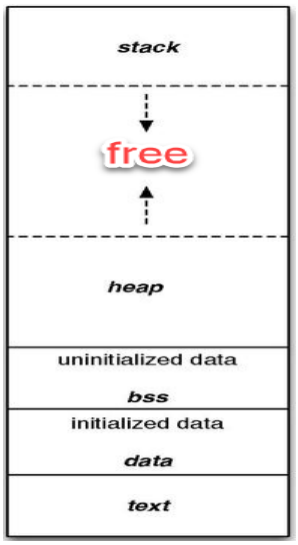

Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addresses

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than handling of stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
5. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap takes is more than a stack.

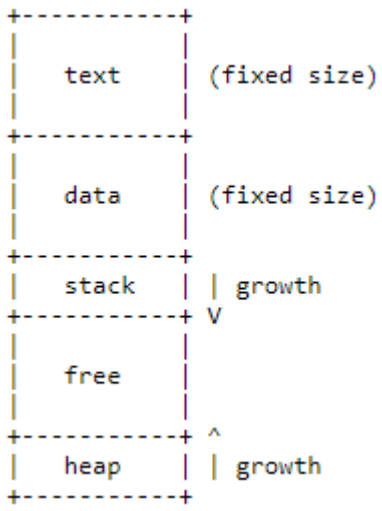


PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	Hard	Easy
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible
Data type structure	Linear	Hierarchical



- Một bộ nhớ chương trình máy tính có thể được phân loại thành hai phần:
  - **Chỉ đọc** (read-only)
  - **Đọc/ghi** (read/write)

- Sự khác biệt này phát triển từ các hệ thống ban đầu giữ chương trình chính của chúng trong bộ nhớ chỉ đọc như **Mask ROM**, **PROM** hoặc **EEPROM**.
- Đưa vào **ROM** với ý tưởng rằng một số phần của bộ nhớ của chương trình không nên được sửa đổi.
- Khi các hệ thống trở nên phức tạp hơn và các chương trình được tải từ phương tiện khác vào **RAM** thay vì thực thi từ **ROM**.



1. Text Segment?

- **Code Segment** hay còn được gọi là **Text Segment** hoặc đơn giản là văn bản.
- Là nơi một phần của tệp đối tượng hoặc phần tương ứng của không gian địa chỉ ảo của chương trình có chứa các hướng dẫn thực thi được lưu trữ và thường có kích thước chỉ đọc và cố định.

2. Data Segment?

- **Data Segment** chứa bất kỳ **biến toàn cục** hoặc **biến tĩnh** nào có giá trị được xác định trước và có thể được sửa đổi.
- Đó là bất kỳ biến nào **không được xác định** trong **hàm** hay **khối hàm** (và do đó có thể được truy cập từ bất kỳ đâu) hoặc được xác định trong hàm nhưng được định nghĩa là **static** để chúng giữ địa chỉ của chúng qua các lần xử lý tiếp theo.
- Ví dụ:

```
int val = 3;
char string[] = "Hello World";
```

- Các giá trị cho các biến này ban đầu được lưu trữ trong **bộ nhớ chỉ đọc** (thường là trong **.text**) và được sao chép vào phân đoạn **.data** trong quy trình khởi động của chương trình.

- **Lưu ý** rằng trong ví dụ trên, nếu các biến này đã được khai báo từ bên trong một hàm, chúng sẽ mặc định được lưu trữ trong khung ngăn xếp **Stack**.

3. BSS?

- **BSS Segment**, còn được gọi là dữ liệu chưa được khởi tạo, thường liền kề với phân đoạn dữ liệu **Data Segment**.
- Phân đoạn BSS chứa tất cả các **biến toàn cục** và **biến tĩnh được khởi tạo về 0** hoặc **không có khởi tạo rõ ràng** trong mã nguồn.
- Chẳng hạn, một biến được định nghĩa là **int i;** sẽ được chứa trong phân đoạn **BSS Segment**.

4. Heap?

- Vùng **heap** thường bắt đầu ở cuối các phân đoạn **.bss** và **.data** và phát triển đến các địa chỉ lớn hơn từ đó.
- Vùng **heap** được quản lý bởi **malloc**, **calloc**, **realloc** và **Free**, có thể sử dụng các lệnh gọi hệ thống **brk** và **sbrk** để điều chỉnh kích thước của nó.
- Vùng **heap** được chia sẻ bởi tất cả các luồng, thư viện dùng chung và các modules được tải động trong một quy trình xử lý.

5. Stack?

- Vùng ngăn xếp **Stack** chứa ngăn xếp chương trình (program stack), cấu trúc **LIFO**, thường nằm ở phần cao hơn của bộ nhớ.
- Một "stack pointer" theo dõi đỉnh của ngăn xếp; nó được điều chỉnh mỗi khi giá trị được "đẩy" lên ngăn xếp.
- Tập hợp các giá trị được đẩy cho một hàm được gọi là "stack frame".
- Một khung ngăn xếp bao gồm tối thiểu một địa chỉ trả lại. Biến tự động cũng được phân bổ trên ngăn xếp Stack.

```
int main()
{
        auto int a;
        int b;
        ....
        return 0;
}
```

- Tham khảo them:
  1. https://www.w3schools.in/cplusplus-tutorial/dynamic-memory-allocation/
  2. https://www.edureka.co/blog/dynamic-memory-allocation-cpp/

3. https://www.cs.fsu.edu/~myers/c++/notes/dma.html
4. https://www.design-reuse.com/articles/25090/dynamic-memory-allocation-fragmentation-c.html