# C++ 1 Les 2

Memory Management

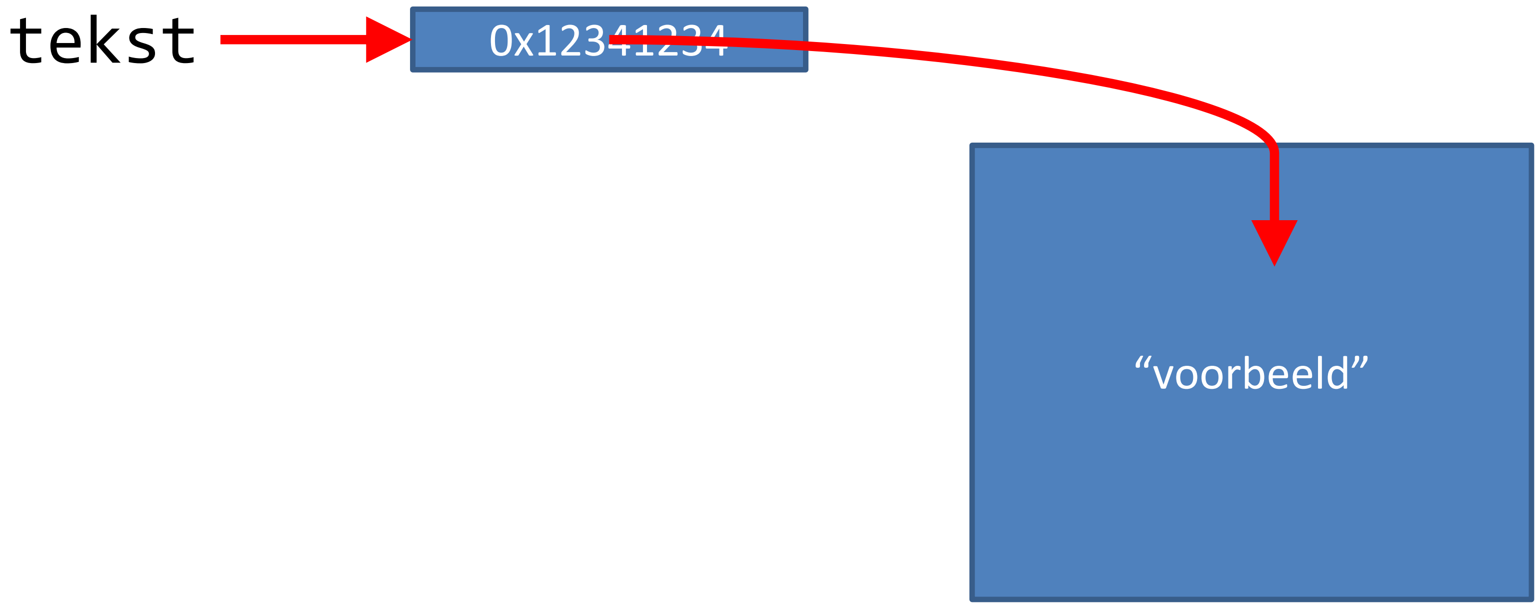
## Leerdoelen

Aan het eind van deze workshop kun je:

- het verschil tussen memory management van C++ en memory management in Java/C# uitleggen.
- uitleggen wat het verschil is tussen code, global, heap, en stack geheugen.

## Voorbeeld Java

```
String tekst = new String("voorbeeld");
```



# Memory Management Java/C#

- Automatic garbage collection
- Minder efficient(?)
- Neemt net iets meer geheugen in

## Memory Management in C++

- Niet automatisch
- Zelf aanmaken
- Zelf vrijgeven

## Geheugenindeling

- Code Space + Registers
- Globals
- Stack
- Heap

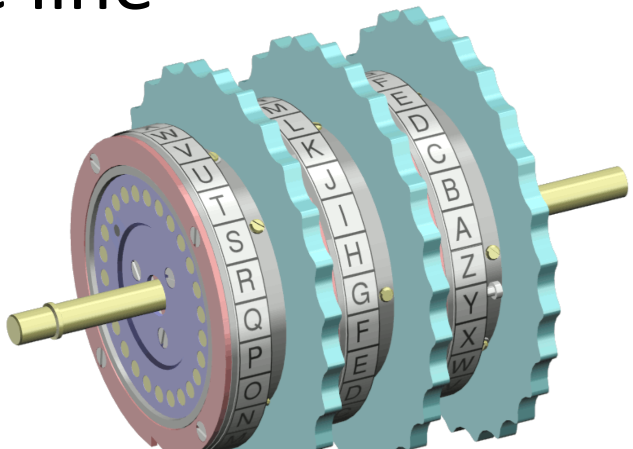
# Code Space + Registers

Code Space:

- Het gecompileerde programma

Registers:

- Pointer naar uit te voeren code line
- Stack Pointer
- e.a.



# Globals

- Globale variabelen
- Statische variabelen

Lelijk, probeer te vermijden!

## Heap

```
void fun()
{
  int* i = new int;
  char* c = nullptr;

  c = new char;

  delete i;
}
```

# Heap

- Maken met: `new`
- Verdwijnt wanneer:
  - Programma eindigt
  - `delete` of `delete[]` wordt aangeroepen
- *bad_alloc* exception indien geen geheugen meer over

# Heap

Bij elke `new` hoort een `delete`!

Anders: Memory Leaks

## Stack

```
char fun(int i)
{
  string str;
  float f[100];

  return 'A';
}
```

## Stack

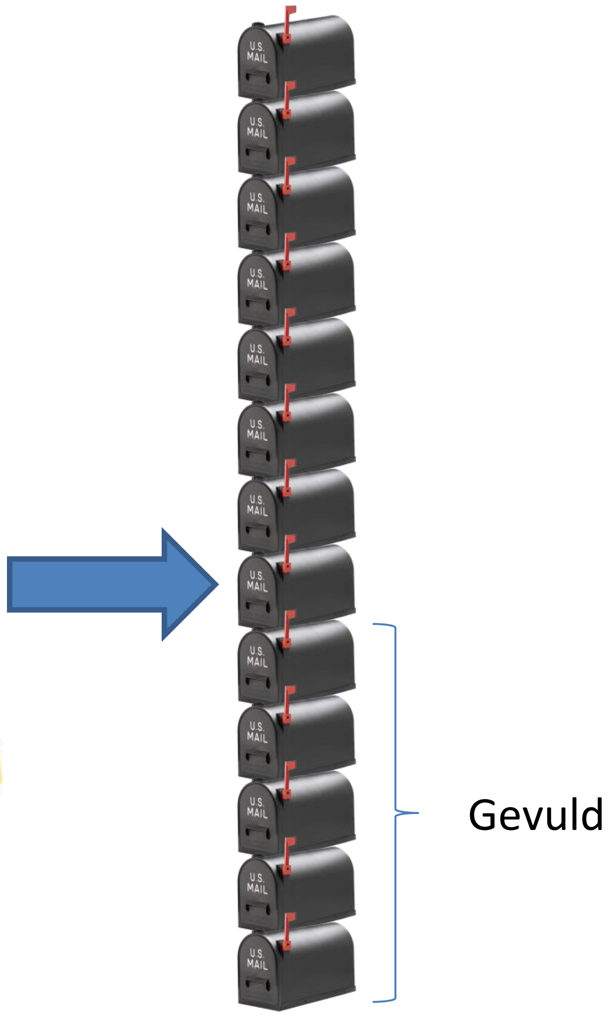
- Last in – first out
- Beperkte hoeveelheid stack geheugen
- *stack overflow* exception/signal (afhankelijk van OS) indien geen stack geheugen meer over

## Stack

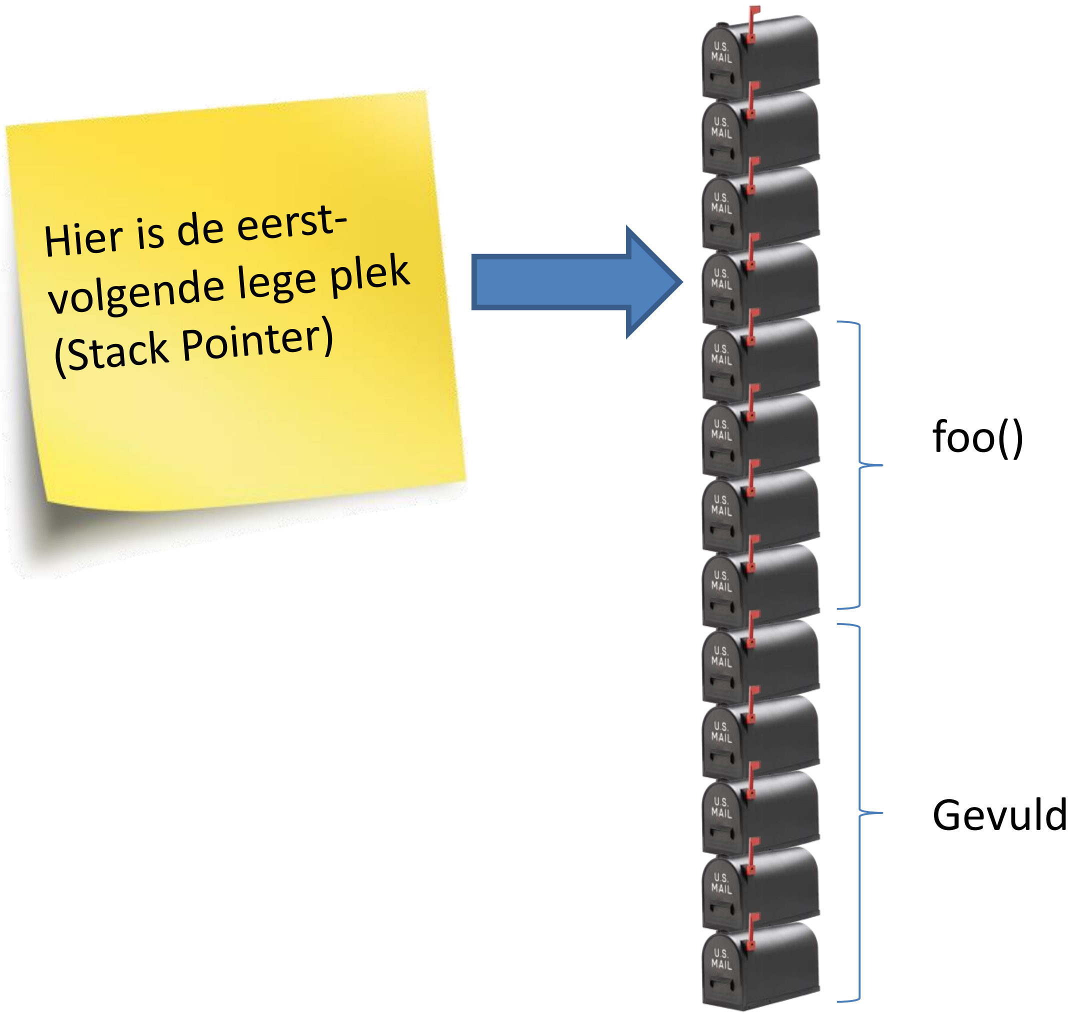
Mailbox = Geheugen
(bevat een variabele)

# Stack

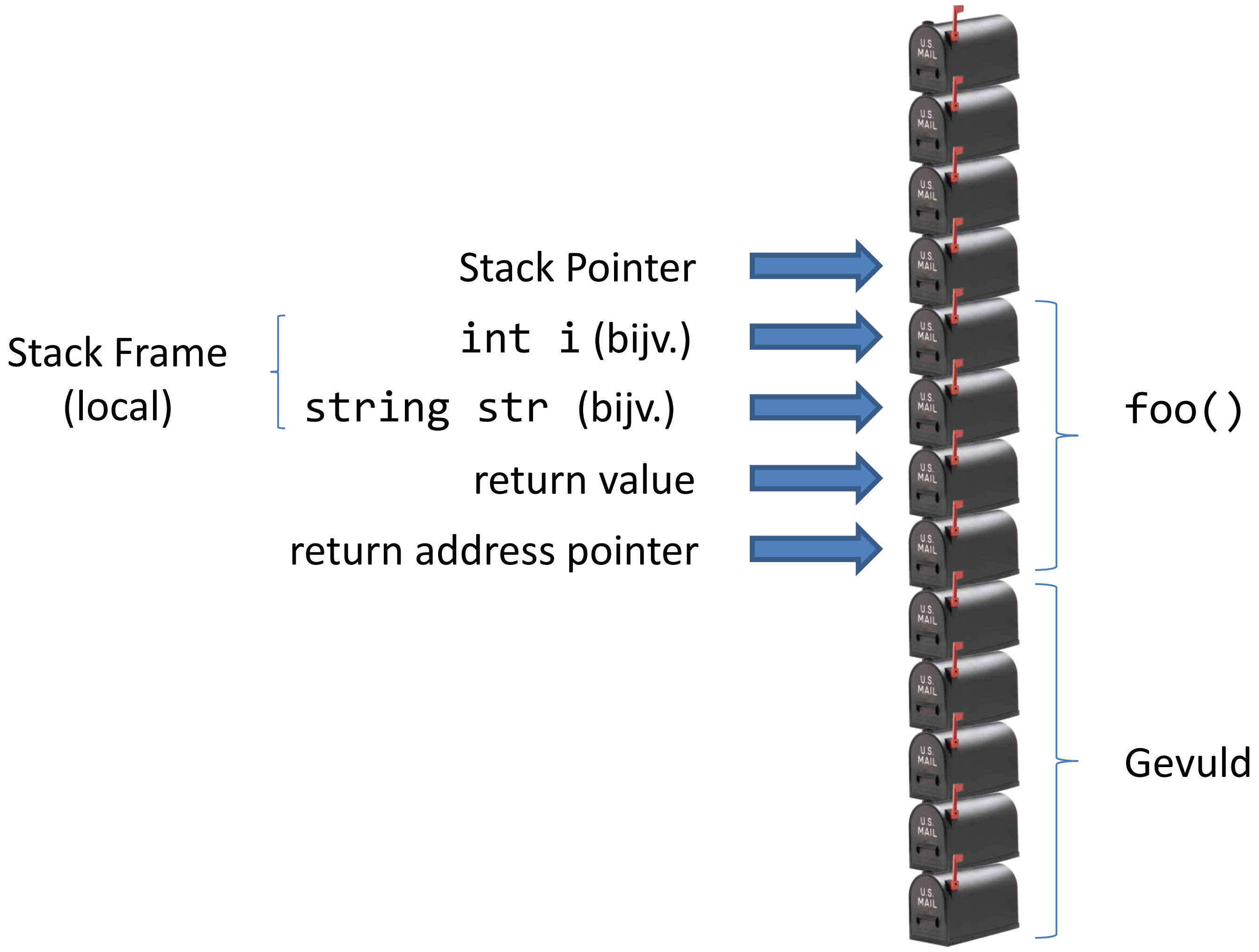
Hier is de eerst-volgende lege plek (Stack Pointer)



# Stack


Hier is de eerst-volgende lege plek (Stack Pointer)
foo()
Gevuld

# Stack



Stack Pointer
Stack Frame (local)
int i (bijv.)
string str (bijv.)
return value
return address pointer
foo()
Gevuld

# Stack (function is called)

1. The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
2. Room is made on the stack for the function’s return type. This is just a placeholder for now.
3. The CPU jumps to the function’s code.
4. The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered “local” to the function.
5. All function arguments are placed on the stack.
6. The instructions inside of the function begin executing.
7. Local variables are pushed onto the stack as they are defined.

# Stack (function terminates)

1. The function’s return value is copied into the placeholder that was put on the stack for this purpose.
2. Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
3. The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn’t assigned to anything, no assignment takes place, and the value is lost.
4. The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.