# System Software Crash Couse

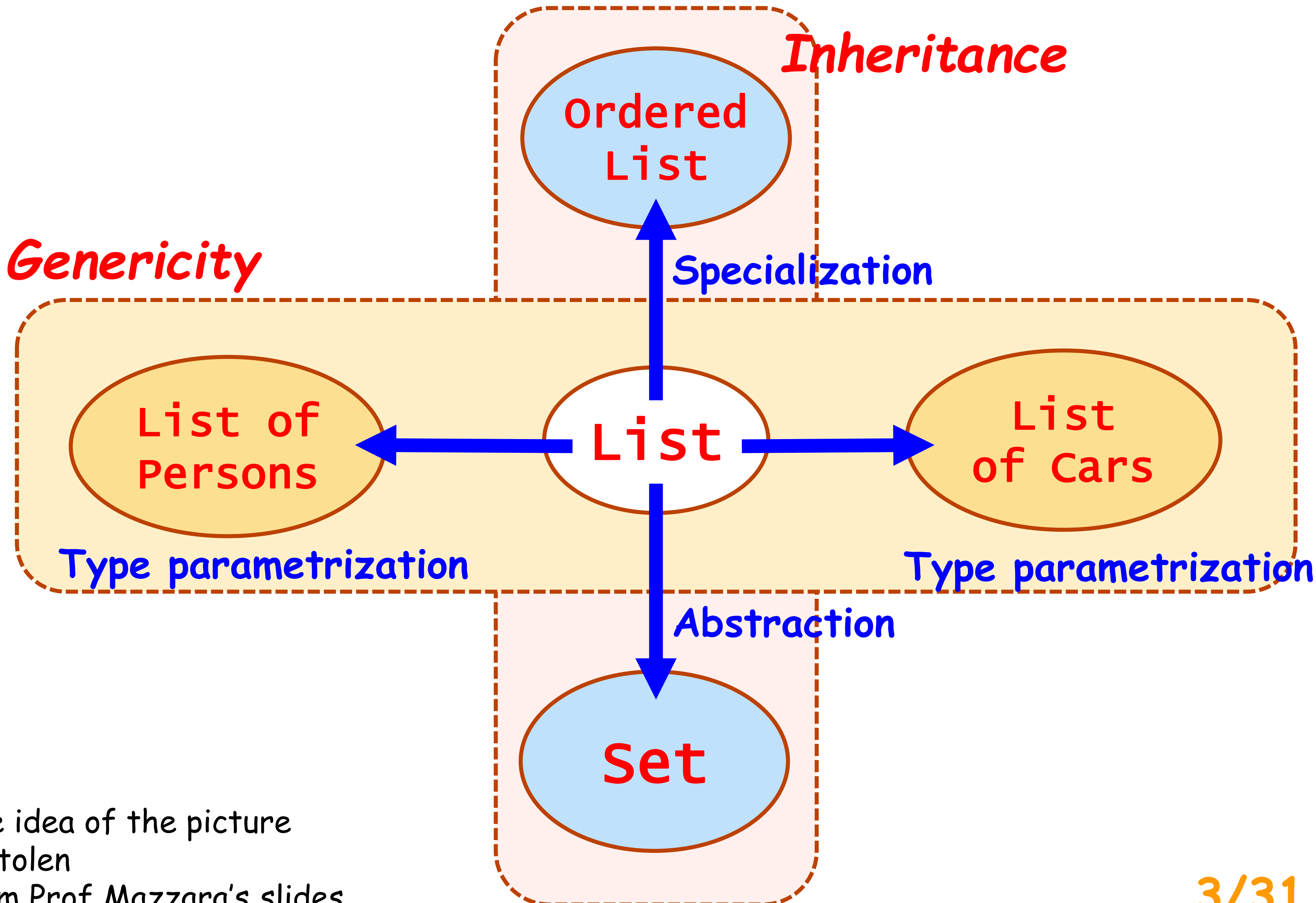
## Samsung Research Russia
## Moscow 2019

## Block G: Advanced C++
## 3-2. Introduction to Generics

Eugene Zouev

# Generic Programming and C++ Templates: Introduction

# Generic Programming: the Underlying Idea




The idea of the picture
is stolen
from Prof Mazzara's slides

# Generic Programming

- An immanent part in any modern language: generic data structures (containers, collections) and generic algorithms are powerful development.
- **What to parametrize**: data or algorithms? (usually both)
- **By what to parametrize**: by types only (C#, Java, Swift) or by types and by values? (Ada, C++)
- What are **requirements** on actual parameters? No requirements at all (C++, but soon will appear) or explicit and efficiently checked requirements (C#, Eiffel, Ada)?

**C++ has the most powerful and comprehensive support for generic programming.**

# The very first example

```
int Max ( int a, int b )
{
    return a>b ? a : b;
}
```

Problems if a program is (really) large:

- Several functions for calculating maximum for **different** parameter **types** (`float`s, `double`s etc.).
- Functions for calculating "maximum"for **user-defined types**.

  - Small independent program:
    we **don't need** any programming technique at all
  - Large and complex program:
    we need a very **powerful** and **advanced** programming technique

# The very first example

**Straightforward solution:**

Define specific Max functions for **all types** used in the program.

```
double Max ( double a, double b )
```

```
float Max ( float a, float b )
{
    return a>b ? a : b;
}
```

float, double are standard types

```
UT2 Max ( UT2 a, UT2 b )
```

```
UT1 Max ( UT1 a, UT1 b )
{
    return a>b ? a : b;
}
```

UT1, UT2 are user-defined types

**Is it a good solution?**

- **Very hard** to maintain (i.e., to test & debug, to prove correctness, to modify etc.).
- **Impossible**: type may be **unknown** beforehand (e.g. in a case of a library).

# The very first example

Let "T" denote "any type"

Keywords

Template **Type Parameter**

```
template < typename T >
T Max ( T a, T b )
{
    return a>b ? a : b;
}
```

Template **Header**

Template **Body**

**Note: "angle brackets" are used to enclose template parameters**

# Using function templates

The general question:

How to use function template?

The general answer:

**Just like ordinary functions!**

<u>Example</u>:
Use Max function template to calculate maximum of two values of type **double**:

```
template < typename T >
T Max ( T a, T b )
{
      return a>b ? a : b;
}

double x, y, res;
// assigning some values to x & y

res = Max(x-1,y+2.5);
```

# Template instantiation:how it works

**Function template**

```
template < typename T >
T Max ( T a, T b )
{
        return a>b ? a : b;
}
```

**Original call**

```
res = Max(x-1,y+2.5);
```

*Compiler takes the types of the actual arguments (suppose* `double`*)*

**Template Instantiation**

*Compiler-generated function with a "mangled" name*

double $\mathtt{Max}_{\mathtt{double}}$( double a, double b )<br>
{<br>
        return a>b ? a : b;<br>
}

**Function-by-template**

*Compiler-generated function call*

**Result call**

res = $\mathtt{Max}_{\mathtt{double}}$(x-1,y+2.5);

# Template instantiation:how it works

**Conclusion**:

- All the job is performed **automatically** by a C++ compiler (this is the requirement of the Standard, i.e., this is the part of the language's semantics).
- Compiler decides which actual types to substitute to the function-by-template instead of formal type(s) by analyzing the **argument types from the call**.

# Template instantiation:how it works

**Remarks**:

- Template instantiation for a specific type is performed **only once**.

  Example: for the following two calls

  ```
  res = Max(x-1,y+2.5);
  res = Max(1.0,res);
  ```

  the compiler will generate only one copy of $\texttt{Max}_{\texttt{double}}$ function, and will use it to generate code for both calls.

- Template instantiation is performed **for every specific set** of the actual types.

  Example: for the following two calls

  ```
  res = Max(x-1,y+2.5);
  int k = Max(1,(int)res);
  ```

  the compiler will generate *two* functions-by-template: $\texttt{Max}_{\texttt{double}}$ for the first call, and $\texttt{Max}_{\texttt{int}}$ for the second one.

# Template instantiation: steps

`f(actual-arguments)`

1. If f is the **usual function**, then go to 3.
2. If f is the **function template** then

   2.1 **Determine the set** $\{T_i\}$ where $T_i$ is the type of the i-th *actual-argument* of the call.

   2.2 If the function-by-template of form $f_{\{Ti\}}$ already **exists** then go to 3.

   2.3 **Generate function-by-template** $f_{\{Ti\}}$ using type $T_i$ as the substitution for the i-th formal type for the f template.

   2.4 **Compile** the function-by-template $f_{\{Ti\}}$ generated on the previous step.
3. **Generate code** for the function **call**.

# Template instantiation: how it works

**Remarks** *(continued)*:

- Template instantiation may cause **problems**.
  Example: separate (independent) compilation

T.h
```
template<typename T>
T Max ( T a, T b )
{
   return a>b?a:b;
}
```

File1.cpp
```
#include "T.h"
 . . .
res = Max(x-1,y+2.5);
 . . .
```

File1.obj

Object code with $\texttt{Max}_{\texttt{double}}$

File2.cpp
```
#include "T.h"
 . . .
res = Max(1.0,res);
 . . .
```

File2.obj

Object code with $\texttt{Max}_{\texttt{double}}$

App.exe

**Executable with two copies of** $\texttt{Max}_{\texttt{double}}$

- Both compilations produce **the same** function-by-template
- Executable contains **two copies** of $\texttt{Max}_{\texttt{double}}$ ("**code bloat**")

# Templates: requirements on types

Taking the very first example again…

```
template < typename T >
T Max ( T a, T b )
{
   return a>b ? a : b;
}
```

Let's introduce a user-defined type…

```
class C {
   int m;
public:
   C(): m(0) { }
};
```

…and apply our template to (the objects of) this type:

```
C c1, c2;
...Max(c1,c2)...
```

What will happen?

# Templates: requirements on types

Let's consider the function-by-template the compiler generates while processing the call `Max(c1,c2)`:

```
// Generated by compiler
C Max_C ( C a, C b )
{
   return a>b ? a : b;
}
```

**Incorrect** function

```
binary '>':
'class C' doesn't define
this operator or...
```

```
class C {
   int m;
public:
   C(): m(0) { }
   bool operator > ( C& c )
   {
      return m > c.m;
   }
};
```

Now $\mathtt{Max}_C$ works.

Add the '>' operator to our C class:

# Function templates: summary

**Max<T>()**

Not a function but a **function template**:
- family of (overloaded) functions, or
- function pattern

*Template Instantiation*

- $\texttt{Max}_{\texttt{int}}()$
- $\texttt{Max}_{\texttt{float}}()$
- $\texttt{Max}_{\texttt{c}}()$

Template **instantiations**:
Concrete functions

**Requirement on actual types**:
Should implement > operator

# Assignment

- Consider the following function:

```
void alignArray ( int* array, int size, int barrier )
{
    for ( int i=0; i<size; i++ )
    {
        if ( array[i] < barrier ) array[i] += 2;
        else if ( array[i] > barrier ) array[i] -= 2;
    }
}
```

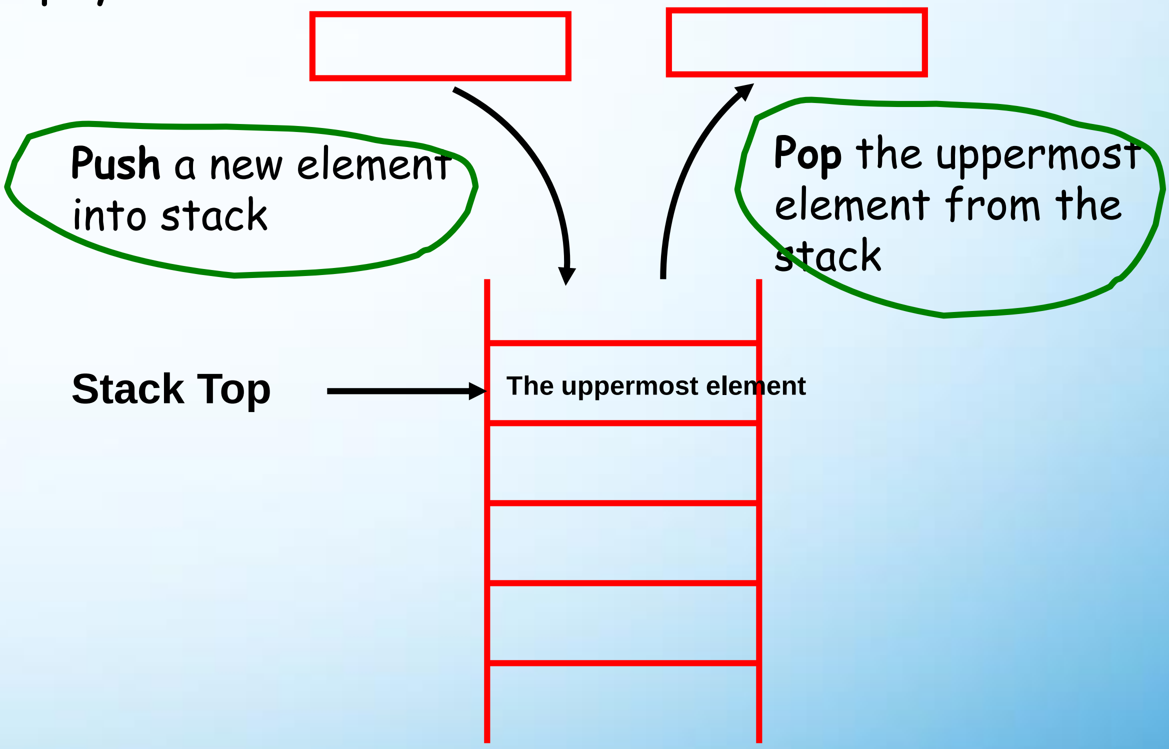
- Using this function as a basis, write a **function template** that can work with arrays of **any type**.
- Declare a class that satisfies the requirements on actual types from your template.
- Write an example: declare the array of class objects and pass it to the function. Show both initial and final states of the array.

# Class Templates: The First Example

## <u>Stack</u>, or LIFO Memory

Usual set of the operators on stack:

- **Push** a new element (value) into stack
- **Pop** the uppermost element from the stack
- **Check** if the stack is empty
- . . .



# Class Templates: The First Example

Implementation in C++: **non-template case**
Stack of integer values (array-based implementation)

```
class Stack
{
   // implementation
   int top;
   int S[100];
public:
   // interface
   Stack() : top(-1) { }
   void push ( int V )
        { S[++top] = V; }
   int pop (void)
        { top--; return S[top+1]; }
   // other operations
   . . .
}
```

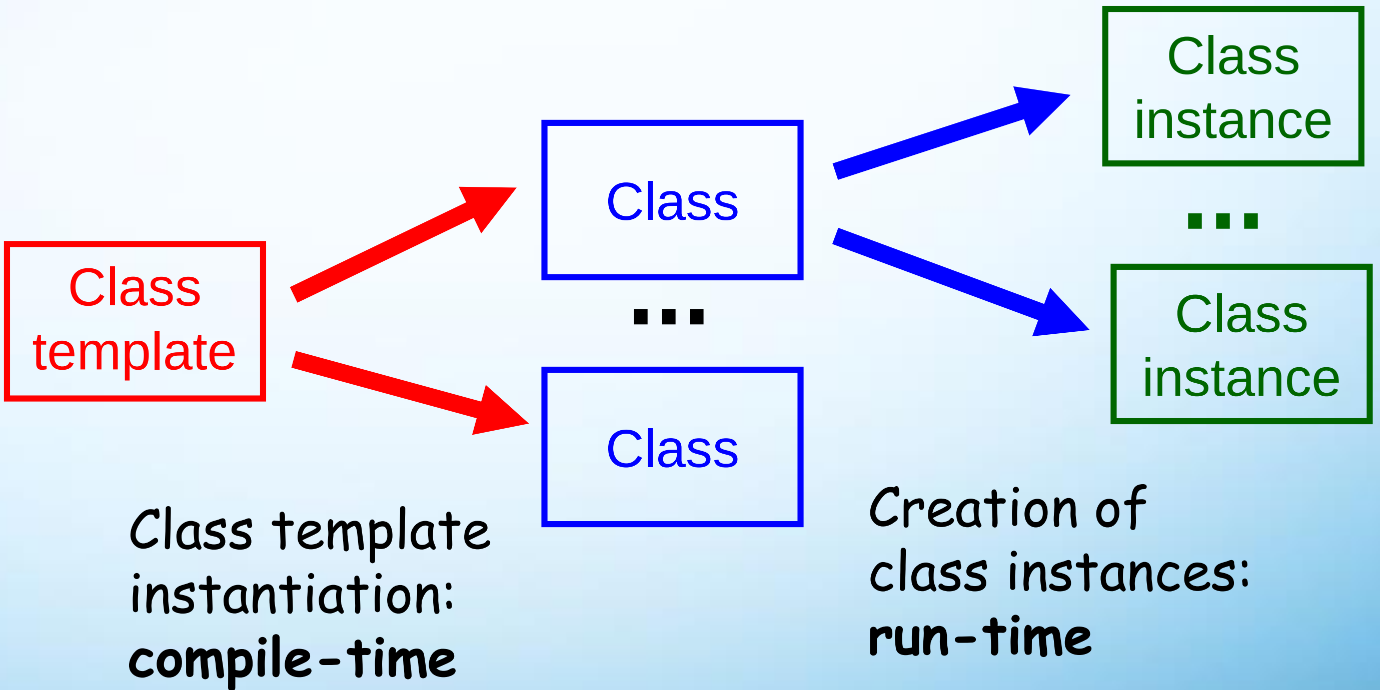
Implementation in C++:
**solution with templates**
Stack of objects of **any type**
(array-based implementation)

```
template < typename T >
class Stack
{
   // implementation
   int top;
   T S[100];
public:
   // interface
   Stack() : top(-1) { }
   void push ( T V )
      { S[++top] = V; }
   T pop (void)
      { top--; return S[top+1]; }
      // other operations
      . . .
}
```

Are there any **design lacks** in this implementation?

# Class Templates

- A class is a type (Std, Chapter 9)
- A class template *is not* a type;
  it is a family of types
- A function template *is not* a function;
  it is a family of (overloaded) functions



# Class Templates

**Almost the same problems as for `Max` template:**

- How **to use** the `Stack` template?
  (I.e., how to make classes from the class template?)

  **Class template instantiation**

- Which are **the requirements** on the actual types?

  **Appropriate set of operators**

- **Optimizing** the template

  **How to weaken the requirements on the actual types?**

- **Improving** the template

  **More generality!**

# Class Template Instantiation (1)

## Template Instantiation: Conceptual Scheme

Class Template C with Formal Type Parameters $T_1 \ldots T_n$

Actual Type Parameters $T_1 \ldots T_n$

**Class by Template**
C $T1 \ldots Tn$

## Template Instantiation: Example and Syntax

Class Template Stack with Formal Type Parameter T

Actual Type Parameter int

**Class by Template**
Stack<int>

# Class Template: Instantiation & Use

```
template < typename T >
class Stack
{
    // implementation
    . . .
public:
    // interface
    Stack() : top(-1) { }
    void push ( T v ) { S[++top] = v; }
    T pop (void) { top--; return S[top+1]; }
}
```

Template Declaration

*Template Name*

```
Stack<int>
```

Template Instantiation

*Actual type*

# Class Template: Instantiation & Use

*Declaration*

```
Stack<int> sint;
```

*Type Specifier* *Declarator*

Object Declaration

**More examples of template**

```
Stack<float> sf1, sf2;

Stack<int>* ArrayOfStacks[10];

typedef Stack<double> SD;
SD sd1, sd2;
```

**How to use objects of class-by-template type:**

```
Stack<int> s;
. . .
s.push(1);
. . .
int from_top = s.pop();
```

# Class Template: Instantiation & Use

- Class template is not a type but a family of types; we should instantiate the template before using it as a type.
- Instantiating the class template means generating the new class-by-template by replacing formal types for corresponding actual types.
- We should always use the notation of explicit template instantiation in order to make a class-by-template:

```
template-name < actual-types >
```

  (there is *no implicit instantiation* for class templates).
- After instantiating the construct of template instantiation denotes a class-by-template generated by compiler.
- Class-by-template behaves just as an ordinary class and may be used everywhere as a type.

# Requirements on Actual Types (1)

```
template < typename T >
class Stack
{
   // implementation
   . . .
public:
   // interface
   void push ( T V ) { S[++top] = V; }
   . . .
}
```

**Which operators** on formal type are used within the template?

Passing an object of type T as parameter:
**Copy constructor**

Copying one object of type T to another:
**Assignment operator**

# Requirements on Actual Types (2)

**Conclusion**:

- To apply the `Stack` template to a user-defined type (i.e., to create a stack of class-type objects) we should provide copy constructor and assignment operator in the class.

  In other words, the requirement on the actual type from the `Stack` template is that the actual type should always have
  - **public copy constructor** and
  - **public assignment operator**.

**Problems** with copy constructor:

- Copying objects while passing them as parameters may be **time-consuming action.**
- Copying objects while passing them as parameters may be simply **impossible**:

```
class C  {
   C ( const C& c ) = delete;
   . . .
};
```

Here class developer **doesn't allow** objects of type C to be passed as parameters.

# Requirements on Actual Types (4)

Hence there are **direct reasons** to weaken the requirement on the actual type from the Stack template.

```
template < typename T >
class Stack
{
    // implementation
    . . .
public:
    // interface
    void push ( T& v ) { S[++top] = v; }
    . . .
}
```

Now ***not a value*** but the ***reference to a value*** is passed as a parameter (*passing-by-reference* in some other languages);
The copy constructor ***will not be invoked*** while passing the reference.

# Requirements on Actual Types (5)

Hence there are **direct reasons** to weaken the requirement on the actual type from the Stack template.

```
template < typename T >
class Stack
{
    // implementation
    . . .
public:
    // interface
    void push ( T& v ) { s[++top] = v; }
    T pop (void) { top--; return s[top+1]; }
    . . .
}
```

C++: returning a value from a function normally means ***copying it***…

How to overcome this problem?
**That's your task ☺**

# Improving Template: Non-type Pars

```
template < typename T >
class Stack
{
    // implementation
    int top;
    T S[100];
public:
    // interface
    Stack() : top(-1) { }
    void push ( T& V )
      { S[++top] = V; }
    T pop (void)
      { top--; return S[top+1]; }
    // other operations
    . . .
}
```

**Is it possible to make the template *more generic*?**

**To parametrize not only the *type* of stack elements but its *size* as well!**

```
template < typename T, int N >
class Stack
{
  // implementation
  int top;
  T S[N];
public:
  // interface
  Stack() : top(-1) { }
  void push ( T& V )
    { S[++top] = V; }
  T pop (void)
    { top--; return S[top+1]; }
  // other operations
  . . .
}
```

Template non-type parameter's syntax has the form of an ordinary **declaration** (or, which is the same, the form of an ordinary **function parameter declaration**).

# Improving Template: Non-type Pars

How to use the template with non-type parameter(s):
Stack of (maximum) *ten integers*:

```
Stack<int,10> s10int;
```

*The list of type and/or non-type template actuals separated by commas and enclosed by angle brackets.*

**Common requirement**:

- Template arguments should be processed during ***compile time***.

**Possible kinds of non-type arguments**
(as a consequence of the previous point):

- ***Constant expression*** of an integral or enumeration type;
- ***Name*** of an object of function with external linkage (i.e., a non-local object or non-static object/function);
- ***Address*** of an object of function with external linkage.