Estruturas de Dados Básicas I

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 231, ramal 231, selan.santos@ufrn.br
UFRN

2023.1

Lista Encadeada – Conteúdo

1. Motivação
2. Introdução
3. Acesso Sequencial da Lista
4. Criação Manual de Lista
5. Idioma de Percorrimento
6. Inserções nas Extremidades da Lista
7. Referências

Motivação e Objetivos

- **Motivação**

Motivação e Objetivos

- **Motivação**
  - Listas implementadas com vetor apresentam baixa **eficiência temporal** quando precisam inserir no meio da lista, causada pelo deslocamento de memória.

Motivação e Objetivos

- **Motivação**
  - Listas implementadas com vetor apresentam baixa **eficiência temporal** quando precisam inserir no meio da lista, causada pelo deslocamento de memória.
  - Em termos de **uso de memória**, nem sempre a memória alocada estará sendo ocupada por elementos da lista.

Motivação e Objetivos

▷ Motivação

- ⋆ Listas implementadas com vetor apresentam baixa **eficiência temporal** quando precisam inserir no meio da lista, causada pelo deslocamento de memória.
- ⋆ Em termos de **uso de memória**, nem sempre a memória alocada estará sendo ocupada por elementos da lista.
- ⋆ **Listas encadeadas** superam estas limitações, viabilizando *inserções eficientes no meio da lista* e *alocando apenas a memória necessária* efetivamente ocupadas por elementos da lista.

Motivação e Objetivos

- **Objetivos**

Motivação e Objetivos

- **Objetivos**
  - Apresentar o conceito e implementação de listas encadeadas.

Motivação e Objetivos

- **Objetivos**
  - Apresentar o conceito e implementação de listas encadeadas.
  - Analisar o custo ou complexidade das principais operações sobre uma lista encadeada.

Motivação e Objetivos

- **Objetivos**
  - Apresentar o conceito e implementação de listas encadeadas.
  - Analisar o custo ou complexidade das principais operações sobre uma lista encadeada.
  - Apresentar algumas variações na implementação de listas encadeadas, como encadeamento duplo, nós cabeça e calda e listas circulares.

Conceitos Básico L.S.E.

- Recorde que a lista encadeada é uma estrutura de dados que pode ser usada para implementar o TAD lista.

Conceitos Básico L.S.E.

- Recorde que a lista encadeada é uma estrutura de dados que pode ser usada para implementar o TAD lista.
- Depende fortemente do uso de **ponteiros** e **alocação dinâmica** para sua implementação.

Conceitos Básico L.S.E.

- Recorde que a lista encadeada é uma estrutura de dados que pode ser usada para implementar o TAD lista.
- Depende fortemente do uso de **ponteiros** e **alocação dinâmica** para sua implementação.
- Por isso, seus algoritmos de manipulação tendem a serem mais complexos que os da lista com vetor.

Anatomia de uma L.S.E.

- São formadas por nós alocados dinamicamente.

Anatomia de uma L.S.E.

- São formadas por nós alocados dinamicamente.
- Cada nó tem um ponteiro `next` que aponta para o próximo nó.

Anatomia de uma L.S.E.

- São formadas por nós alocados dinamicamente.
- Cada nó tem um ponteiro `next` que aponta para o próximo nó.
- A frente da lista é um ponteiro para o primeiro nó da lista.

Anatomia de uma L.S.E.

- São formadas por nós alocados dinamicamente.
- Cada nó tem um ponteiro `next` que aponta para o próximo nó.
- A frente da lista é um ponteiro para o primeiro nó da lista.
- Cada nó é alocado no *heap* por meio de chamadas a `new`.

Anatomia de uma L.S.E.

- São formadas por nós alocados dinamicamente.
- Cada nó tem um ponteiro `next` que aponta para o próximo nó.
- A frente da lista é um ponteiro para o primeiro nó da lista.
- Cada nó é alocado no *heap* por meio de chamadas a `new`.
- Memória do nó persiste alocada até ser liberada via `delete`.

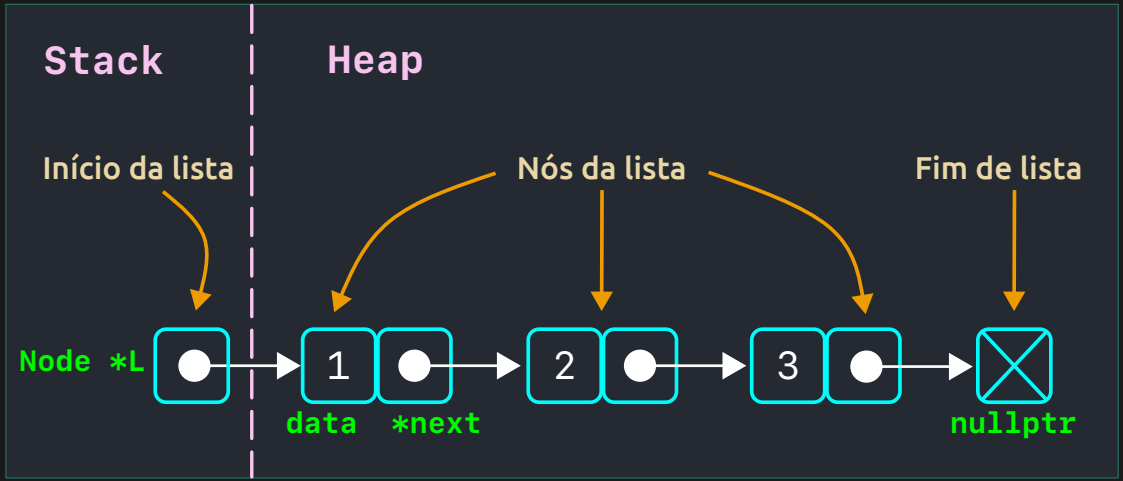
Anatomia de uma L.S.E.

- São formadas por nós alocados dinamicamente.
- Cada nó tem um ponteiro `next` que aponta para o próximo nó.
- A frente da lista é um ponteiro para o primeiro nó da lista.
- Cada nó é alocado no *heap* por meio de chamadas a `new`.
- Memória do nó persiste alocada até ser liberada via `delete`.
- Último nó da lista aponta para `nullptr`.

Representação de um Nó em C++

```
struct Node {
   int data;     // informação do nó
   Node *next;   // próximo nó.
};
```

Anatomia Visual da L.S.E.



Acesso sequencial em L.S.E.

- Para acessar cada elemento de uma lista utilizamos um ponteiro de percorrimento, que é deslocado ao longo da lista via campo `next`.

Acesso sequencial em L.S.E.

- Para acessar cada elemento de uma lista utilizamos um ponteiro de percorrimento, que é deslocado ao longo da lista via campo `next`.
- O acesso linear aos elementos é mais demorado que o acesso (constante) indexado de vetores.

Acesso sequencial em L.S.E.

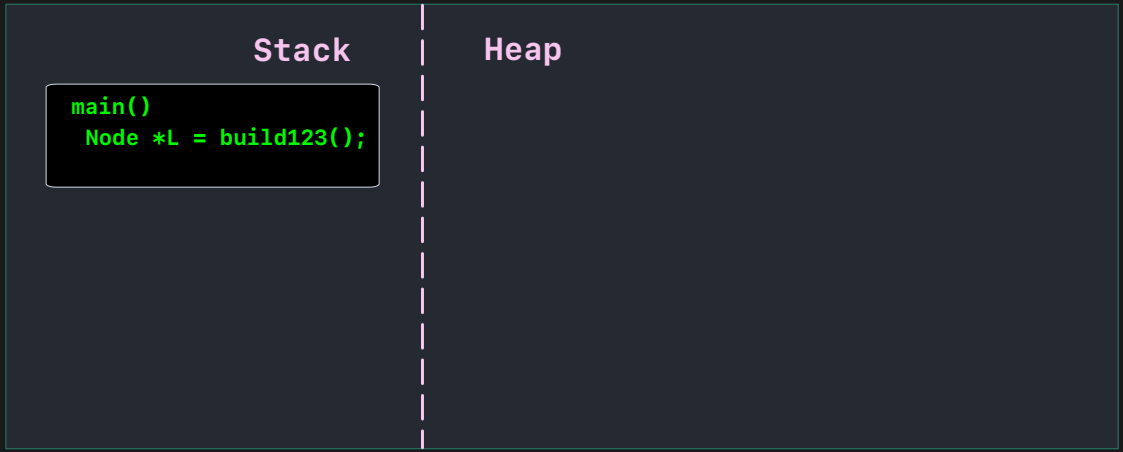
- Para acessar cada elemento de uma lista utilizamos um ponteiro de percorrimento, que é deslocado ao longo da lista via campo `next`.
- O acesso linear aos elementos é mais demorado que o acesso (constante) indexado de vetores.
- Veremos a representação da memória (*heap* vs *stack*) com uma lista que foi criada via função `build123()`.

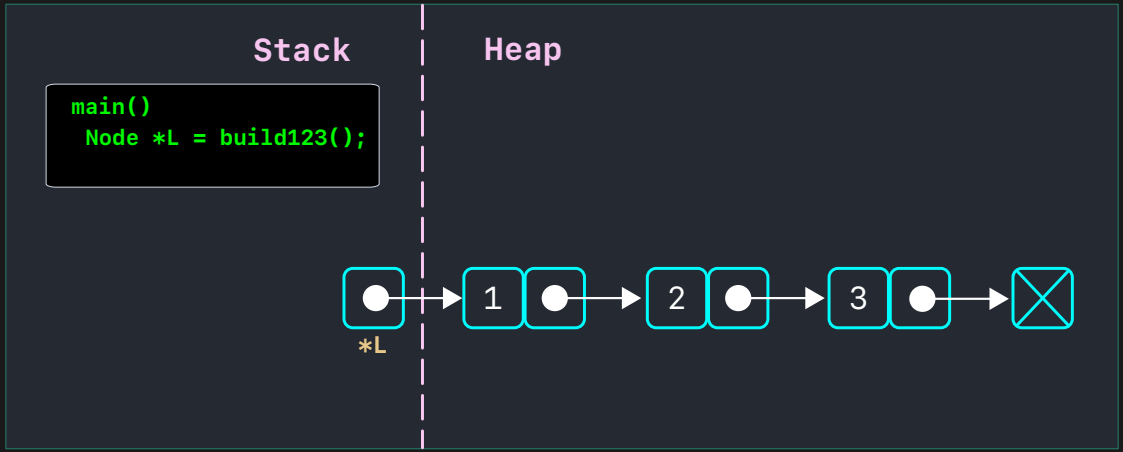
Acesso sequencial em L.S.E.

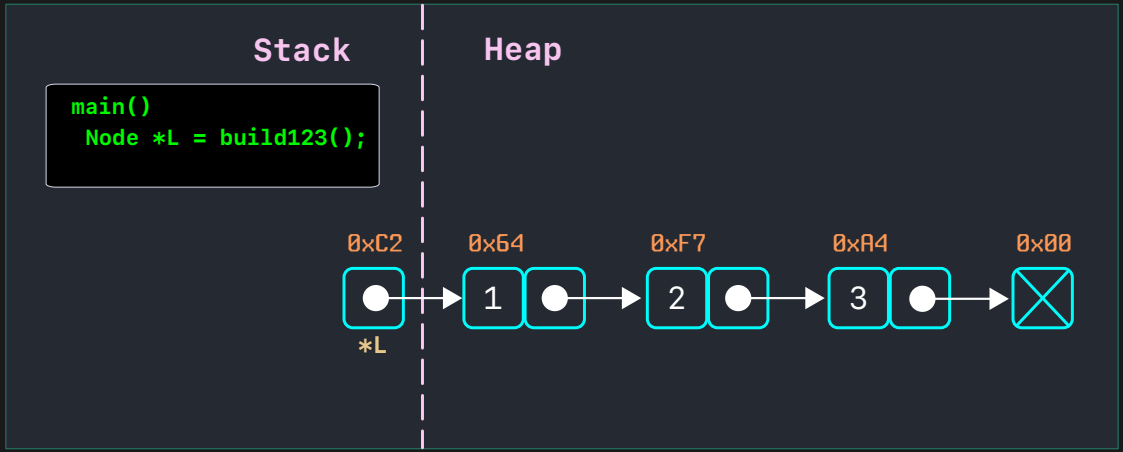
- Para acessar cada elemento de uma lista utilizamos um ponteiro de percorrimento, que é deslocado ao longo da lista via campo `next`.
- O acesso linear aos elementos é mais demorado que o acesso (constante) indexado de vetores.
- Veremos a representação da memória (*heap* vs *stack*) com uma lista que foi criada via função `build123()`.

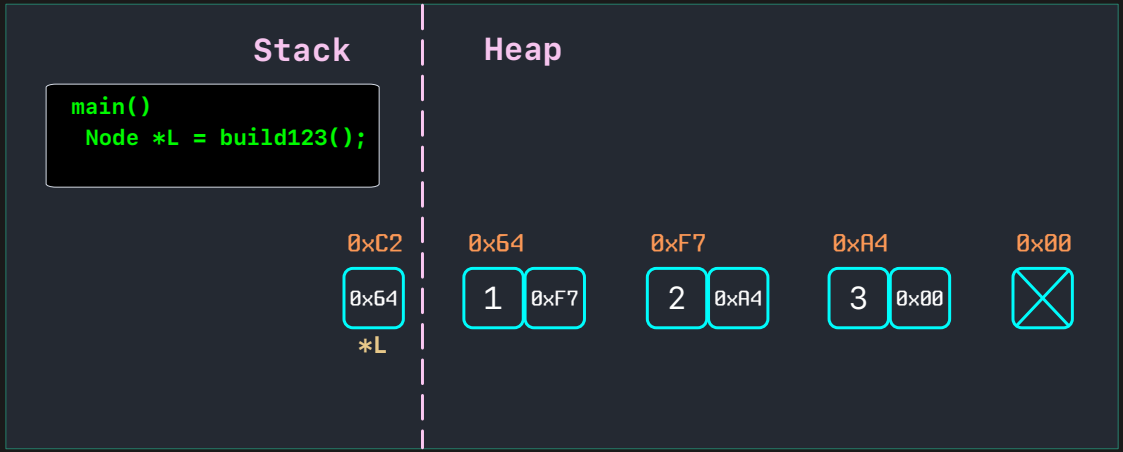
Acesso sequencial em L.S.E.

- Para acessar cada elemento de uma lista utilizamos um ponteiro de percorrimento, que é deslocado ao longo da lista via campo `next`.
- O acesso linear aos elementos é mais demorado que o acesso (constante) indexado de vetores.
- Veremos a representação da memória (*heap* vs *stack*) com uma lista que foi criada via função `build123()`. Note que `L` é a variável que armazena o começo da lista.


Stack
Heap
main()
Node *L = build123();


Stack
Heap
main()
Node *L = build123();
1
2
3
*L


Stack
Heap
main()
Node *L = build123();
0xC2
0x64
0xF7
0xA4
0x00
1
2
3
*L


Stack
Heap
main()
Node *L = build123();
0xC2
0x64
*L
0x64
1
0xF7
0xF7
2
0xA4
0xA4
3
0x00
0x00

Características de L.S.E.

A lista do exemplo tem comprimento 3. Uma lista vazia ou nula tem comprimento **zero** e é representada por um apontador para `nullptr`.

Características de L.S.E.

A lista do exemplo tem comprimento 3. Uma lista vazia ou nula tem comprimento **zero** e é representada por um apontador para `nullptr`.

Por isso, precisamos verificar se a lista é vazia antes de realizar operações sobre a lista. Caso contrário, o acesso pode gerar *segmentation fault*.

Características de L.S.E.

A lista do exemplo tem comprimento 3. Uma lista vazia ou nula tem comprimento **zero** e é representada por um apontador para `nullptr`.

Por isso, precisamos verificar se a lista é vazia antes de realizar operações sobre a lista. Caso contrário, o acesso pode gerar *segmentation fault*.

Alguns algoritmos precisam tratar o caso de lista vazia em separado, em outros isso não é necessário.

Criação de uma L.S.E.

Vamos ver agora como implementar a função `build123()`.

Criação de uma L.S.E.

Vamos ver agora como implementar a função `build123()`.

Adotaremos o paradigma imperativo para as operações em lista.

Criação de uma L.S.E.

Vamos ver agora como implementar a função `build123()`.

Adotaremos o paradigma imperativo para as operações em lista.

Portanto, precisamos passar a lista como primeiro argumento para cada função, OU retornar ponteiro para início da lista.

Stack

```
main()
  Node *L = build123();
```

Heap

Stack

```
main()
  Node *L = build123();
```

*L

Heap

Stack

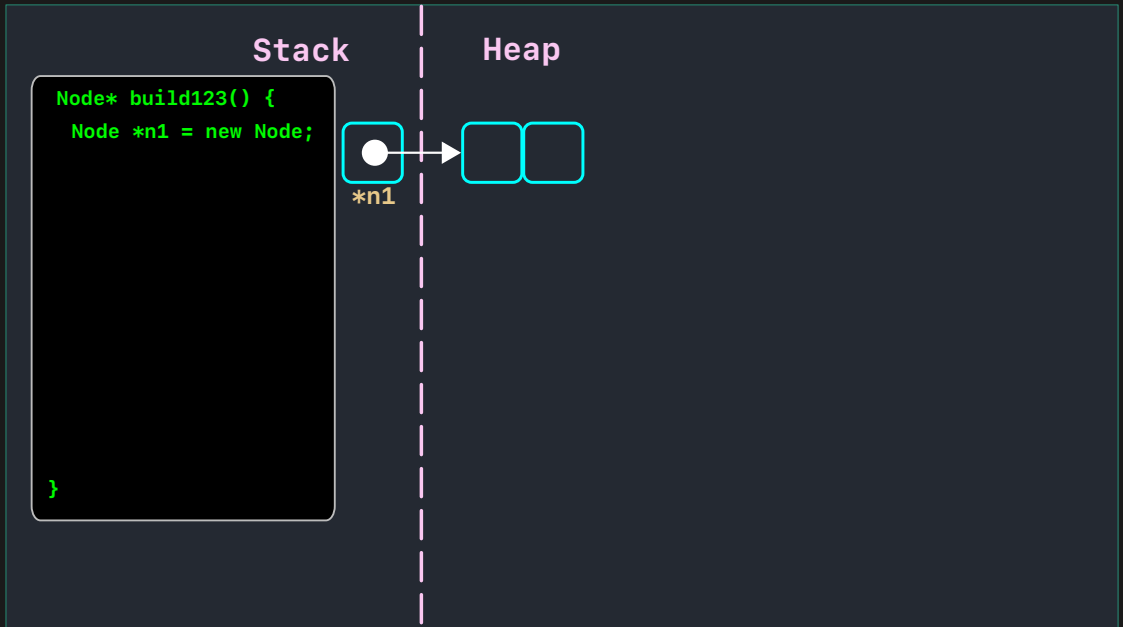
Heap

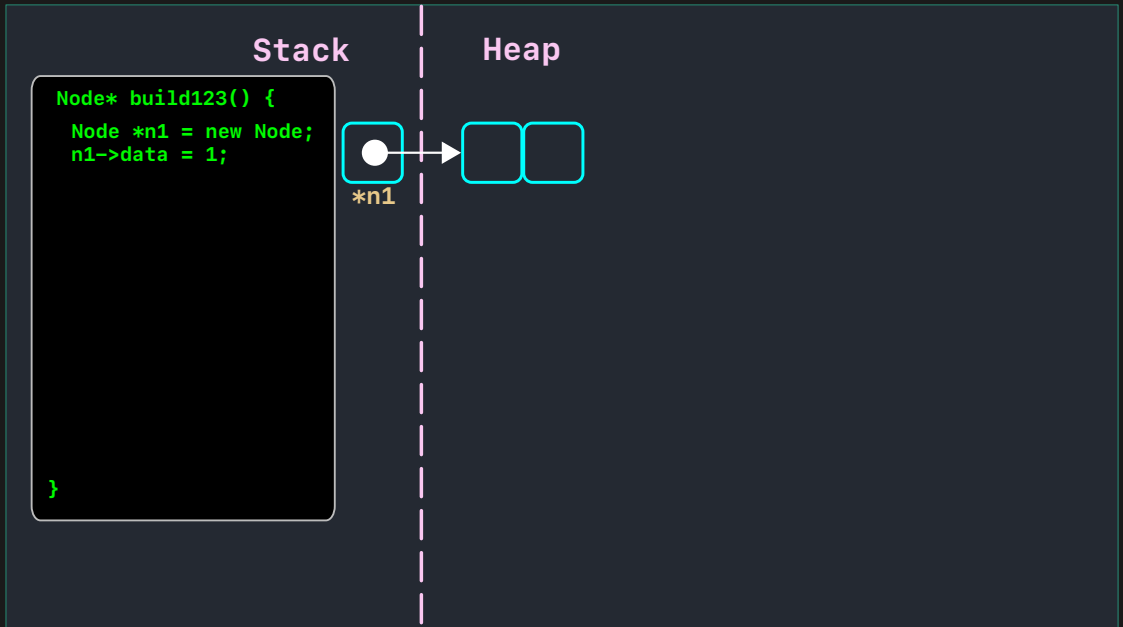
```
Node* build123() {



}
```

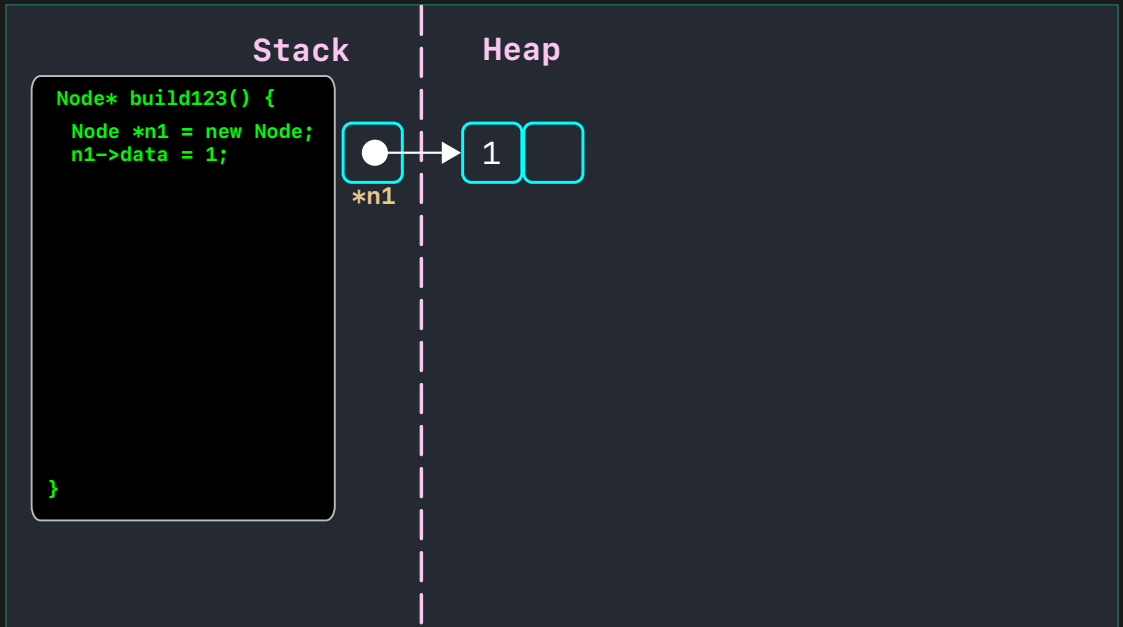
Stack

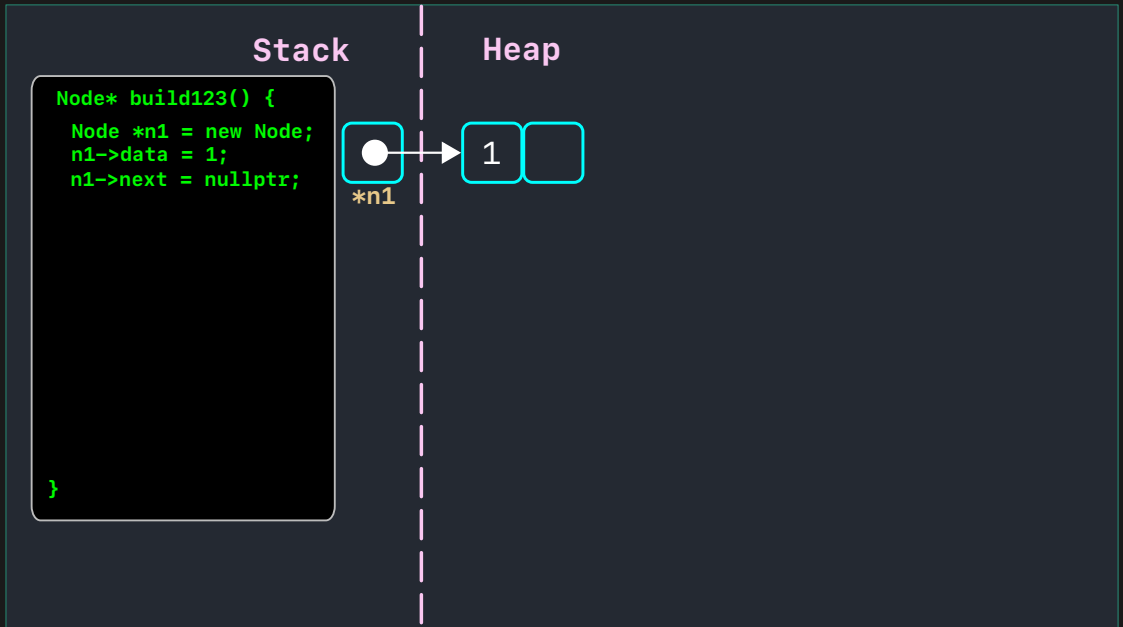
```
Node* build123() {
  Node *n1 = new Node;




}
```

Heap


Stack
Heap
Node* build123() {
Node *n1 = new Node;
}
*n1


Stack
Heap
Node* build123() {
Node *n1 = new Node;
n1->data = 1;
}
*n1


Stack
Heap
Node* build123() {
Node *n1 = new Node;
n1->data = 1;
}
1
*n1


Stack
Heap
Node* build123() {
Node *n1 = new Node;
n1->data = 1;
n1->next = nullptr;
}
1
*n1

Stack

Heap

```
Node* build123() {
  Node *n1 = new Node;
  n1->data = 1;
  n1->next = nullptr;


}
```

Stack

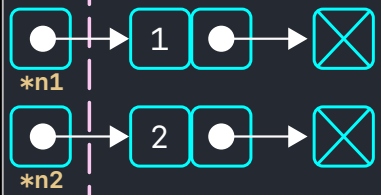
Heap

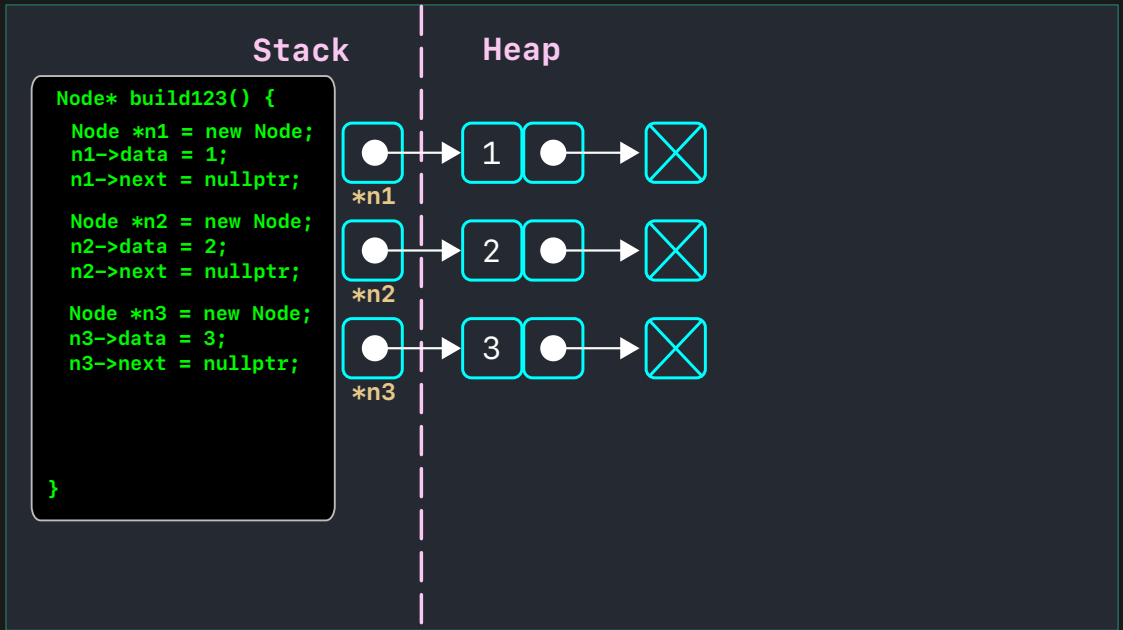
```
Node* build123() {
  Node *n1 = new Node;
  n1->data = 1;
  n1->next = nullptr;

  Node *n2 = new Node;
  n2->data = 2;
  n2->next = nullptr;




}
```

Stack

Heap

```
Node* build123() {
  Node *n1 = new Node;
  n1->data = 1;
  n1->next = nullptr;

  Node *n2 = new Node;
  n2->data = 2;
  n2->next = nullptr;




}
```

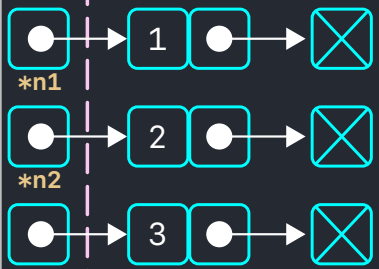



Stack
Heap
Node* build123() {
Node *n1 = new Node;
n1->data = 1;
n1->next = nullptr;
Node *n2 = new Node;
n2->data = 2;
n2->next = nullptr;
Node *n3 = new Node;
n3->data = 3;
n3->next = nullptr;
}
*n1
*n2
*n3
1
2
3

Stack

Heap

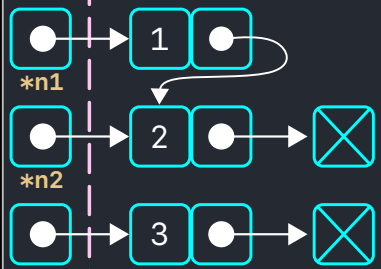
```
Node* build123() {
 Node *n1 = new Node;
 n1->data = 1;
 n1->next = nullptr;

 Node *n2 = new Node;
 n2->data = 2;
 n2->next = nullptr;

 Node *n3 = new Node;
 n3->data = 3;
 n3->next = nullptr;

 n1->next = n2;


}
```



Stack

Heap

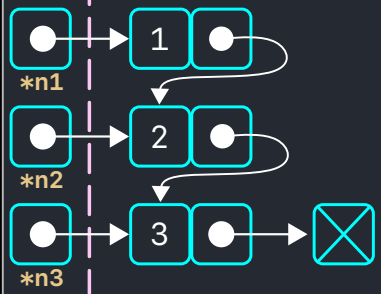
```
Node* build123() {
 Node *n1 = new Node;
 n1->data = 1;
 n1->next = nullptr;

 Node *n2 = new Node;
 n2->data = 2;
 n2->next = nullptr;

 Node *n3 = new Node;
 n3->data = 3;
 n3->next = nullptr;

 n1->next = n2;


}
```

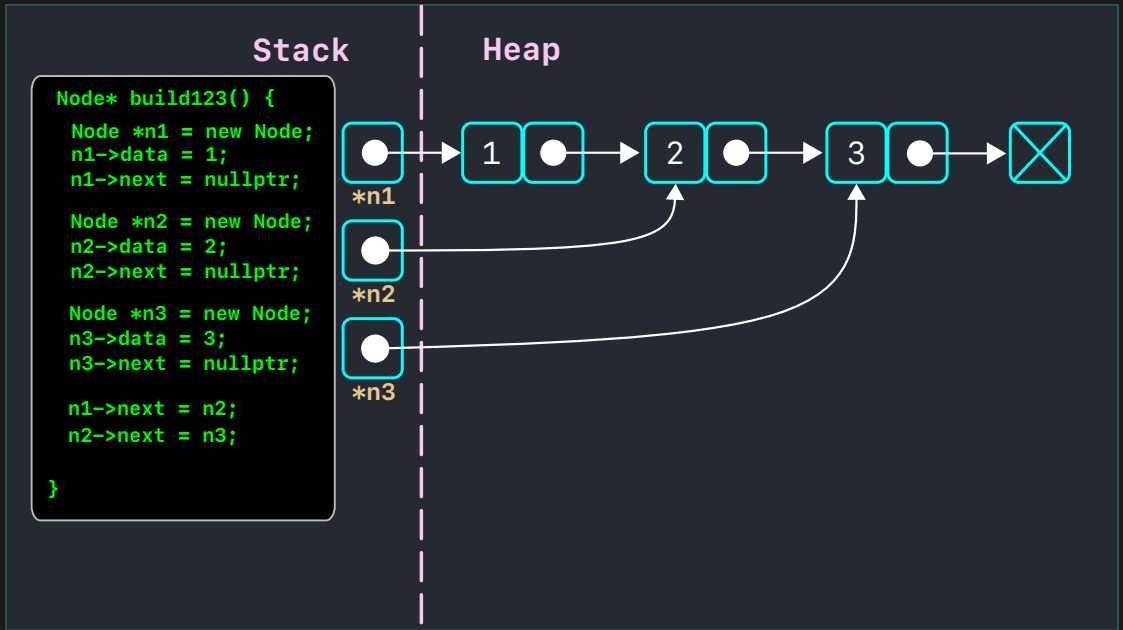


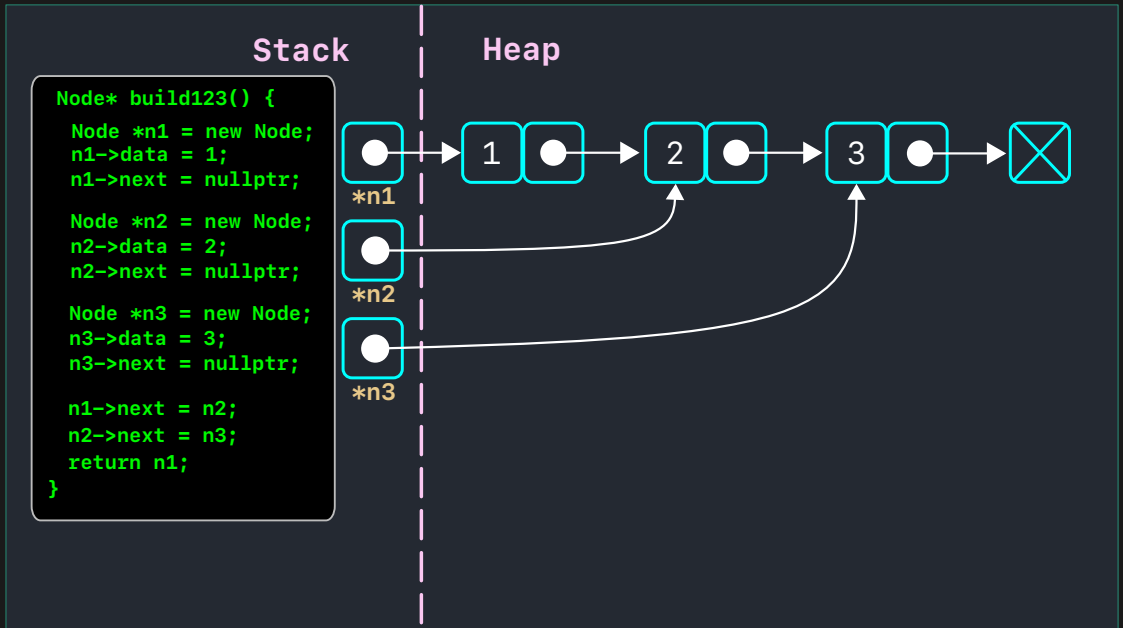
Stack

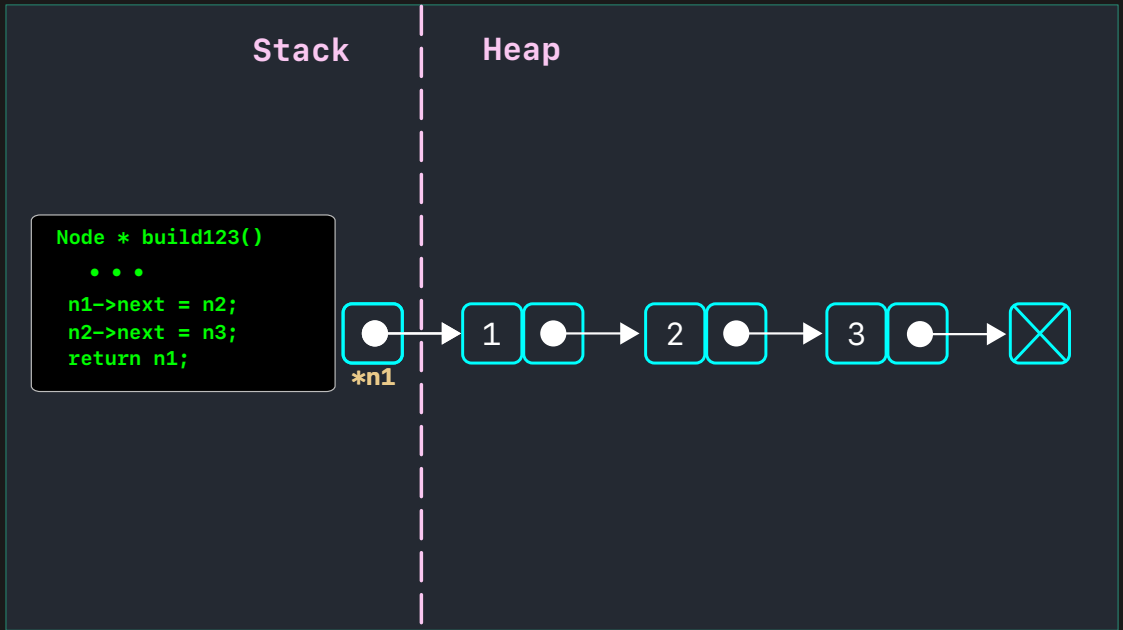
Heap

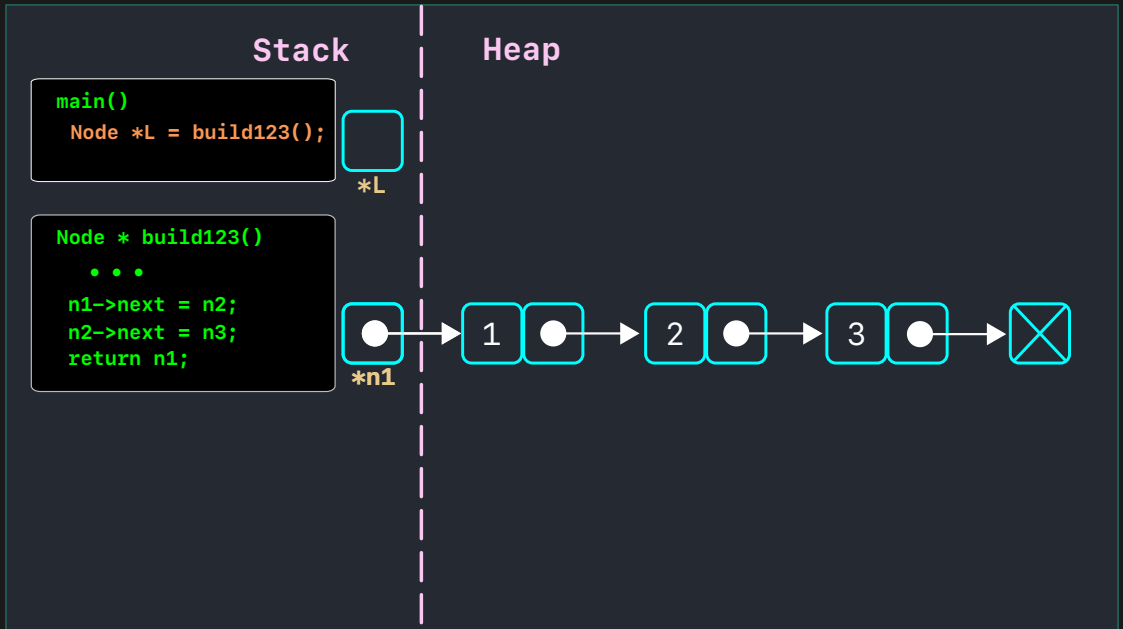
```
Node* build123() {
 Node *n1 = new Node;
 n1->data = 1;
 n1->next = nullptr;

 Node *n2 = new Node;
 n2->data = 2;
 n2->next = nullptr;

 Node *n3 = new Node;
 n3->data = 3;
 n3->next = nullptr;

 n1->next = n2;
 n2->next = n3;

}
```

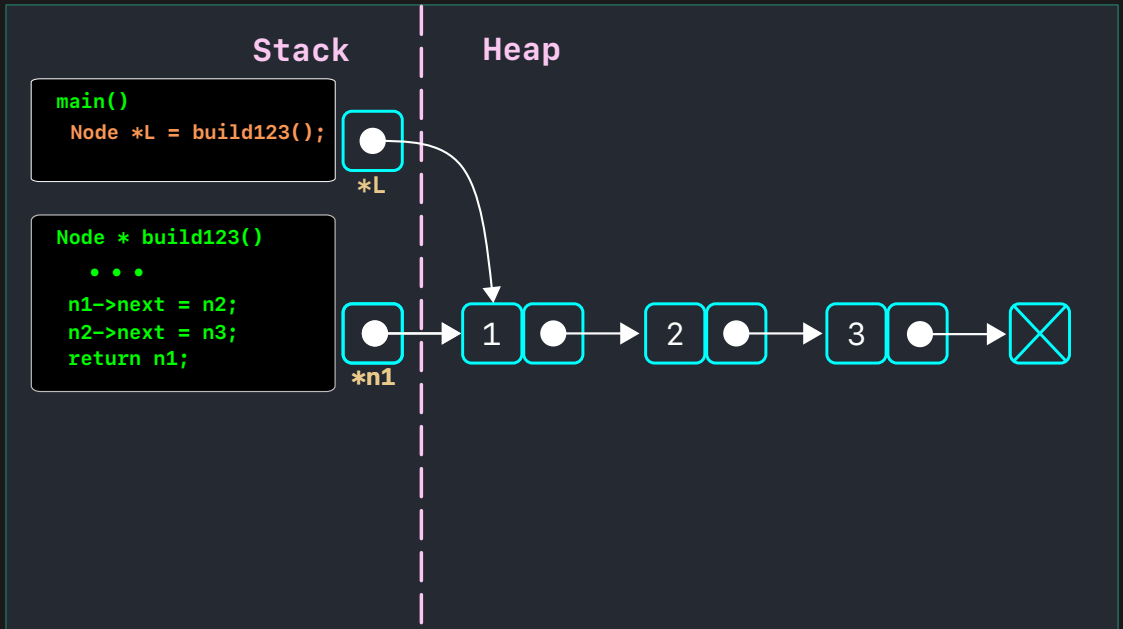


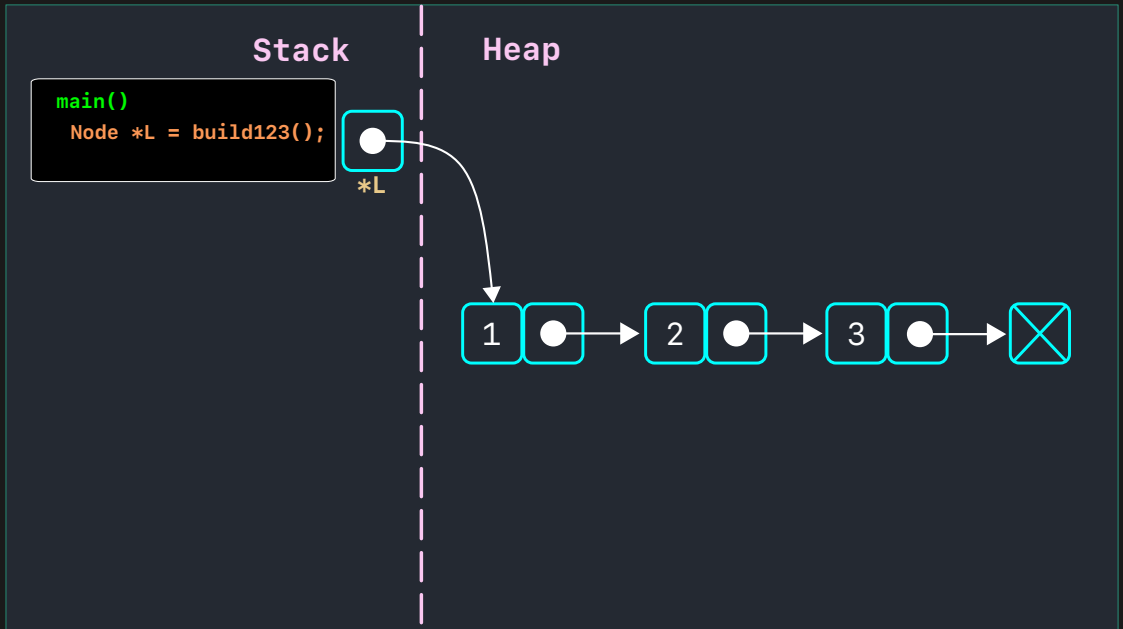

Stack
Heap
Node* build123() {
Node *n1 = new Node;
n1->data = 1;
n1->next = nullptr;
Node *n2 = new Node;
n2->data = 2;
n2->next = nullptr;
Node *n3 = new Node;
n3->data = 3;
n3->next = nullptr;
n1->next = n2;
n2->next = n3;
}
*n1
*n2
*n3
1
2
3


Stack
Heap
Node* build123() {
Node *n1 = new Node;
n1->data = 1;
n1->next = nullptr;
Node *n2 = new Node;
n2->data = 2;
n2->next = nullptr;
Node *n3 = new Node;
n3->data = 3;
n3->next = nullptr;
n1->next = n2;
n2->next = n3;
return n1;
}
*n1
*n2
*n3
1
2
3


Stack
Heap
Node * build123()
• • •
n1->next = n2;
n2->next = n3;
return n1;
*n1
1
2
3


Stack
Heap
main()
Node *L = build123();
*L
Node * build123()
• • •
n1->next = n2;
n2->next = n3;
return n1;
*n1
1
2
3


Stack
Heap
main()
Node *L = build123();
*L
Node * build123()
• • •
n1->next = n2;
n2->next = n3;
return n1;
*n1
1
2
3


Stack
Heap
main()
Node *L = build123();
*L
1
2
3

Percorrendo uma L.S.E.

A função `length()` recebe uma lista já pronta como argumento e retorna seu comprimento.

Percorrendo uma L.S.E.

A função `length()` recebe uma lista já pronta como argumento e retorna seu comprimento.

`length()` demonstra o idioma de programação para **percorrimento** da lista.

Stack

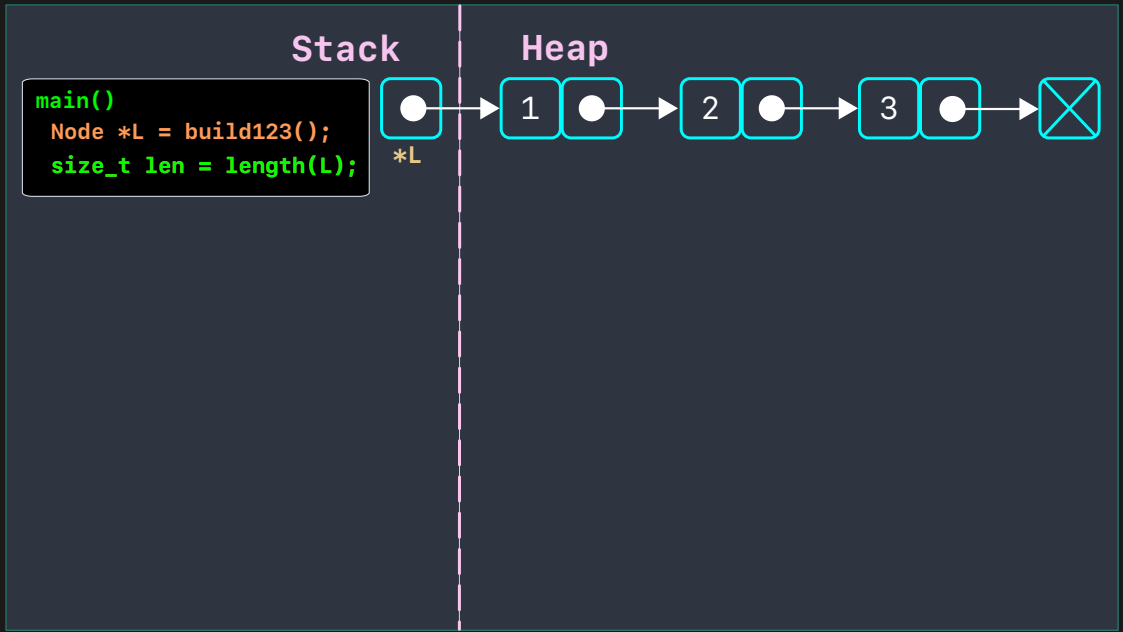
Heap

```
main()
 Node *L = build123();
 size_t len = length(L);
```

Stack

Heap

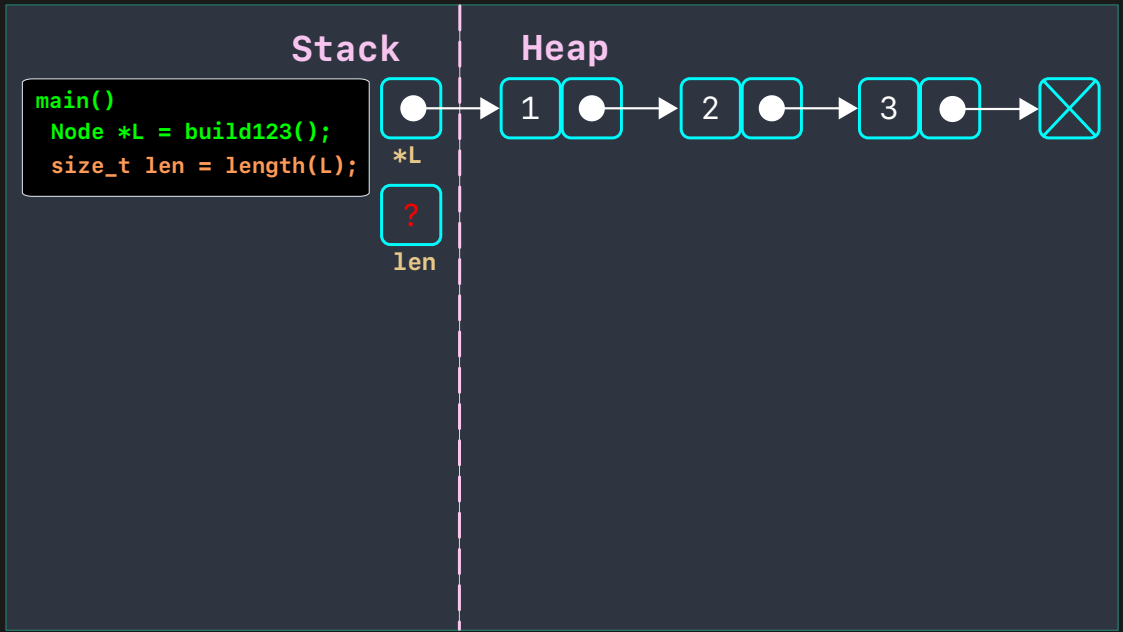
```
main()
  Node *L = build123();
  size_t len = length(L);
```


Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3

Stack

Heap

```
main()
 Node *L = build123();
 size_t len = length(L);
```

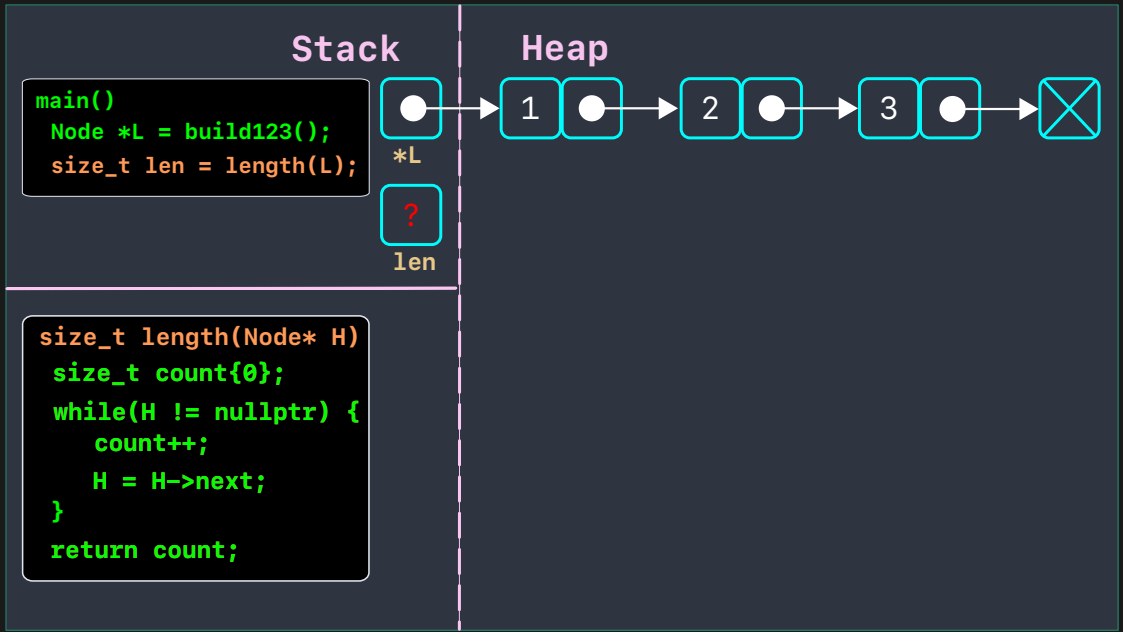

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
?
len
1
2
3

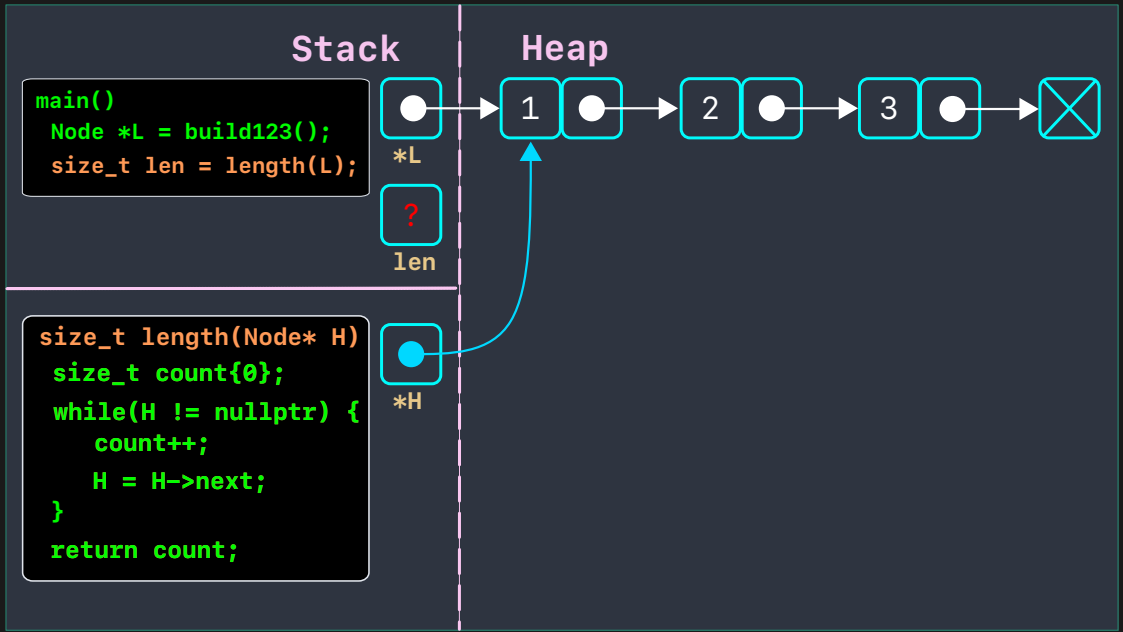
Stack

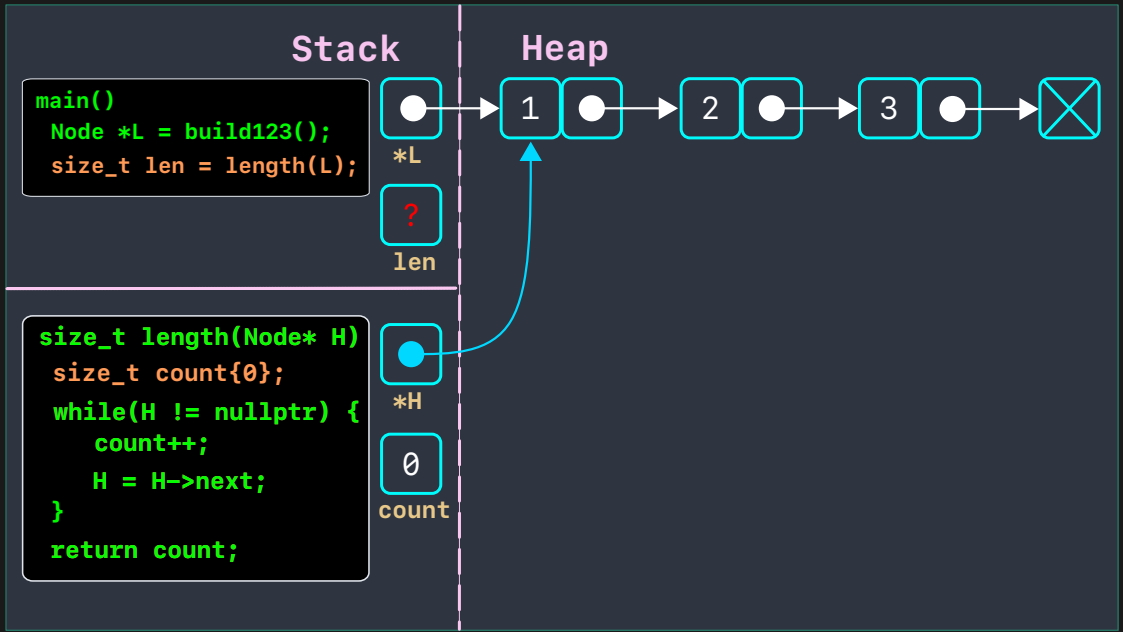
Heap

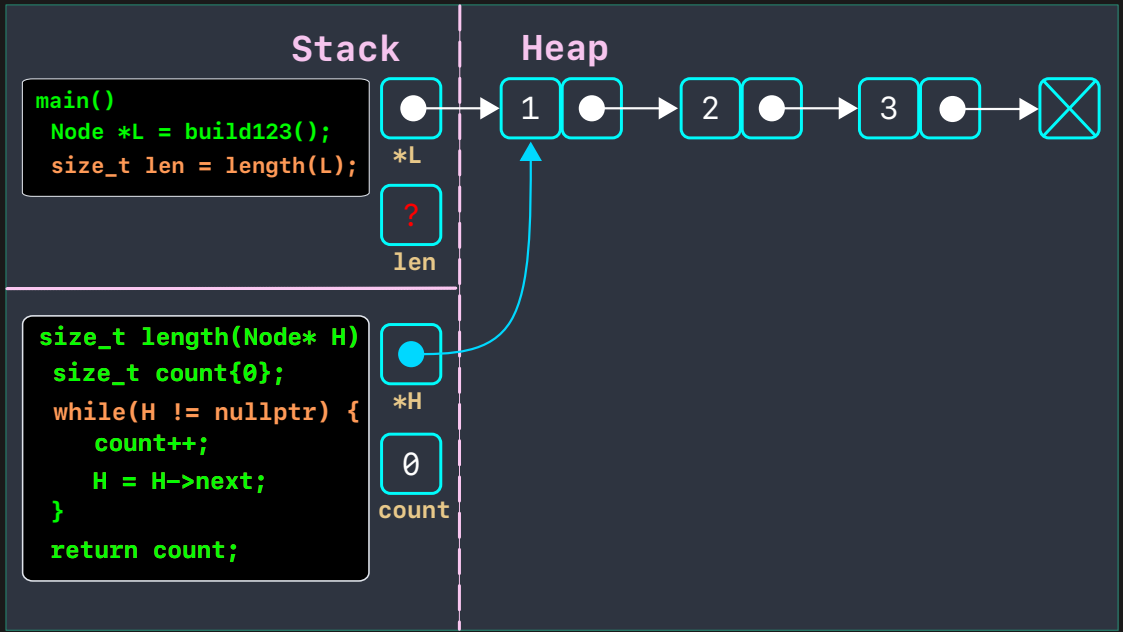
```
main()
 Node *L = build123();
 size_t len = length(L);
```

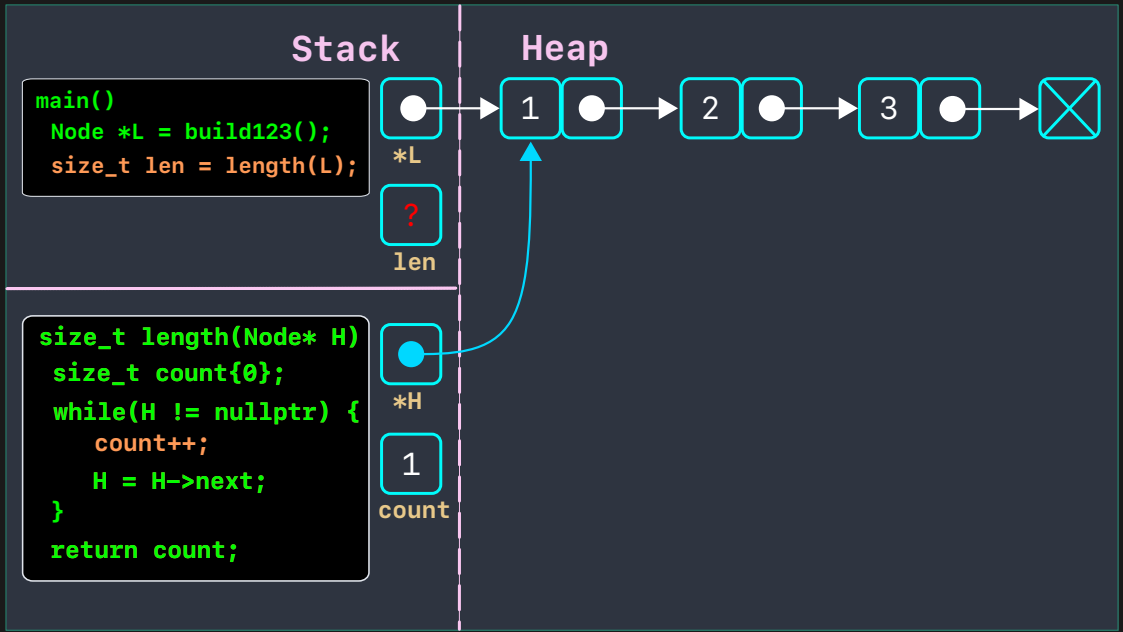

```
size_t length(Node* H)
 size_t count{0};
 while(H != nullptr) {
    count++;
    H = H->next;
 }
 return count;
```

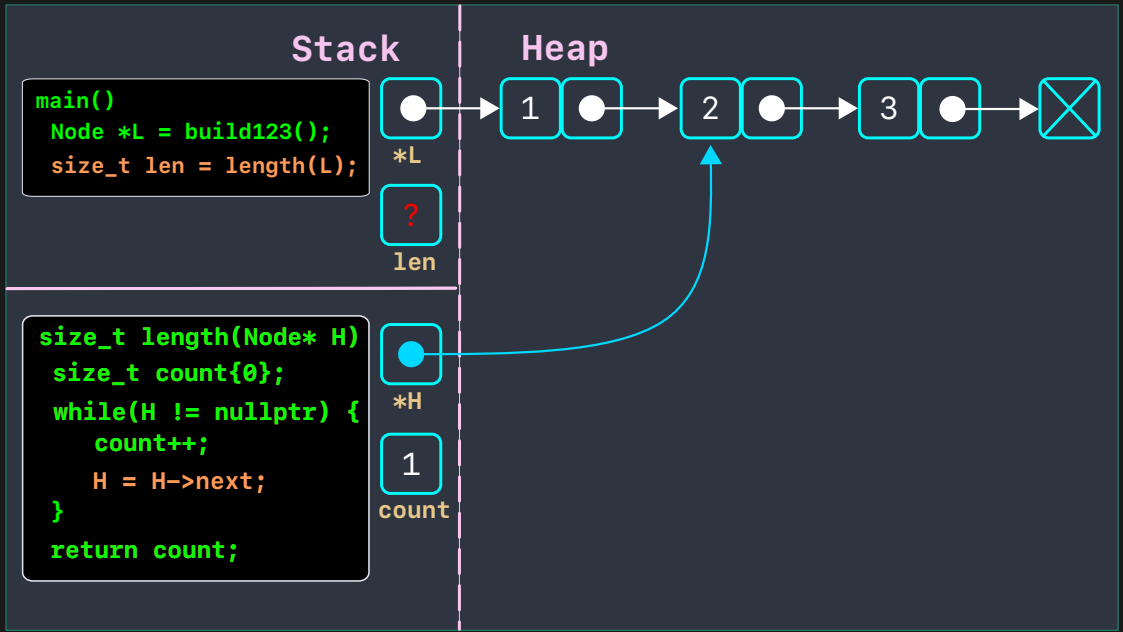

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
?
len
1
2
3
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;

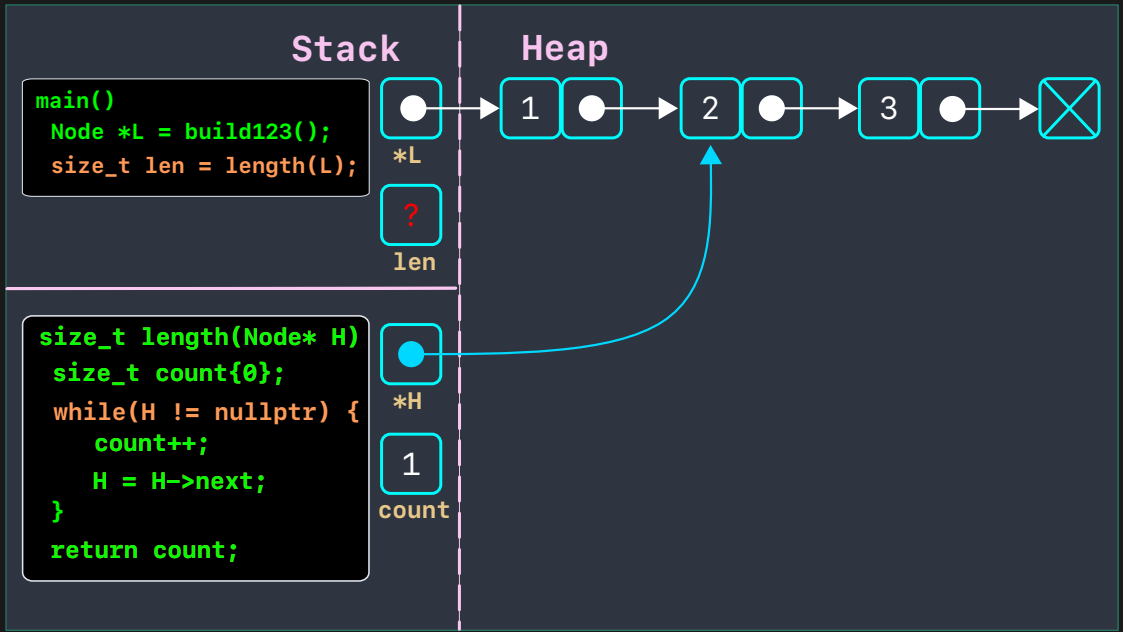

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H

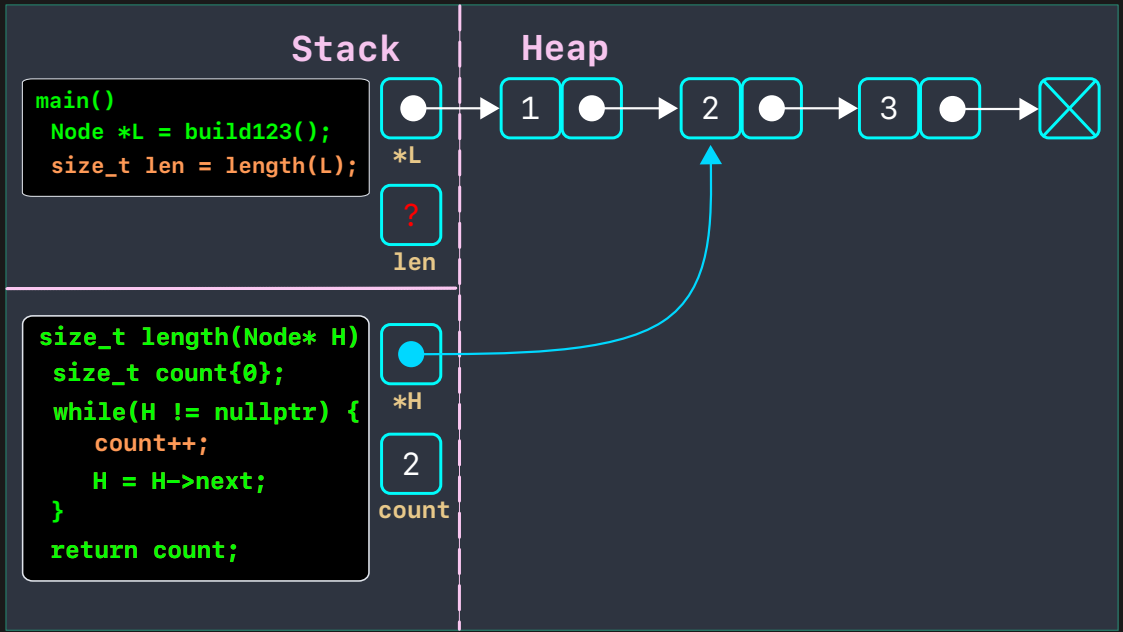

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
0
count

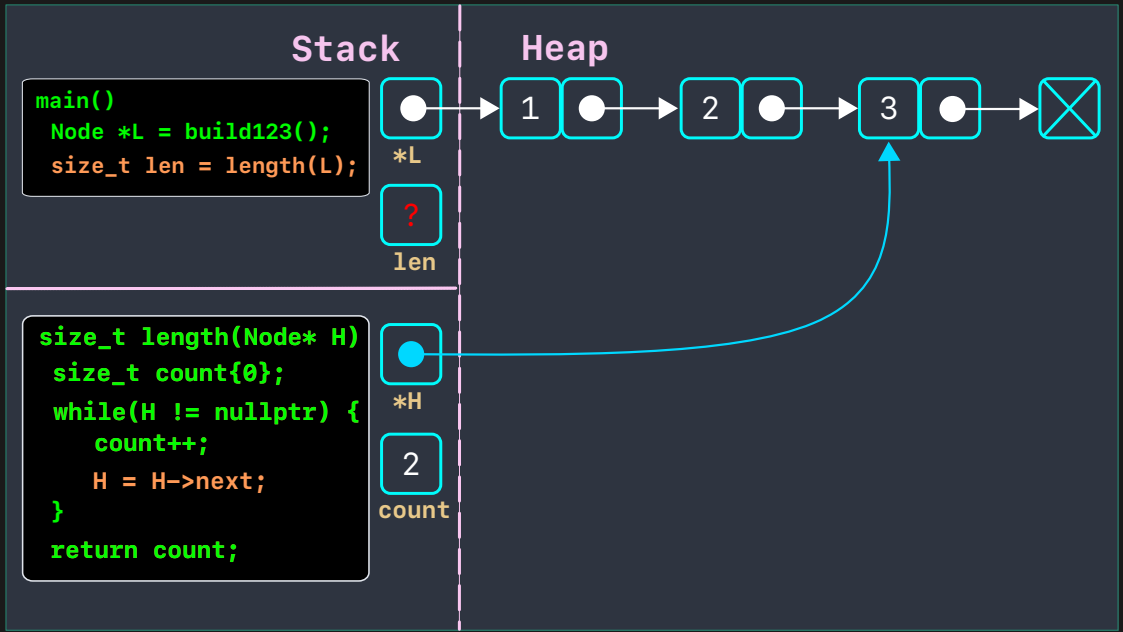

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
0
count

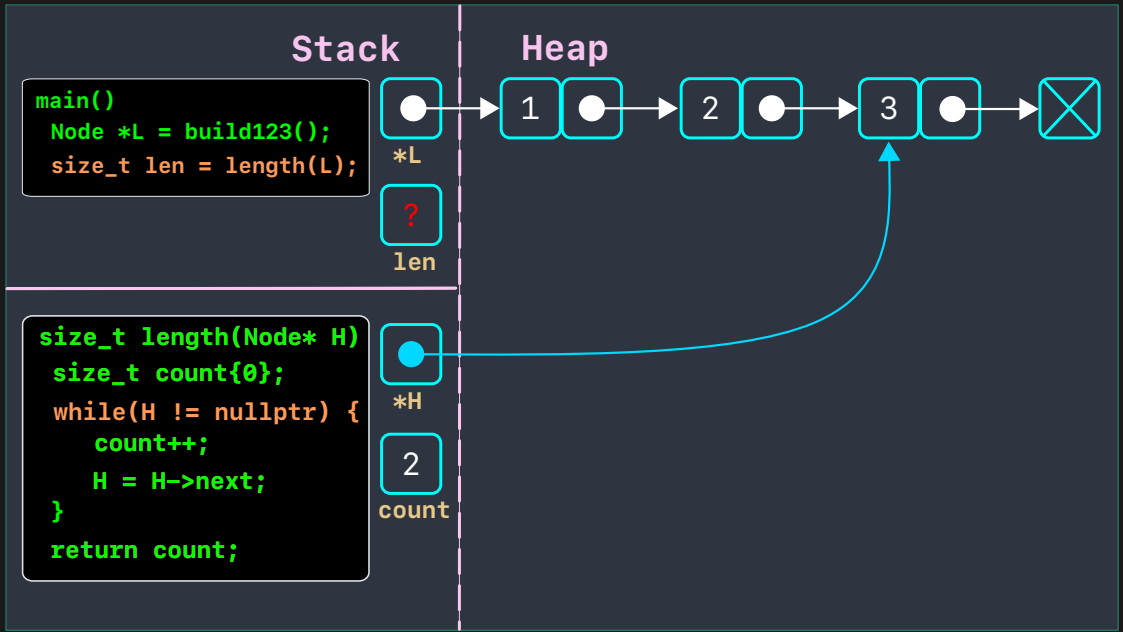

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
1
count

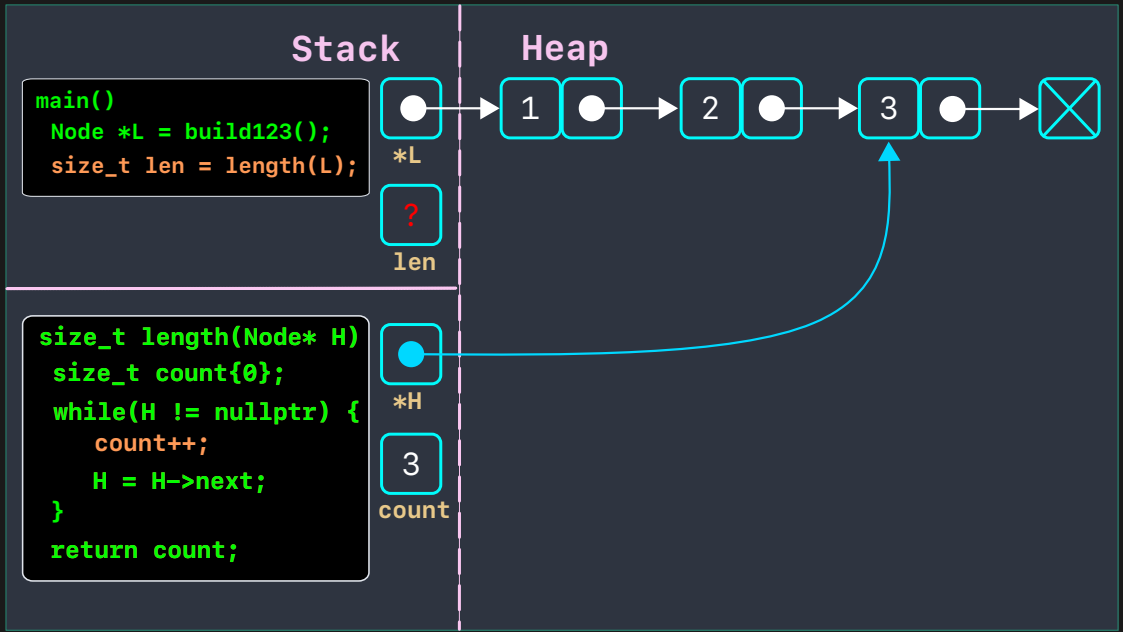

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
1
count

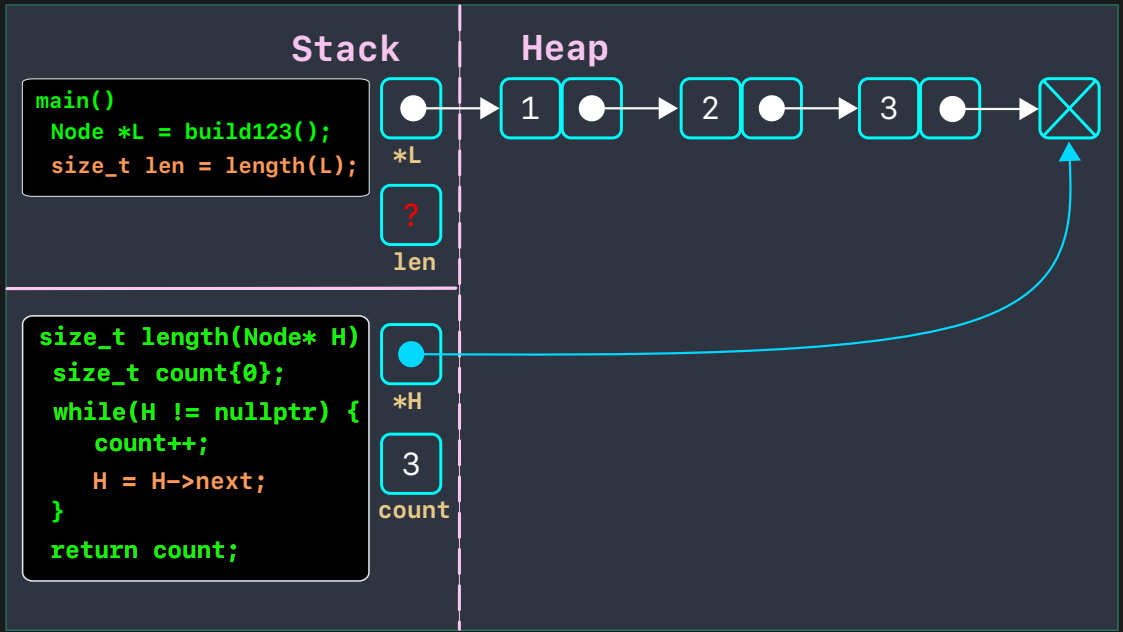

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
1
count

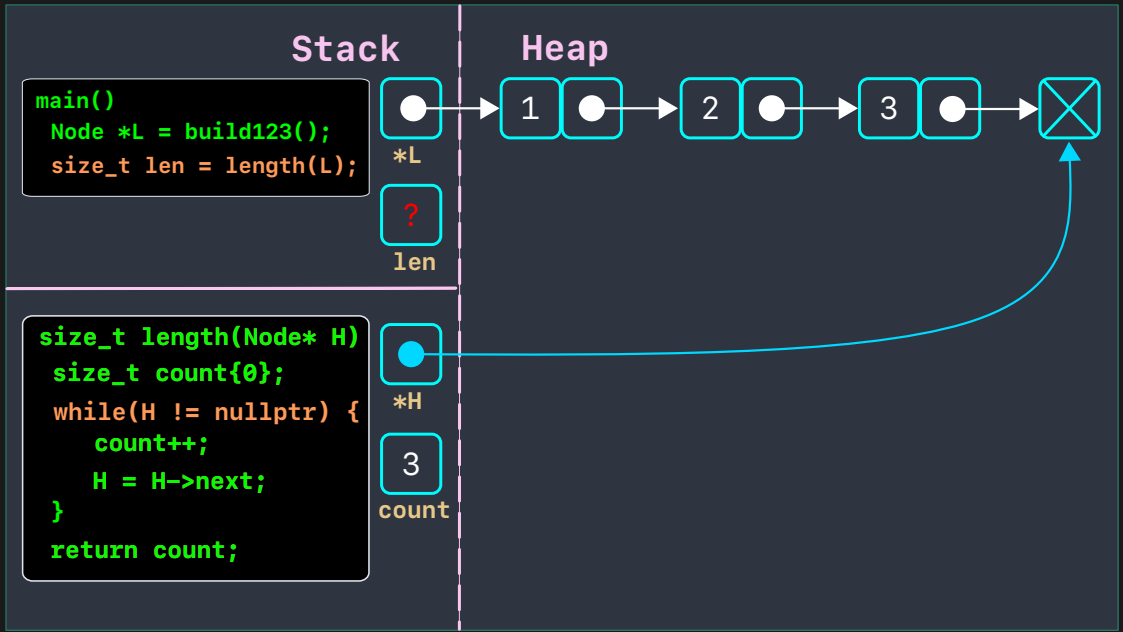

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
2
count

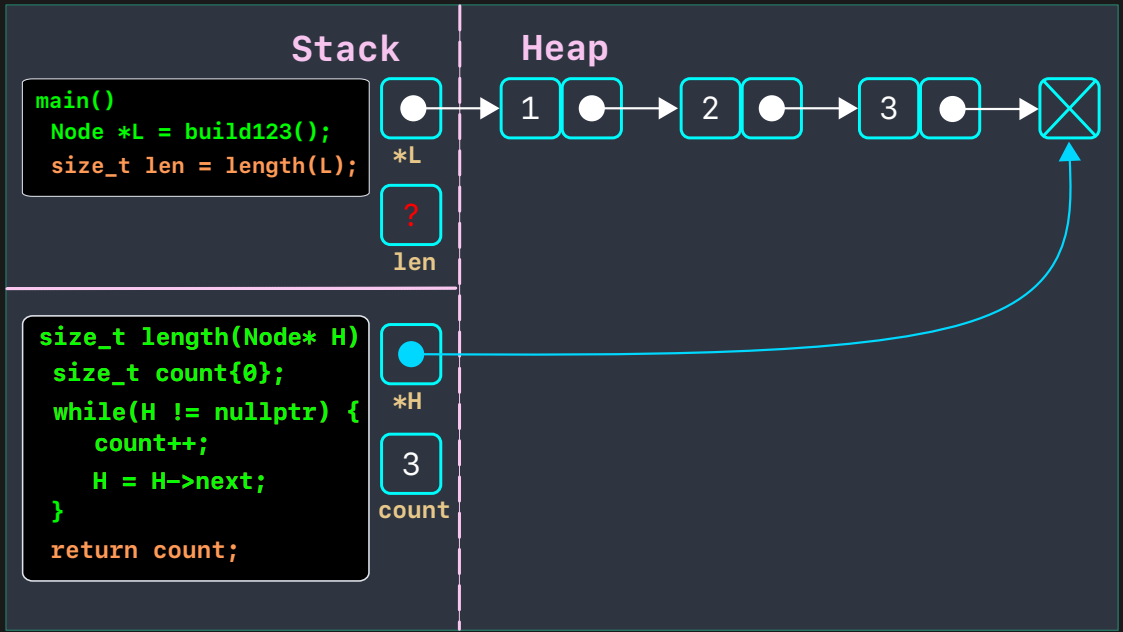

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
2
count

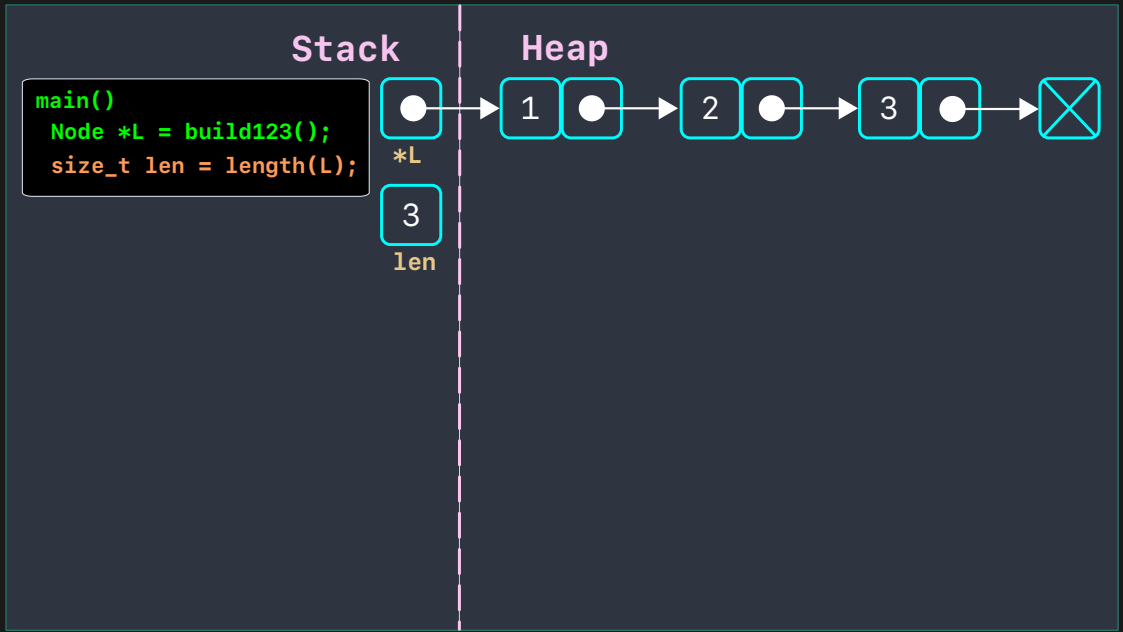

Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
2
count


Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
3
count


Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
3
count


Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
3
count


Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
1
2
3
?
len
size_t length(Node* H)
size_t count{0};
while(H != nullptr) {
count++;
H = H->next;
}
return count;
*H
3
count


Stack
Heap
main()
Node *L = build123();
size_t len = length(L);
*L
3
len
1
2
3

Percorrendo uma L.S.E.

Será que o algoritmo apresentado funciona para lista vazia?

Percorrendo uma L.S.E.

Será que o algoritmo apresentado funciona para lista vazia?

```
size_t length(Node* H){
   size_t count{0};
   while(H != nullptr) {
       count++;
       H = H->next;
   }
   return count;
}
```

Inserindo na Frente da Lista

A função `push_front()` apresentada a seguir recebe uma lista já pronta como argumento e um novo valor a ser inserido na frente da lista.

Inserindo na Frente da Lista

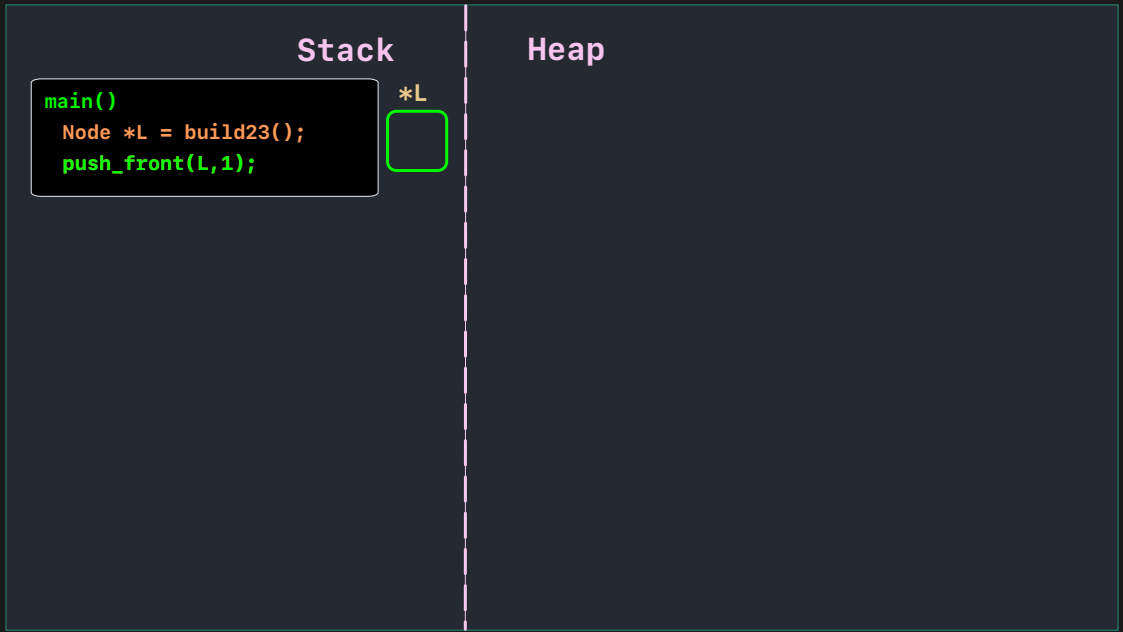
A função `push_front()` apresentada a seguir recebe uma lista já pronta como argumento e um novo valor a ser inserido na frente da lista.

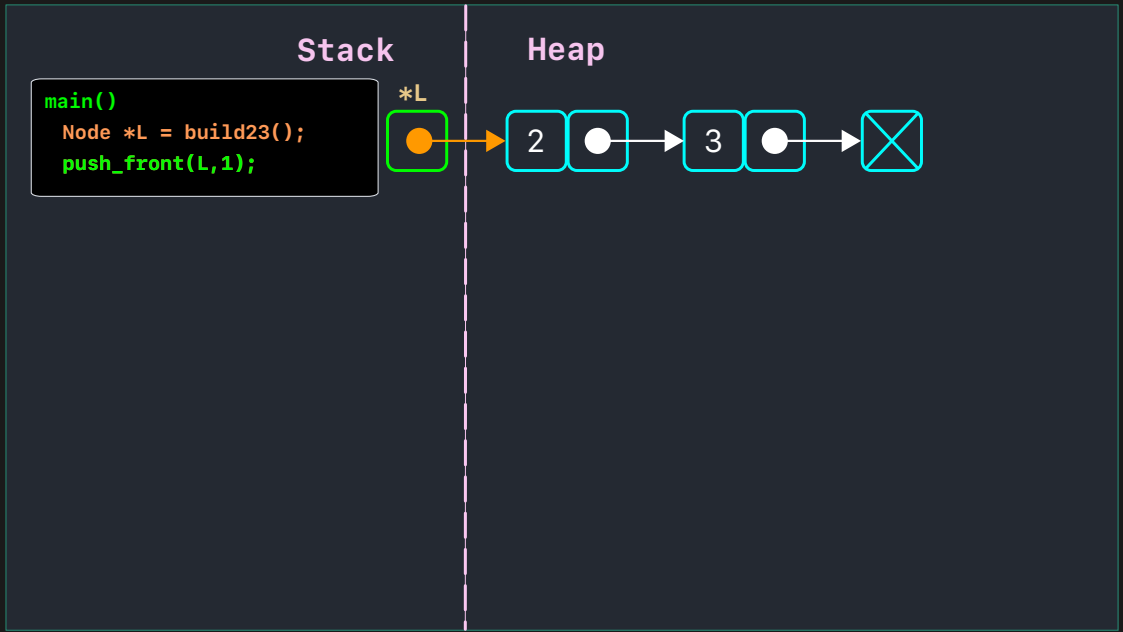
Como precisamos alterar o ponteiro para a lista no lado cliente, precisamos receber a lista na função por referência.

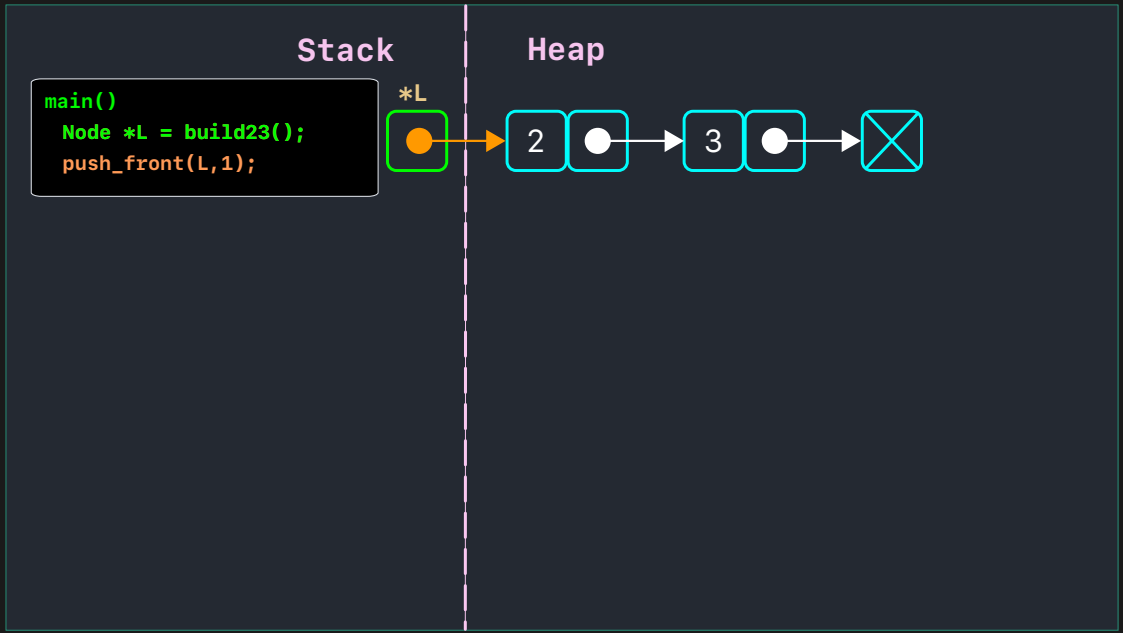
Stack

```
main()
  Node *L = build23();
  push_front(L,1);
```

Heap


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3

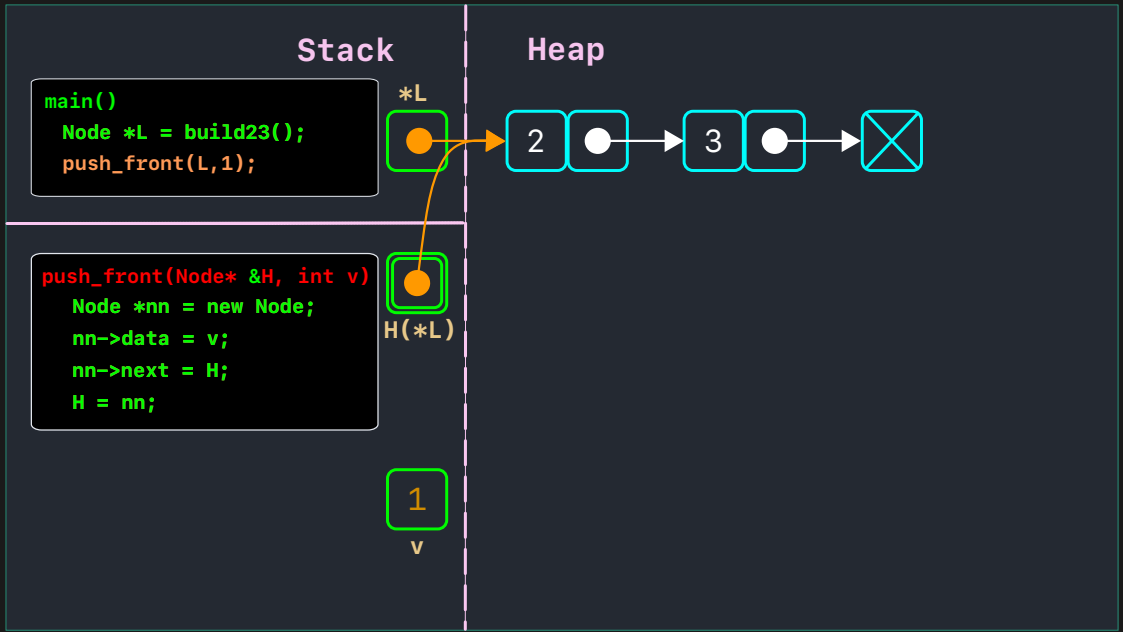

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3

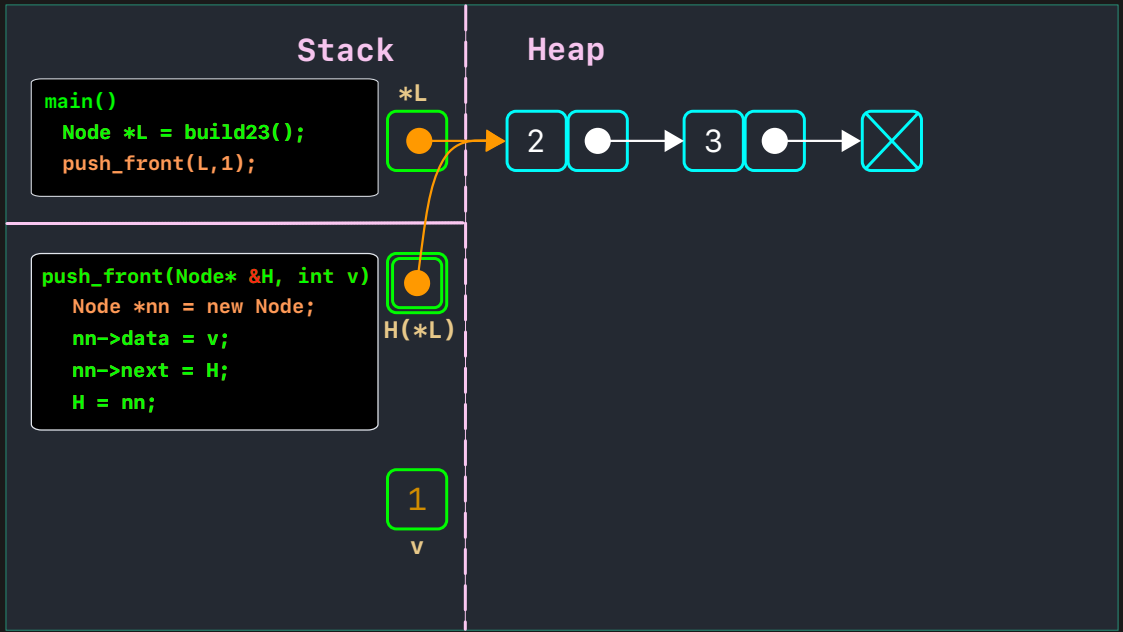
Stack

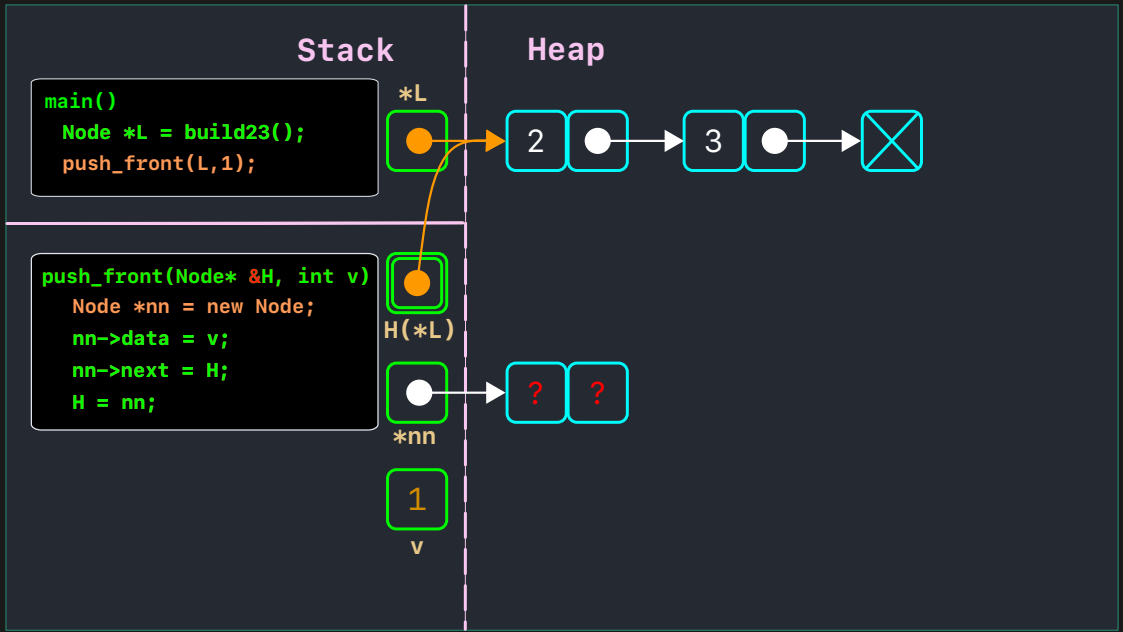
Heap

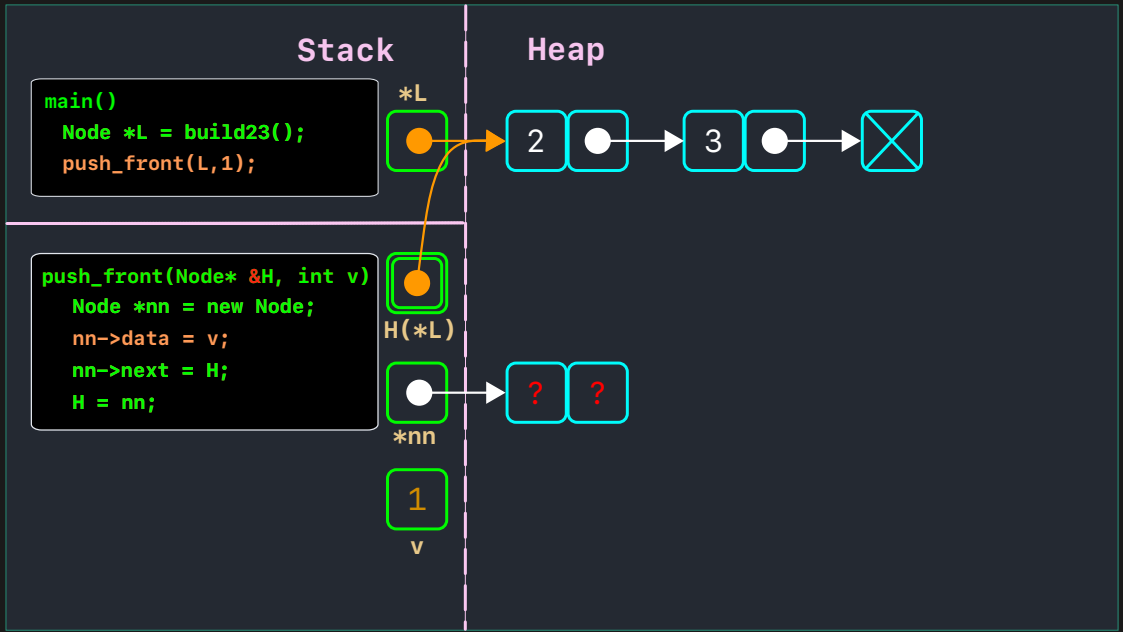
```
main()
  Node *L = build23();
  push_front(L,1);
```

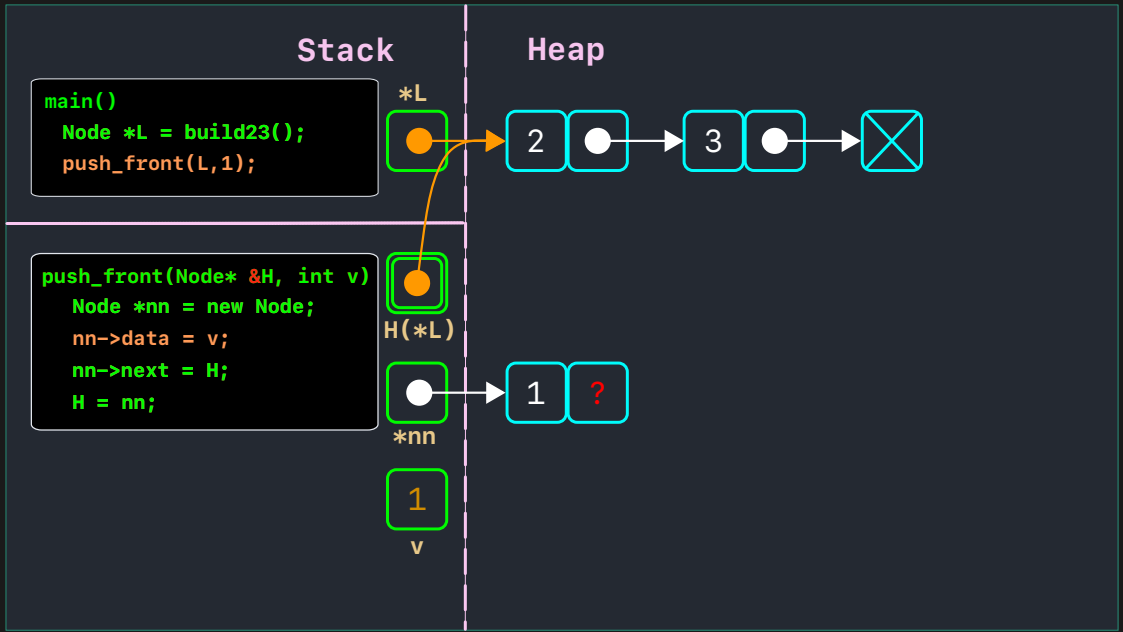

```
push_front(Node* &H, int v)
   Node *nn = new Node;
   nn->data = v;
   nn->next = H;
   H = nn;
```

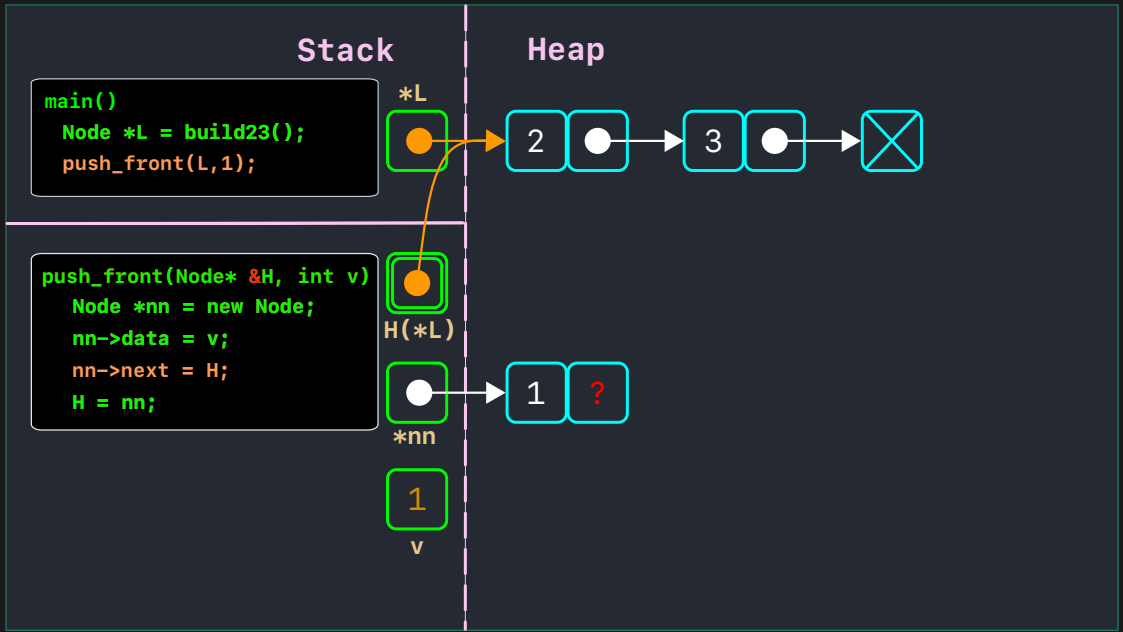

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
v

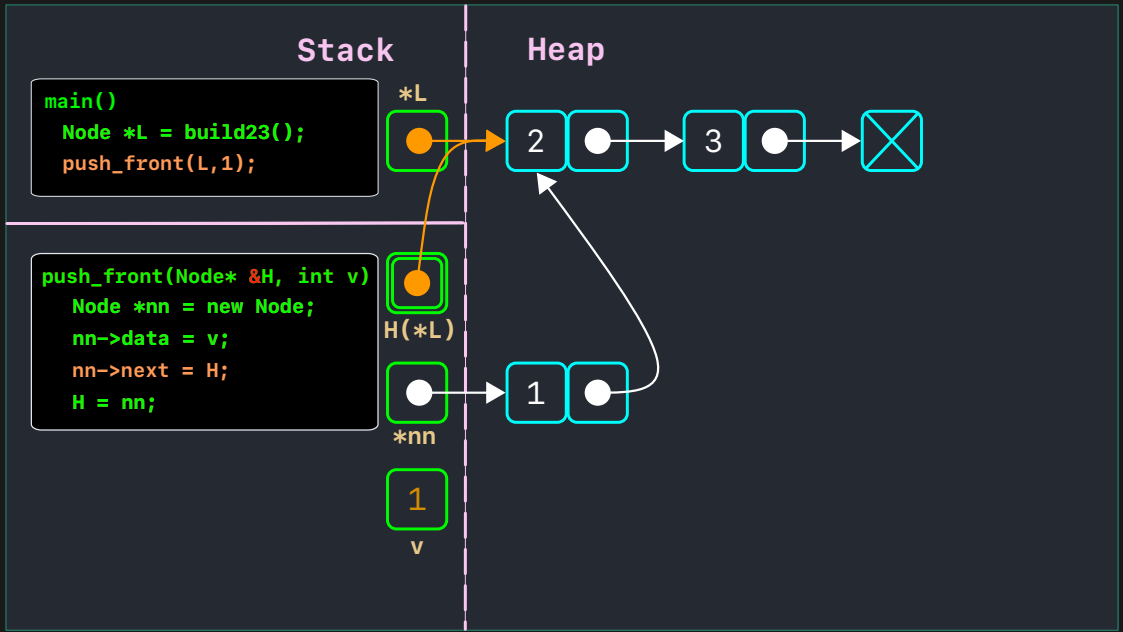

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
v

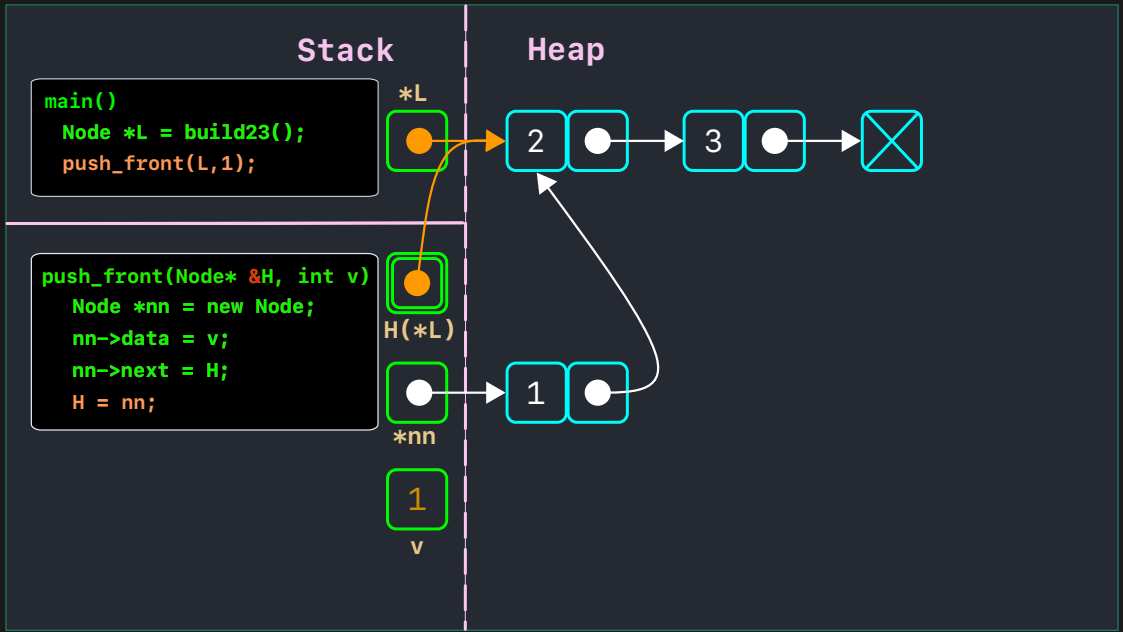

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
?
?
*nn
1
v

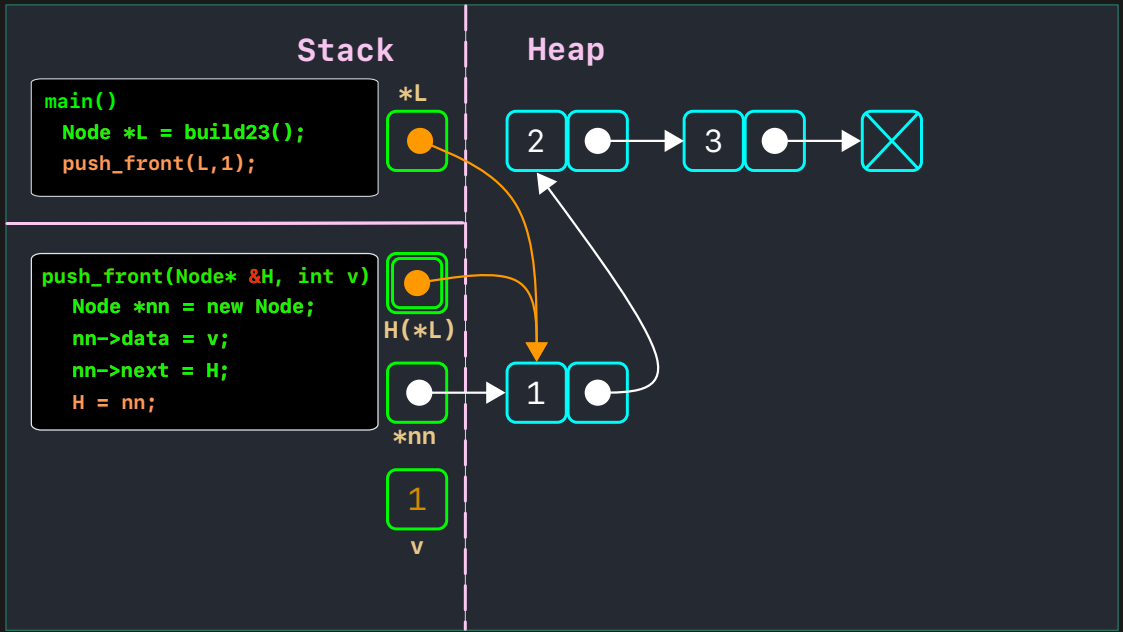

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
?
?
*nn
1
v

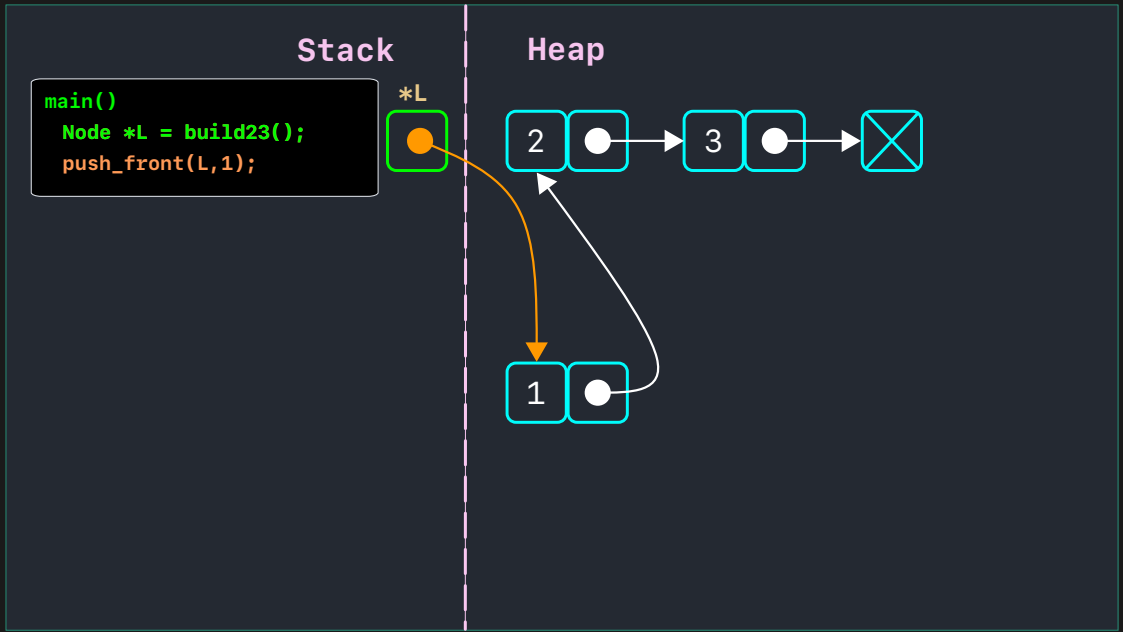

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
?
*nn
1
v

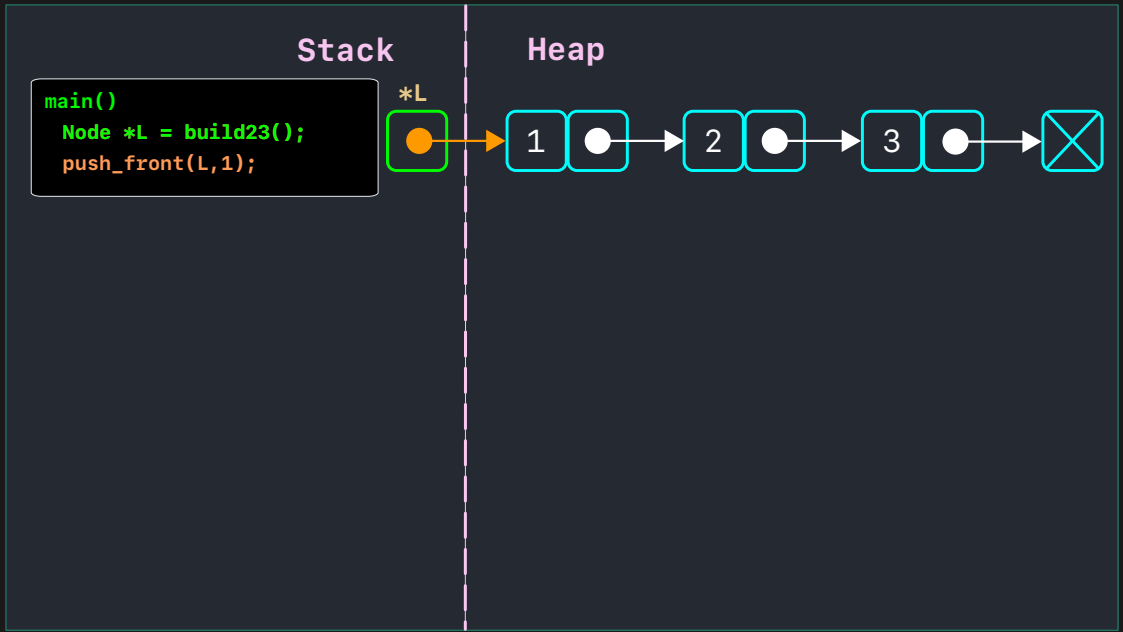

Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
?
*nn
1
v


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
*nn
1
v


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
*nn
1
v


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
push_front(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = H;
H = nn;
H(*L)
1
*nn
1
v


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
2
3
1


Stack
Heap
main()
Node *L = build23();
push_front(L,1);
*L
1
2
3

Inserindo na Frente da Lista

Será que o algoritmo apresentado funciona para lista vazia?

Inserindo na Frente da Lista

Será que o algoritmo apresentado funciona para lista vazia?

```
push_front(Node* &H,int v) {
    Node *nn = new Node;
    nn->data = v;
    nn->next = H;
    H = nn;
}
```

Inserindo no Final da Lista

Para inserir no final é um pouco mais elaborado. Vejamos.

1. Precisamos obter um ponteiro para o último nó da lista através do idioma de programação de percorrimento.

Inserindo no Final da Lista

Para inserir no final é um pouco mais elaborado. Vejamos.

1. Precisamos obter um ponteiro para o último nó da lista através do idioma de programação de percorrimento.
2. Criamos o novo nó e o conectamos à lista via campo `next` do nó obtido no passo anterior.

Inserindo no Final da Lista

Para inserir no final é um pouco mais elaborado. Vejamos.

1. Precisamos obter um ponteiro para o último nó da lista através do idioma de programação de percorrimento.
2. Criamos o novo nó e o conectamos à lista via campo `next` do nó obtido no passo anterior.
3. Caso não exista último nó na lista (lista vazia), o ponteiro de início de lista (no cliente) deve apontar para o nó criado.

Inserindo no Final da Lista

Mas antes vamos criar uma função auxiliar `get_last(Node *L)` que retorna um ponteiro para o último nó da lista.

Inserindo no Final da Lista

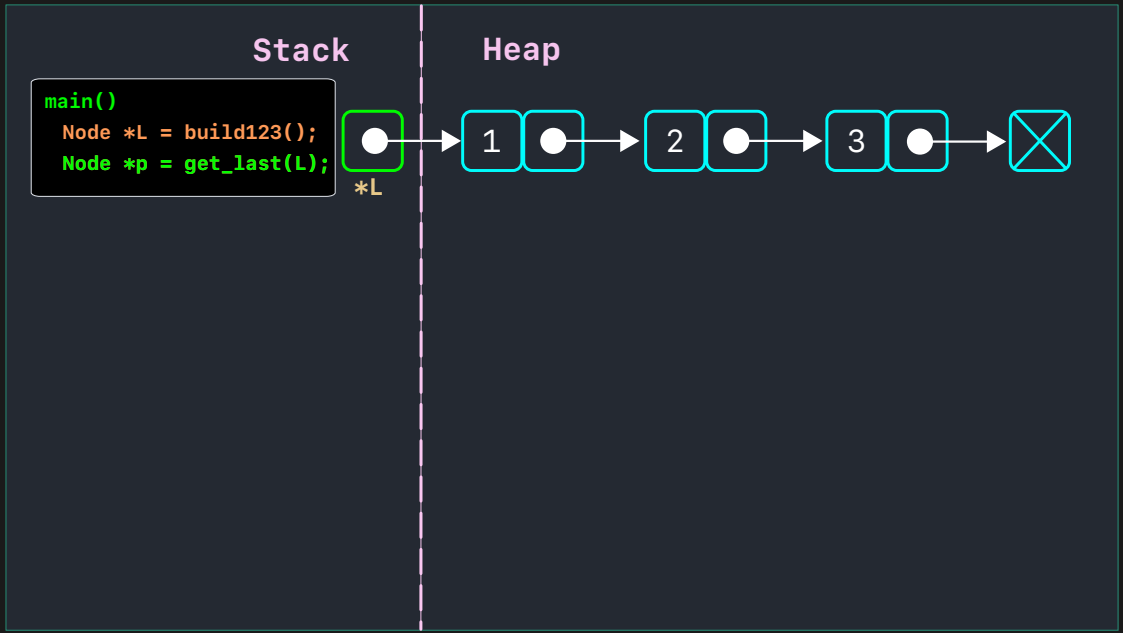
Mas antes vamos criar uma função auxiliar `get_last(Node *L)` que retorna um ponteiro para o último nó da lista.

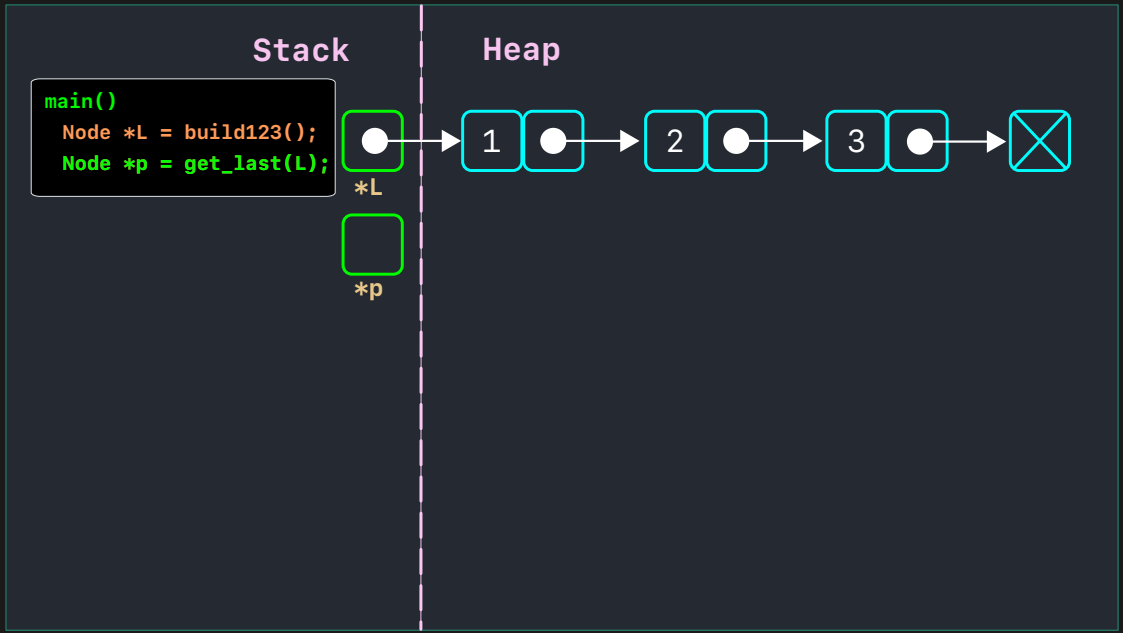
Se a lista `L` passada for vazia a função deve retornar `nullptr`, indicando que não existe nó na lista.

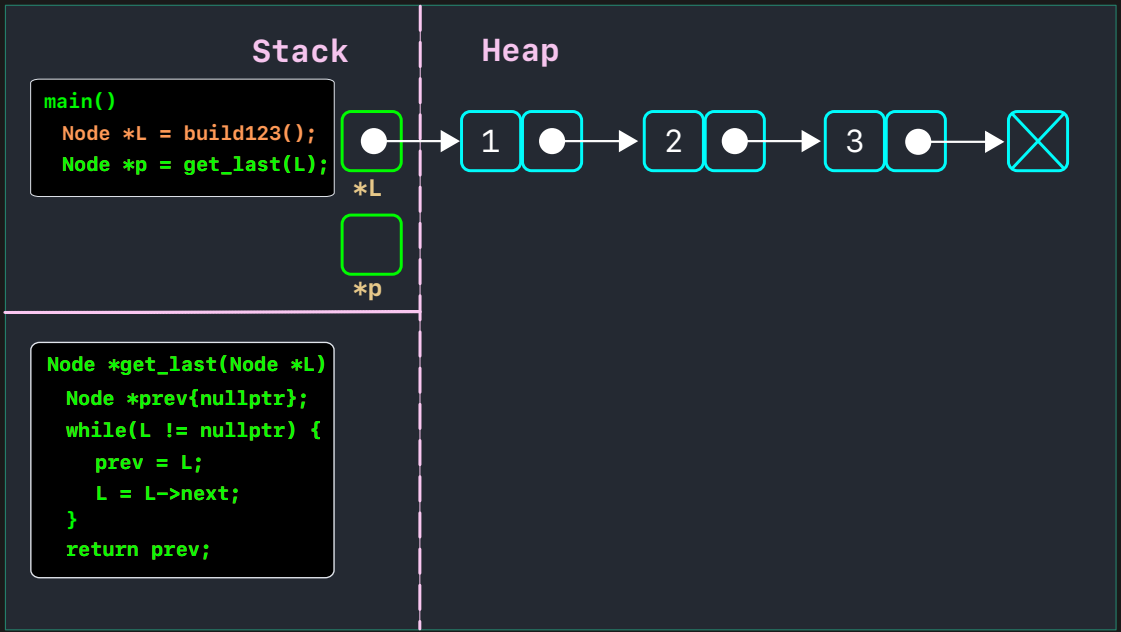
Stack

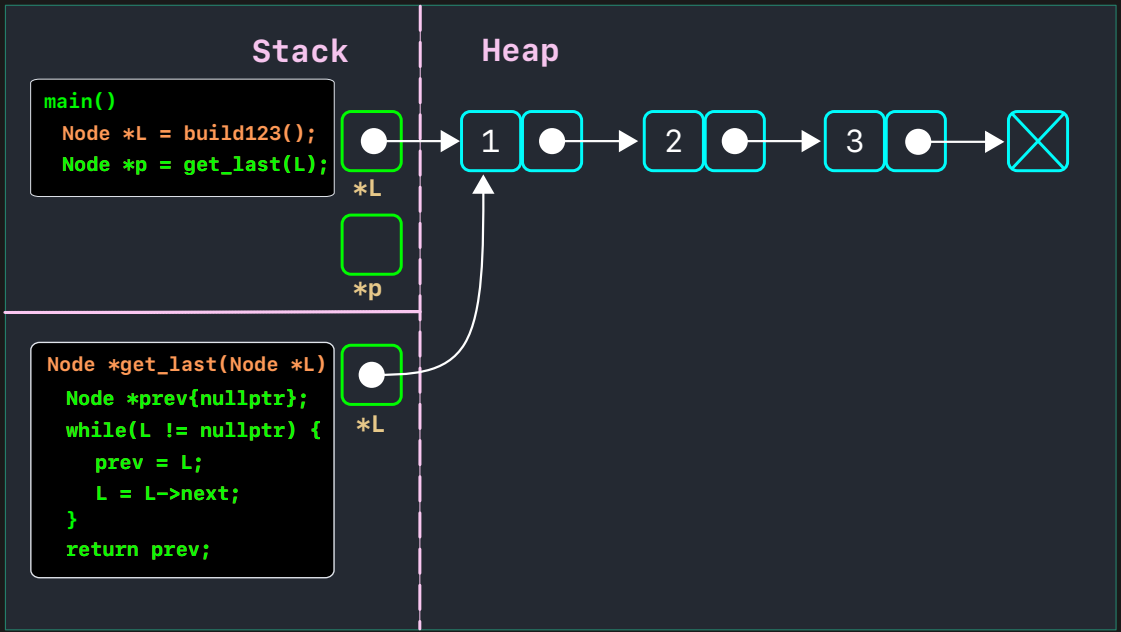
Heap

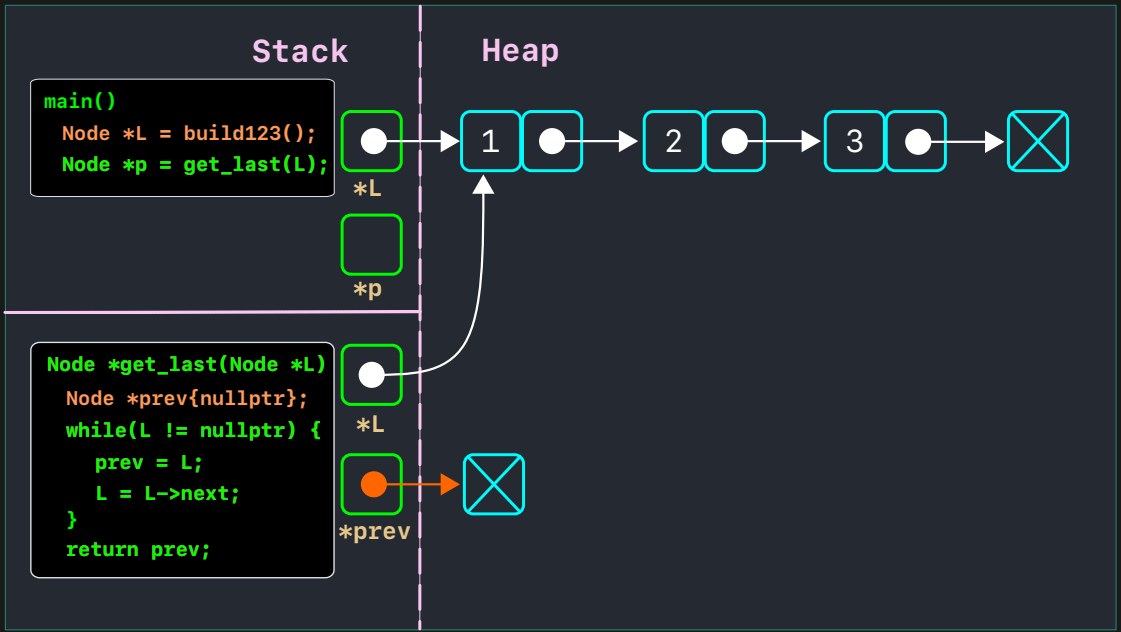
```
main()
  Node *L = build123();
  Node *p = get_last(L);
```

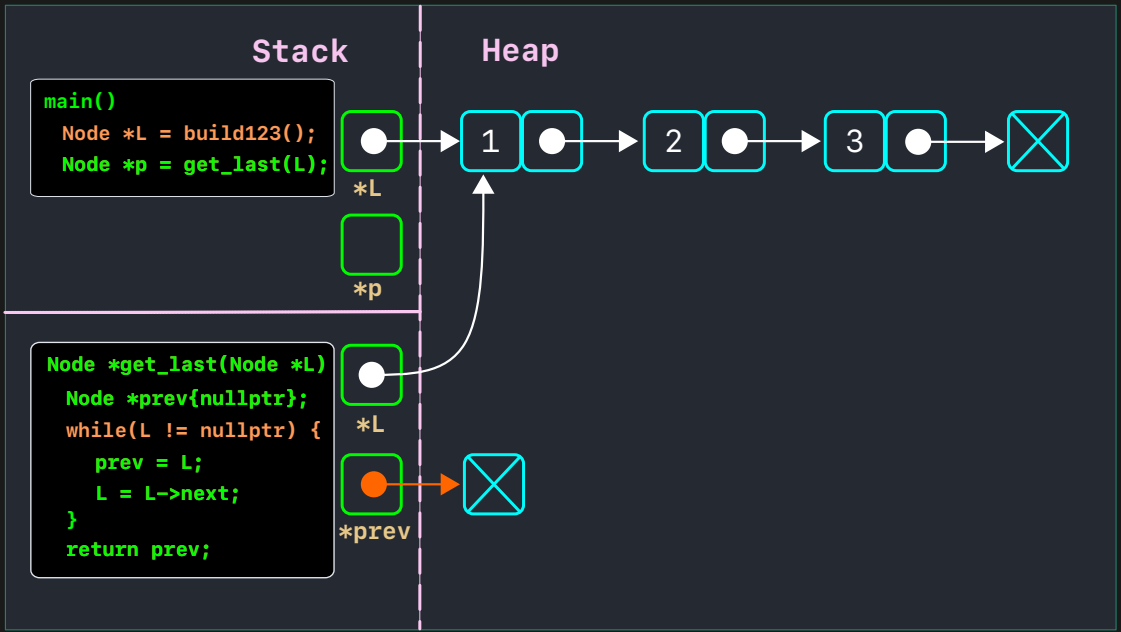

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
1
2
3

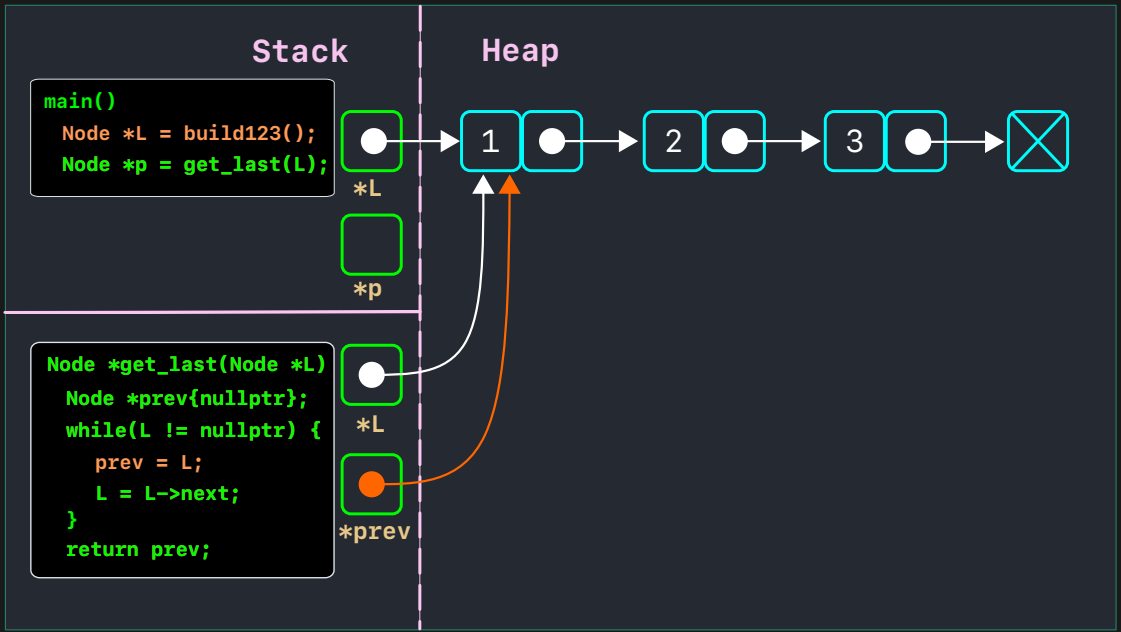

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3

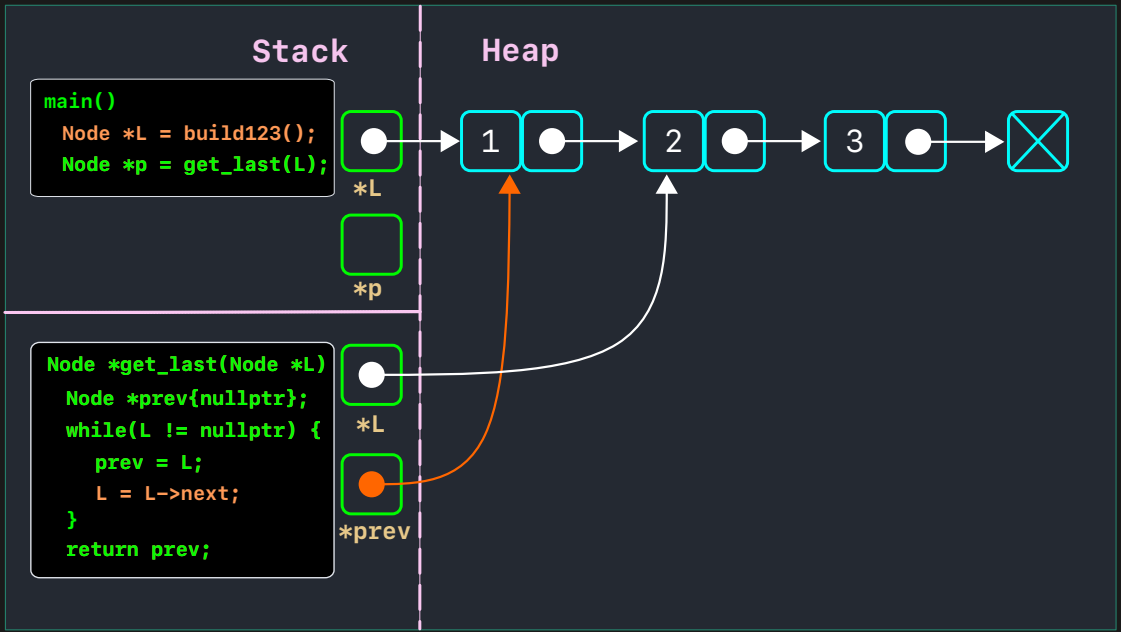

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;

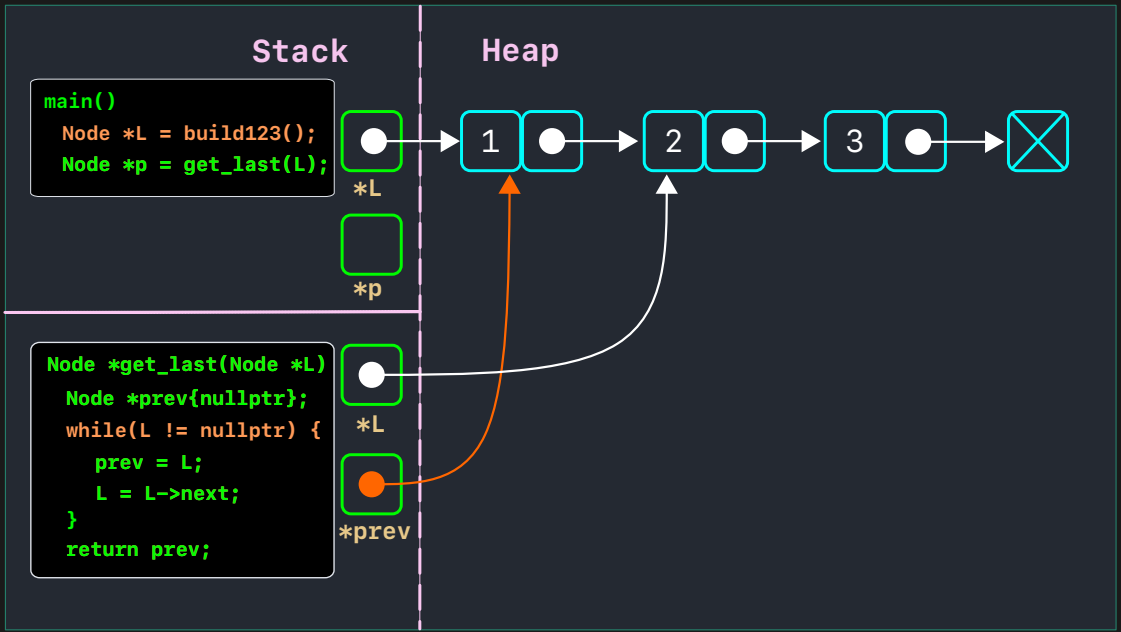

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L

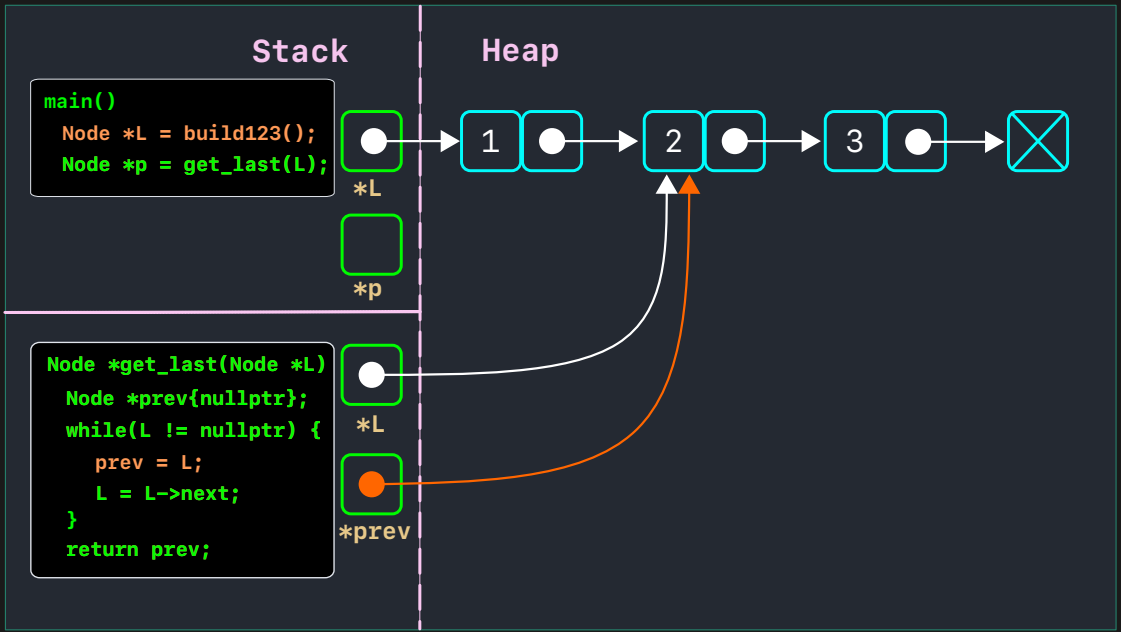

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

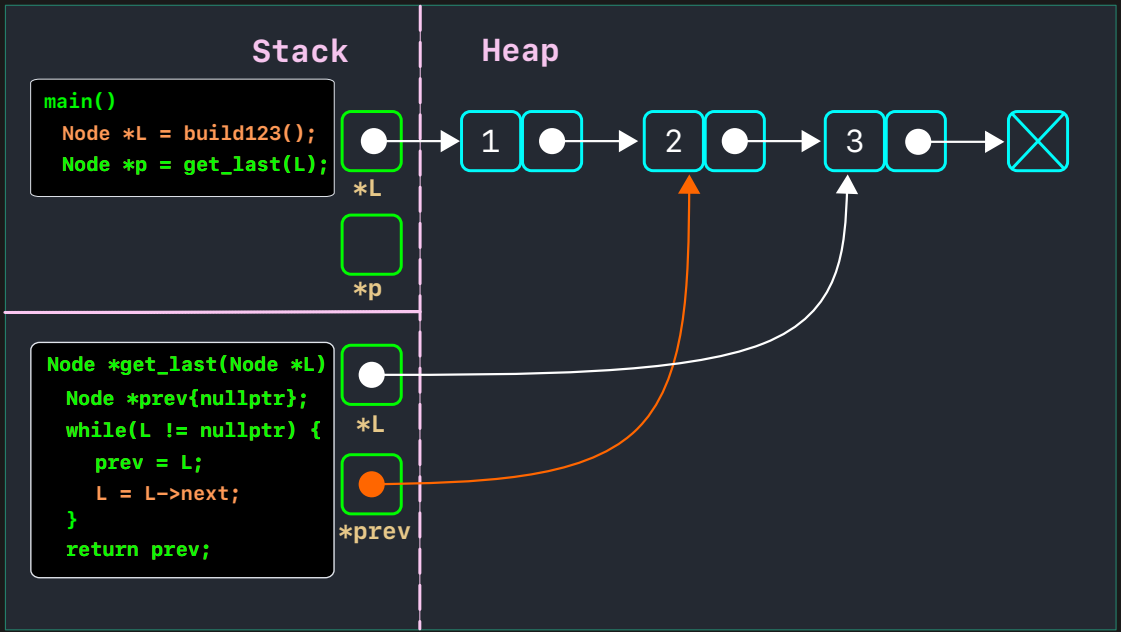

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

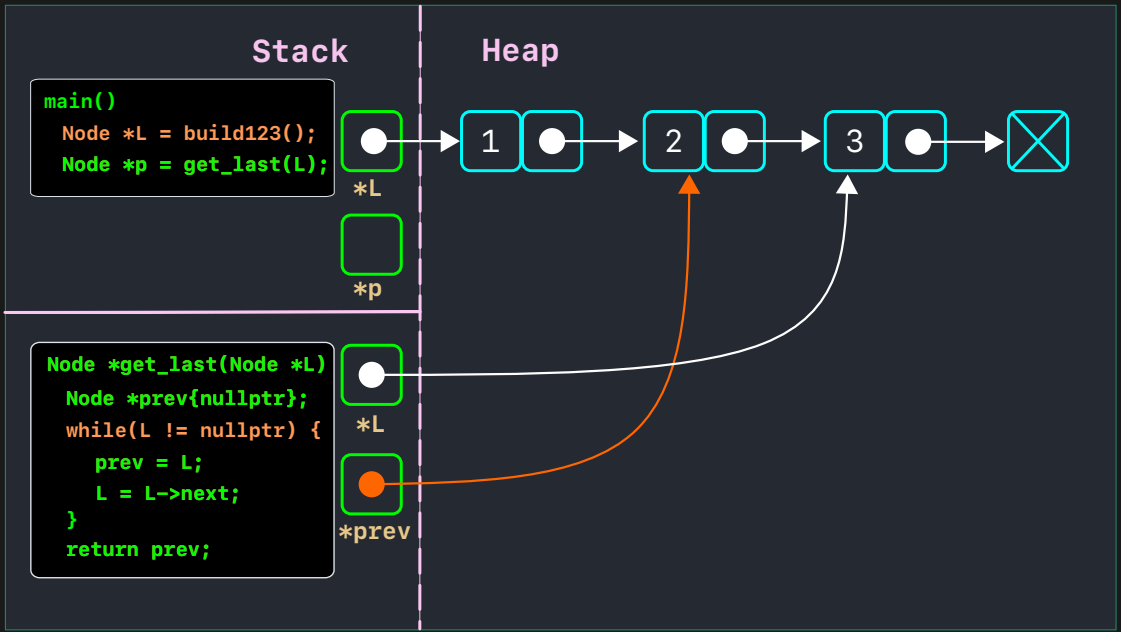

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

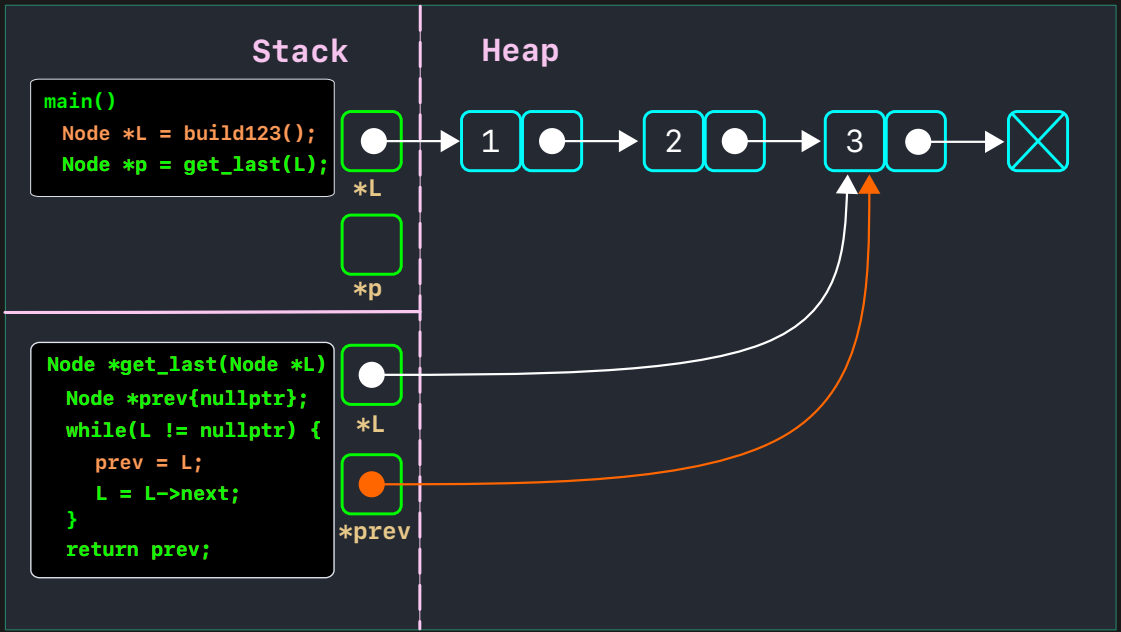

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

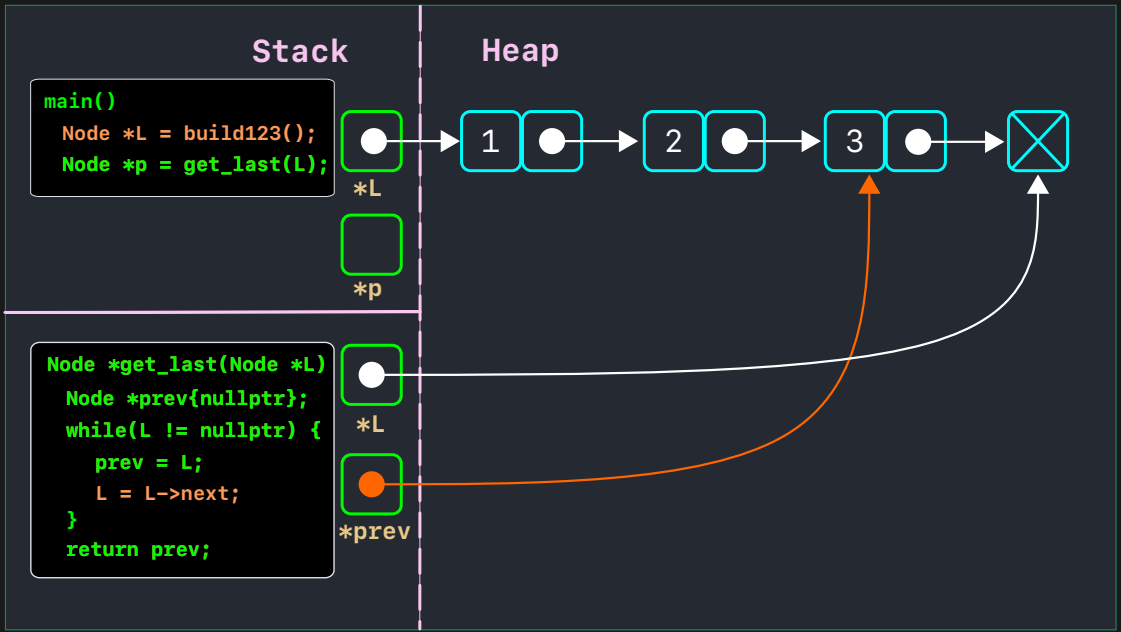

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

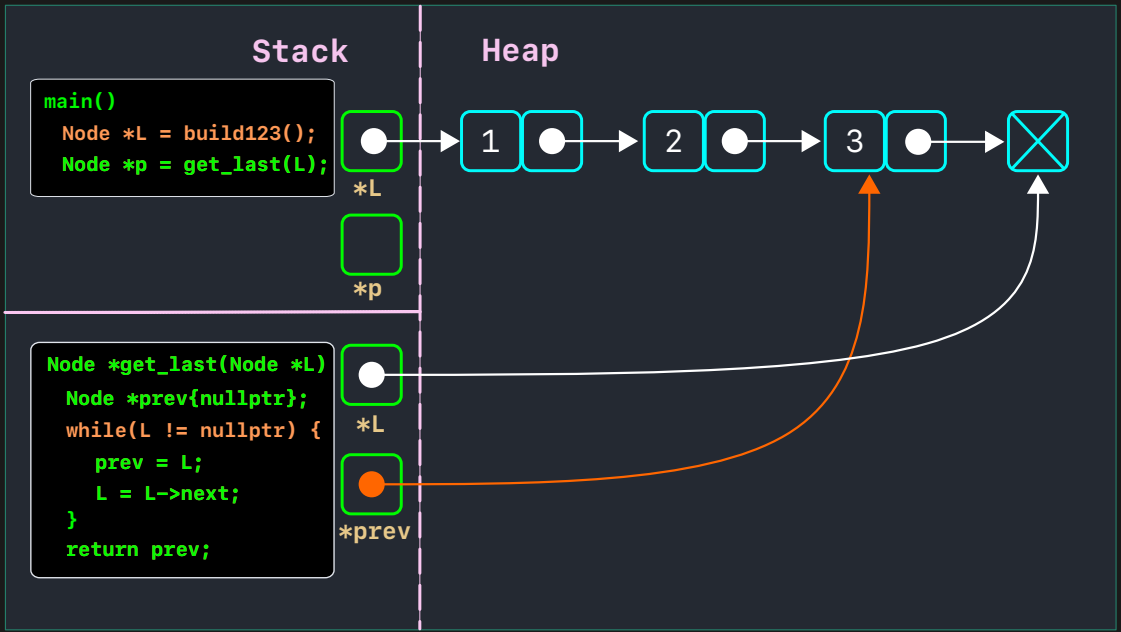

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

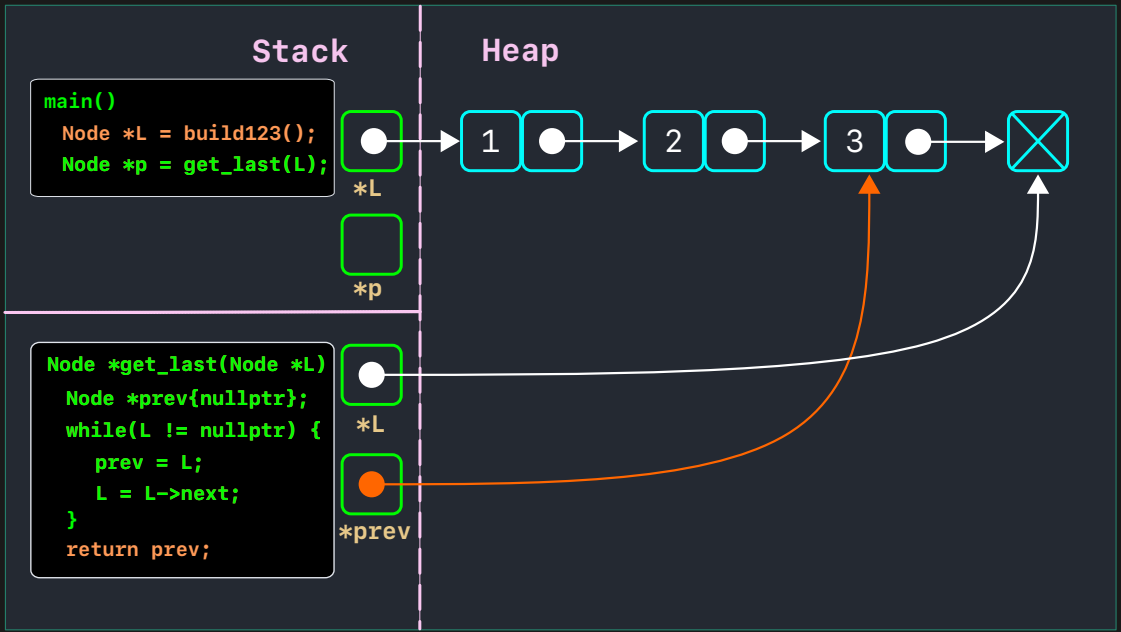

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

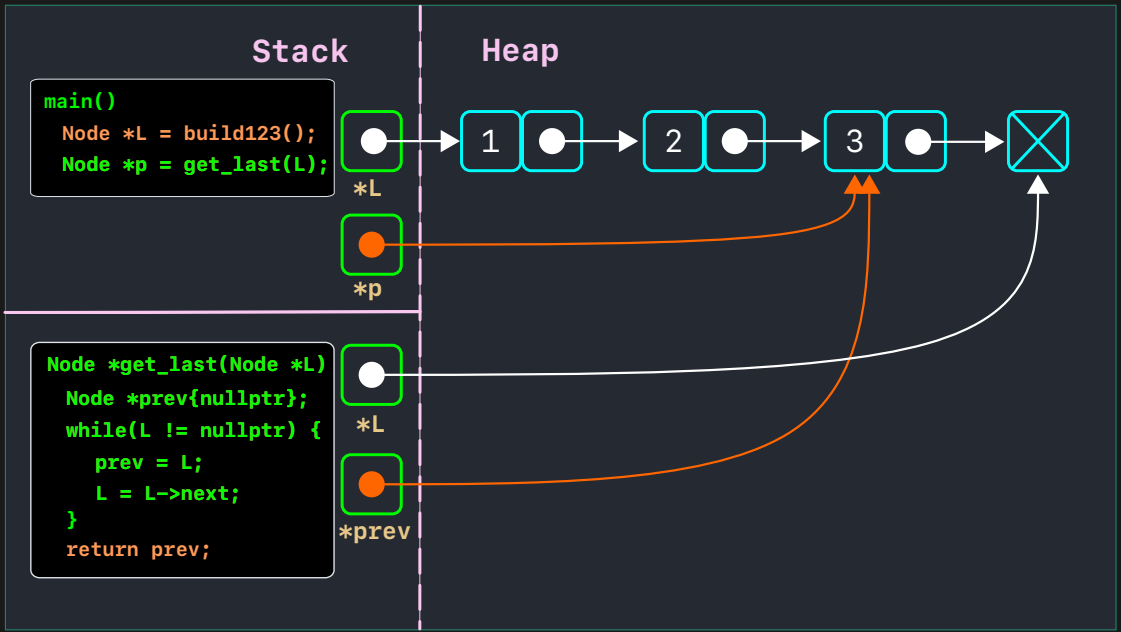

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev

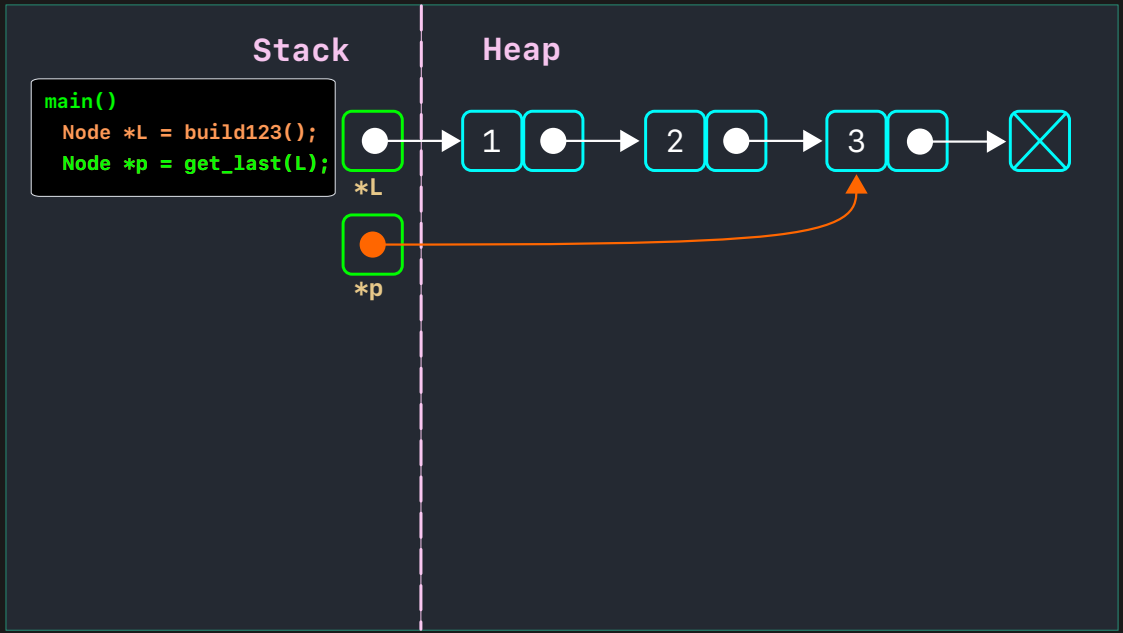

Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev


Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev


Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev


Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev


Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3
Node *get_last(Node *L)
Node *prev{nullptr};
while(L != nullptr) {
prev = L;
L = L->next;
}
return prev;
*L
*prev


Stack
Heap
main()
Node *L = build123();
Node *p = get_last(L);
*L
*p
1
2
3

Recuperando o Último Nó da Lista

Note que o algoritmo apresentado funciona mesmo para uma lista vazia, retornando `nullptr` conforme prometido.

```
Node *get_last(Node *L) {
  Node *prev{nullptr};
  while(L != nullptr) {
     prev = L;
     L = L->next;
  }
  return prev;
}
```

Inserindo na Frente da Lista

Agora vamos ver como fica a função `push_back()` que utiliza a função `get_last()` recém apresentada.

Inserindo na Frente da Lista

Agora vamos ver como fica a função `push_back()` que utiliza a função `get_last()` recém apresentada.

Para determinar se precisamos passar o ponteiro para o início da lista por referência precisamos perguntar:

- *Existe a possibilidade de precisarmos alterar o ponteiro head que aponta para o início da lista, lá no código cliente?*

Inserindo na Frente da Lista

Agora vamos ver como fica a função `push_back()` que utiliza a função `get_last()` recém apresentada.

Para determinar se precisamos passar o ponteiro para o início da lista por referência precisamos perguntar:

- *Existe a possibilidade de precisarmos alterar o ponteiro head que aponta para o início da lista, lá no código cliente?*
- Se a resposta for **sim**, precisamos passar o ponteiro para o início da lista por referência.

Stack

Heap

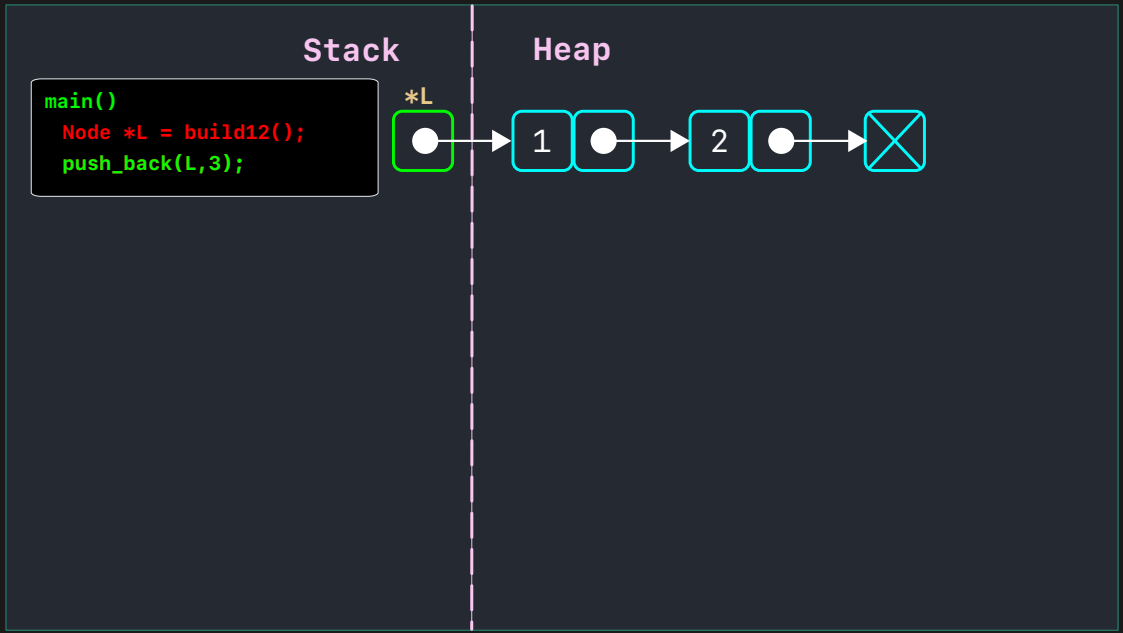
```
main()
  Node *L = build12();
  push_back(L,3);
```

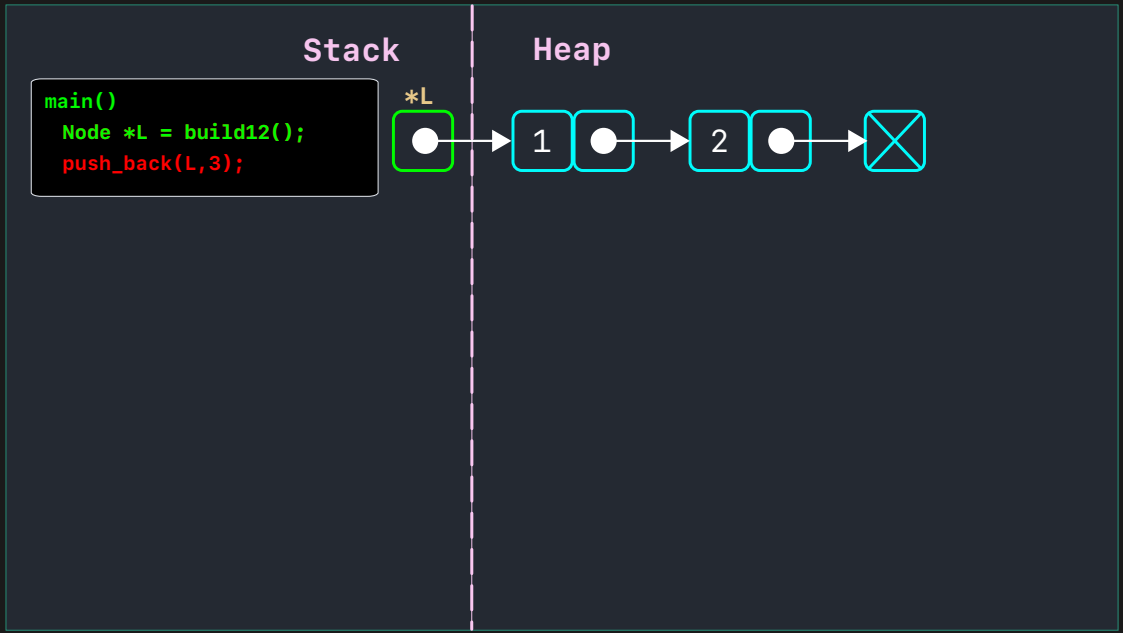
Stack

Heap

```
main()
  Node *L = build12();
  push_back(L,3);
```

*L


Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2

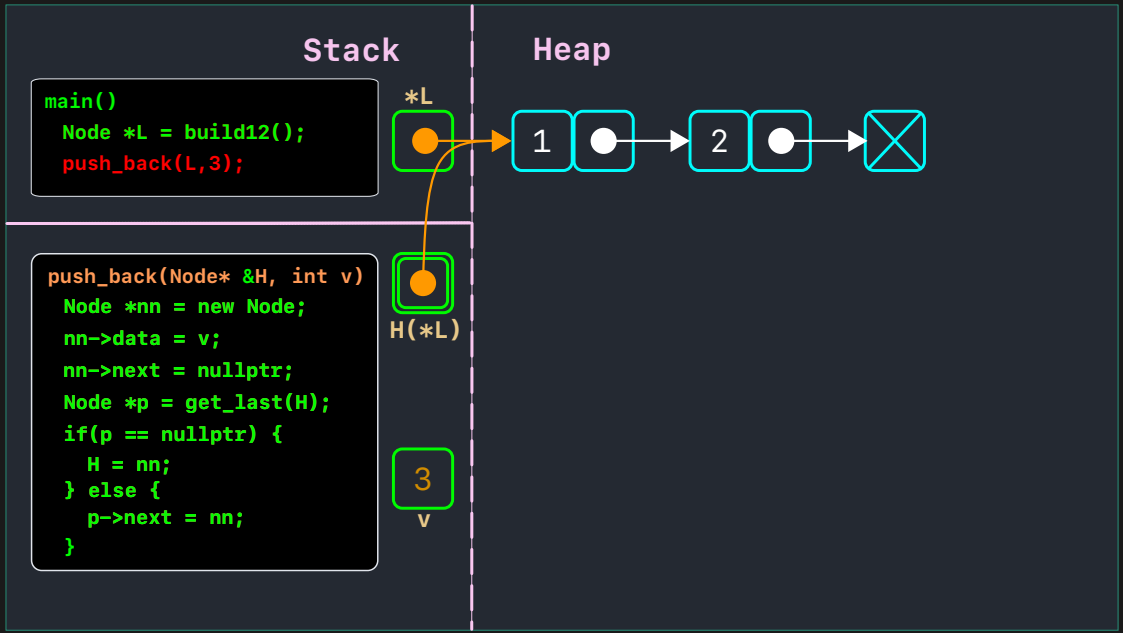

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2

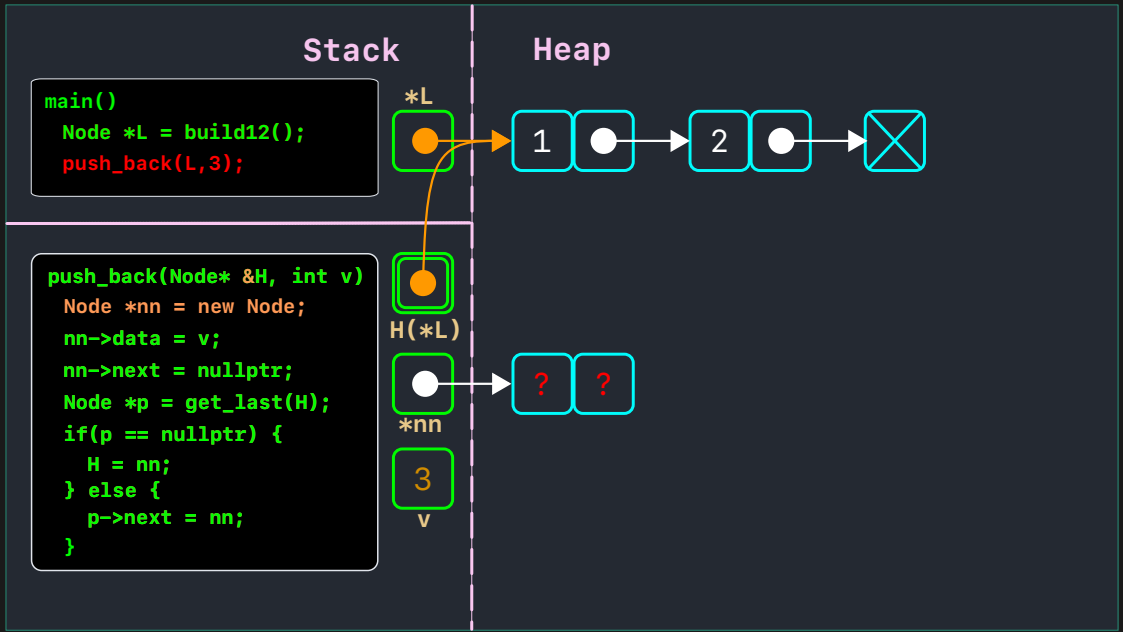
Stack

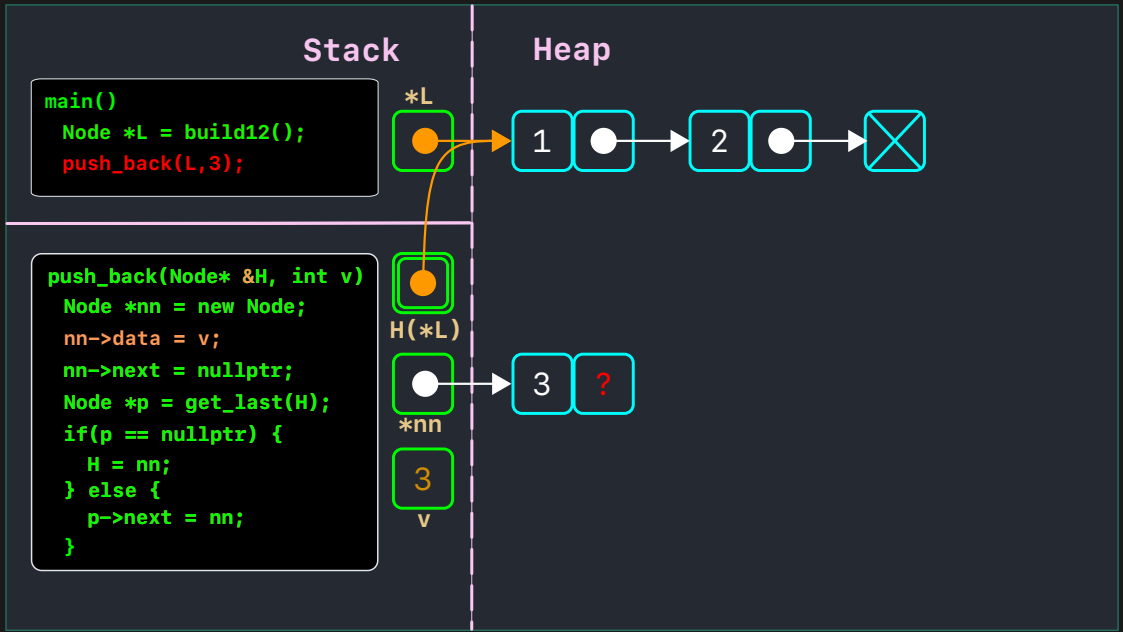
Heap

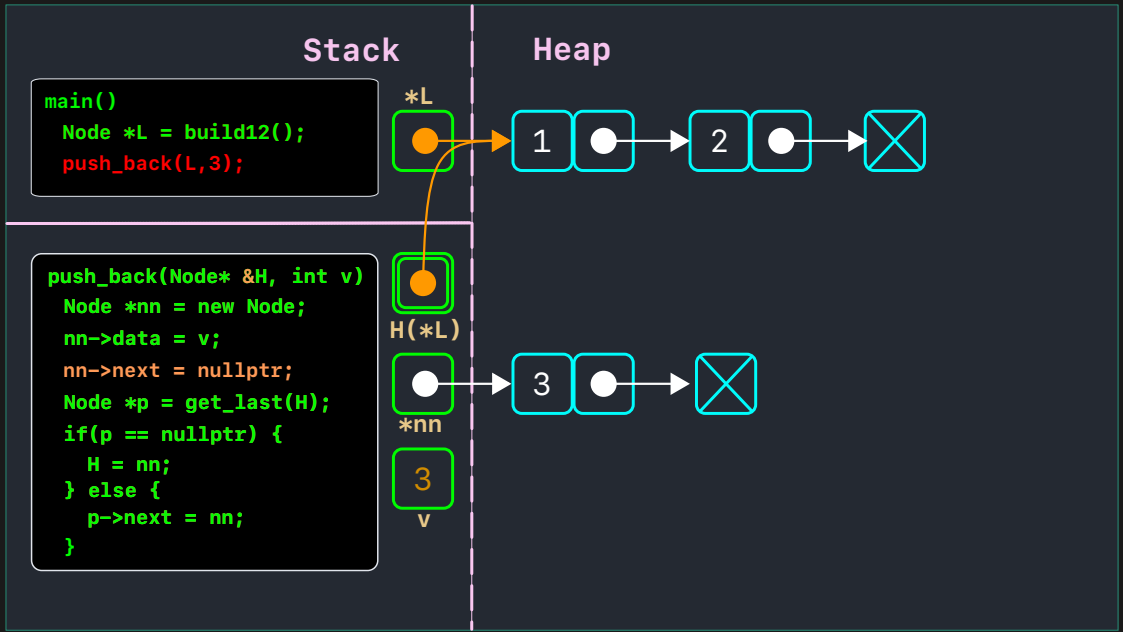
```
main()
  Node *L = build12();
  push_back(L,3);
```

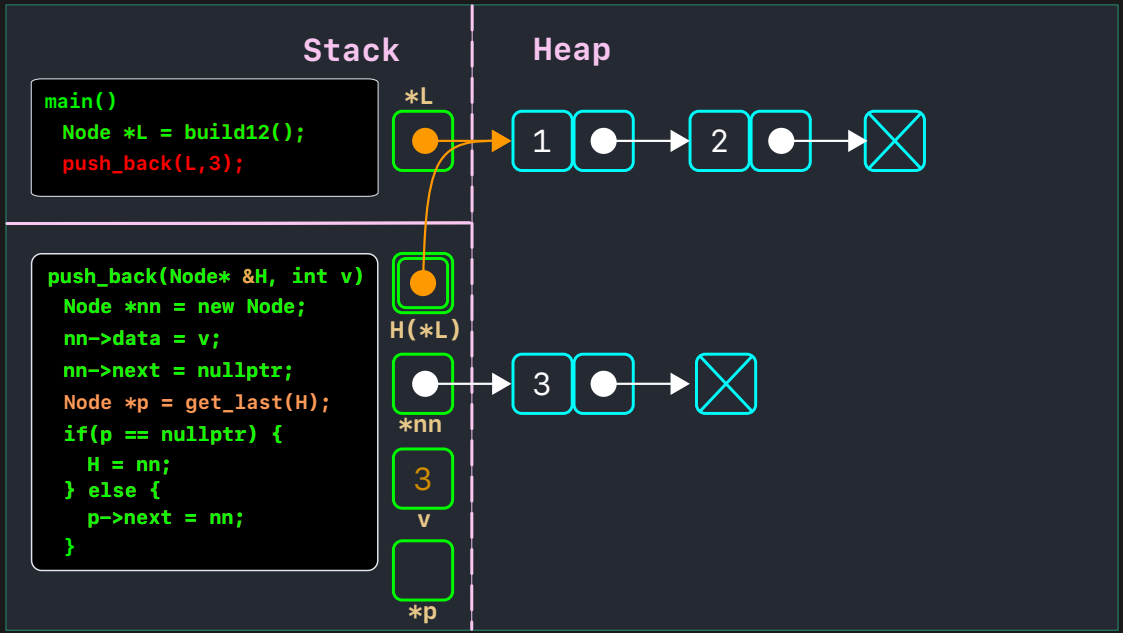

```
push_back(Node* &H, int v)
  Node *nn = new Node;
  nn->data = v;
  nn->next = nullptr;
  Node *p = get_last(H);
  if(p == nullptr) {
    H = nn;
  } else {
    p->next = nn;
  }
```

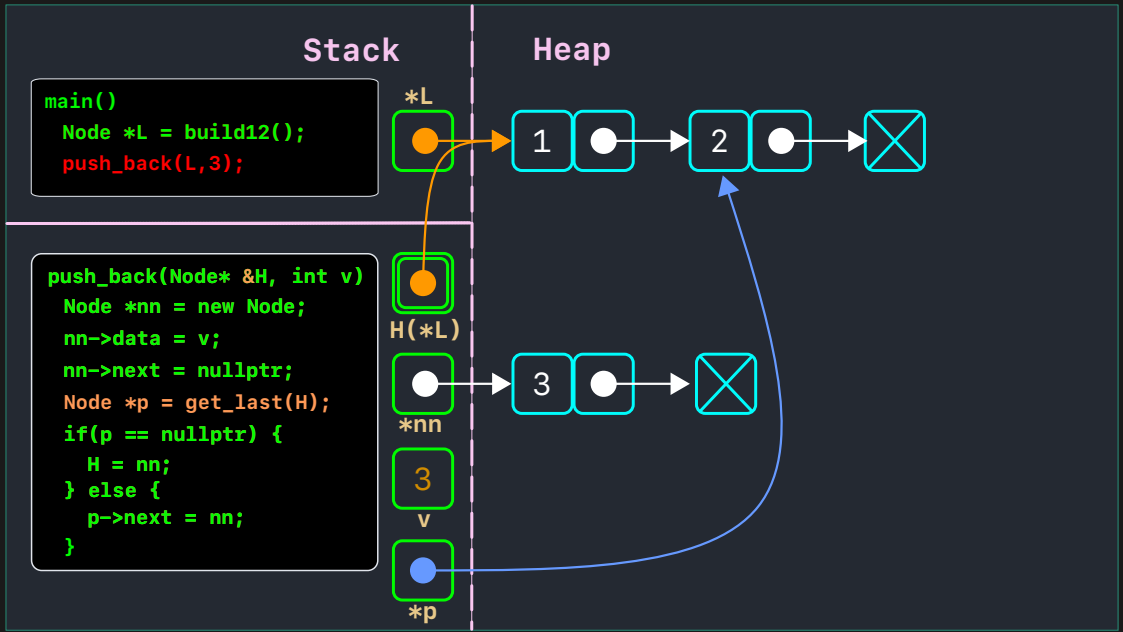

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
v

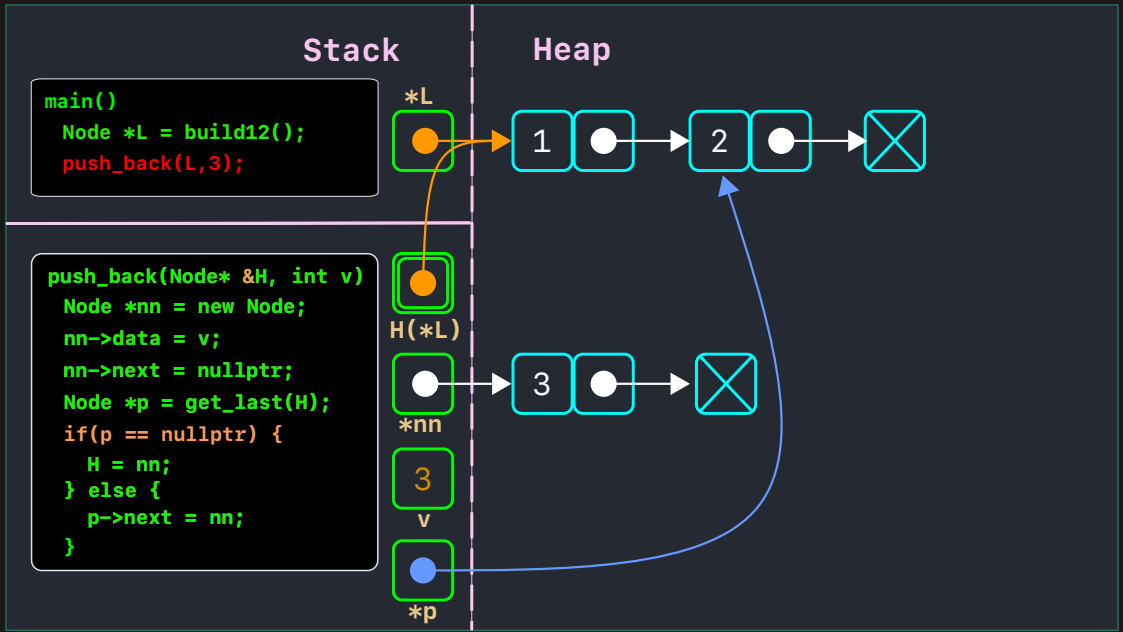

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
?
?
*nn
3
v

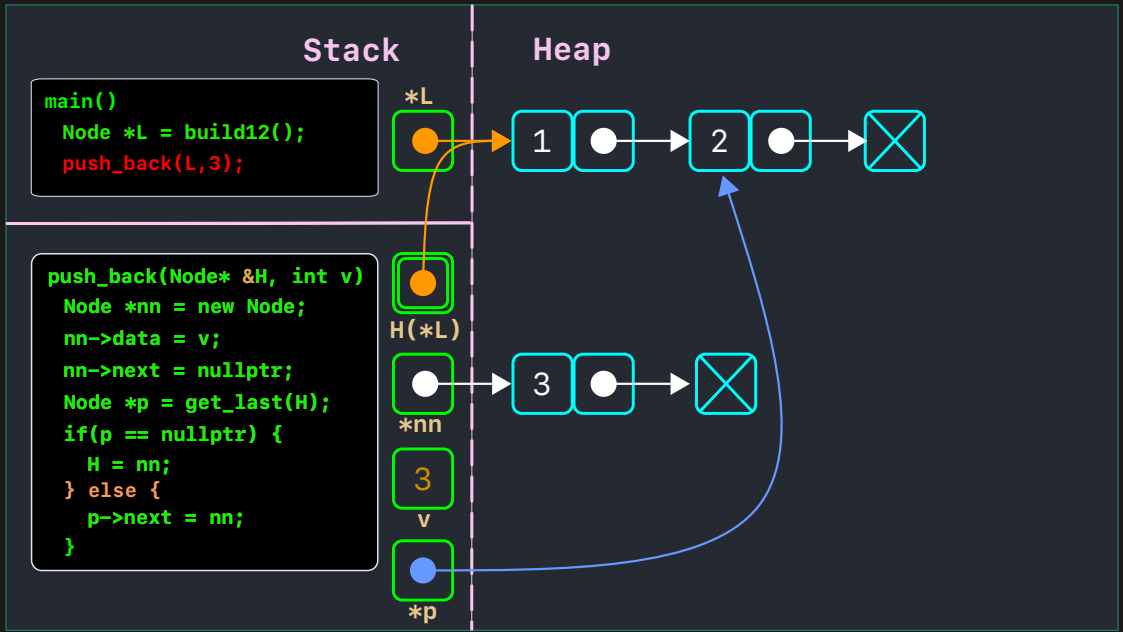

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
?
*nn
3
v

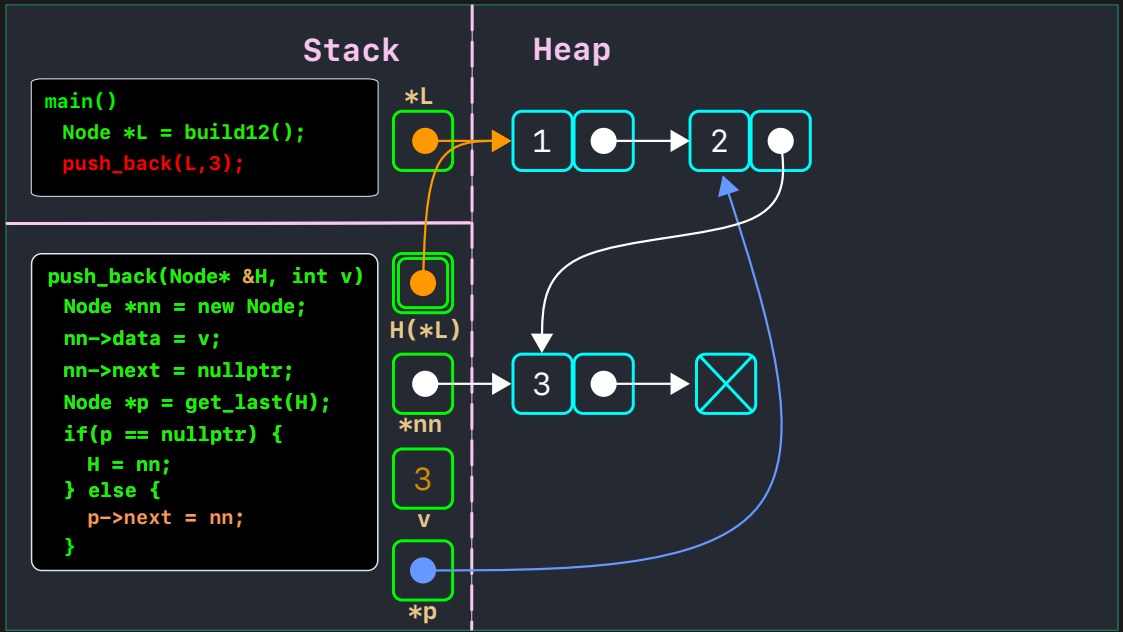

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
*nn
3
v

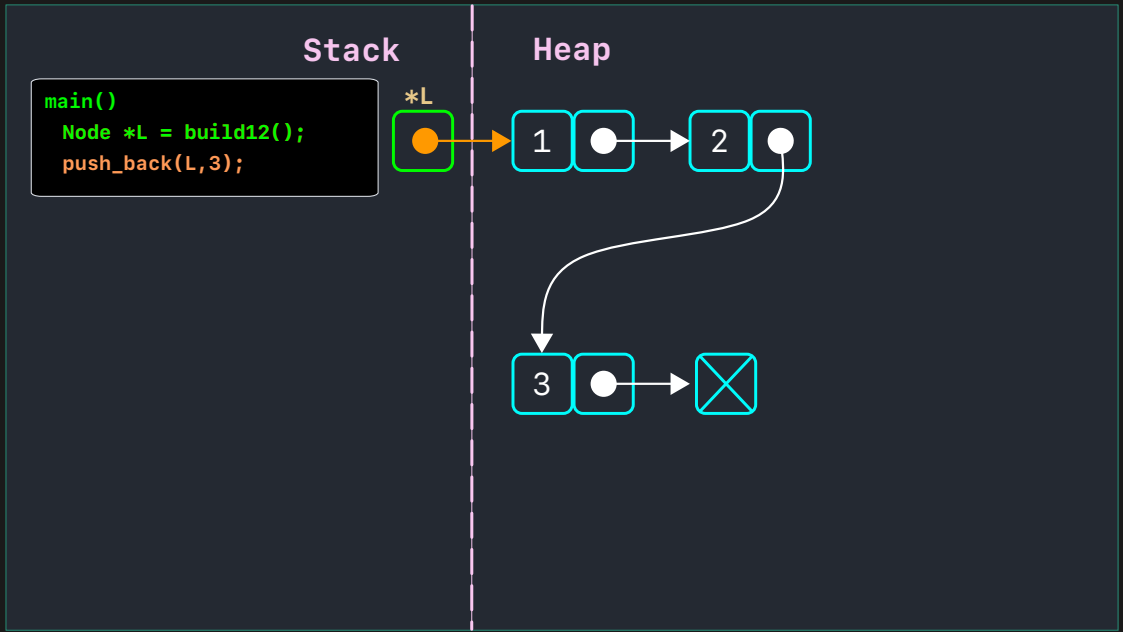

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
*nn
3
v
*p

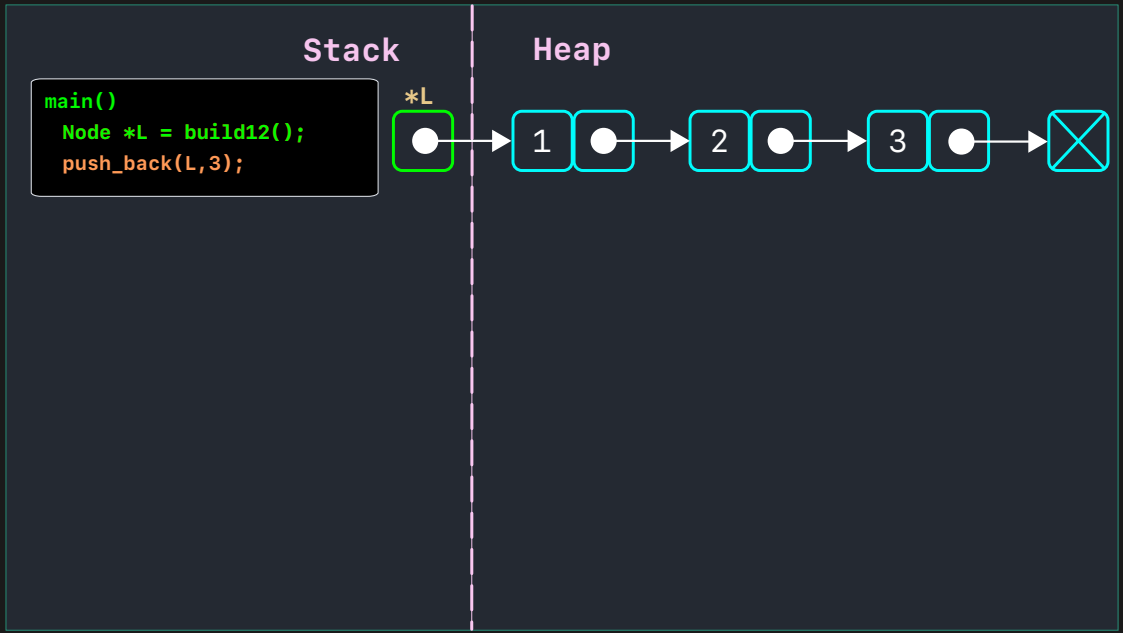

Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
*nn
3
v
*p


Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
*nn
3
v
*p


Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
*nn
3
v
*p


Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
3
*nn
3
v
*p


Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
3


Stack
Heap
main()
Node *L = build12();
push_back(L,3);
*L
1
2
3

Inserindo no Final da Lista

Será que o algoritmo apresentado funciona para lista vazia?

```
push_back(Node* &H, int v) {
  Node *nn = new Node;
  nn->data = v;
  nn->next = nullptr;
  Node *p = get_last(H);
  if(p == nullptr)   H = nn;
  else               p->next = nn;
}
```

Stack

Heap

```
main()
  Node *L = nullptr;
  push_back(L,1);
```

Stack

```
main()
  Node *L = nullptr;
  push_back(L,1);
```

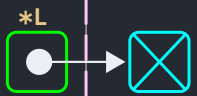
*L

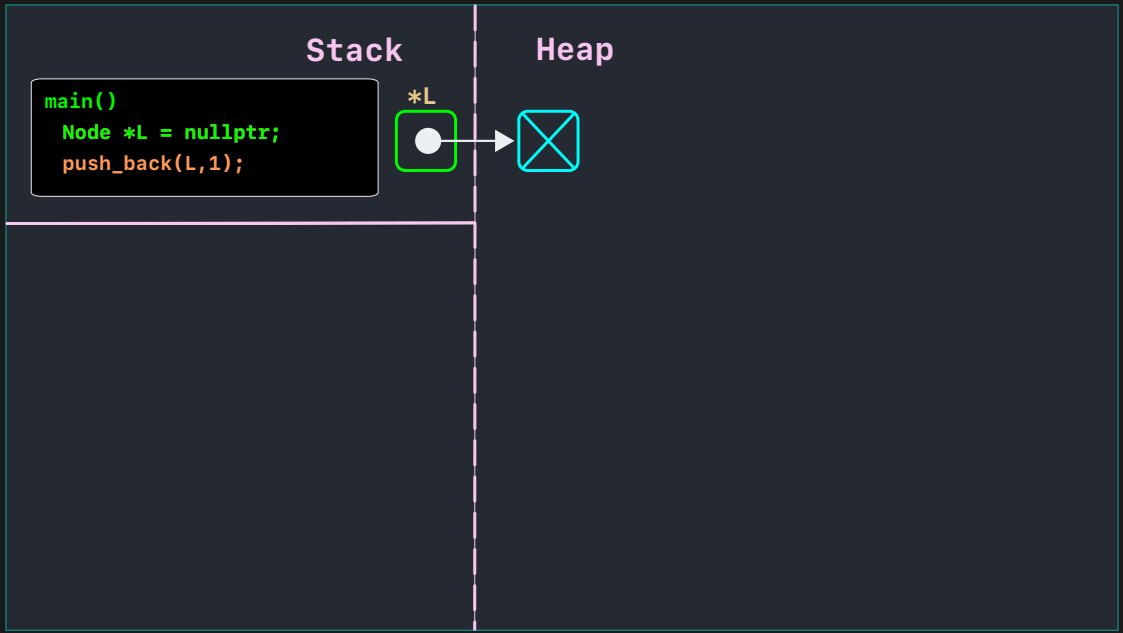
Heap

Stack

Heap

```
main()
  Node *L = nullptr;
  push_back(L,1);
```

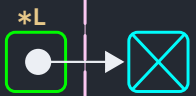



Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L

Stack

Heap

```
main()
  Node *L = nullptr;
  push_back(L,1);
```

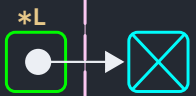



```
push_back(Node* &H, int v)
  Node *nn = new Node;
  nn->data = v;
  nn->next = nullptr;
  Node *p = get_last(H);
  if(p == nullptr) {
    H = nn;
  } else {
    p->next = nn;
  }
```

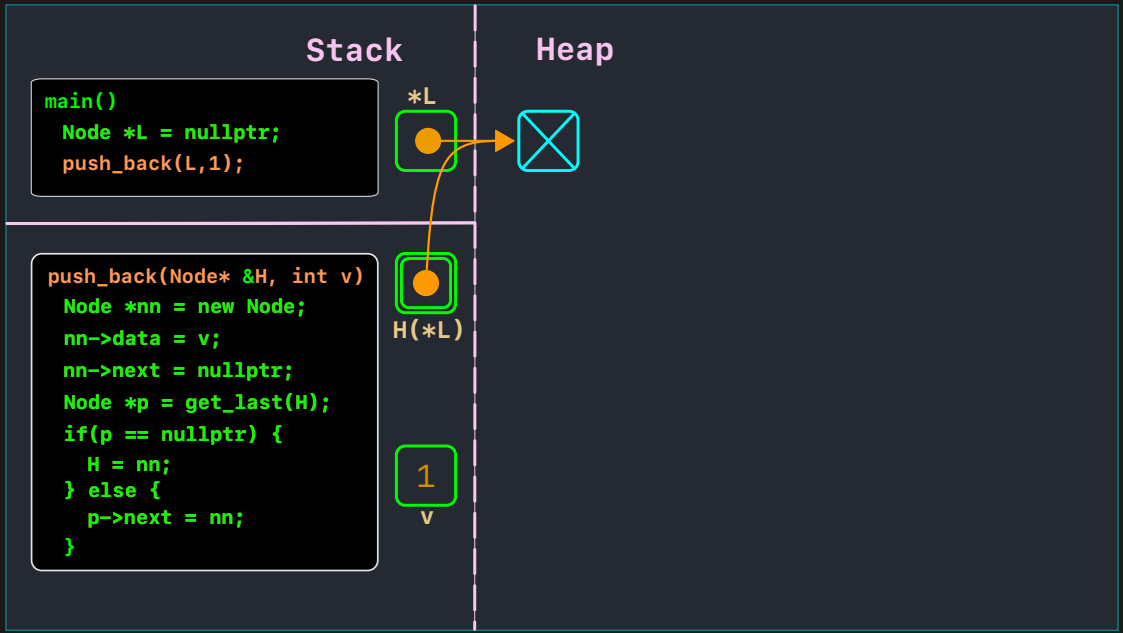
Stack

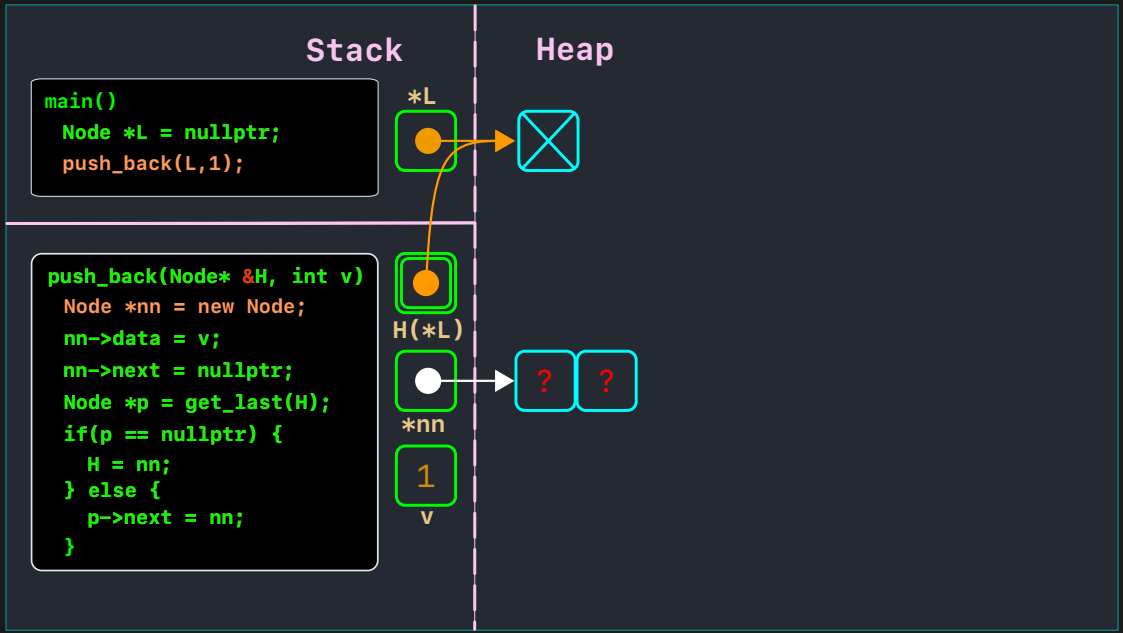
Heap

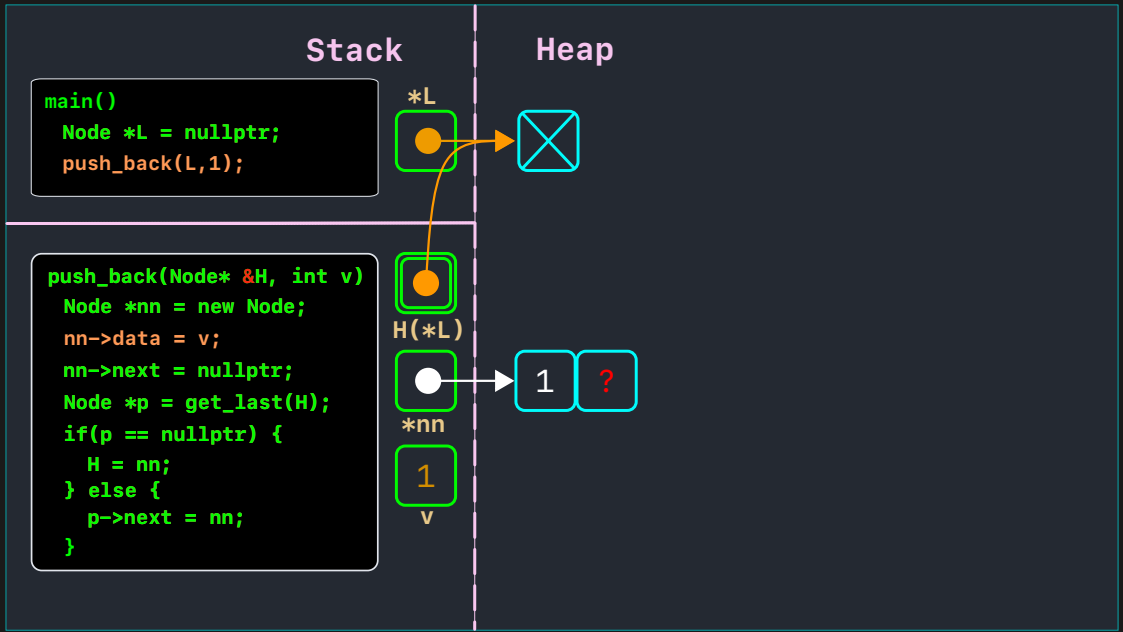
```
main()
  Node *L = nullptr;
  push_back(L,1);
```

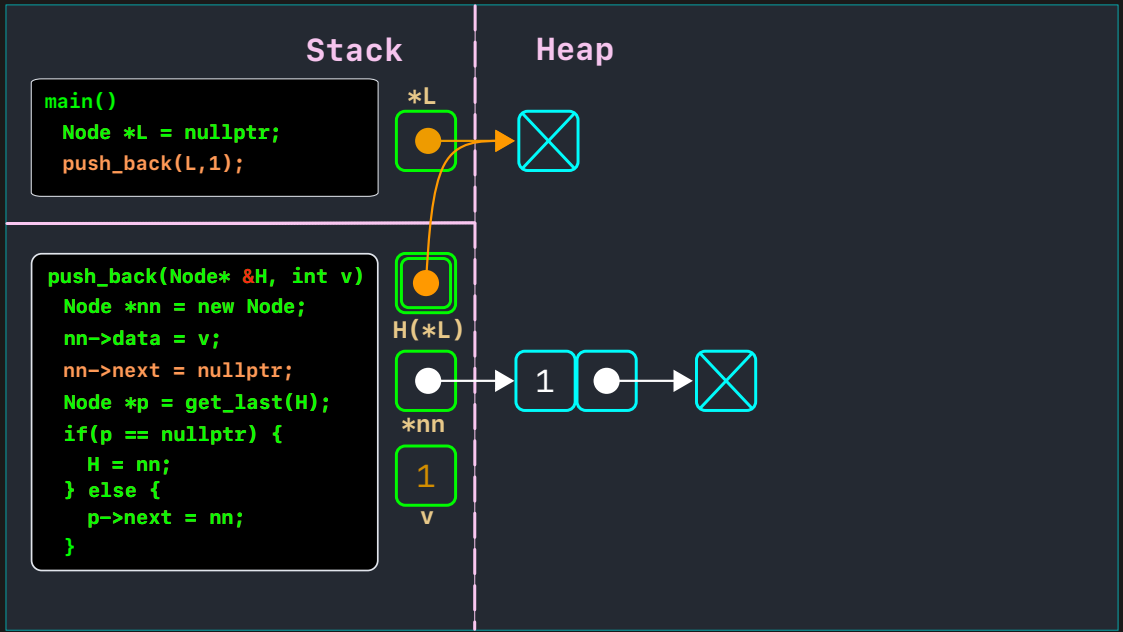


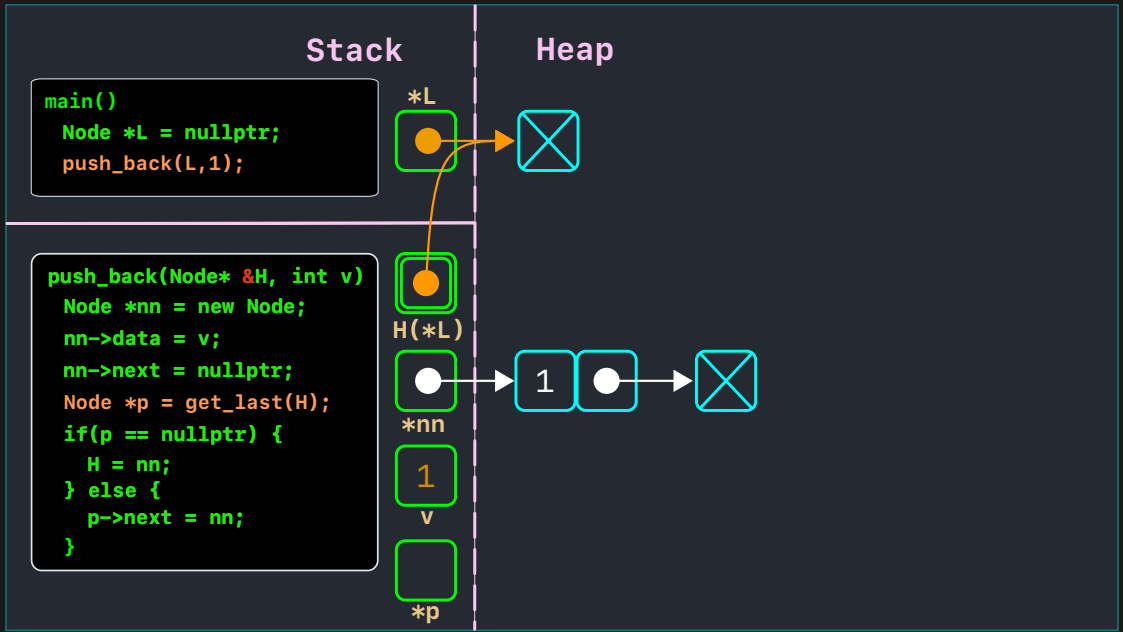

```
push_back(Node* &H, int v)
  Node *nn = new Node;
  nn->data = v;
  nn->next = nullptr;
  Node *p = get_last(H);
  if(p == nullptr) {
    H = nn;
  } else {
    p->next = nn;
  }
```

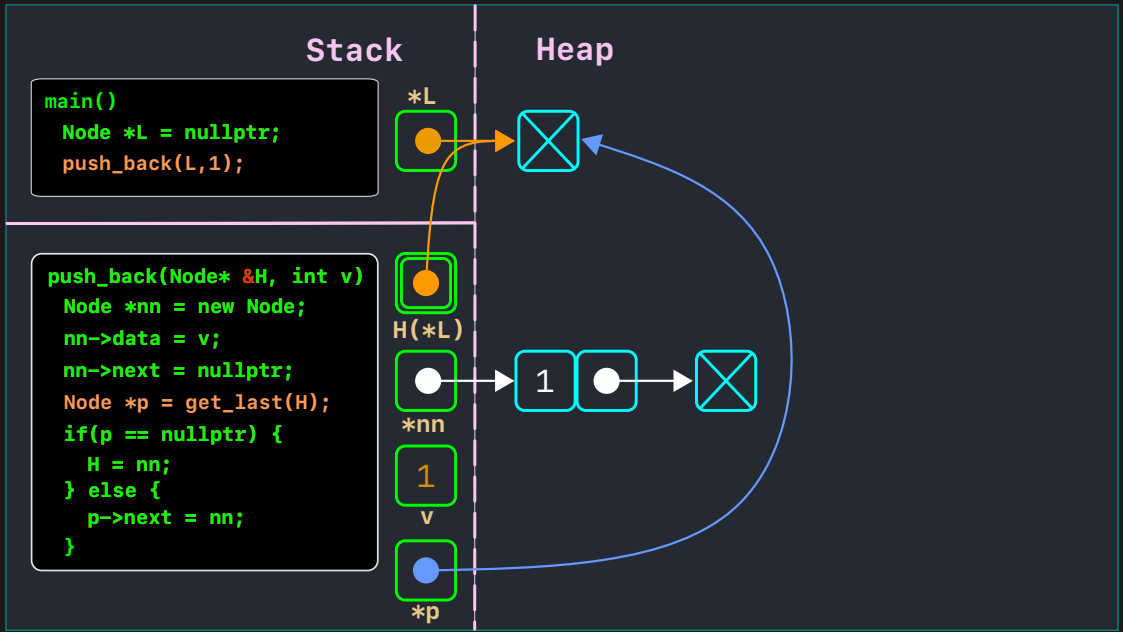

Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
v

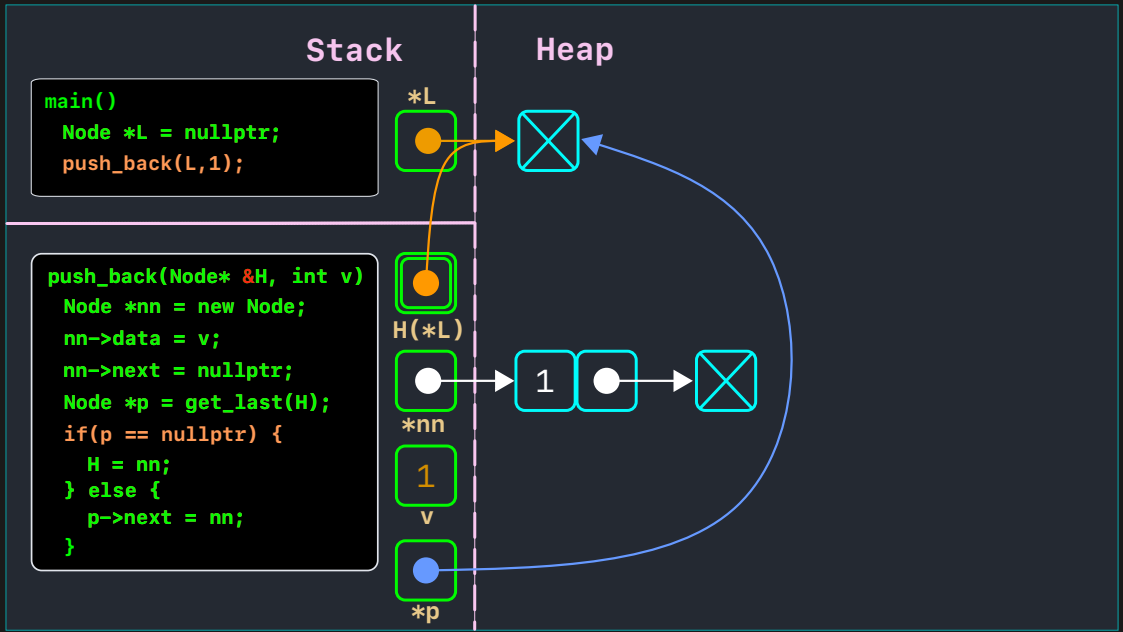

Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
?
?
*nn
1
v

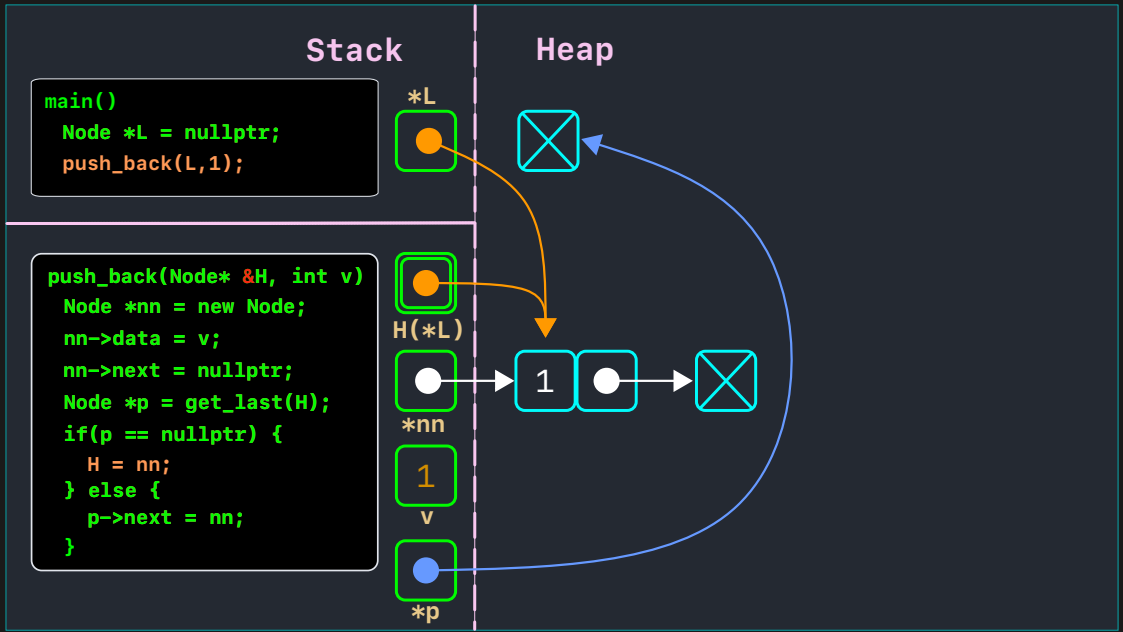

Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
?
*nn
1
v

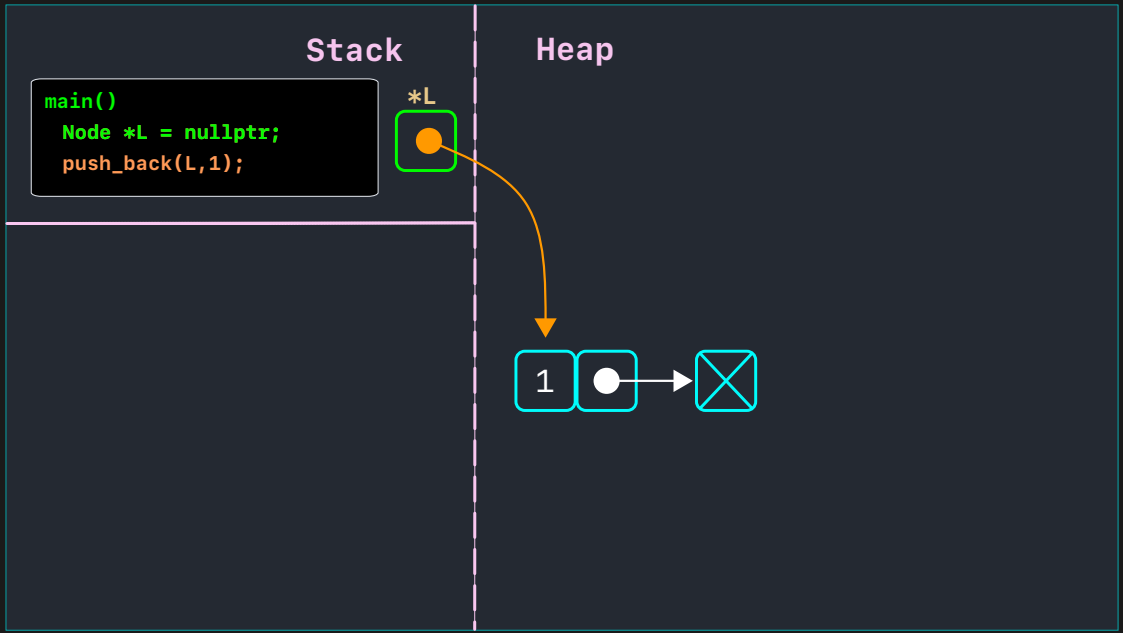

Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
*nn
1
v

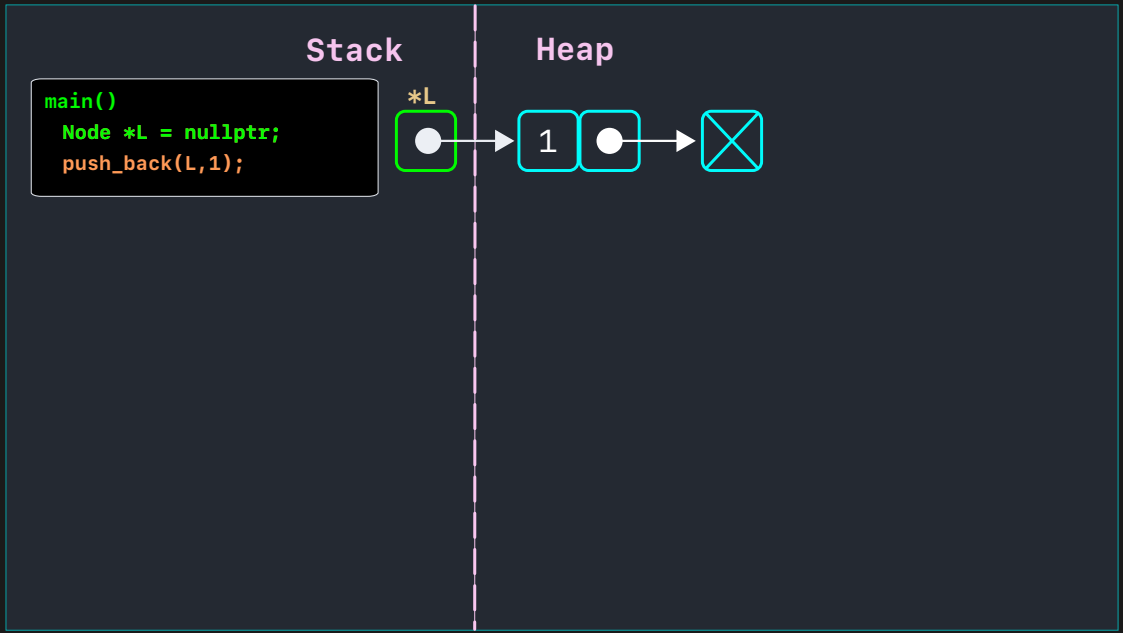

Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
*nn
1
v
*p


Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
*nn
1
v
*p


Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
*nn
1
v
*p


Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
push_back(Node* &H, int v)
Node *nn = new Node;
nn->data = v;
nn->next = nullptr;
Node *p = get_last(H);
if(p == nullptr) {
H = nn;
} else {
p->next = nn;
}
H(*L)
1
*nn
1
v
*p


Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
1


Stack
Heap
main()
Node *L = nullptr;
push_back(L,1);
*L
1

Outras Operações sobre Lista

Nos próximos slides vamos abordar

- Inserção no meio da lista;

Outras Operações sobre Lista

Nos próximos slides vamos abordar

- Inserção no meio da lista;
- Remoção de elementos da lista;

Outras Operações sobre Lista

Nos próximos slides vamos abordar

- Inserção no meio da lista;
- Remoção de elementos da lista;
- Busca e alteração de informação na lista, e;

Outras Operações sobre Lista

Nos próximos slides vamos abordar

- Inserção no meio da lista;
- Remoção de elementos da lista;
- Busca e alteração de informação na lista, e;
- Variações de lista, como nó cabeça e lista duplamente encadeada.

Referências

Nick Parlante.
Pointers and Memory, Document #102.
Computer Science Education Library, Stanford University.
`http://cslibrary.stanford.edu/102`

Nick Parlante.
Linked List Basics, Document #103.
Computer Science Education Library, Stanford University.
`http://cslibrary.stanford.edu/103`