

Processes

Johan Montelius

KTH

VT26

arrays

recap

- collection of data structures accessed by index
- assumed constant time access
- fixed size*

Many programming languages provide dynamic arrays that can grow in size.

arrays

recap

- collection of data structures accessed by index
- assumed constant time access
- fixed size*

Many programming languages provide dynamic arrays that can grow in size.

arrays

recap

- collection of data structures accessed by index
- assumed constant time access
- fixed size*

Many programming languages provide dynamic arrays that can grow in size.

stacks

A *stack* is an abstract data structure that provides two operations: `push(val)` and `pop()`. The `push` operation will add a new item to the stack and the `pop` operations will remove and return the last pushed item.

How do we implement this using an array?

stacks

A *stack* is an abstract data structure that provides two operations: `push(val)` and `pop()`. The `push` operation will add a new item to the stack and the `pop` operations will remove and return the last pushed item.

How do we implement this using an array?

a first try

```
typedef struct stack_t {  
    int top;  
    int *array;  
} stack_t;  
  
stack_t *stack_new(int size) {  
    stack_t *stk = (stack_t*)malloc(sizeof(stack_t));  
    stk->array = (int*)malloc(size*sizeof(int));  
    stk->top = 0;  
    return stk;  
}
```

push()

```
void push(stack_t *stk, int val) {  
    stk->array[stk->top] = val;  
    stk->top += 1;  
}
```

What if top is greater than the size of the array?

push()

```
void push(stack_t *stk, int val) {  
    stk->array[stk->top] = val;  
    stk->top += 1;  
}
```

What if top is greater than the size of the array?

pop()

```
int pop(stack_t *stk) {  
    stk->top -= 1;  
    int val = stk->array[stk->top];  
    return val;  
}
```

What if top is 0?

pop()

```
int pop(stack_t *stk) {  
    stk->top -= 1;  
    int val = stk->array[stk->top];  
    return val;  
}
```

What if top is 0?

stacks

- efficient push and pop operations
- efficient memory usage
- extremely useful when implementing a programming language

static vs dynamic

- How large stack should be allocated?
- What should we do when the stack is full?
- Can we allow the stack to grow?

static vs dynamic

- How large stack should be allocated?
- What should we do when the stack is full?
- Can we allow the stack to grow?

static vs dynamic

- How large stack should be allocated?
- What should we do when the stack is full?
- Can we allow the stack to grow?

static vs dynamic

- How large stack should be allocated?
- What should we do when the stack is full?
- Can we allow the stack to grow?

allocate a new large stack

- Allocate an array of some size.
- Detect that you're about to push above the limit.
- If so, allocate a larger array and copy the content to the new array, then push.

allocate a new large stack

- Allocate an array of some size.
- Detect that you're about to push above the limit.
- If so, allocate a larger array and copy the content to the new array, then push.

allocate a new large stack

- Allocate an array of some size.
- Detect that you're about to push above the limit.
- If so, allocate a larger array and copy the content to the new array, then push.

a dynamic stack

```
typedef struct stack_t {  
    :  
    int size;  
} stack_t;  
  
stack_t *stack_new(int size) {  
    :  
    stk->size = size;  
    :  
}
```

push

```
    :  
    if (stk->top == stk->size) {  
        int size = stk->size * 2;  
        stk->size = size;  
        int *copy = (int*)malloc(size*sizeof(int));  
        for (int i = 0; i < stk->top; i++) {  
            copy[i] = stk->array[i];  
        }  
        free(stk->array);  
        stk->array = copy;  
    }  
    :
```

```
:  
if ((stk->top < stk->size/4) && (MIN < stk->size/2)) {  
    int size = stk->size/2;  
    stk->size = size;  
  
    int *copy = (int*)malloc(size*sizeof(int));  
    for (int i = 0; i < stk->top; i++) {  
        copy[i] = stk->array[i];  
    }  
    free(stk->array);  
    stk->array = copy;  
}  
:
```

runtime complexity

- What is the run time complexity of push and pop in a static stack?
- What is the run time complexity of push and pop in a dynamic stack?

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

cost of a push operation in dynamic stack

Assume a stack of size 16, first 16 push operations are constant.

- you pay a price for copying 16 elements ...
- then its a constant cost for the next 16 push operations
- you pay a price for copying 32 element ...
- then its a constant cost for the next 32 push operations
-

A push operation could be very costly.

amortized cost

Assume a stack of size 16 with 8 items.

- the next eight push operation you pay one crown but also put two crowns in a bowl
- when it's time to copy you have 16 crowns in the bowl that will pay for the copy
- the next 16 push operations pay a crown but also put two crowns in a bowl
-

The *amortized cost* is constant.

amortized cost

Assume a stack of size 16 with 8 items.

- the next eight push operation you pay one crown but also put two crowns in a bowl
- when it's time to copy you have 16 crowns in the bowl that will pay for the copy
- the next 16 push operations pay a crown but also put two crowns in a bowl
-

The *amortized cost* is constant.

amortized cost

Assume a stack of size 16 with 8 items.

- the next eight push operation you pay one crown but also put two crowns in a bowl
- when it's time to copy you have 16 crowns in the bowl that will pay for the copy
- the next 16 push operations pay a crown but also put two crowns in a bowl
-

The *amortized cost* is constant.

amortized cost

Assume a stack of size 16 with 8 items.

- the next eight push operation you pay one crown but also put two crowns in a bowl
- when it's time to copy you have 16 crowns in the bowl that will pay for the copy
- the next 16 push operations pay a crown but also put two crowns in a bowl
-

The *amortized cost* is constant.

what about pop?

Should we shrink the size if we pop half of the elements?

- pros
- cons

what about pop?

Should we shrink the size if we pop half of the elements?

- pros
- cons

what about pop?

Should we shrink the size if we pop half of the elements?

- pros
- cons

what about pop?

Should we shrink the size if we pop half of the elements?

- pros
- cons

what about pop?

Should we shrink the size if we pop half of the elements?

- pros
- cons

the call stack

stack
frame
of bar

```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    z = foo(3, 4)  
    return z;  
}
```

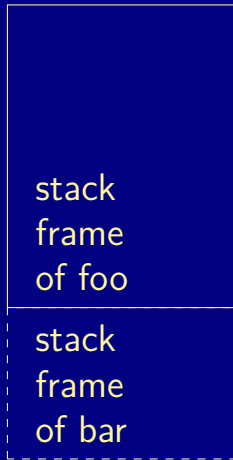
the call stack

stack
frame
of bar

```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    —————> z = foo(3, 4)  
    return z;  
}
```

the call stack



```
int foo(int x, int y) {  
    → return x + y;  
}
```

```
int bar() {  
    int z;  
    z = foo(3, 4)  
    return z;  
}
```

the call stack

stack
frame
of bar

```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    z = foo(3, 4)  
    —————> return z;  
}
```

the frame

The *conceptual stack frame* contains:

- the arguments of the procedure
- place to hold local variables
- magic information to be able to return from a call

the frame

The *conceptual stack frame* contains:

- the arguments of the procedure
- place to hold local variables
- magic information to be able to return from a call

the frame

The *conceptual stack frame* contains:

- the arguments of the procedure
- place to hold local variables
- magic information to be able to return from a call

the frame

The *conceptual stack frame* contains:

- the arguments of the procedure
- place to hold local variables
- magic information to be able to return from a call

Implementation of the stack

The CPU can be described by:

- the instruction pointer (IP): a reference to the next instruction to execute
- a stack pointer (SP): a reference to the top of the stack
- a base pointer (BP): a reference to the current stack frame
- general purpose registers: used by a process to store data

Implementation of the stack

The CPU can be described by:

- the instruction pointer (IP): a reference to the next instruction to execute
- a stack pointer (SP): a reference to the top of the stack
- a base pointer (BP): a reference to the current stack frame
- general purpose registers: used by a process to store data

Implementation of the stack

The CPU can be described by:

- the instruction pointer (IP): a reference to the next instruction to execute
- a stack pointer (SP): a reference to the top of the stack
- a base pointer (BP): a reference to the current stack frame
- general purpose registers: used by a process to store data

Implementation of the stack

The CPU can be described by:

- the instruction pointer (IP): a reference to the next instruction to execute
- a stack pointer (SP): a reference to the top of the stack
- a base pointer (BP): a reference to the current stack frame
- general purpose registers: used by a process to store data

Implementation of the stack

The CPU can be described by:

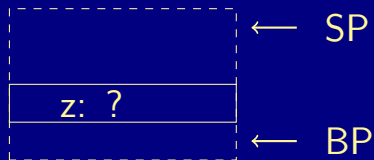
- the instruction pointer (IP): a reference to the next instruction to execute
- a stack pointer (SP): a reference to the top of the stack
- a base pointer (BP): a reference to the current stack frame
- general purpose registers: used by a process to store data

Implementation of the stack

The CPU can be described by:

- the instruction pointer (IP): a reference to the next instruction to execute
- a stack pointer (SP): a reference to the top of the stack
- a base pointer (BP): a reference to the current stack frame
- general purpose registers: used by a process to store data

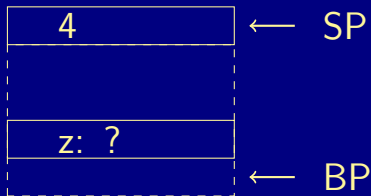
the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP —————> z = foo(3, 4);  
    return z;  
}
```

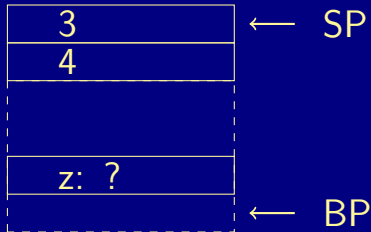
the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

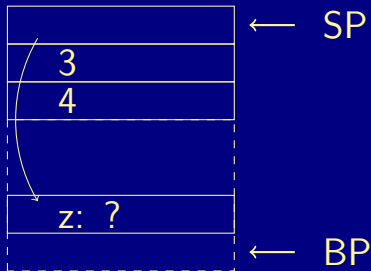
the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

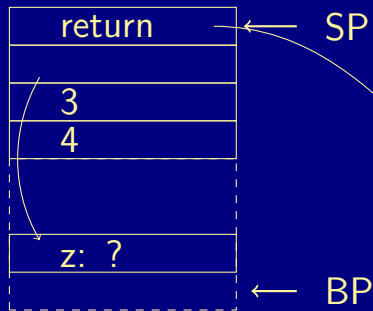
the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

the call stack

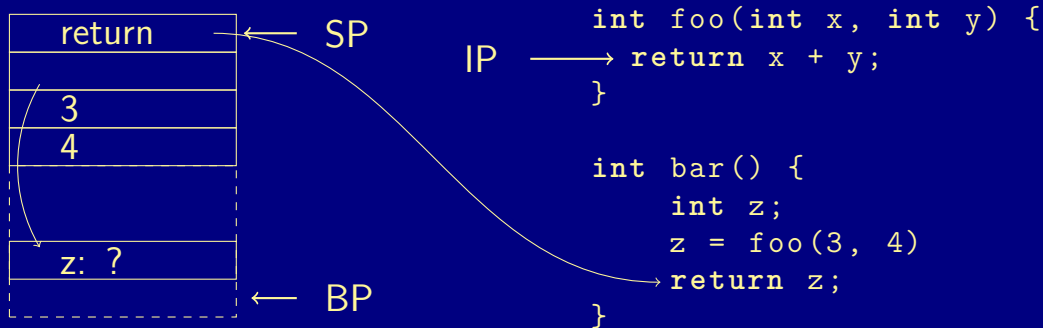


```
int foo(int x, int y) {  
    return x + y;  
}
```

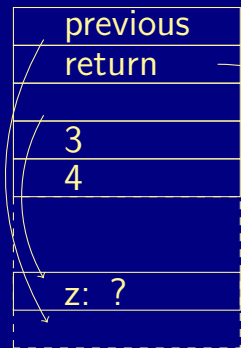
```
int bar() {  
    int z;  
    z = foo(3, 4);  
    return z;  
}
```

IP

the call stack



the call stack

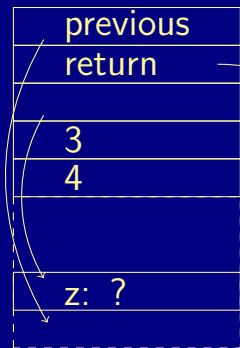


← SP

```
int foo(int x, int y) {  
    IP → return x + y;  
}
```

```
int bar() {  
    int z;  
    z = foo(3, 4)  
    return z;  
}
```

the call stack



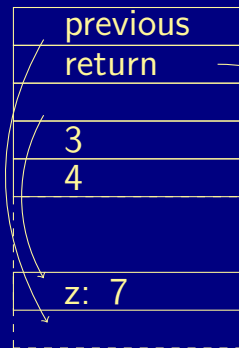
← SP/BP

IP →

```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    z = foo(3, 4)  
    return z;  
}
```

the call stack

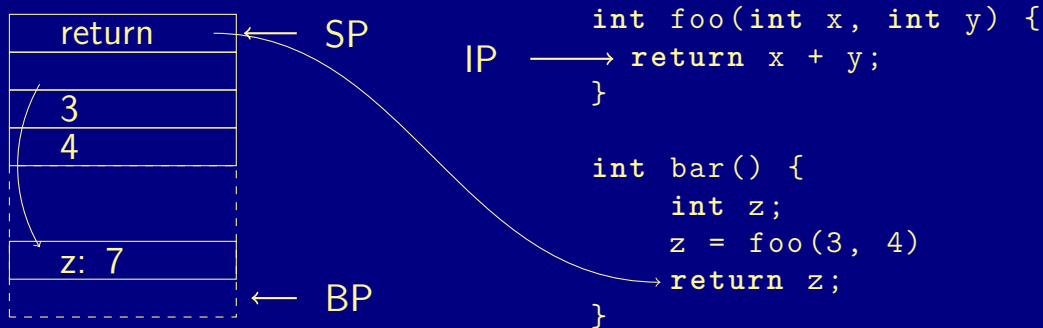


← SP/BP

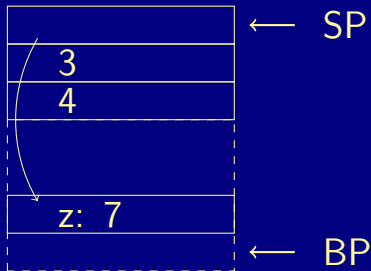
```
int foo(int x, int y) {  
    IP → return x + y;  
}
```

```
int bar() {  
    int z;  
    z = foo(3, 4)  
    return z;  
}
```

the call stack



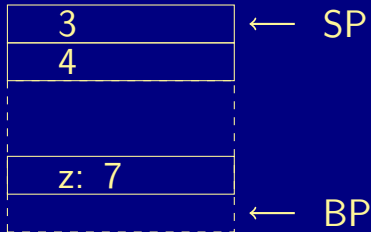
the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

the call stack



```
int foo(int x, int y) {  
    return x + y;  
}
```

```
int bar() {  
    int z;  
    IP → z = foo(3, 4);  
    return z;  
}
```

the call stack

Data structures placed on the stack will not be available once we return from a call.

The space required to store the structures can be reused immediately.

Not all data structures could be allocated on the stack (not the ones that should live for longer).

Data structures that should survive a return should be placed on the *heap*.

the call stack

Data structures placed on the stack will not be available once we return from a call.

The space required to store the structures can be reused immediately.

Not all data structures could be allocated on the stack (not the ones that should live for longer).

Data structures that should survive a return should be placed on the *heap*.

the call stack

Data structures placed on the stack will not be available once we return from a call.

The space required to store the structures can be reused immediately.

Not all data structures could be allocated on the stack (not the ones that should live for longer).

Data structures that should survive a return should be placed on the *heap*.

the call stack

Data structures placed on the stack will not be available once we return from a call.

The space required to store the structures can be reused immediately.

Not all data structures could be allocated on the stack (not the ones that should live for longer).

Data structures that should survive a return should be placed on the *heap*.

the call stack

Data structures placed on the stack will not be available once we return from a call.

The space required to store the structures can be reused immediately.

Not all data structures could be allocated on the stack (not the ones that should live for longer).

Data structures that should survive a return should be placed on the *heap*.

programming languages

- Java : all primitive data structures (integers, floats ..) are placed on the stack, all other (new) are on the heap.
- C : data structures are normally allocated on the stack, only if you use malloc will it be placed on the heap. A call to free will remove the data structure.
- C++ : similar to C but now we use the new directive to create and delete to remove

How do we remove data structures from the heap in Java?

programming languages

- Java : all primitive data structures (integers, floats ..) are placed on the stack, all other (new) are on the heap.
- C : data structures are normally allocated on the stack, only if you use `malloc` will it be placed on the heap. A call to `free` will remove the data structure.
- C++ : similar to C but now we use the `new` directive to create and `delete` to remove

How do we remove data structures from the heap in Java?

programming languages

- Java : all primitive data structures (integers, floats ..) are placed on the stack, all other (new) are on the heap.
- C : data structures are normally allocated on the stack, only if you use `malloc` will it be placed on the heap. A call to `free` will remove the data structure.
- C++ : similar to C but now we use the `new` directive to create and `delete` to remove

How do we remove data structures from the heap in Java?

the process

What does the *process* look like in memory when we run a program?



simplistic view, in reality much more complicated

the process

What does the *process* look like in memory when we run a program?



simplistic view, in reality much more complicated

the process

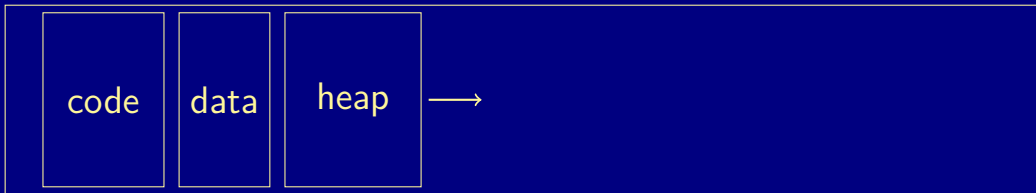
What does the *process* look like in memory when we run a program?



simplistic view, in reality much more complicated

the process

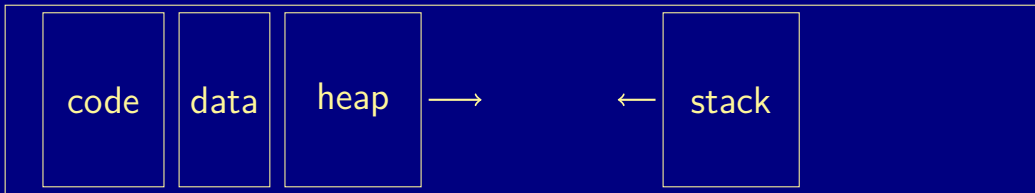
What does the *process* look like in memory when we run a program?



simplistic view, in reality much more complicated

the process

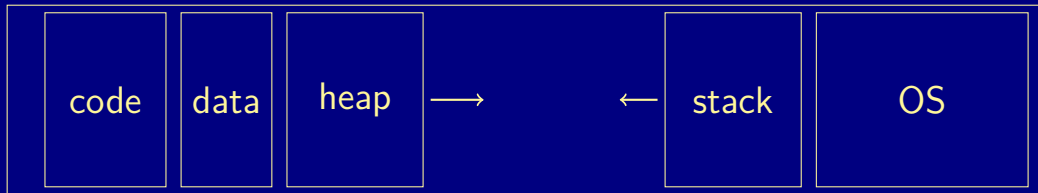
What does the *process* look like in memory when we run a program?



simplistic view, in reality much more complicated

the process

What does the *process* look like in memory when we run a program?



simplistic view, in reality much more complicated