# 1INF25
# 2025-1
# H-0583
# Mg. Erasmo Gómez

# About this presentation

| | | |
|---|---|---|
| #1 | All about memory allocation | Introduction. |
| #2 | Functions | what, where, when, why, how ... |
| #3 | Operator Overloading | applications in cpp |

# All about memory allocation

# Main program

## 01

### Stack

Contiguous allocation

```
int main() {
    int a = 3, b = 75, c;
    c = f( x: a, y: b);
    ....
    ...
    return 0;
}
```

## 02

### Heap

Allocation during execution of instructions

## 03

### Identifiers

Each value for memory allocation

## 04

### Step by Step

Executed step by step and calls each function involved

# Call a function

## 01

### Separate thread

Each call has its own memory allocated in stack

## 02

### Referenced

All variables inside functions are allocated on heap.

```
int f(int x, int y){
    int t;
    t = x + y;
    return t;
}
```

## 03

### Life is short

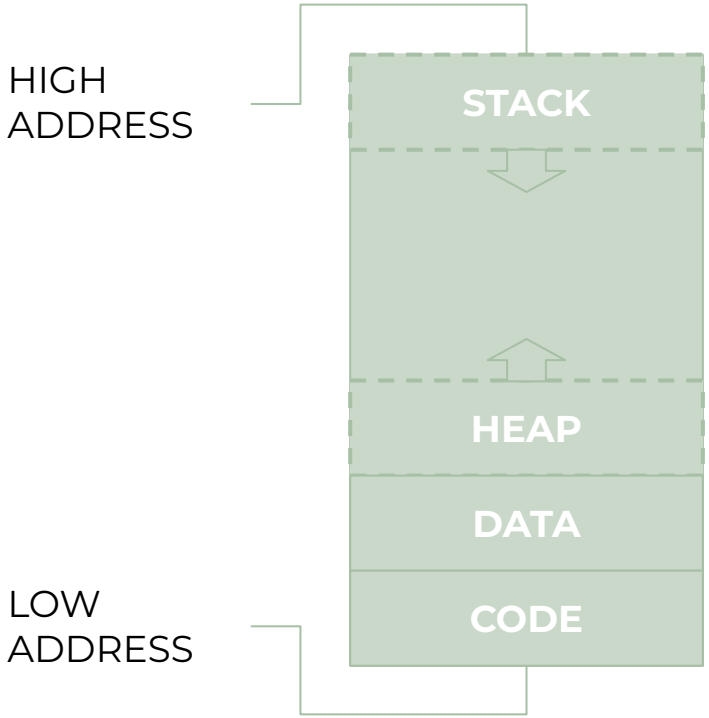
All inside dies when the block is finished executing

## 04

### Duplicity

As reference some memory are duplicated

# How a program is stored in memory?




Local variables and parameters stored here: Grows as programs call functions, shrinks on return from function.

Dynamically allocated memory : grows as program allocate memory

Global variable stored here

Functions instructions stored here

# Arrays and pointers ...

01

## Array declaration

int arr[5] ... arr[0]..arr[4]
Int arr[5] = {1, 2, 3, 4, 5};
Int arr[5]{1, 2, 3, 4, 5};

02

## Pointers

Int *p;
P = new int[5];
P = new int[5]{1,2,3,4,5};

03

## Memory

Variable's memory address is stored independently than values

04

## By Value

void ByValue(int *x)

05

## By Reference

void ByReference(int *&x)

06

## After

ByValue *x still has the value
ByReference *x replace the original value

# Function parameters

| | By Value | By Reference |
|---|---|---|
| **Invocation:** | f(a); | f(a); |
| **Headers: .hpp** | **T** f(int); | **T** f(int &); |
| **Sources: .cpp** | **T** f(int a){<br>…<br>} | **T** f(int &a){<br>…<br>} |

f: Function name
T: Returned data type
a: Parameter name

# Default values to call a function

| | |
|---|---|
| **Headers:** | **T** f(int = 10, int = 7); |
| **Sources:** | **T** f(int a, int b){<br>…<br>} |
| **Invocation:** | f(a,b);<br>f(a);<br>f(); |

f: Function name
T: Returned data type
a: Parameter name

# Let’s code!

# Function and Operator **overloading**

# Let's suppose we have defined these two functions

```
int f(int a){
    return a + 5;
}
```

```
int f(int a){
    return a * 10;
}
```

# What would happen if at some point in the program we do ...

```
b = f(10);
```

reference to **f** is ambiguous

# but about now?

```
int f(int a){
    return a + 5;
}
```

```
int f(int a, int b){
    return a * 10;
}
```

# What would happen if at some point in the program we do ...

```
b = f(10);

b = f(10,8);
```

Execute 1st function

Execute 2nd function

# the same for:

```
int f(int a){
    …
}
int f(int *a){
    …
}
```

```
int f(struct St a){
    …
}

int f(class Cl a){
    …
}
```

# Function Overloading

## Definition

Property of CPP that allows define two or more functions with the same name

## Require:

Parameters must be different whether by number or data type

# Cpp interpret any expression as follow:

Operand1 **Operator** Operand2

a + b

**Operator** (Operand1, Operand2)

**We have functions right there!**

## We can overload:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ++ | -- | + | - | * | / | % | \| |
| & | ^ | ~ | >> | << | == | != | > |
| >= | <= | not | and | or | = | += | -= |
| *= | /= | %= | \|= | &= | *= | ^= | >>= |
| <<= | [] | () | -> | new | | delete | |

## We can’t overload:

| | | |
|---|---|---|
| :: | . | ?. |

# Restrictions for Operator Overloading:

- We can’t create new operators.  Like @.
- We can’t change the operator’s priority
- We can't change the associative property.
- We need to respect unary operators.
- Zero tolerance about ambiguity.

# Let’s code!

# Summary

## Memory Allocation

- By executing a program we **reserve** memory in a **clever** way.
- **Static** allocation in stack and **dynamic** allocation in heap

## Functions

- We can **overload** functions so that they **behave** as we want.
- Passing parameters as **value** and as **reference** is **key** to **practice**.

## Operators

- We have so many operators that we can **overload**, we need to respect **how they were constructed** in order to maintain **congruence**.

# Thanks