Dynamic memory

Programação (L.EIC009)

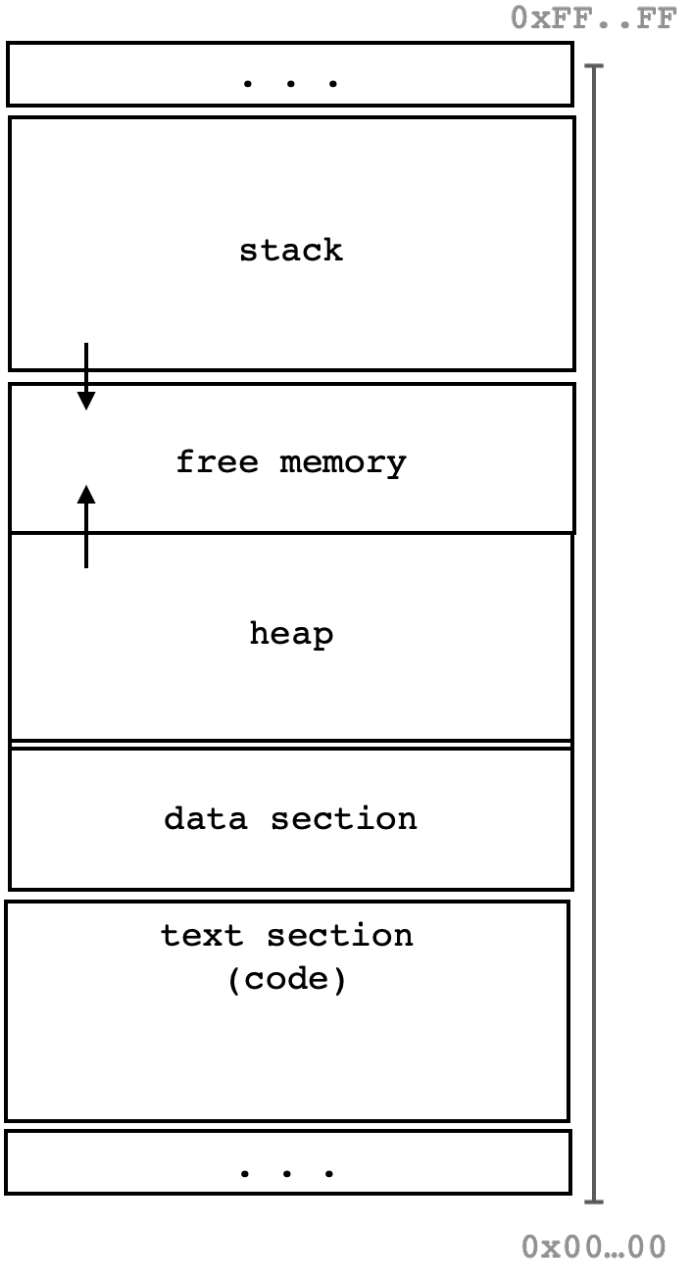
Eduardo R. B. Marques, DCC/FCUP

Outline

- Memory regions. What is dynamic memory?
- Dynamic memory in C++: `new` and `delete`
- Case-study example: a simple vector

Memory organisation for a program

Memory sections



Memory sections (cont.)

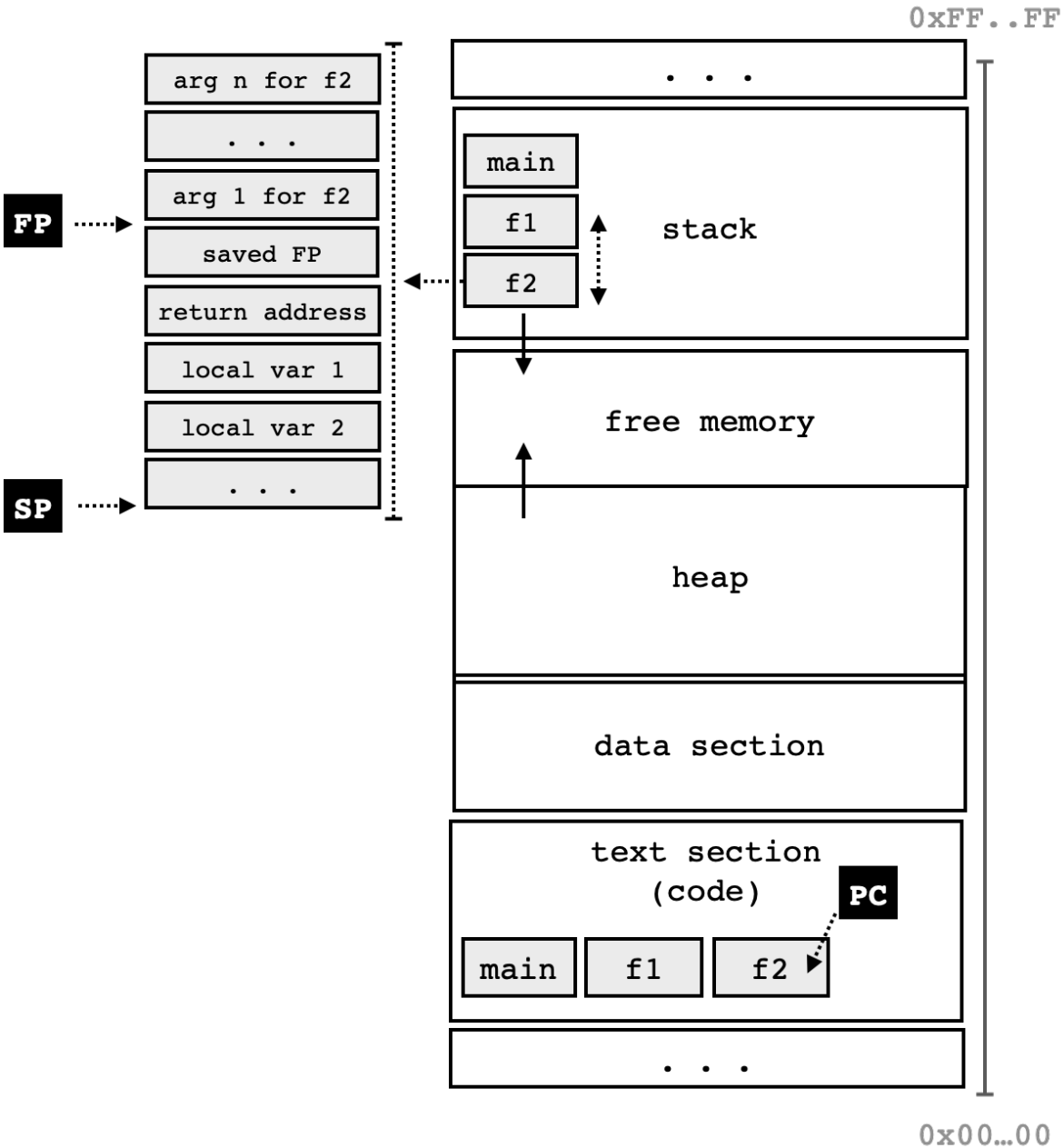
Text: the binary machine code that results from compiling the program (it is called the "text" section for historical reasons)

Data: memory used by global variables and constants.

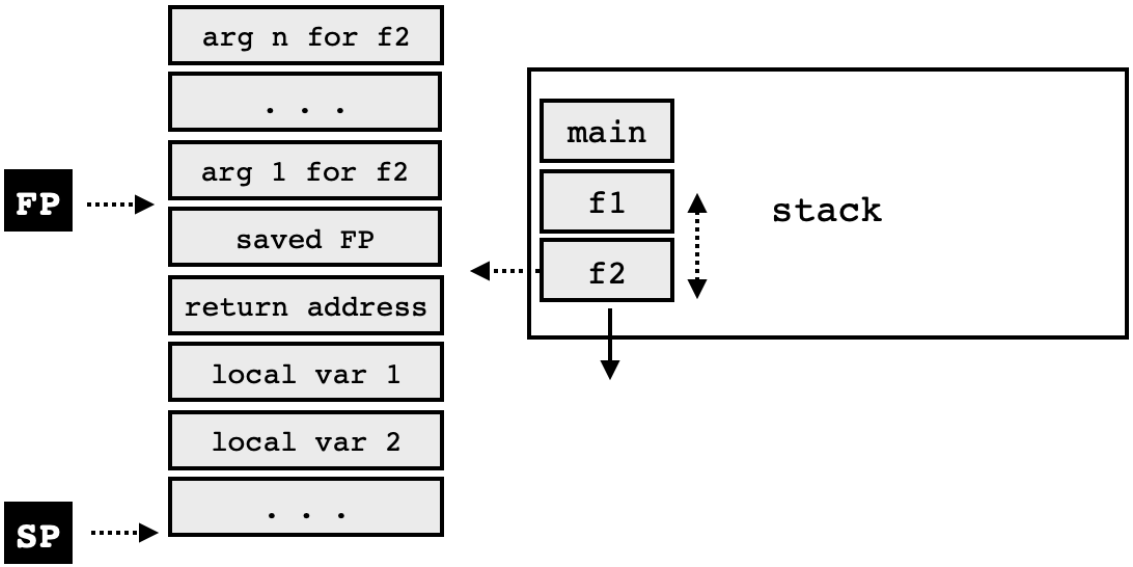
Stack: the execution stack, containing one stack frame per function invocation (usually "grows down" in the memory address space).

Heap: **dynamic memory** - managed explicitly by the program in C/C++, or garbage collection mechanisms in many languages (e.g., Python, C#, Java) - usually "grows up" in the memory address space.

The execution stack



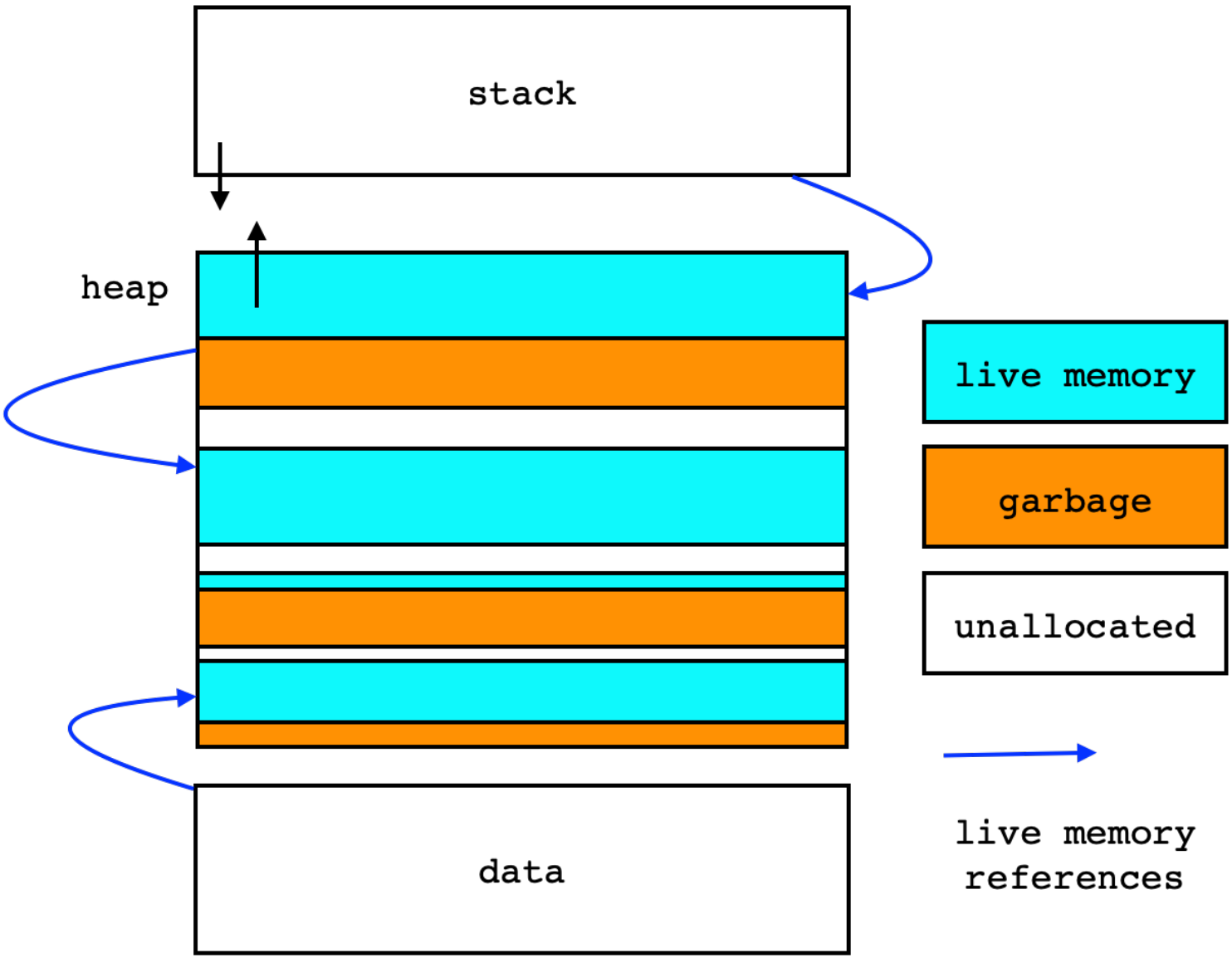
The execution stack (cont.)



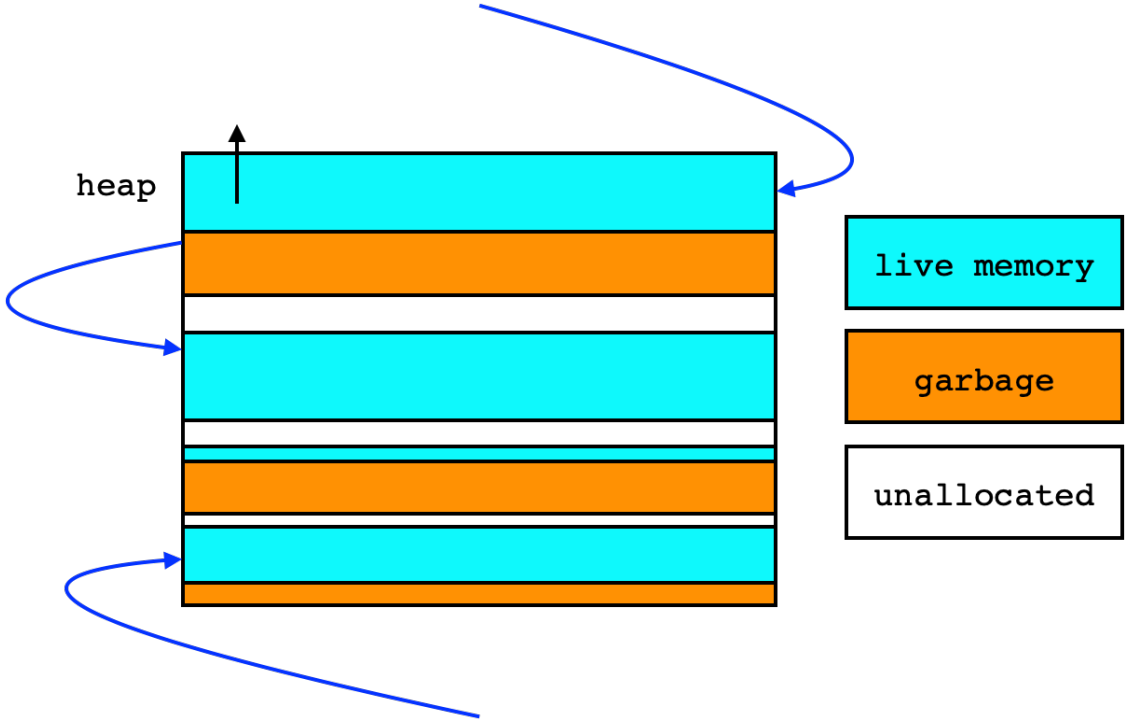
A stack frame contains the necessary space for a function invocation, including arguments, local variables, and call management information such as the return address.

A programmer need not be concerned with the low-level details of function invocation. The execution stack is managed automatically during execution. A compiler generates machine code to handle function calls and returns.

The heap


stack
heap
data
live memory
garbage
unallocated
live memory
references

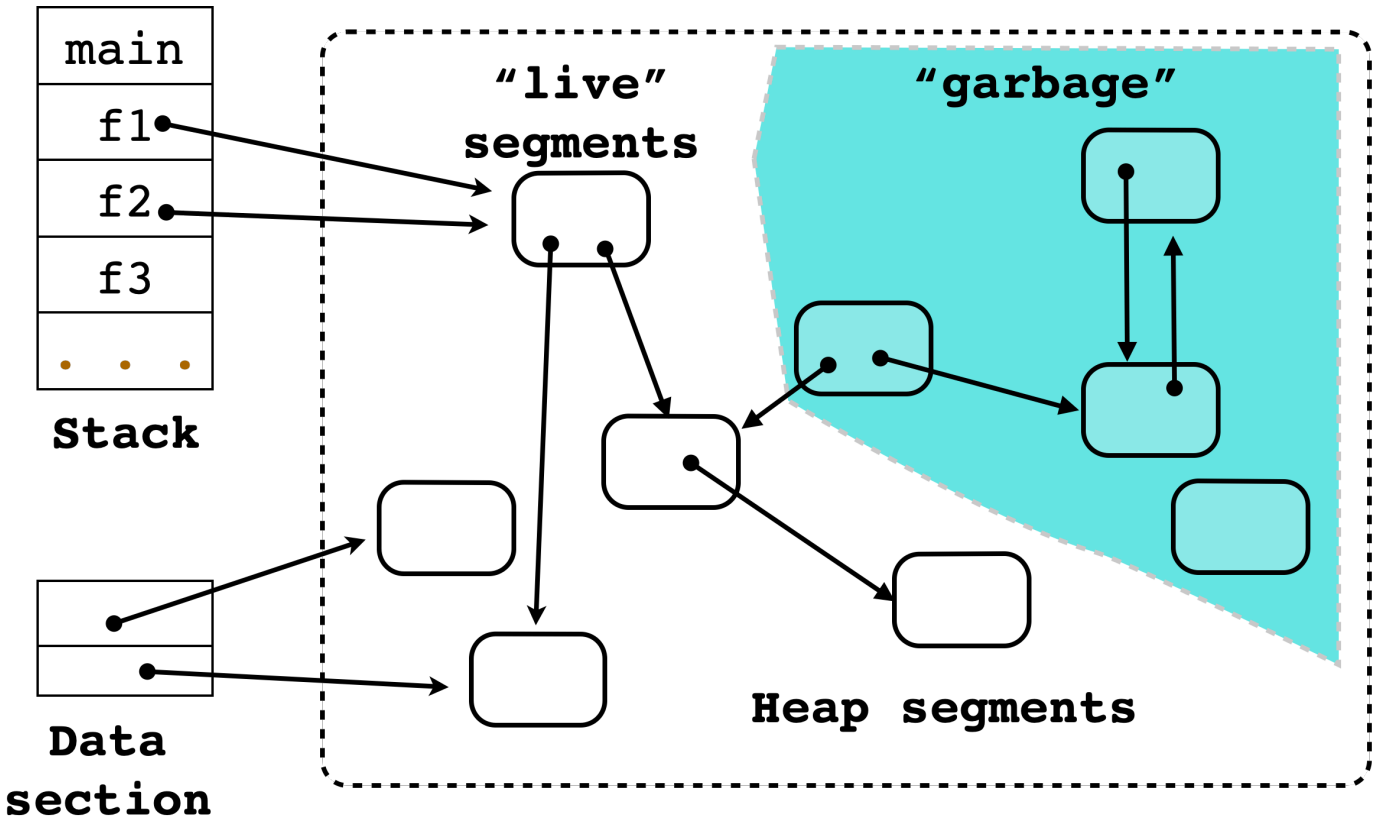
The heap (cont.)



The heap contains memory segments than can either be:

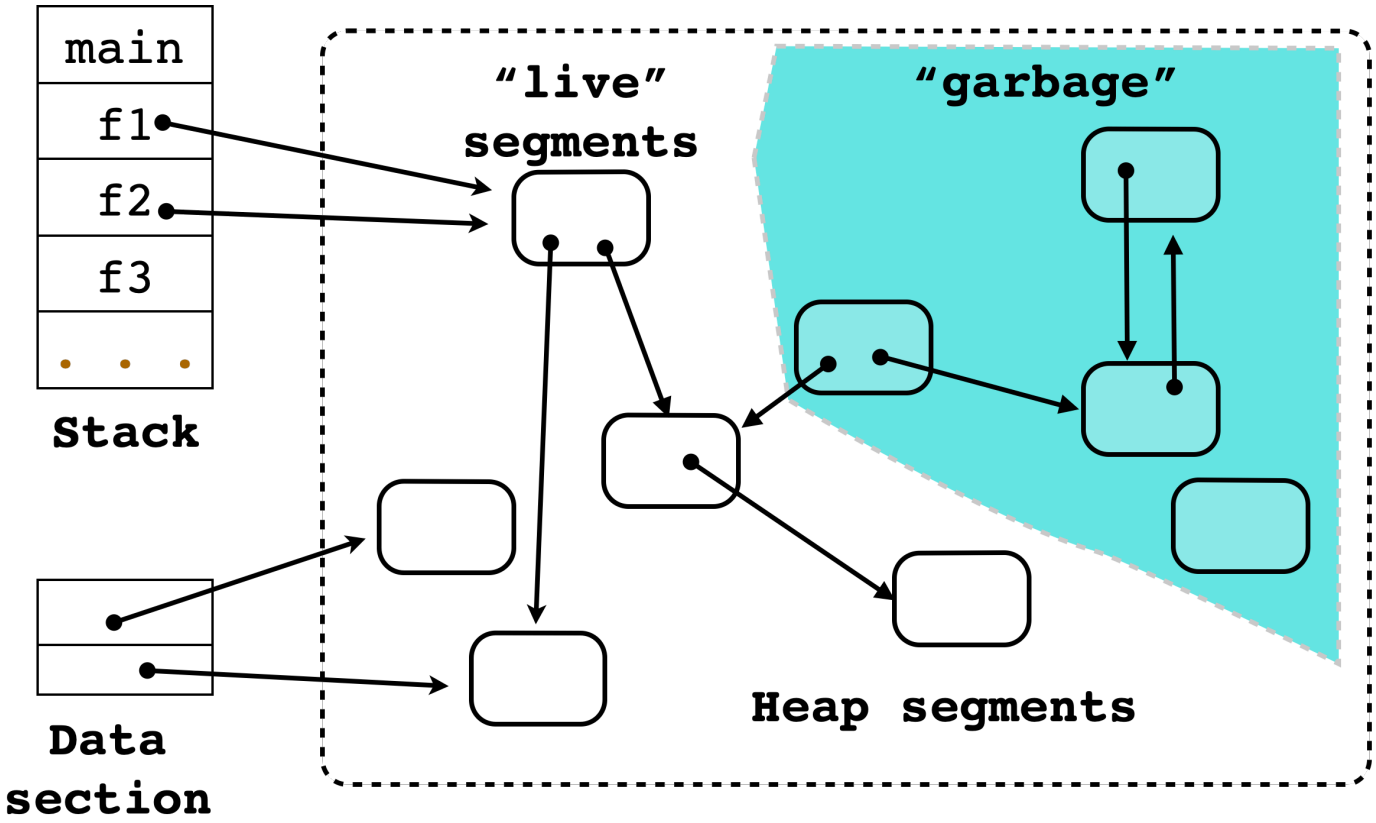
- **live** : allocated memory that is being used by a program, referenced by the execution stack or global data;
- **garbage**: allocated memory that is no longer being used by a program, but has not been freed yet;
- **unallocated**: memory segments that are available for execution;

The heap (cont.)



A heap segment may be created by a particular function invocation but many times outlives that invocation, i.e., the segment may be freed up long after the original creator function invocation ended its execution.

The heap (cont.)



Garbage collection mechanisms identify and free garbage segments automatically in many programming languages, e.g., Python, Java, C#, … **but in C/C++ we must avoid their existence through explicit programming.**

Dynamic memory management in C/C++

As we know already, C/C++ programs are not memory-safe (e.g., as we discussed for buffer overflows). The use of dynamic memory (heap memory) raises a number of additional issues, such as:

- **memory leaks**: garbage that is never freed up, taking up space for no purpose;
- **dangling references**: references to segments that are no longer allocated;
- and, to avoid these problems, **correct programming** of dynamic memory management!

Dynamic memory in C++: `new` and `delete`

`new` and `delete`

C++ defines operators **`new`** and **`delete`** to manage dynamic memory.

`new` allocates (new) memory:

```
// Allocate memory for one item of type T.
T* simple_ptr = new T;

// Allocate memory for an array of type T.
T*  array_ptr = new T[size];
```

`delete` frees up memory previously allocated using **`new`**:

```
// Free allocated memory for one item.
delete simple_ptr;

// Free allocated array.
delete [] array_ptr;
```

Illustration of syntax

For one item of memory:

```
int* pi; = new int; // primitive type
time_of_day* pt = new time_of_day; // struct type
. . .
delete pi;
delete pt;
```

For arrays - note that the array size need not be a constant expression:

```
int n; cin >> n;
int* iarr = new int[n];
time_of_day* tarr = new time_of_day[20];
. . .
delete [] iarr;
delete [] tarr;
```

Illustration of syntax (cont.)

Initialisers (for arrays, struct types, etc) can be employed with `new`:

```
int* iarr = new int[10] { 59, 0 };
time_of_day* pt = new time_of_day { 23, 58 };
time_of_day* tarr =
   new time_of_day[3] { {23, 57},
                        *pt,
                        { 23, iarr[0] } };
```

Arbitrary values will be stored initially if no initialisers are used, as in the case of variable declarations.

Example

A simple program that prints in reverser order `n` values read from the standard input, where `n` is a runtime input value.

```
int main() {
  int n; cout << "n? "; cin >> n;
  // Allocate array.
  double* arr = new double[n];
  // Read values
  for (int i = 0; i < n; i++) cin >> arr[i];
  // Print value in reverse order
  for (int i = n - 1; i >= 0; i--) cout << arr[i] << '\n';
  // Free array.
  delete [] arr;
  return 0;
}
```

Errors in handling dynamic memory

Correct use of `new` and `delete`

Two crucial rules:

- ("Free after use") Every memory segment allocated with `new` must eventually be set free using `delete`. Otherwise the memory segment will become a **memory leak**, that is, memory that is no longer used by the program but that it still allocated.
- ("No use after free") A freed up memory segment must not be used after being freed up. Any reference to a freed up memory segment is called a **dangling reference** and its use is called a **use-after-free** error.

Also important:

- **Buffer overflows** (as one would expect!) result from accessing dynamically allocated arrays beyond their boundaries.
- **Double-free anomalies**: memory segments should not be freed up more than once.

Memory leaks

```
// Memory is allocated
int n = ...;
int* arr = new int[n];
// Do something with arr
arr[0] = 1;
...
// delete not ever called for arr ...
```

No free after use! Program does not release dynamically allocated memory.

The accumulation of memory leaks may eventually cause the program to run out of memory.

Or maybe not :) ... read a funny story here

Memory leaks (cont.)

```
// Memory is allocated
int n = ...;
int* arr = new int[n];
// Do something with arr
arr[0] = 1;
...
// delete not ever called for arr to free memory ...
```

ASan detects leaks ...

```
Direct leak of 12 byte(s) in 1 object(s) allocated from:
#0 0x7fea2b077097 in operator new[](unsigned long) ...
#1 0x4010bd in main /home/runner/examples-12/06/x.cpp:2

SUMMARY: AddressSanitizer: 12 byte(s) leaked in 1 allocation(s
```

Dangling references

```
  // Memory is allocated
  int n = ...;
  int* arr = new int[n];
  ...
  // Memory is freed up
  delete [] arr;

  // arr is at this point a dangling reference
  // Memory is used after being freed up.
  arr[0] = 123;
}
```

Use after free! Program uses already released memory, called a dangling reference.

Dangling references (cont.)

```
  // Memory is allocated
  int n = ...;
  int* arr = new int[n];
  ...
  // Memory is freed up
  delete [] arr;

  // arr is at this point a dangling reference
  // Memory is used after being freed up.
  arr[0] = 123;
}
```

ASan detects use-after-free anomalies too.

```
AddressSanitizer: heap-use-after-free on address ...
    #0 0x401176 in main /home/runner/examples-12/06/x.cpp:5
```

Double-free anomalies

```
  // Memory is allocated
  int n = ...;
  int* arr = new int[n];
  ...
  // Memory is freed up more than once
  delete [] arr;
  delete [] arr;
}
```

```
AddressSanitizer: attempting double-free ...
```

Heap-buffer overflows

Buffer overflows can occur too ...

```
int n = ...;
int* x = new int[n];
x[n] = 0;
```

```
AddressSanitizer: heap-buffer-overflow ...
```

Case-study example: a simple vector

Outline

`simple_vector` data type: a container for a sequence of `int` elements with growable size (a sort of "growable array"):

```
struct simple_vector { ... };
simple_vector* build(int n);
void destroy(simple_vector* sv);
void add(simple_vector* sv,  int x);
int size(const simple_vector* sv);
int get(const simple_vector* sv, int i);
```

- `build(n)` returns a new `simple_vector` with capacity for `n` elements;
- `destroy(sv)`: frees up the memory associated to `sv`;
- `add(sv, x)`: adds value `x` to the end of the vector, growing the internal capacity by a factor of 2 if needed;
- `size(sv)`: get number of elements stored in (added to) `sv`;
- `get(sv, i)`: gets value stored in `sv` at position `i`;

Intended use

Illustration of intended use:

```
simple_vector* sv = build(2);
add(sv, 2);
add(sv, 5);
add(sv, 1); // grows capacity to 4
add(sv, 3);
for (int i = 0; i < size(sv); i++)
    cout << get(sv, i) << '\n';
destroy(sv);
```

```
2
5
1
3
```

The data structure

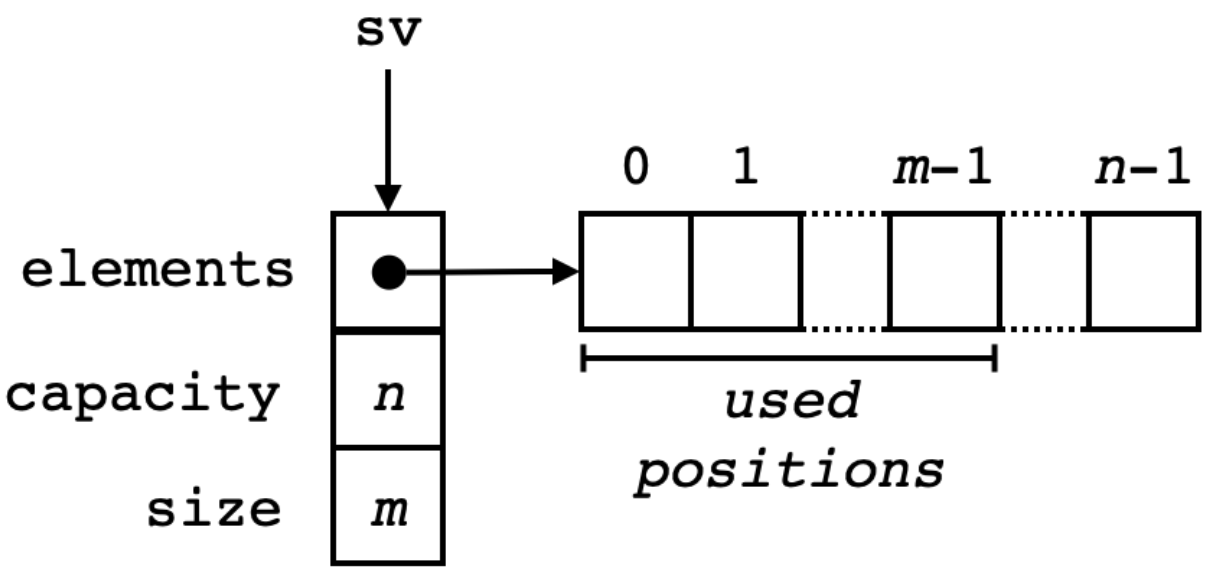
```
struct simple_vector {
  int* elements;
  int capacity;
  int size;
};
```

Fields:

- `elements`: dynamically allocated array of elements;
- `capacity`: the current capacity, i.e., the length of the array;
- `size`: the number of elements actually stored, i.e., array positions effectively used so far (`size <= capacity`);

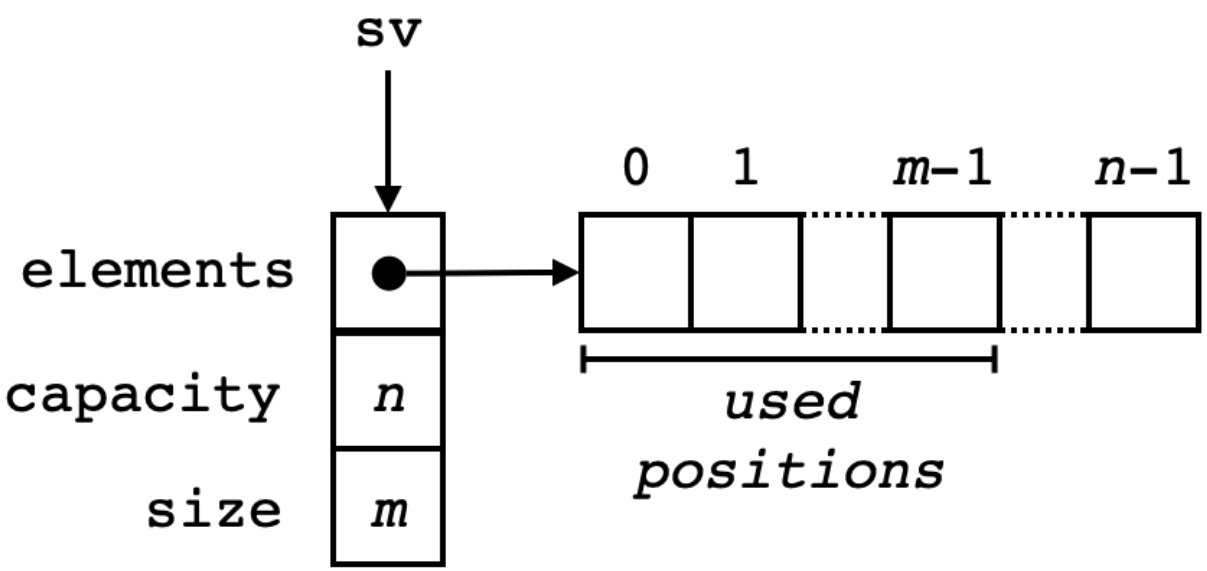
The data structure (cont.)

```
struct simple_vector {
  int* elements;
  int capacity;
  int size;
};
```



The more obvious functions

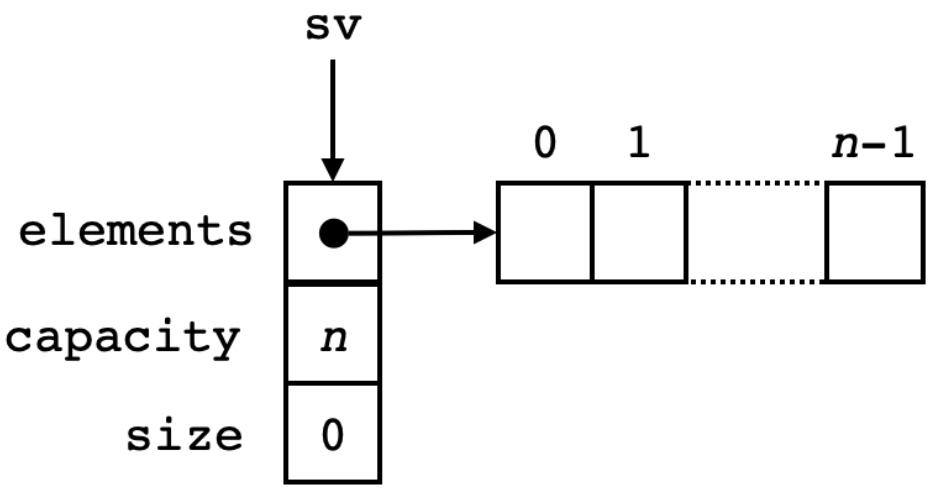
```
int size(const simple_vector* sv) {
  return sv->size;
}
int get(const simple_vector* sv, int i) {
  return sv->elements[i];
}
```



Construction

```
simple_vector* build(int n) {
  simple_vector* sv = new simple_vector;
  sv->elements = new int[n];
  sv->capacity = n;
  sv->size = 0;
  return sv;
}
```

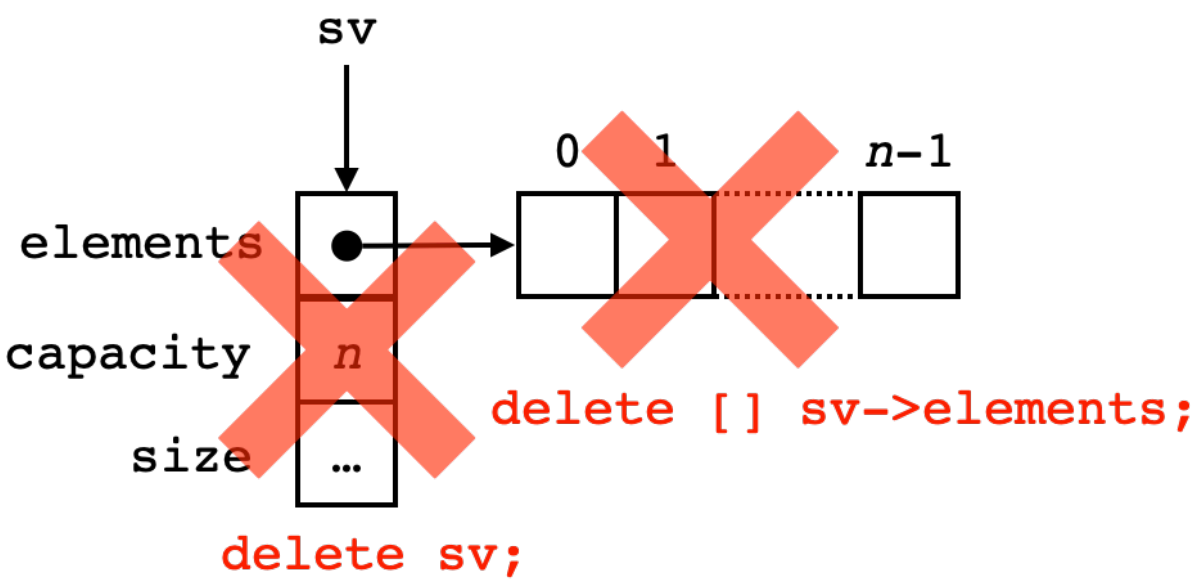
`build(n)` allocates memory for the new vector `sv` and then for `sv->elements` (an array of size `n`).



Destruction

```
void destroy(simple_vector* sv) {
  delete [] sv->elements;
  delete sv;
}
```

In symmetry, `destroy(sv)` sets free the memory for `sv->elements` and then for `sv`.



Destruction (cont.)

```
void destroy(simple_vector* sv) {
  delete [] sv->elements;
  delete sv;
}
```

We must execute two `delete` instructions in correspondence to the two allocations using `new`. If we forget any of them, we will have **memory leaks**.

Their order is also important. Inverting the order shown would lead to a **use-after-free** situation: freeing `sv` first will make `sv` a dangling reference in the access to `sv->elements`.

Finally, we should not repeat any of the `delete` instructions, as we would have a **double-free** anomaly.

Growing the vector

```
void add(simple_vector* sv, int x) {
  if (sv->capacity == sv->size) {
    // vector is full ... need to grow elements array
    . . .
  }
  sv->elements[sv->size] = x;
  sv->size++;
}
```

`add(sv,x)` needs to grow internal array `sv->elements` when the capacity is exceeded, i.e., `sv->size == sv->capacity`.

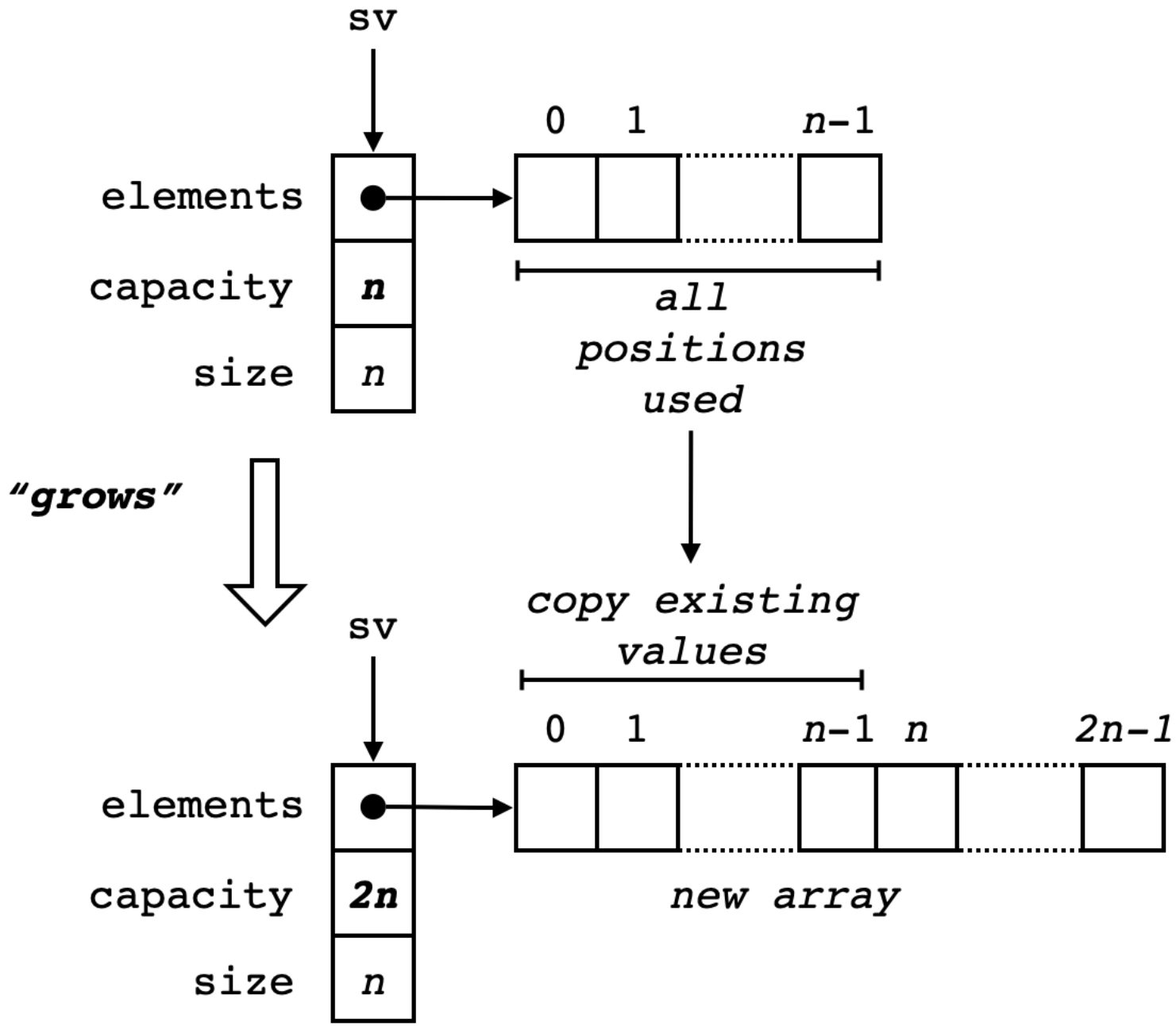
Otherwise, a buffer overflow will occur when executing `sv->elements[sv->size] = x;` …

Growing the vector (cont.)

The "growing part":

```
void add(simple_vector* sv, int x) {
  if (sv->capacity == sv->size) {
    // Create new array
    int new_capacity = 2 * sv->capacity;
    int* new_array = . . . ;
    // Copy elements from old to new array
    . . .
    // Free up memory for old array
    . . .
    // Point to new array
    sv->elements = new_array;
    sv->capacity = new_capacity;
  }
  sv->elements[sv->size] = x;
  sv->size++;
}
```

Growing the vector (cont.)



Growing the vector (cont.)

```
void add(simple_vector* sv, int x) {
  if (sv->capacity == sv->size) {
    // Create new array
    int new_capacity = 2 * sv->capacity;
    int* new_array = new int[new_capacity] ;
    // Copy elements from old to new array
    for (int i = 0; i < sv->capacity; i++)
      new_array[i] = sv->elements[i];
    // Free memory for old array
    delete [] sv->elements;
    // Point to new array
    sv->elements = new_array;
    sv->capacity = new_capacity;
  }
  sv->elements[sv->size] = x;
  sv->size++;
}
```