Introduction to Programming

Week – *8*, Lecture – *3*

Stacks and Queues

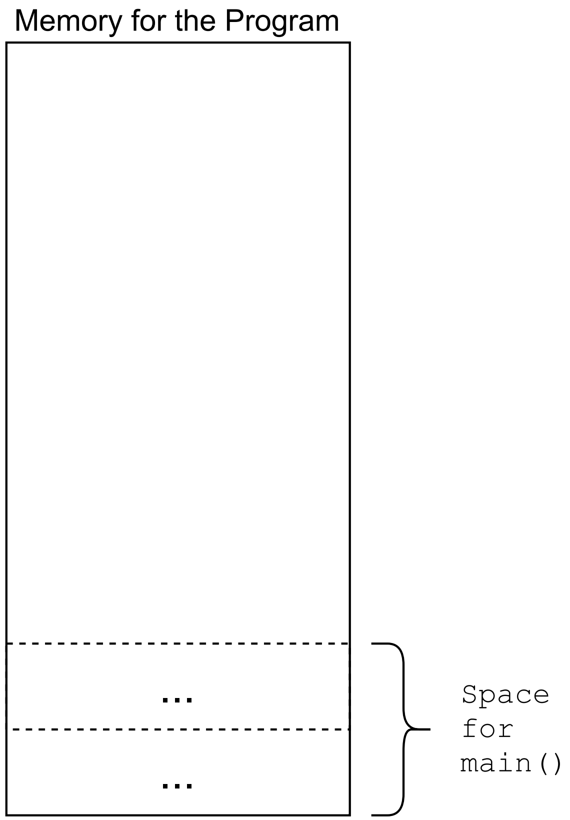
SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Example: How does a program execute?

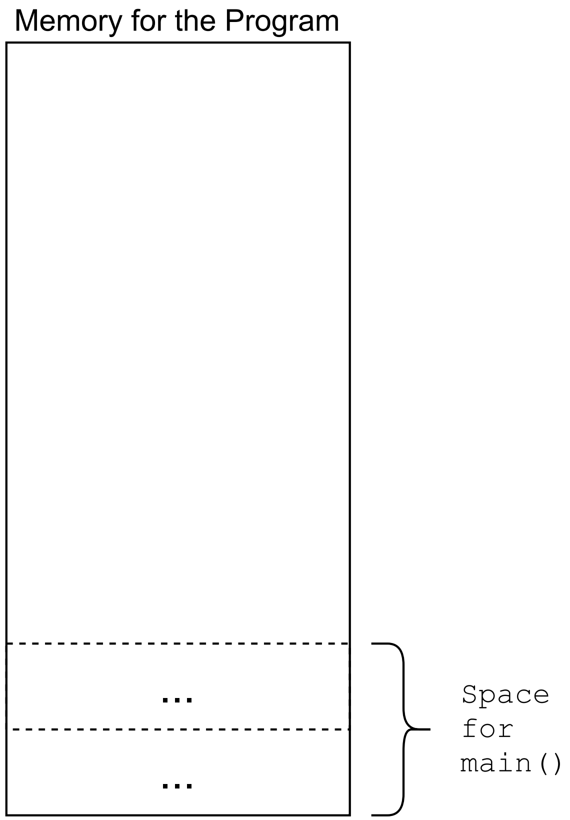
```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```



Assume that we are trying to execute this simple C program

Example: How does a program execute?

```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```



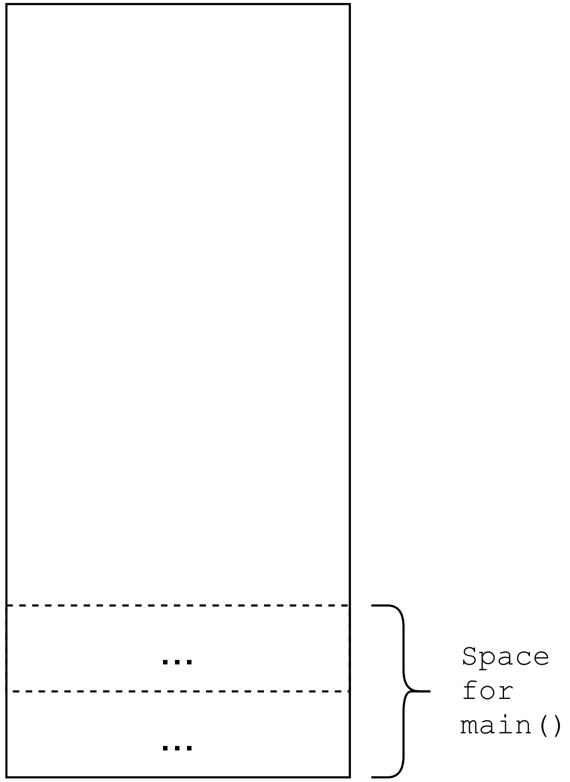
Assume that we are trying to execute this simple C program

It has four functions – `main()`, `f1()`, `f2()` and `f3()`

Example: How does a program execute?

```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```

Memory for the Program



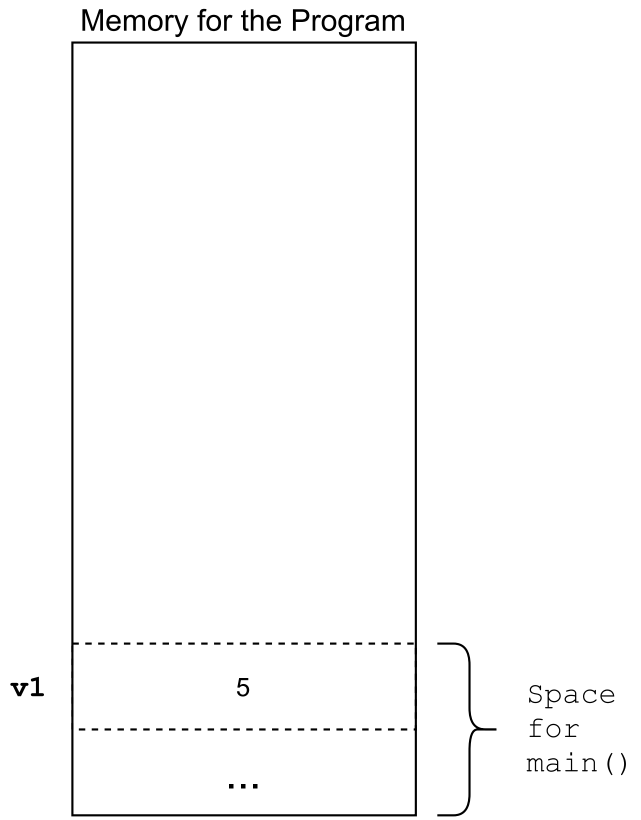
Assume that we are trying to execute this simple C program

It has four functions – `main()`, `f1()`, `f2()` and `f3()`

The execution starts from the first statement in `main()`

Example: How does a program execute?

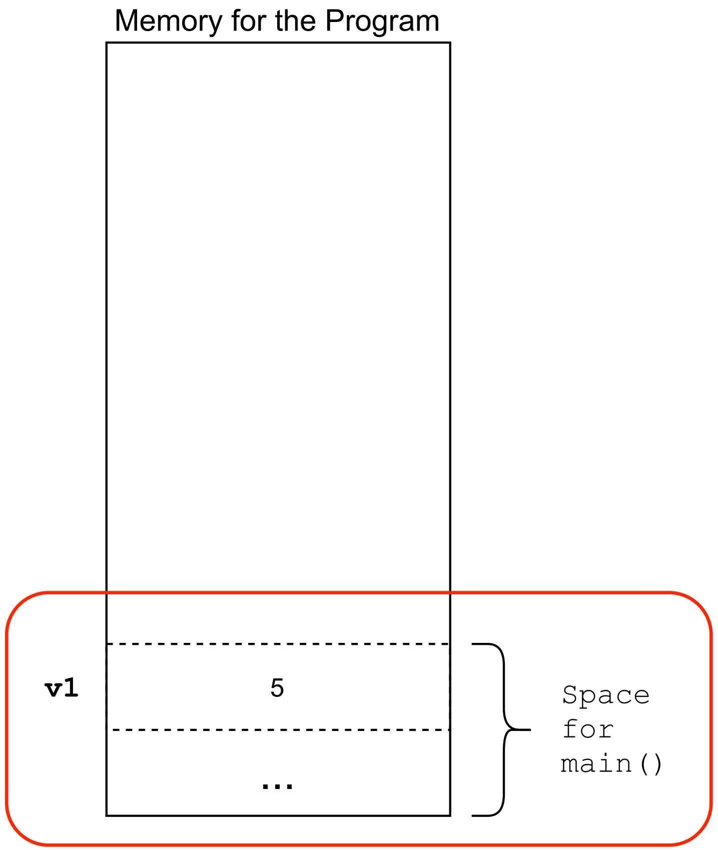
```
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```



A variable called `v1` is allocated in some "designated" space in the Main Memory

Example: How does a program execute?

```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```

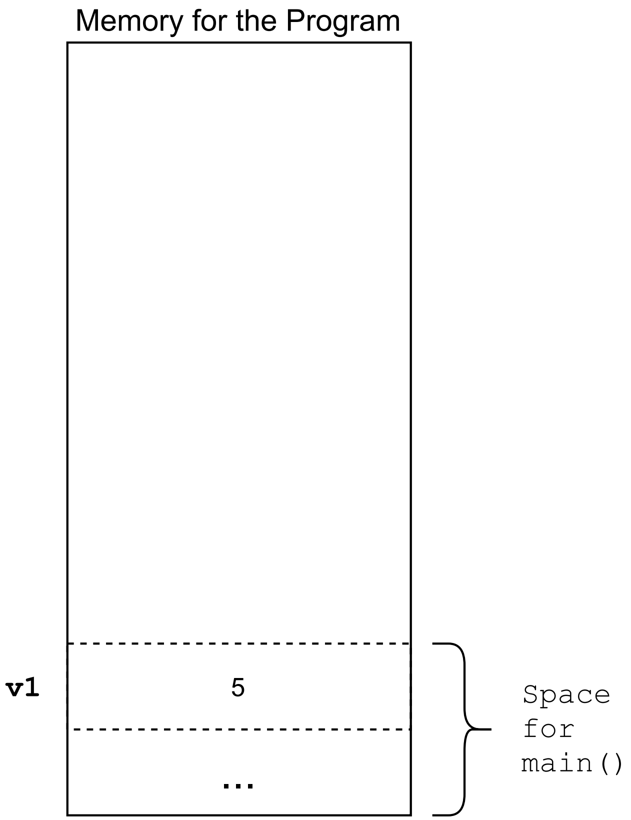


A variable called `v1` is allocated in some "designated" space in the Main Memory

This memory block is reserved for executing the `main()` function only

Example: How does a program execute?

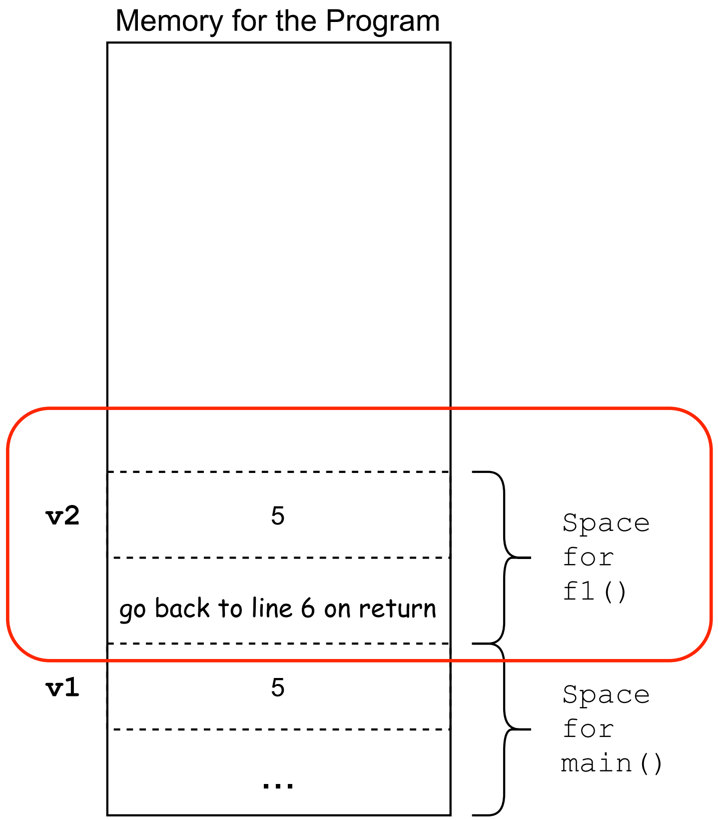
```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```



The next statement in `main()` is a call to the function `f1()`

Example: How does a program execute?

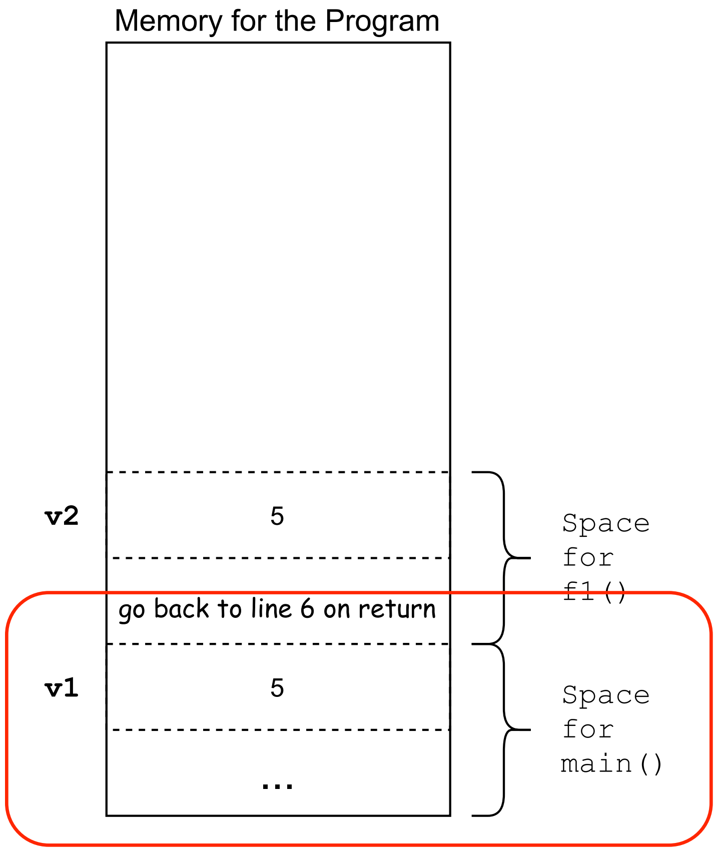
```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



At this point, some space in the memory is allocated for executing `f1()`

Example: How does a program execute?

```
1  #include<stdio.h>
2  int main()
3  {
4      int v1 = 5;
5      f1(v1);
6      return 0;
7  }
8  void f1(int v2)
9  {
10     f2();
11     f3(v2);
12     return;
13 }
14 void f2()
15 {
16     printf("Hey !!");
17     return;
18 }
19 void f3(int v3)
20 {
21     printf("%d", v3);
22     return;
23 }
```

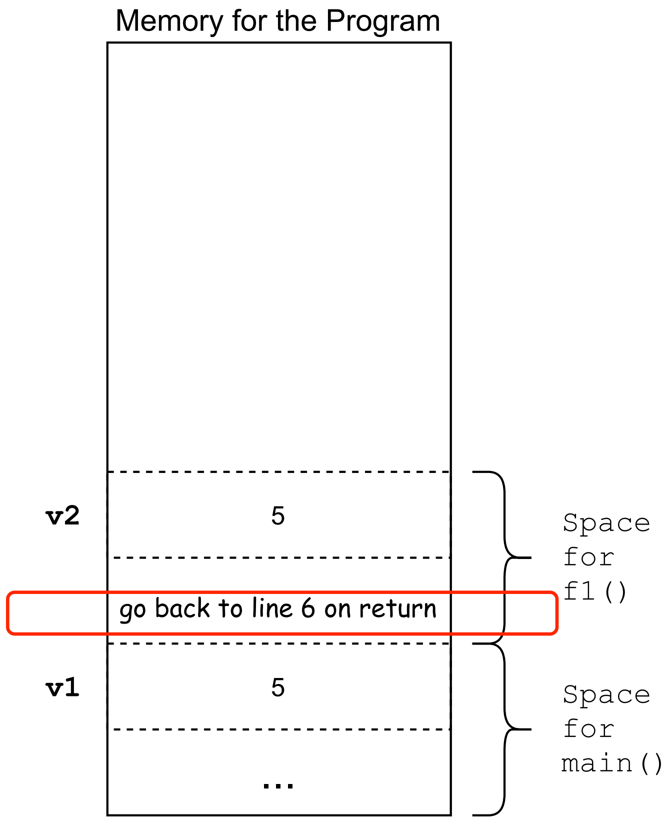


At this point, some space in the memory is allocated for executing `f1()`

The memory block for `main()` is also in the memory, but it is separated from this block

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



At this point, some space in the memory is allocated for executing `f1()`

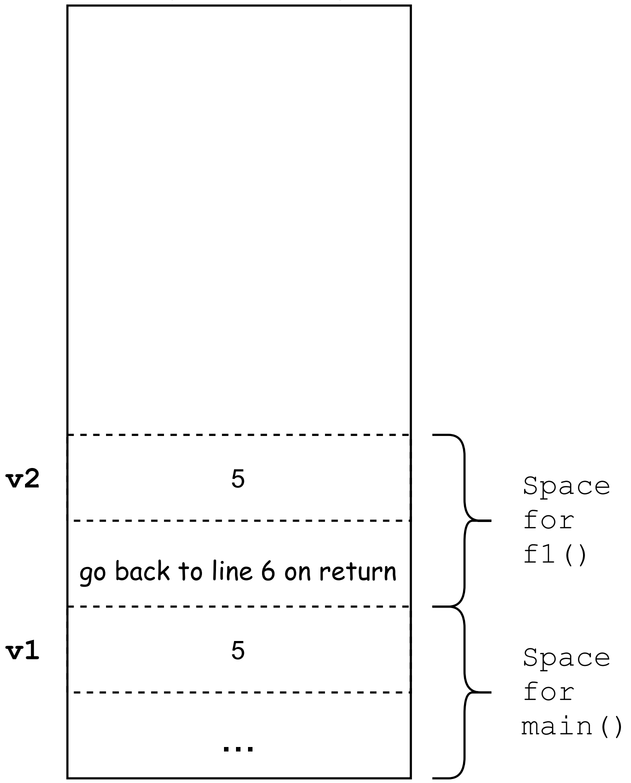
The memory block for `main()` is also in the memory, but it is separated from this block

Some information is saved in the block, for resumption of `main()` on return of `f1()`

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```

Memory for the Program



At this point, some space in the memory is allocated for executing `f1()`

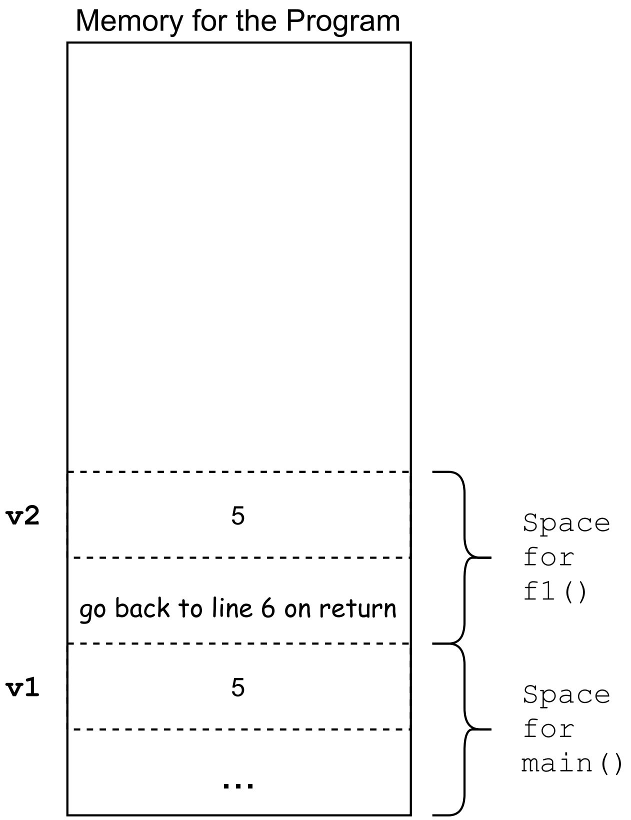
The memory block for `main()` is also in the memory, but it is separated from this block

Some information is saved in the block, for resumption of `main()` on return of `f1()`

Local variable `v2`, which is a formal parameter for `f1()`, gets allocated in this block

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



At this point, some space in the memory is allocated for executing `f1()`

The memory block for `main()` is also in the memory, but it is separated from this block

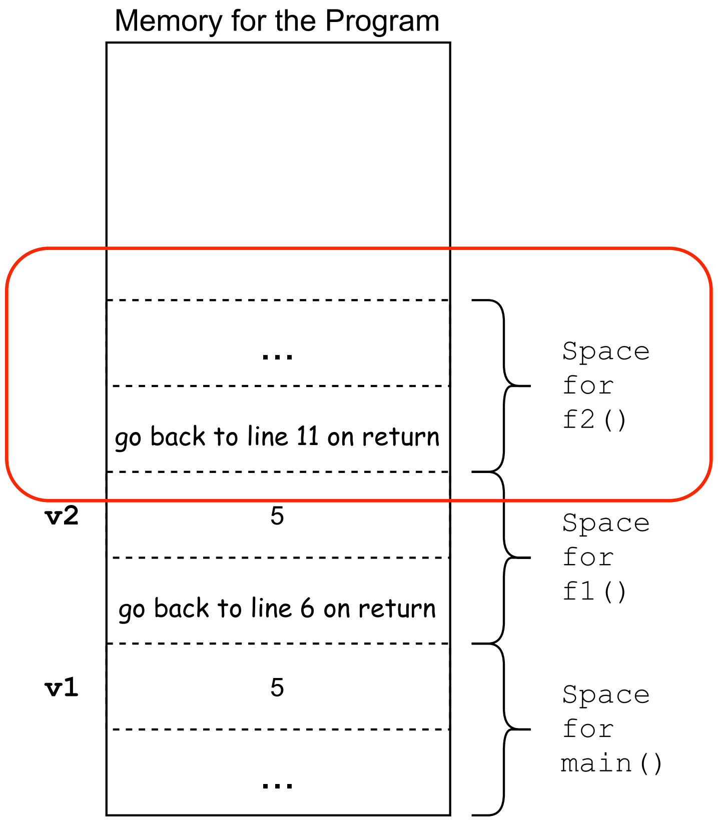
Some information is saved in the block, for resumption of `main()` on return of `f1()`

Local variable `v2`, which is a formal parameter for `f1()`, gets allocated in this block

The first statement in `f1()` is a call to the function `f2()`

Example: How does a program execute?

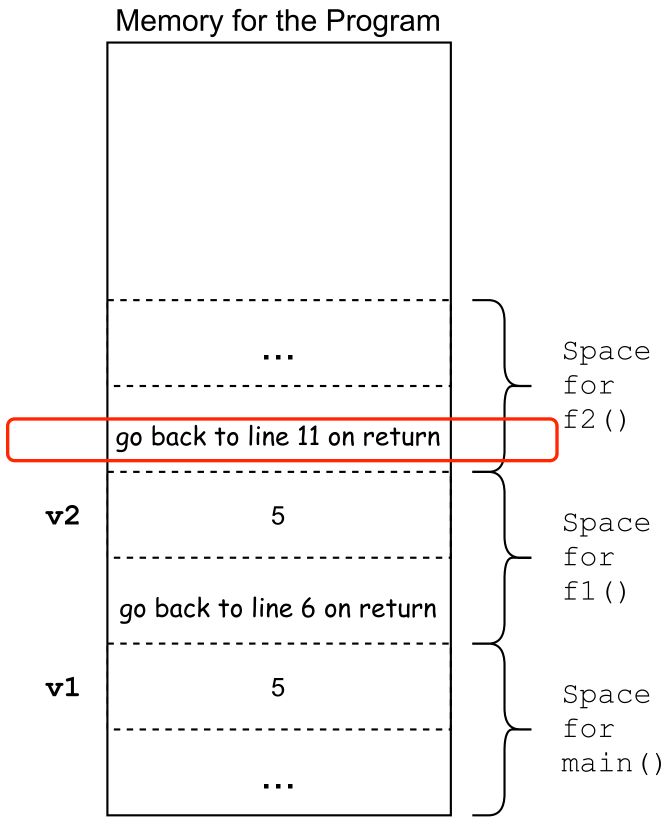
```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



Again, another memory block is allocated, for executing `f2()` …

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```

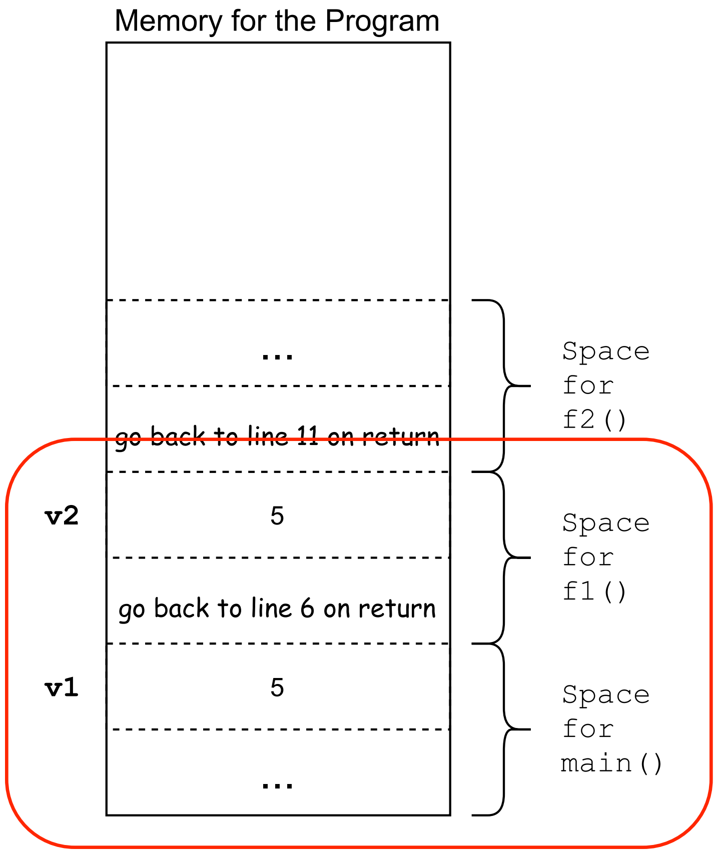


Again, another memory block is allocated, for executing `f2()` …

… and some information to return the control back to `f1()` is also stored

Example: How does a program execute?

```
1  #include<stdio.h>
2  int main()
3  {
4      int v1 = 5;
5      f1(v1);
6      return 0;
7  }
8  void f1(int v2)
9  {
10     f2();
11     f3(v2);
12     return;
13 }
14 void f2()
15 {
16     printf("Hey !!");
17     return;
18 }
19 void f3(int v3)
20 {
21     printf("%d", v3);
22     return;
23 }
```



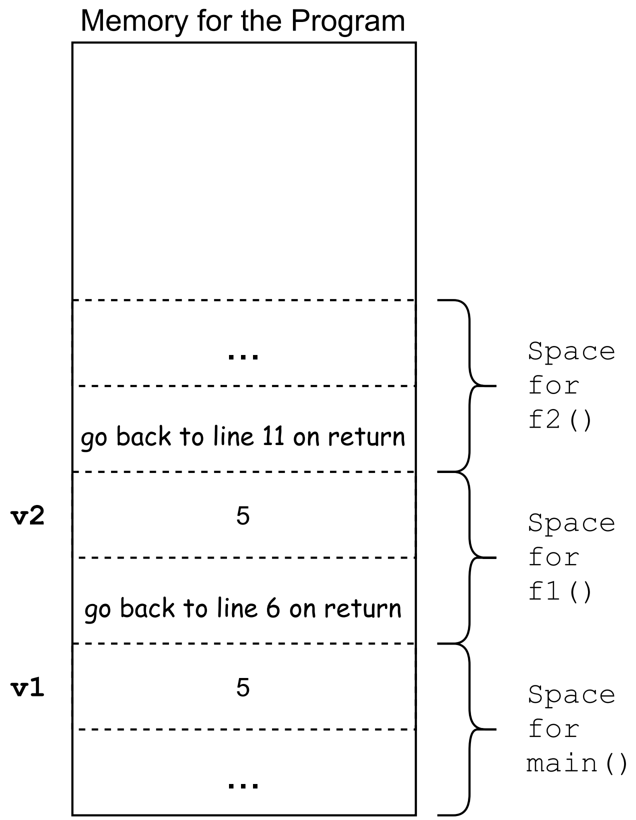
Again, another memory block is allocated, for executing `f2()` …

… and some information to return the control back to `f1()` is also stored

The blocks for `f1()` and `main()` remain intact

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



Again, another memory block is allocated, for executing `f2()` …

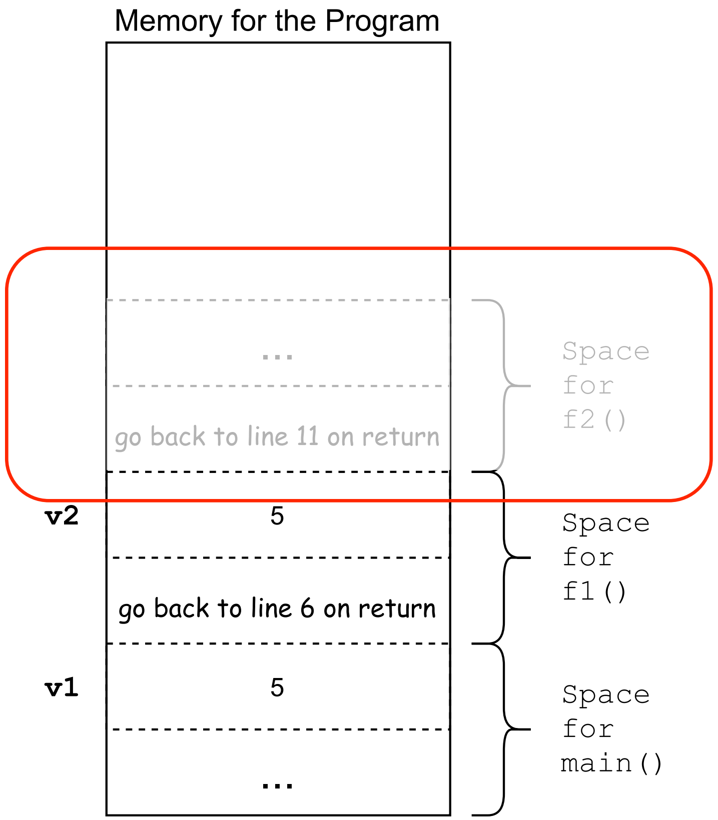
… and some information to return the control back to `f1()` is also stored

The blocks for `f1()` and `main()` remain intact

The `printf()` statement executes, and since there are no other statements, `f2()` returns

Example: How does a program execute?

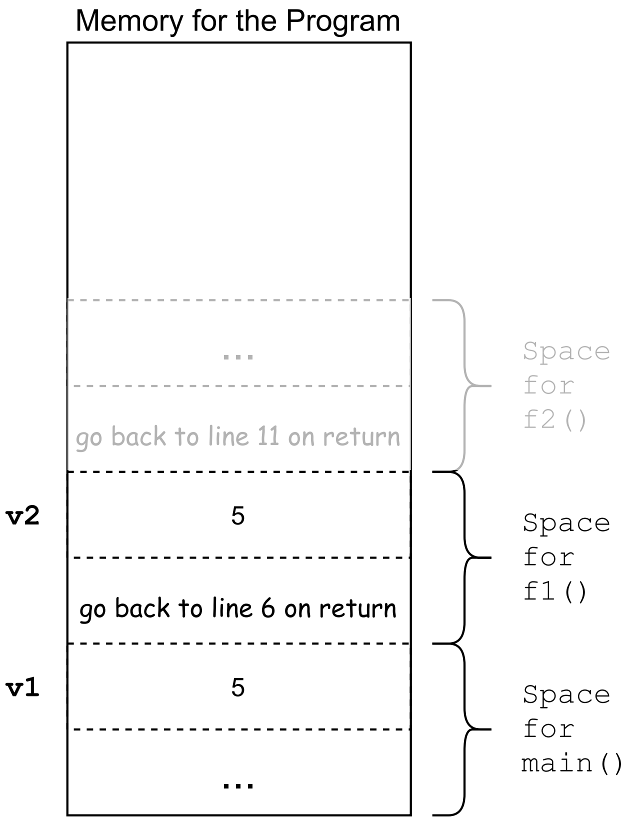
```c
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



As soon as the return statement executes, the block for `f2()` gets freed

Example: How does a program execute?

```
1  #include<stdio.h>
2  int main()
3  {
4      int v1 = 5;
5      f1(v1);
6      return 0;
7  }
8  void f1(int v2)
9  {
10     f2();
11     f3(v2);
12     return;
13 }
14 void f2()
15 {
16     printf("Hey !!");
17     return;
18 }
19 void f3(int v3)
20 {
21     printf("%d", v3);
22     return;
23 }
```



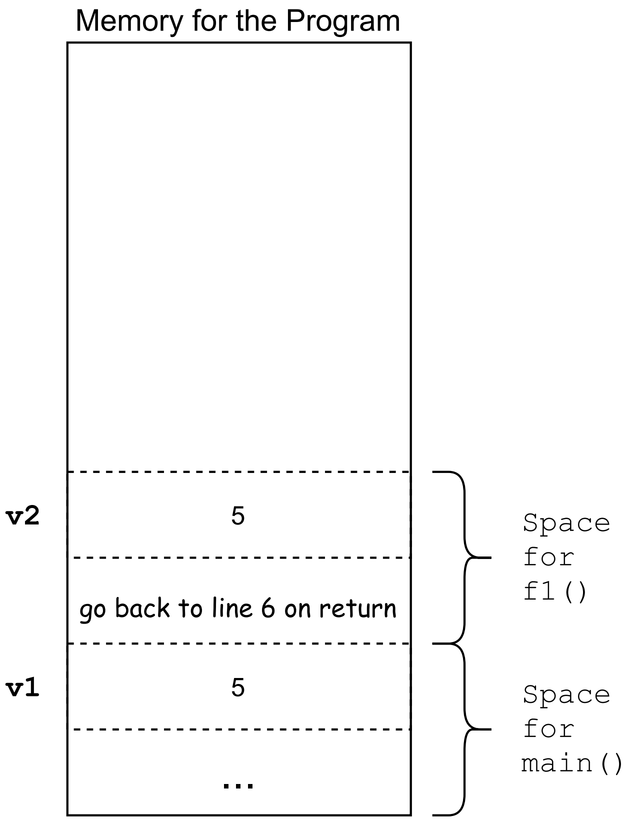
As soon as the return statement executes, the block for `f2()` gets freed

The return information is used at this point, to go back to the correct position in code

SAURABH SRIVASTAVA | DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING | IIT KANPUR

Example: How does a program execute?

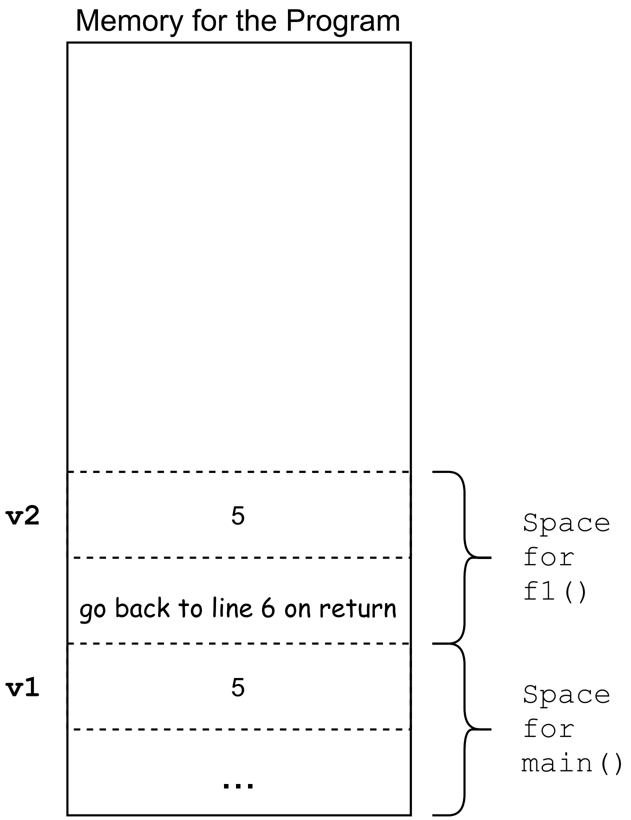
```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



The control now returns to `f1()`

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```

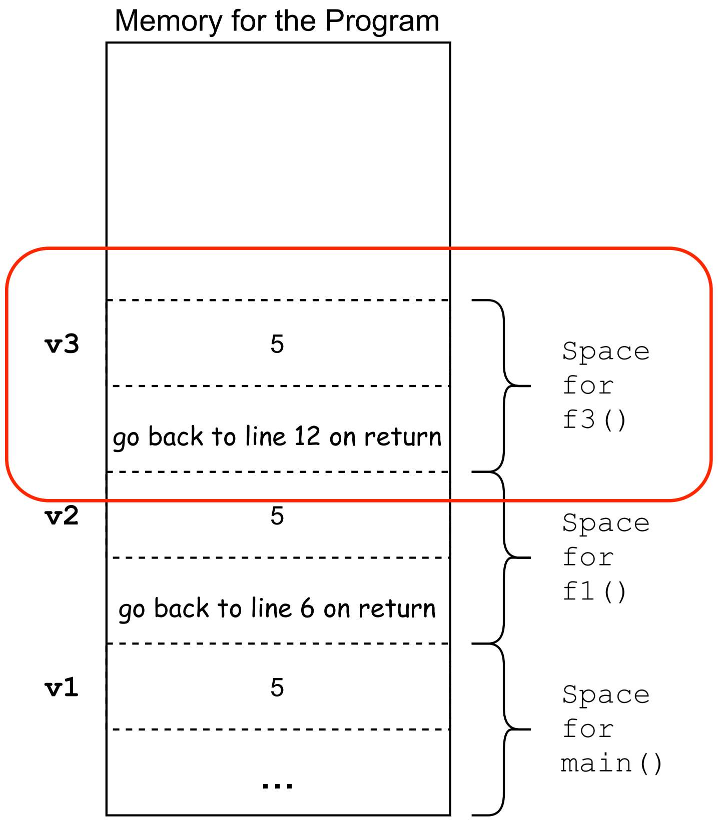


The control now returns to `f1()`

The next statement now, is a call to `f3()`

Example: How does a program execute?

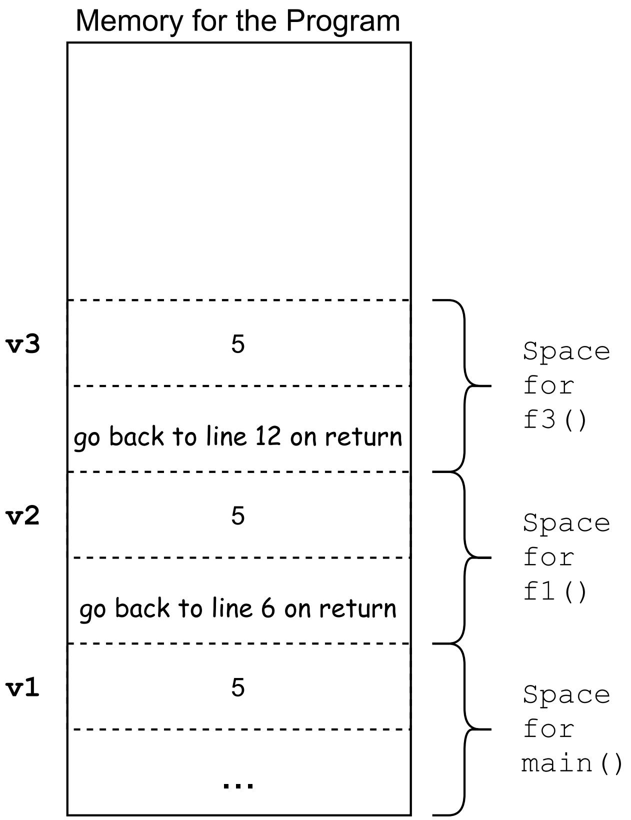
```c
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



Another memory block gets allocated, this time for executing `f3()`

Example: How does a program execute?

```
1  #include<stdio.h>
2  int main()
3  {
4      int v1 = 5;
5      f1(v1);
6      return 0;
7  }
8  void f1(int v2)
9  {
10     f2();
11     f3(v2);
12     return;
13 }
14 void f2()
15 {
16     printf("Hey !!");
17     return;
18 }
19 void f3(int v3)
20 {
21     printf("%d", v3);
22     return;
23 }
```

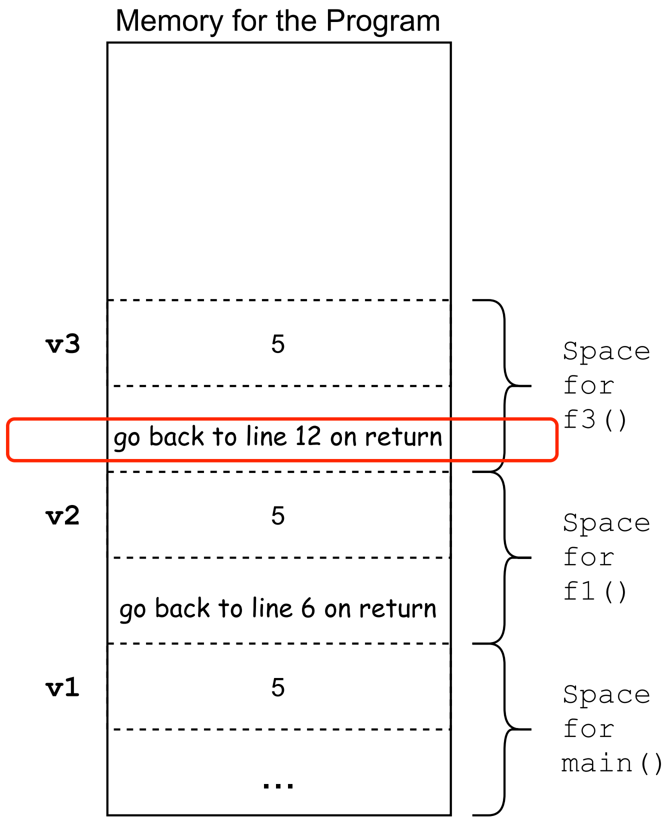


Another memory block gets allocated, this time for executing `f3()`

The formal parameter `v3`, is allocated in that block

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



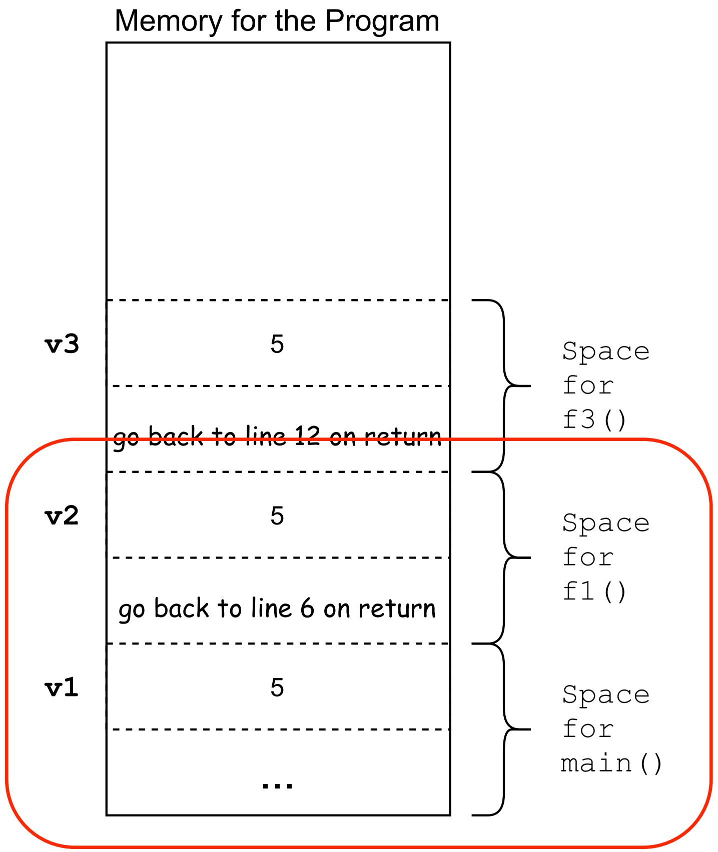
Another memory block gets allocated, this time for executing `f3()`

The formal parameter `v3`, is allocated in that block

The return address information is also stored

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



Another memory block gets allocated, this time for executing `f3()`

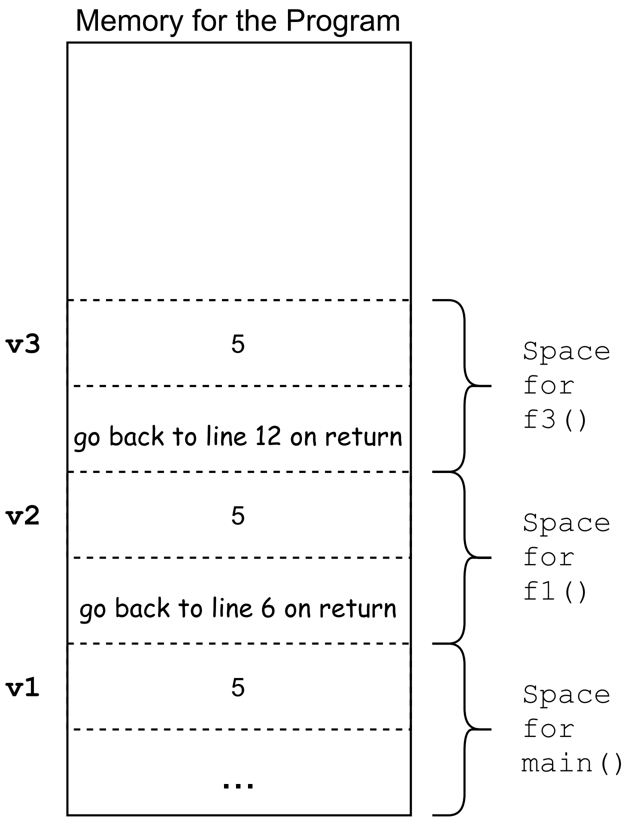
The formal parameter `v3`, is allocated in that block

The return address information is also stored

The blocks for `f1()` and `main()` still remain intact

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



Another memory block gets allocated, this time for executing `f3()`

The formal parameter `v3`, is allocated in that block

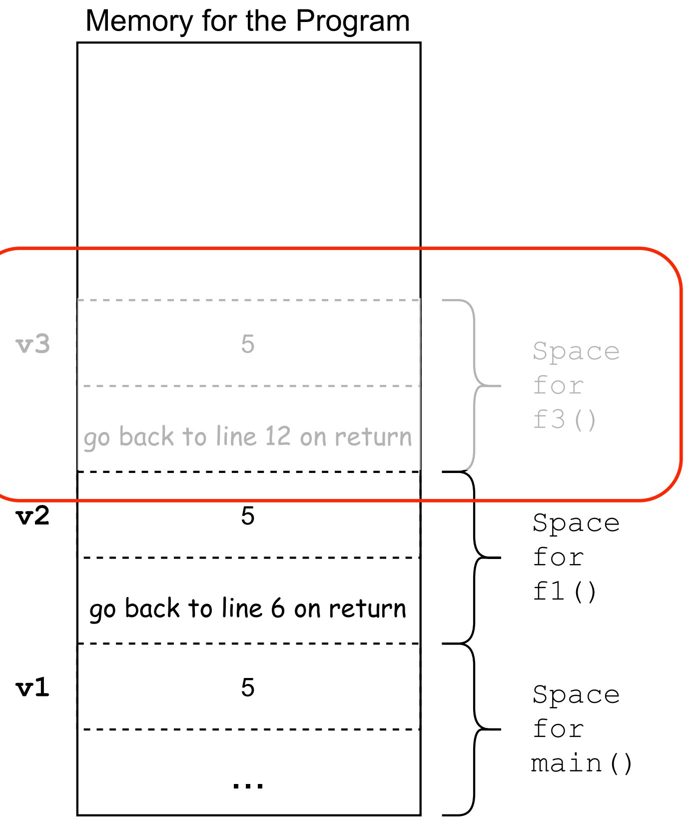
The return address information is also stored

The blocks for `f1()` and `main()` still remain intact

The `printf()` statement executes, and since there are no other statements, `f3()` returns

Example: How does a program execute?

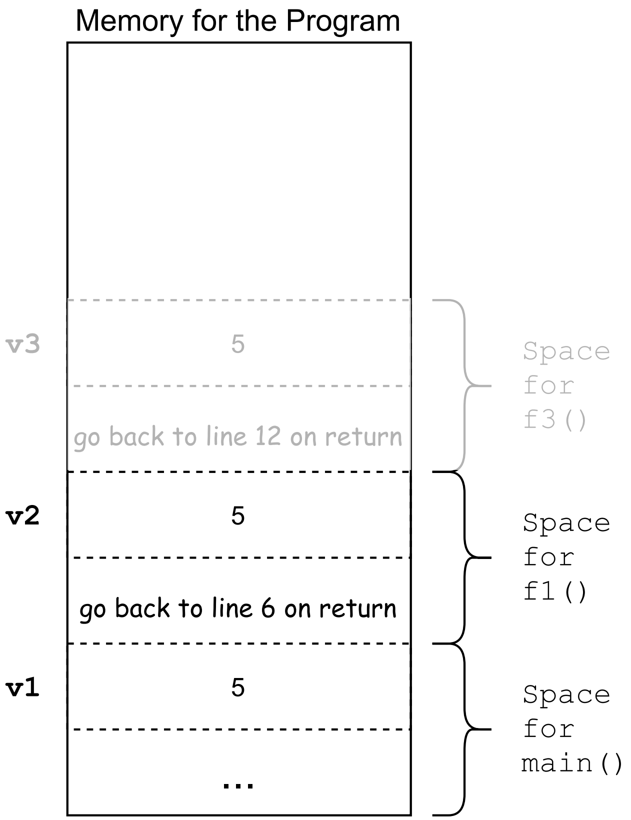
```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



The memory block for `f3()`, too, is freed when the return statement is executed

Example: How does a program execute?

```
1  #include<stdio.h>
2  int main()
3  {
4      int v1 = 5;
5      f1(v1);
6      return 0;
7  }
8  void f1(int v2)
9  {
10     f2();
11     f3(v2);
12     return;
13 }
14 void f2()
15 {
16     printf("Hey !!");
17     return;
18 }
19 void f3(int v3)
20 {
21     printf("%d", v3);
22     return;
23 }
```

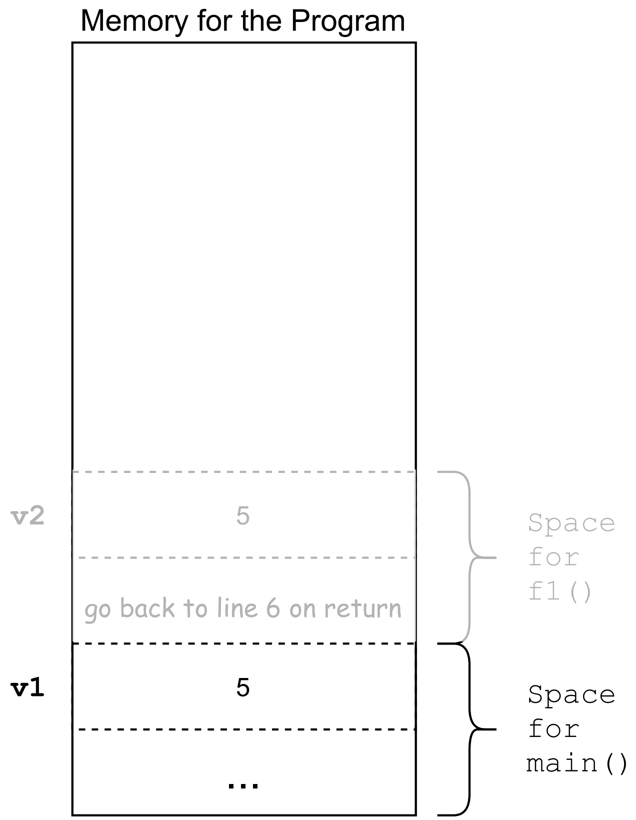


The memory block for `f3()`, too, is freed when the return statement is executed

Again, the return information helps in taking the control back to `f1()`

Example: How does a program execute?

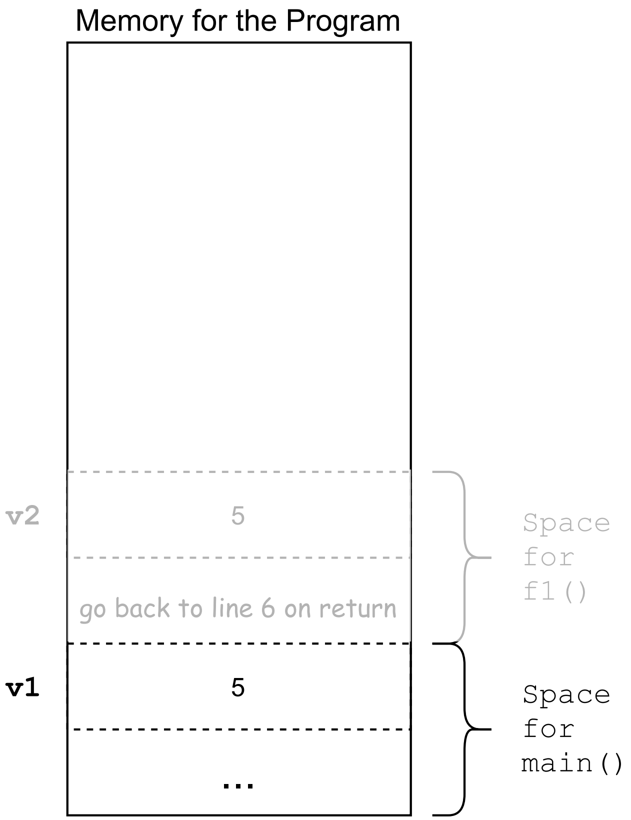
```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



Now, `f1()` too, has completed execution, and can now return

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```

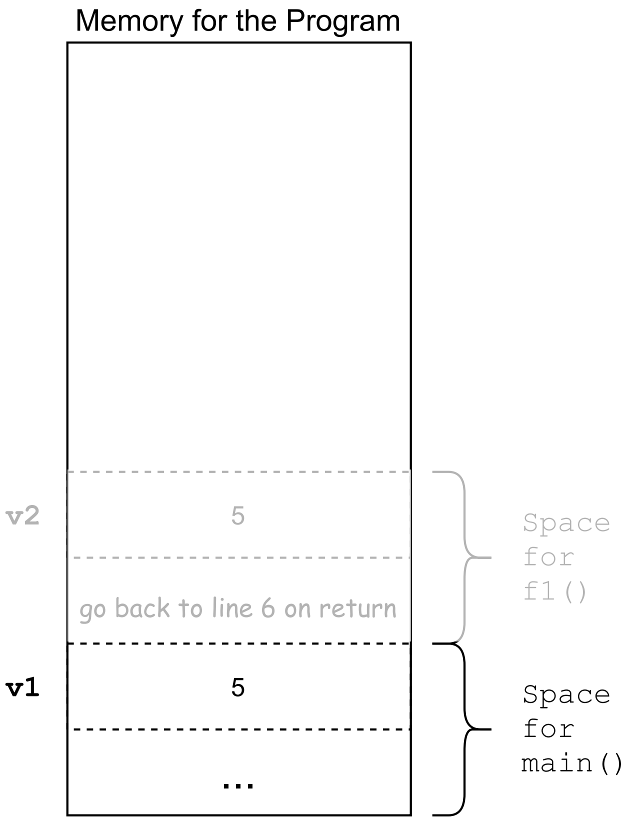


Now, `f1()` too, has completed execution, and can now return

Its memory block, too, is now freed

Example: How does a program execute?

```
1   #include<stdio.h>
2   int main()
3   {
4       int v1 = 5;
5       f1(v1);
6       return 0;
7   }
8   void f1(int v2)
9   {
10      f2();
11      f3(v2);
12      return;
13  }
14  void f2()
15  {
16      printf("Hey !!");
17      return;
18  }
19  void f3(int v3)
20  {
21      printf("%d", v3);
22      return;
23  }
```



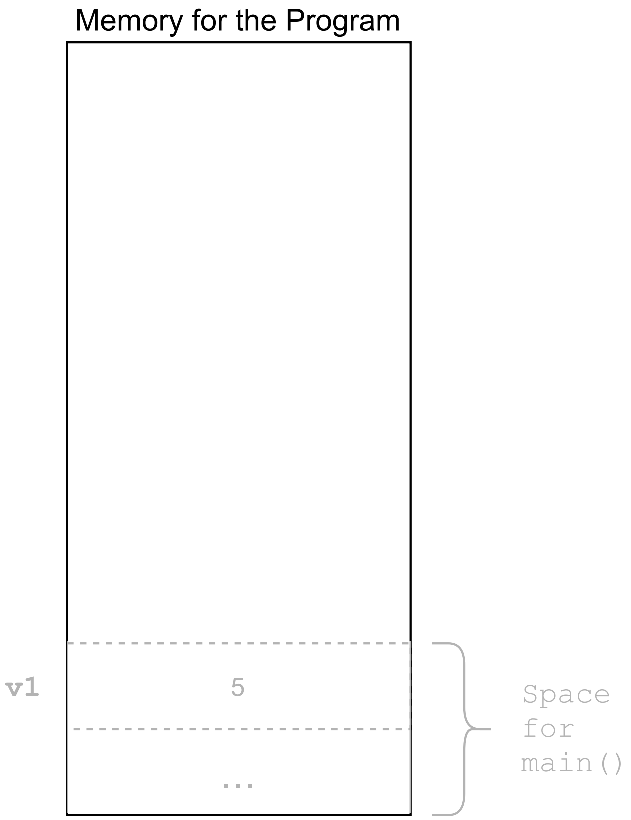
Now, `f1()` too, has completed execution, and can now return

Its memory block, too, is now freed

With the help of the return information, control can now go back to `main()`

Example: How does a program execute?

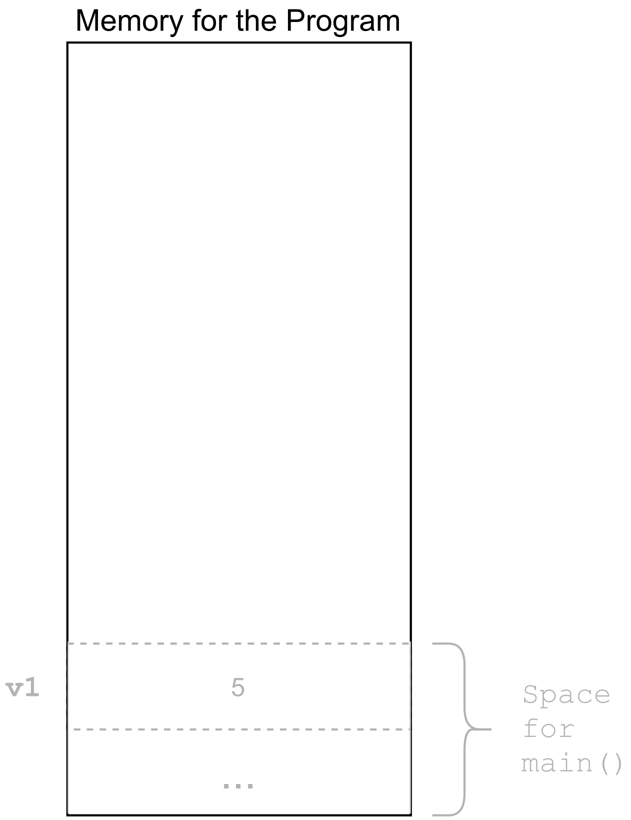
```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```



Finally, the `main()` too, completes and returns

Example: How does a program execute?

```c
#include<stdio.h>
int main()
{
    int v1 = 5;
    f1(v1);
    return 0;
}
void f1(int v2)
{
    f2();
    f3(v2);
    return;
}
void f2()
{
    printf("Hey !!");
    return;
}
void f3(int v3)
{
    printf("%d", v3);
    return;
}
```



Finally, the `main()` too, completes and returns

At this point, the execution of the program ends, and the memory allocated to the respective process is freed in totality

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

In a stack, the data that is stored last, is taken out first, and vice versa

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

In a stack, the data that is stored last, is taken out first, and vice versa

One way to implement a stack, is by using restricted insertions and deletions over an array

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

In a stack, the data that is stored last, is taken out first, and vice versa

One way to implement a stack, is by using restricted insertions and deletions over an array

- The restriction is that the values can only be added or removed from the higher end of the array

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

In a stack, the data that is stored last, is taken out first, and vice versa

One way to implement a stack, is by using restricted insertions and deletions over an array

- The restriction is that the values can only be added or removed from the higher end of the array
- We call the currently highest index of the array, where some element is present, as the *top* of the stack

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

In a stack, the data that is stored last, is taken out first, and vice versa

One way to implement a stack, is by using restricted insertions and deletions over an array

- The restriction is that the values can only be added or removed from the higher end of the array
- We call the currently highest index of the array, where some element is present, as the *top* of the stack

The operation to add an element to a stack is called *push()* and it takes the element as argument

The concept of a Stack

One aspect of the memory blocks that we saw, is that they follow a *last in first out* pattern…

- … i.e. the block for the function which was called most recently in the function call chain, is freed first

This type of organisation of the memory is called a *stack*

A stack is a type of *data structure*

- A data structure is just a way to organise data, so that it is useful for some purpose

In a stack, the data that is stored last, is taken out first, and vice versa

One way to implement a stack, is by using restricted insertions and deletions over an array

- The restriction is that the values can only be added or removed from the higher end of the array
- We call the currently highest index of the array, where some element is present, as the *top* of the stack

The operation to add an element to a stack is called *push()* and it takes the element as argument

The operation to delete an element from a stack is called *pop()* and it returns the deleted element

Example: Stack using an array

0	1	2	3	4
12	x	x	x	x

top = 0

Assume that we have a single element in the stack right now

Example: Stack using an array

0	1	2	3	4
12	x	x	x	x

`top = 0`

Assume that we have a single element in the stack right now

The `top` right now points to index `0`

Example: Stack using an array

0	1	2	3	4
12	X	X	X	X

`top = 0`

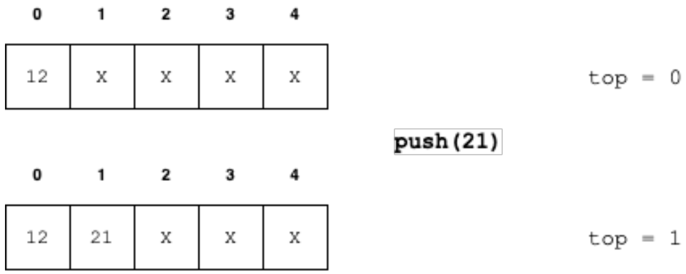
`push(21)`

0	1	2	3	4
12	21	X	X	X

`top = 1`

If we push an element, `21`, it is inserted at position `1`

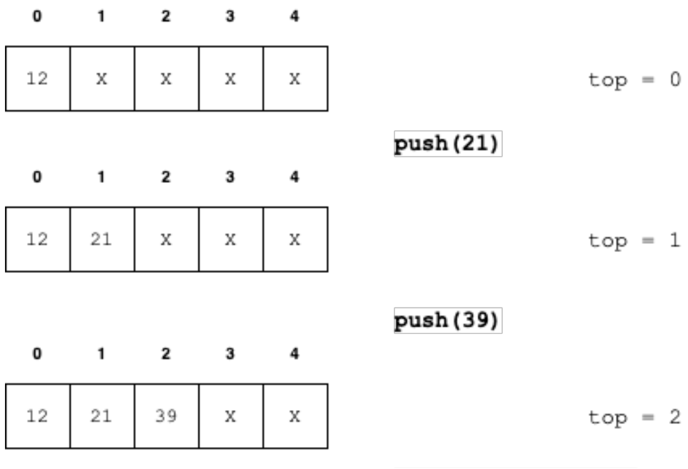
Example: Stack using an array



If we push an element, `21`, it is inserted at position `1`

The `top` now points to index `1`

Example: Stack using an array



If we push another element, 39, it is inserted at position 2

Example: Stack using an array

0	1	2	3	4
12	X	X	X	X

`top = 0`

`push(21)`

0	1	2	3	4
12	21	X	X	X

`top = 1`

`push(39)`

0	1	2	3	4
12	21	39	X	X

`top = 2`

If we push another element, `39`, it is inserted at position `2`

The `top` now points to index `2`

Example: Stack using an array

0	1	2	3	4
12	X	X	X	X

`top = 0`

`push(21)`

0	1	2	3	4
12	21	X	X	X

`top = 1`

`push(39)`

0	1	2	3	4
12	21	39	X	X

`top = 2`

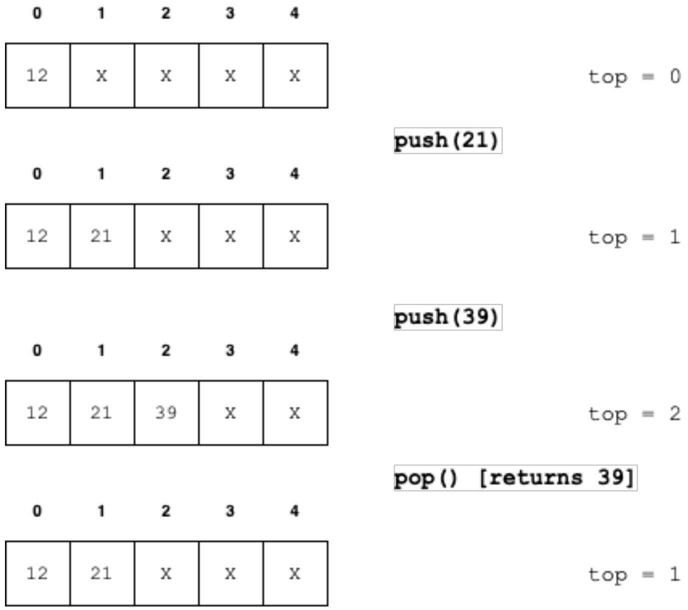
`pop() [returns 39]`

0	1	2	3	4
12	21	X	X	X

`top = 1`

If we pop an element, the element at index `top` is removed

Example: Stack using an array



If we pop an element, the element at index `top` is removed

The removed element, i.e., `39`, is returned, and `top` now points to `1`

Example: Stack using an array

0	1	2	3	4
12	X	X	X	X

top = 0

`push(21)`

0	1	2	3	4
12	21	X	X	X

top = 1

`push(39)`

0	1	2	3	4
12	21	39	X	X

top = 2

`pop() [returns 39]`

0	1	2	3	4
12	21	X	X	X

top = 1

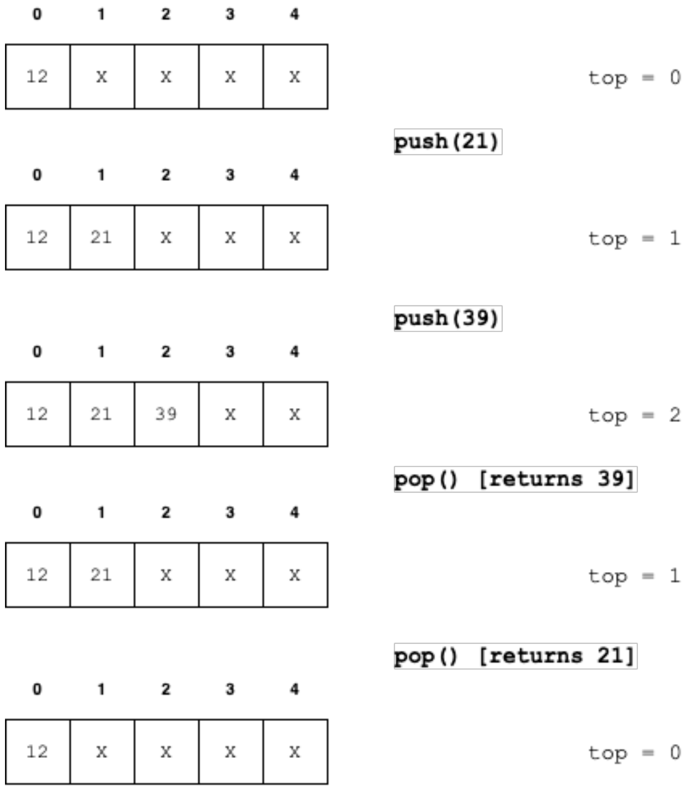
`pop() [returns 21]`

0	1	2	3	4
12	X	X	X	X

top = 0

Another pop will remove `21` from the stack

Example: Stack using an array



Another pop will remove `21` from the stack

The `top` now points to index `0`

Example: Stack using an array

0	1	2	3	4
12	x	x	x	x

top = 0

`push(21)`

0	1	2	3	4
12	21	x	x	x

top = 1

`push(39)`

0	1	2	3	4
12	21	39	x	x

top = 2

`pop() [returns 39]`

0	1	2	3	4
12	21	x	x	x

top = 1

`pop() [returns 21]`

0	1	2	3	4
12	x	x	x	x

top = 0

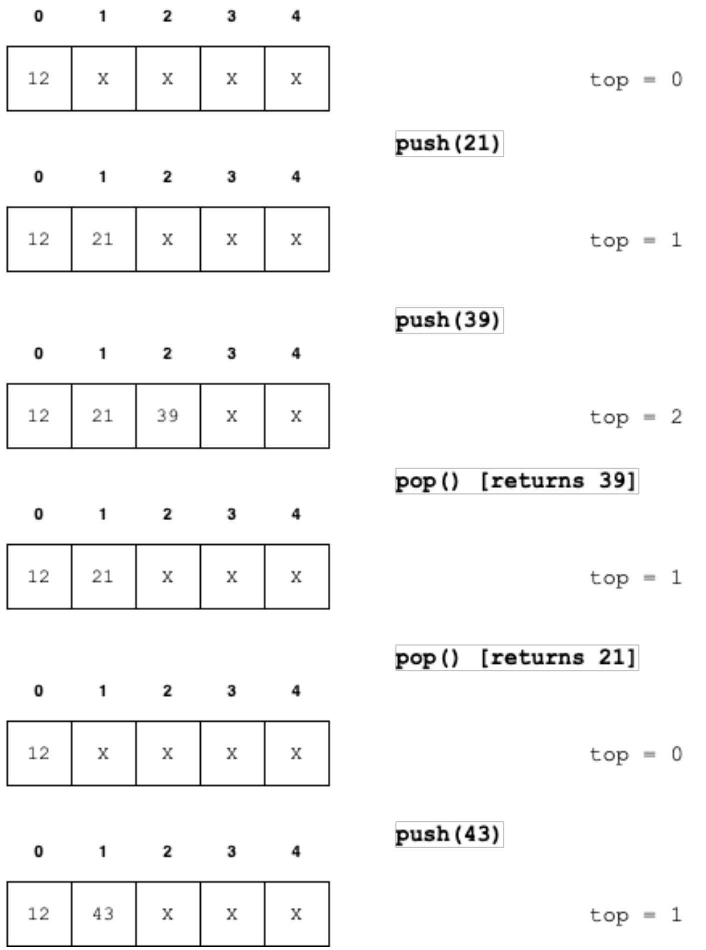
`push(43)`

0	1	2	3	4
12	43	x	x	x

top = 1

Another push operation, say with element `43`, adds it to index `1`

Example: Stack using an array



Another push operation, say with element `43`, adds it to index `1`

The `top` again increases to index `1`

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

Another data structure with a lot of utility is called *queue*

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

Another data structure with a lot of utility is called *queue*

In a queue, insertions and deletions are still restricted to ends, but they occur at different ends

- Instead of last in first out, a queue is a *first in first out* data structure

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

Another data structure with a lot of utility is called *queue*

In a queue, insertions and deletions are still restricted to ends, but they occur at different ends

- Instead of last in first out, a queue is a *first in first out* data structure

The end where insertions are done, is called the rear end…

- … and a variable that points to the index of the last inserted element is called *rear*

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

Another data structure with a lot of utility is called *queue*

In a queue, insertions and deletions are still restricted to ends, but they occur at different ends

- Instead of last in first out, a queue is a *first in first out* data structure

The end where insertions are done, is called the rear end…

- … and a variable that points to the index of the last inserted element is called *rear*

Similarly, the other end, where deletions occur, is called the front end…

- … and a variable that points to the element which will go out next is called *front*

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

Another data structure with a lot of utility is called *queue*

In a queue, insertions and deletions are still restricted to ends, but they occur at different ends

- Instead of last in first out, a queue is a *first in first out* data structure

The end where insertions are done, is called the rear end…

- … and a variable that points to the index of the last inserted element is called *rear*

Similarly, the other end, where deletions occur, is called the front end…

- … and a variable that points to the element which will go out next is called *front*

The counterparts of push() and pop() are called *enqueue()* and *dequeue()* respectively

A counterpart of Stack – Queue

With stack, the insertions and deletions are restricted at just one end

Another data structure with a lot of utility is called *queue*

In a queue, insertions and deletions are still restricted to ends, but they occur at different ends

- Instead of last in first out, a queue is a *first in first out* data structure

The end where insertions are done, is called the rear end…

- … and a variable that points to the index of the last inserted element is called *rear*

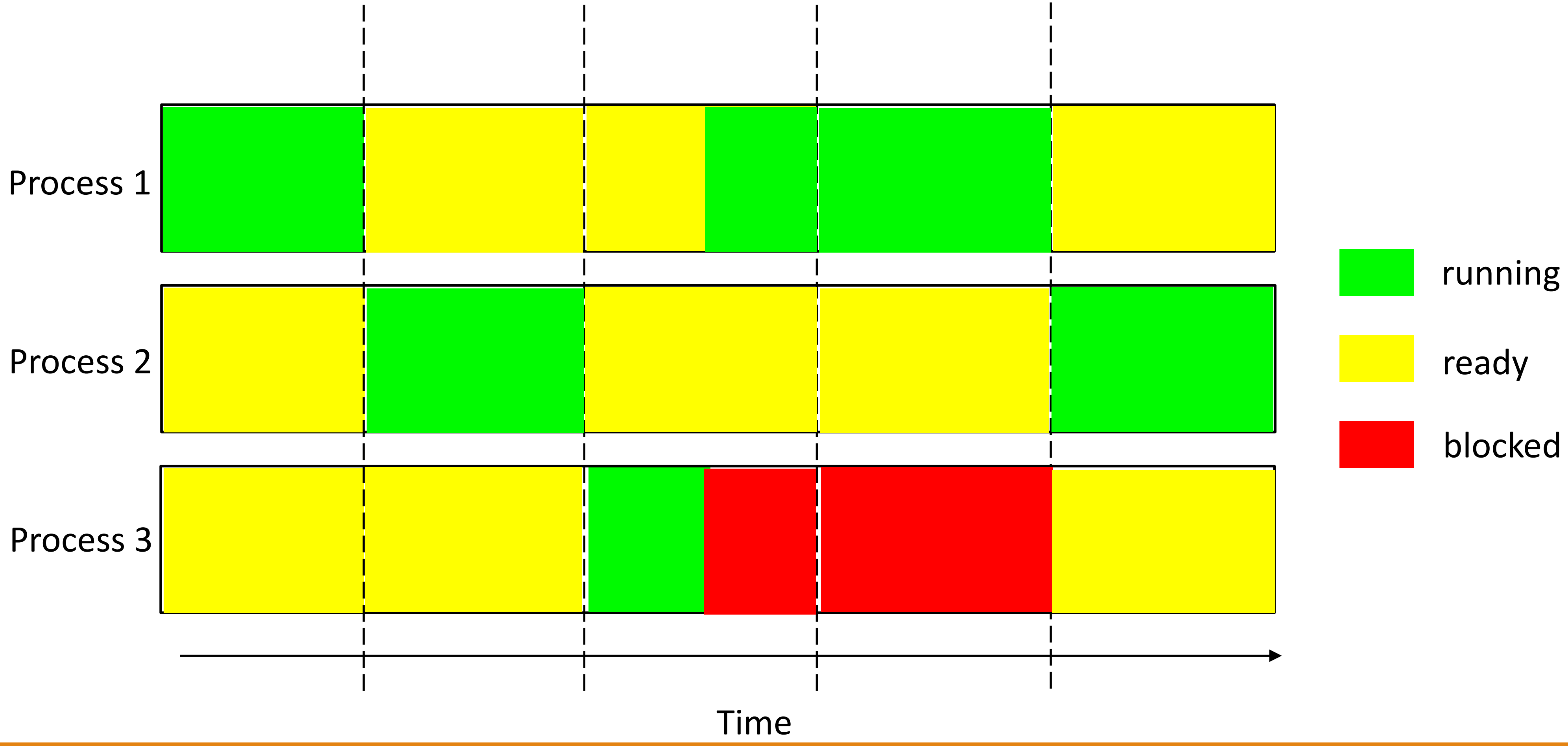
Similarly, the other end, where deletions occur, is called the front end…

- … and a variable that points to the element which will go out next is called *front*

The counterparts of push() and pop() are called *enqueue()* and *dequeue()* respectively

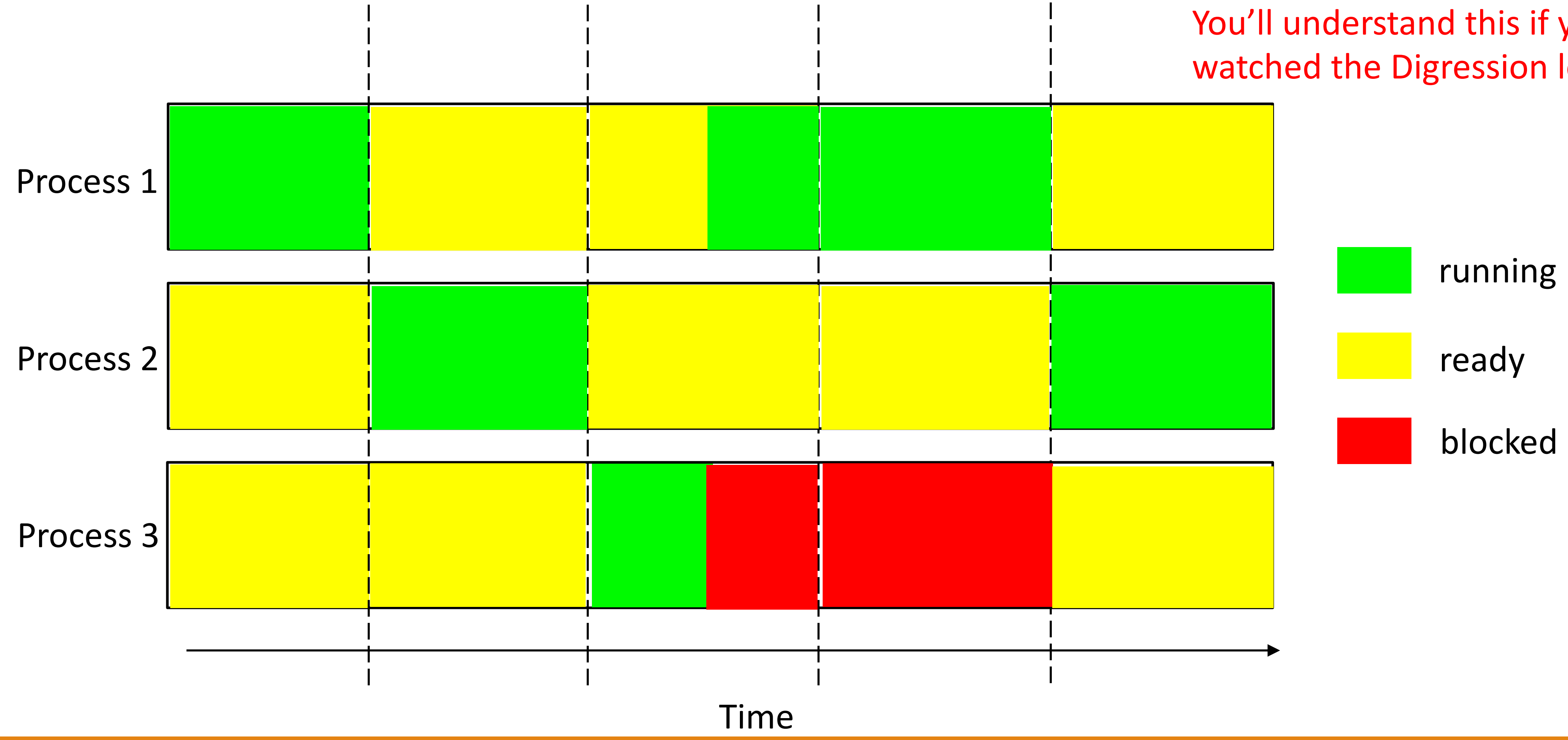
Queues are often used in holding details for processes, so that they can be scheduled judiciously

Example: Scheduling Processes



Example: Scheduling Processes

You’ll understand this if you watched the Digression lecture :P



Example: Queue using an array

0	1	2	3	4
12	x	x	x	x

```
rear = 0
front = 0
```

Assume that we have a single element in the queue right now

Example: Queue using an array

0	1	2	3	4
12	x	x	x	x

```
rear = 0
front = 0
```

Assume that we have a single element in the queue right now

Both `rear` and `front` point to index `0`

Example: Queue using an array

0	1	2	3	4
12	X	X	X	X

rear = 0
front = 0

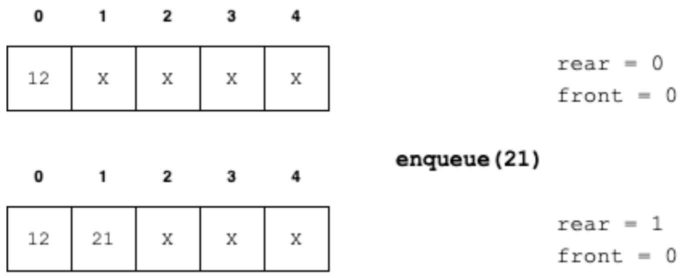
enqueue(21)

0	1	2	3	4
12	21	X	X	X

rear = 1
front = 0

If we enqueue an element, `21`, in the queue, it is added at index `1`

Example: Queue using an array



If we enqueue an element, `21`, in the queue, it is added at index `1`

While `rear` now points to index `1`, `front` remains unchanged

Example: Queue using an array

0	1	2	3	4
12	X	X	X	X

```
rear = 0
front = 0
```

enqueue(21)

0	1	2	3	4
12	21	X	X	X

```
rear = 1
front = 0
```

enqueue(39)

0	1	2	3	4
12	21	39	X	X

```
rear = 2
front = 0
```

Another enqueue request, for element `39`, is then executed

Example: Queue using an array

0	1	2	3	4
12	x	x	x	x

rear = 0
front = 0

enqueue(21)

0	1	2	3	4
12	21	x	x	x

rear = 1
front = 0

enqueue(39)

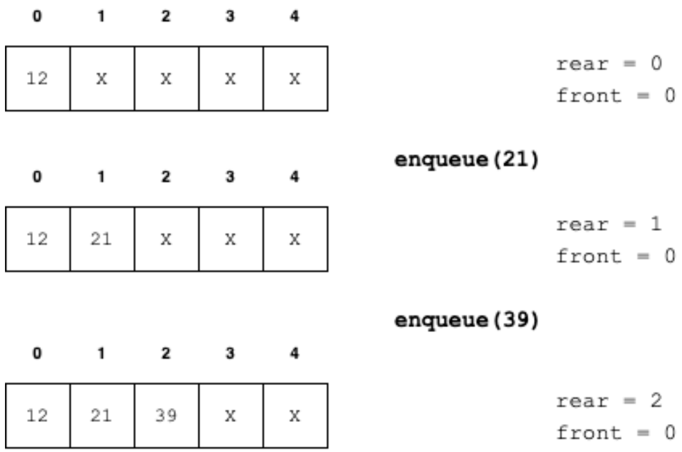
0	1	2	3	4
12	21	39	x	x

rear = 2
front = 0

Another enqueue request, for element `39`, is then executed

The element is added at index `2`

Example: Queue using an array



Another enqueue request, for element `39`, is then executed

The element is added at index `2`

It means `rear` now points to `2`; `front` remains at `0`

Example: Queue using an array

0	1	2	3	4
12	X	X	X	X

```
rear = 0
front = 0
```

enqueue(21)

0	1	2	3	4
12	21	X	X	X

```
rear = 1
front = 0
```

enqueue(39)

0	1	2	3	4
12	21	39	X	X

```
rear = 2
front = 0
```

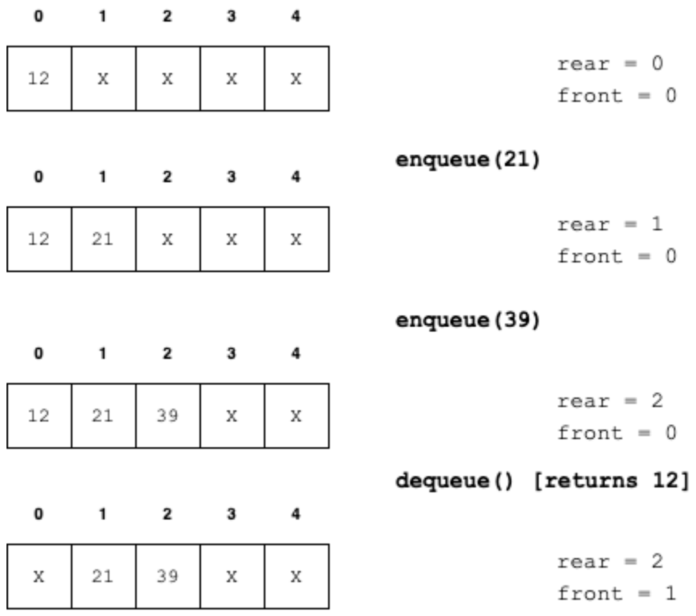
dequeue() [returns 12]

0	1	2	3	4
X	21	39	X	X

```
rear = 2
front = 1
```

On receiving a dequeue request, element `12` is removed from the queue and returned

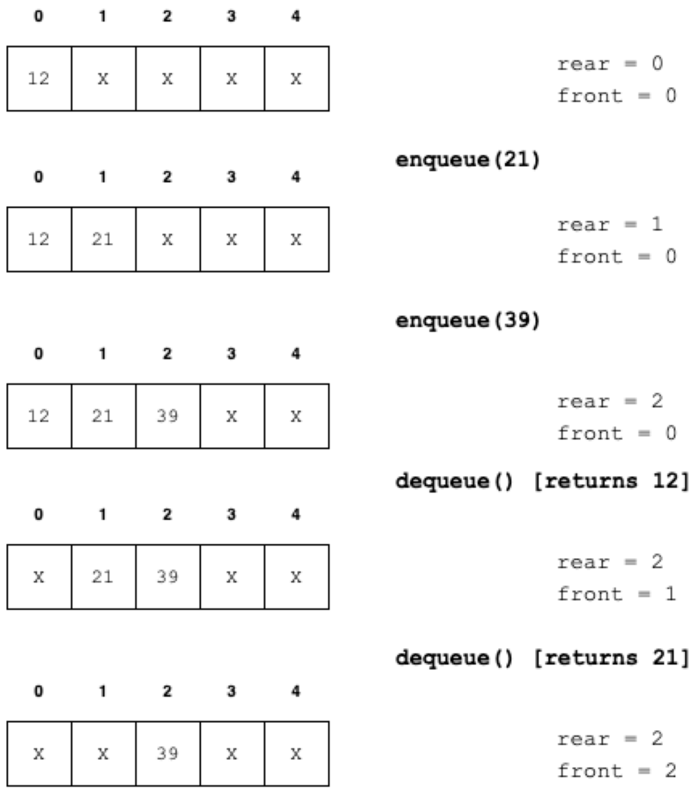
Example: Queue using an array



On receiving a dequeue request, element `12` is removed from the queue and returned

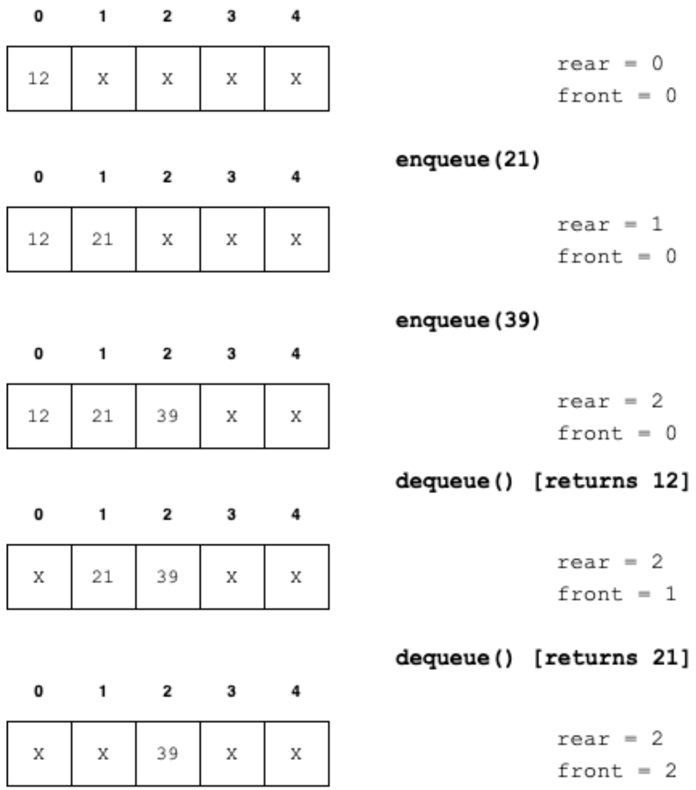
The `front` now moves to index `1`, while `rear` points to the same index as before

Example: Queue using an array



One more dequeue request removes `21` this time the queue, and the same is returned

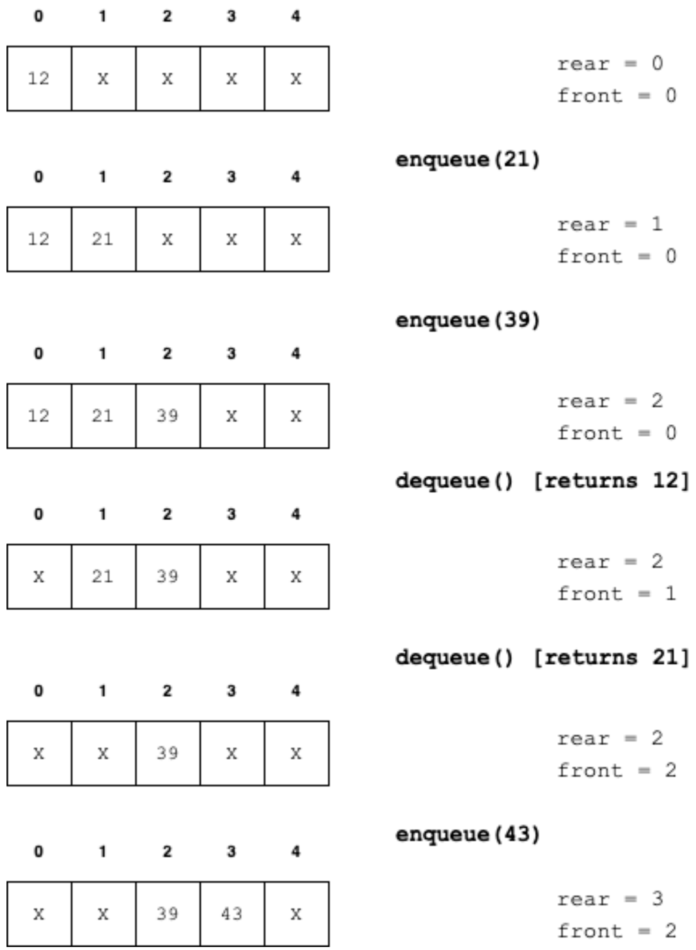
Example: Queue using an array



One more dequeue request removes `21` this time the queue, and the same is returned

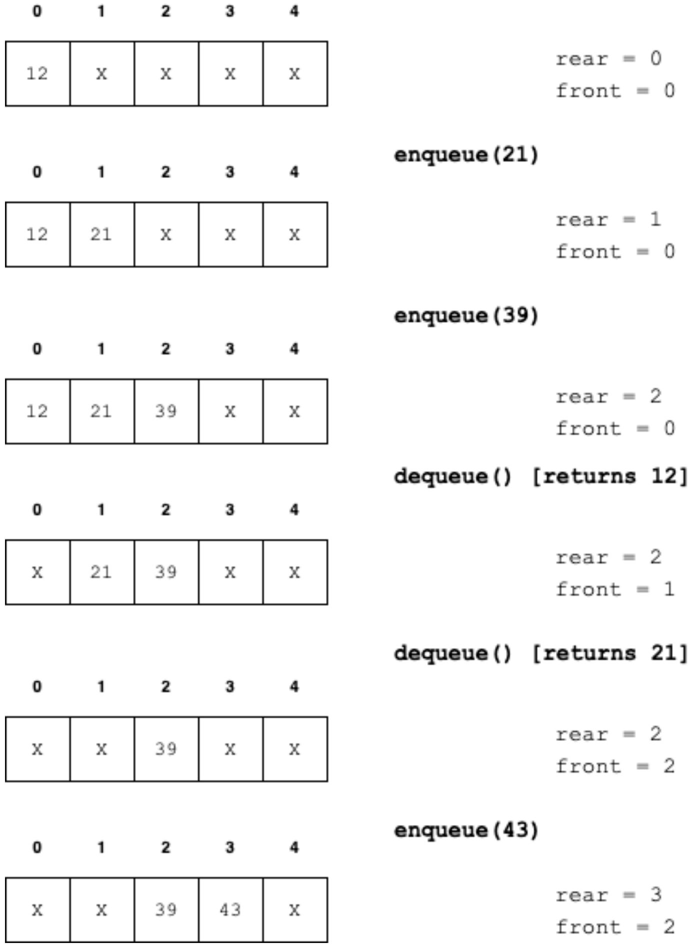
The `front` moves further to index `2`, joining `rear`

Example: Queue using an array



And again, another enqueue operation, this time for element `43`, results in its addition in the queue at index `3`

Example: Queue using an array



And again, another enqueue operation, this time for element `43`, results in its addition in the queue at index `3`

This results in advancement of `rear` to index `3`, leaving `front` at index `2`

Uses of Stacks and Queues

We saw how stacks are used to implement a system for chains of function calls

Uses of Stacks and Queues

We saw how stacks are used to implement a system for chains of function calls

In fact, if you have a recursive solution to a difficult problem…

- … and for performance reasons, you want to convert it to an iterative solution…
- … using a stack can provide a straightforward conversion

Uses of Stacks and Queues

We saw how stacks are used to implement a system for chains of function calls

In fact, if you have a recursive solution to a difficult problem…

- … and for performance reasons, you want to convert it to an iterative solution…
- … using a stack can provide a straightforward conversion

Queues are used for many purposes in Computer Science

- Protocols that requires a “first-come-first-serve” model usually use queues in background
- A common example is usage in any type of message delivery system

Uses of Stacks and Queues

We saw how stacks are used to implement a system for chains of function calls

In fact, if you have a recursive solution to a difficult problem…

- … and for performance reasons, you want to convert it to an iterative solution…
- … using a stack can provide a straightforward conversion

Queues are used for many purposes in Computer Science

- Protocols that requires a "first-come-first-serve" model usually use queues in background
- A common example is usage in any type of message delivery system

However, the formulation of queues that we saw in this lecture is seldom used

- It is because we will not be able to enqueue any new element, when `rear` reaches the end of the array
- Usually, the queues that are used more practically, are *circular* in nature (see **Additional Reading** slide)

Uses of Stacks and Queues

We saw how stacks are used to implement a system for chains of function calls

In fact, if you have a recursive solution to a difficult problem…

- … and for performance reasons, you want to convert it to an iterative solution…
- … using a stack can provide a straightforward conversion

Queues are used for many purposes in Computer Science

- Protocols that requires a “first-come-first-serve” model usually use queues in background
- A common example is usage in any type of message delivery system

However, the formulation of queues that we saw in this lecture is seldom used

- It is because we will not be able to enqueue any new element, when `rear` reaches the end of the array
- Usually, the queues that are used more practically, are *circular* in nature (see **Additional Reading** slide)

This was just an introduction to two most common data structures

- We will have a look at one more important data structure – *linked lists* – in the next week

Homework !!

Revisit the program called `BasicArrayOperations.c` from Week 5

- Change it to create two programs – `StackOperations.c` and `QueueOperations.c`
- Modify the code to emulate the working of a stack and a queue respectively
- See if you can handle the problem with formulation of a queue that we discussed
  Hint: May be shifting elements of the array could lead to a solution

Additional Reading

A more practical version of queue is called *circular queue*

- A circular queue allows the rear and the front variables to "wrap-around"…
- … i.e., they can go from 0 to size-1 and vice versa, if applicable

Read more about circular queues

- You may start here: https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/