

# System Software Crash Course

Samsung Research Russia  
Moscow 2019

Block G: Advanced C++  
5. Generic Programming  
Eugene Zouev

# Generic Programming: Introduction

## Previous lectures: To Remind

- Function & Class Templates
- Template Type & Non-type Parameters
- Template Instantiation: Implicit & Explicit
- Explicit Specialization
- Partial Specialization

## Plan for Today

- Metaprogramming Example
- Functional Objects & Templates

1

# Metaprogramming Example: Range Types

# Range Types (1)

A typical code & related problems:

```
int currentDay, currentMonth;  
.  
.  
.  
currentDay = 70;  
    // Nonsense, but OK for compiler!  
currentDay = currentMonth+1;  
    // Looks strange for human's point of view,  
    // but again this is OK for compiler!
```

A better solution:

```
type DayOfMonth = 1..31;           // Pascal
```

```
type DayOfMonth is Integer range 1..31; // Ada
```

```
var currentDay : DayOfMonth;
```

```
var currentMonth : 1..12;
```

```
.  
. .  
. . .
```

```
currentDay := 70; // compiler error
```

```
currentDay := currentMonth+1;
```

```
    // if type of currentMonth is not the same as the
```

```
    // type of currentDay then compiler might report
```

```
    // a warning; otherwise it may add some checking code
```

# Range Types (2)

C++ language: no range types; but:

1. **Classes**: universal mechanism for defining new types.

2. **Templates!**

The very first (naïve) attempt:

```
class RANGE
{
    int leftBorder;
    int rightBorder;
    int value;
public:
    // interface
    . . .
};
```

# Range Types (3)

## 1. Constructor with 3 params: initialization semantics

```
class RANGE
{
    ...
    // interface
    RANGE(int v, int l, int r )
    {
        leftBorder = l;
        rightBorder = r;
        value = v;
    }
};
```

```
RANGE range(0,-10,10);
```

## 2. Default constructor: to prevent uninitialized instances

```
class RANGE
{
    ...
    // interface
    RANGE() = delete
};
```

```
RANGE range; // error!
```

# Range Types (4)

## 3. Copy constructor: initializing semantics

```
RANGE::RANGE ( const RANGE& r )  
{  
    leftBorder = r.LeftBorder;  
    rightBorder = r.rightBorder;  
    value = r.value;  
};
```

```
RANGE range1(range);
```

## 4. Assignment operators: copying semantics

```
RANGE& RANGE::operator=(RANGE& r)  
{  
    value = r.value;  
    return *this;  
}  
RANGE& RANGE::operator=(int v)  
{  
    value = v;  
    return *this;  
}
```

```
range2 = range1;
```

```
range2 = 1;
```

# Range Types (5)

A problem:

```
RANGE range1(0,-5,5);  
RANGE range2(3,1,10);
```

```
range1 = range2;
```

What does it mean  
*semantically*?

5. Other operators: to make **RANGE** type as much similar to integral types as possible

```
RANGE& RANGE::operator++(void)  
{  
    value++;  
    return *this;  
}  
RANGE::operator int()  
{  
    return value;  
}
```

```
range2++;
```

```
int i = range2;
```



# Range Types (6)

## 6. Error handling

```
void RANGE::check(void)
{
    if ( value<leftBorder ||
        value>rightBorder )
        throw 1;
}
```

```
RANGE& RANGE::operator++(void)
{
    value++;
    check();
    return *this;
}
```

```
RANGE::RANGE(int v,int l,int r )
{
    leftBorder = l;
    rightBorder = r;
    value = v;
    check();
}
```

# Range Types (7)

## Is this solution really suitable?

### Problem 1:

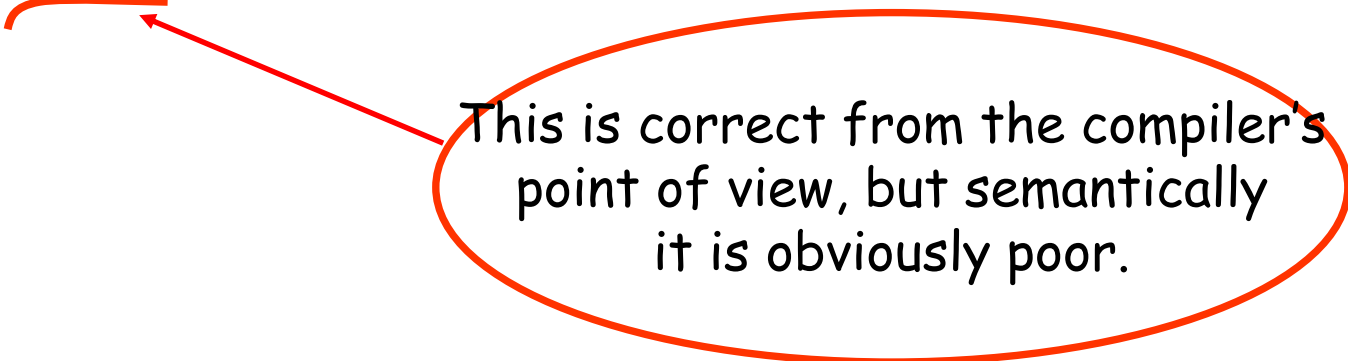
- Range boundaries are attributes of *values*, but not attributes of *types*

```
RANGE a(0,-5,5);  
RANGE b(3,1,10);
```

Formally, *a* and *b* are variables of the *same type*;

But we wanted to have *different types* for *different ranges*

```
a = b;
```



This is correct from the compiler's point of view, but semantically it is obviously poor.

# Range Types (8)

## Problem 2:

- Any **RANGE** instance carries *three* integer values:

one of them (**value**) has useful semantics;  
the other two (**leftBorder**, **rightBorder**) represent boundaries  
and they will be definitely not changed  
(it's a nonsense to change boundaries during instance's lifetime)

## Solution:

**Parametrize RANGE:**

re-design it as a *template*  
considering boundaries as the template  
*parameters*.

# Range Types (9)

## Solution:

```
template < int leftBorder, int rightBorder >
class RANGE
{
    int value;    // the single member!
    RANGE() { }   // private default constructor
public:
    // interface
    RANGE ( int v )           { value = v; check(); }
    RANGE ( const RANGE& r )   { value = r.value; }
    RANGE& operator=(RANGE& r){ value = r.value;return *this; }
    RANGE& operator++(void)    { value++;check();return *this; }
    operator int()             { return value; }
    . . .
};
```

Here **a** and **b** are the variables of the *different* types..

Hence the assignment causes the compiler error

```
RANGE<-5,5> a(0);
RANGE<1,10> b(3);
```

```
a = b; // error
```

# Range Types (10)

## Two Remarks:

1.

```
class A { . . . };  
class B { . . . };
```

Different types

```
A a;  
B b;
```

```
a = b; // error
```

Illegal, because types  
of **a** and **b** are different.

```
RANGE<-5,5>  
RANGE<1,10>  
RANGE<100,1000>  
. . .
```

These instantiations form  
the *single family* of types,  
but all of the types are *different*.

2. Shorthand:

```
typedef RANGE<-5,5> myTinyInt;
```

```
myTinyInt i = 2;
```

**New syntax!**

```
using myTinyInt = RANGE<-5,5>;
```

**The task for your homework:**

**Write the complete implementation of the `RANGE` template providing:**

- Constructor(s) and destructor
- Arithmetic and relational operators
- Increment and decrement
- Conversion function `RANGE→long`
- A simple checking and exception handling mechanism

**Show a couple of “realistic” (practical) examples of using the `RANGE` template.**

2

# Functional Objects And Templates

# Functional Objects & Templates (1)

**Problem:**

Find the first array element which is equal to a given value.

**Solution:**

```
const int* find1 ( const int* pool, int n, int x )
{
    const int* p = pool;
    for ( int i = 0; i<n; i++ )
    {
        if ( *p == x ) return p; // success
        p++;
    }
    return 0; // fail
}
```

**Example of use:**

```
int A[100];
...
int* p = find1(A,100,5); // p points to the 1st element of A
                        // which is equal to 5, or p==0.
```



# Functional Objects & Templates (2)

**More general problem:**

Find the first array element which satisfies a condition.

**Solution:**

```
const int* find2 ( const int* pool, int n, bool (*cond)(int)  )
{
    const int* p = pool;
    for ( int i = 0; i<n; i++ )
    {
        if ( cond(*p) ) return p; // success
        p++;
    }
    return 0; // fail
}
```

↑  
Pointer to function calculating the condition

→  
Call by pointer

**Example of use:**

```
int A[100];
bool cond_e5 ( int x ) { return x==5; }
int* p = find2(A,100,cond_e5); // p points to the 1st element of
                                // A which is equal to 5, or p==0.
```

# Functional Objects & Templates (3)

More examples of use:

```
int A[100];  
  
bool cond_eq5 ( int x ) { return x==5; }  
int* p1 = find2(A,100,cond_eq5);  
  
bool cond_range_0_100 ( int x )  
{  
    return (x>=0) && (x<=100);  
}  
int* p2 = find2(A,100,cond_range_0_100);
```

Well-known example: `qsort()` function from the Std library:

```
void qsort ( void* base,           // first element  
             size_t num,          // number of elements  
             size_t width,        // element size  
             int (*compare)(const void*, const void*) );  
                                // comparing function
```

# Functional Objects & Templates (4)

**Summary:** the general principle of the technique:

Passing the pointer to a *callback function* containing the expression on the array element.

**Callback function: pros & cons:**

- 😊 **Flexible and general mechanism:**  
we can specify (virtually) any condition;
- 😞 **Inefficient:** repeated code generates a lot of overhead (while the expression the callback function evaluates is typically very simple);
- 😞 **Inefficient:** slows down pipelined machines (function calls force the processor to stall).

# Functional Objects & Templates (5)

The ideal solution should be:

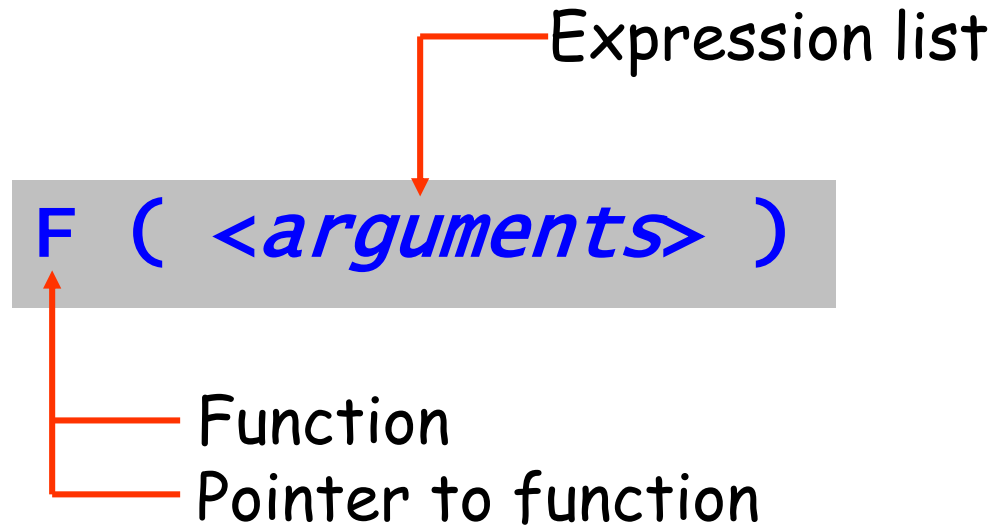
- 😊 Flexible and general as callbacks;
- 😊 More efficient than callbacks.

The basis for the better solution:

- **Classes** with user-defined call operators;
- **Templates!**

# Functional Objects: a Side-step (1)

Function call:



Examples:

```
int F ( int x) { return expr; }  
int (*pF)(int) = F;  
.  
.  
.  
int a = F(1);  
int b = pF(1);
```

# Functional Objects: a Side-step (2)

Function call:

Expression list

**F** ( *<arguments>* )

Function

Pointer to function

*Object of a type with the call operator*

Example:

```
class C {  
public:  
    int operator()(int x) { return expr; }  
};  
.  
.  
.  
C c;  
int z = c(1); // ≡ c.operator()(1);
```

# Functional Objects: a Side-step (3)

If **F** is an object of a type with the call operator then the construct like

```
F ( <arguments> )
```

is equivalent to:

```
F.operator() ( <arguments> )
```



*Just a special “name”*

# Functional Objects: Definition

- If a type has the call operator **operator()** then the type is called **functional type**.
- If an object is of a functional type it is called **functional object**.

## Remarks:

The call operator may be either **built-it** or **user-defined**.

The simplest case: the *C/C++* **pointer-to-function type** is *the functional type*.



# Functional Objects & Templates (6)

Coming back to `find2()` example:

Let's introduce the special "comparing" class...

```
class greater_than_5
{
public:
    bool operator()(int x) const { return x>5; } // inline
};
```

...and modify the `find2()` function:

```
const int* find2 ( const int* pool, int n, greater_than_5 c )
{
    const int* p = pool;
    for ( int i = 0; i<n; i++ )
    {
        if ( c(*p) ) return p; // success
        p++;
    }
    return 0; // fail
}
```

`c(*p)` is equivalent to `c.operator()(*p)`

We are passing the object with statically known `operator()` method which is inline by default

# Functional Objects & Templates (7)

And now... *generalize* "comparing class":

```
template < typename T, T N >
class greater
{
public:
    bool operator()(T x)const { return x>N; } // inline
};
```

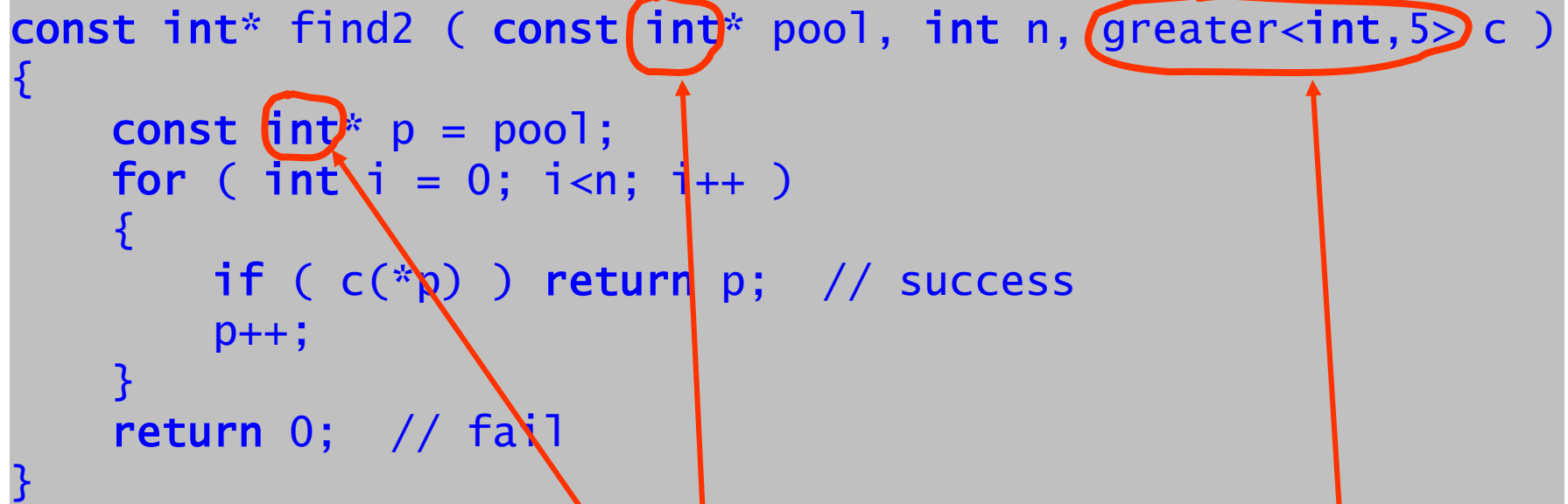
find2() function using the generic comparator:

```
const int* find2 ( const int* pool, int n, greater<int,5> c )
{
    const int* p = pool;
    for ( int i = 0; i<n; i++ )
    {
        if ( c(*p) ) return p; // success
        p++;
    }
    return 0; // fail
}
```

# Functional Objects & Templates (8)

Now consider the `find2()` function itself:

```
const int* find2 ( const int* pool, int n, greater<int,5> c )
{
    const int* p = pool;
    for ( int i = 0; i < n; i++ )
    {
        if ( c(*p) ) return p; // success
        p++;
    }
    return 0; // fail
}
```



The lacks:

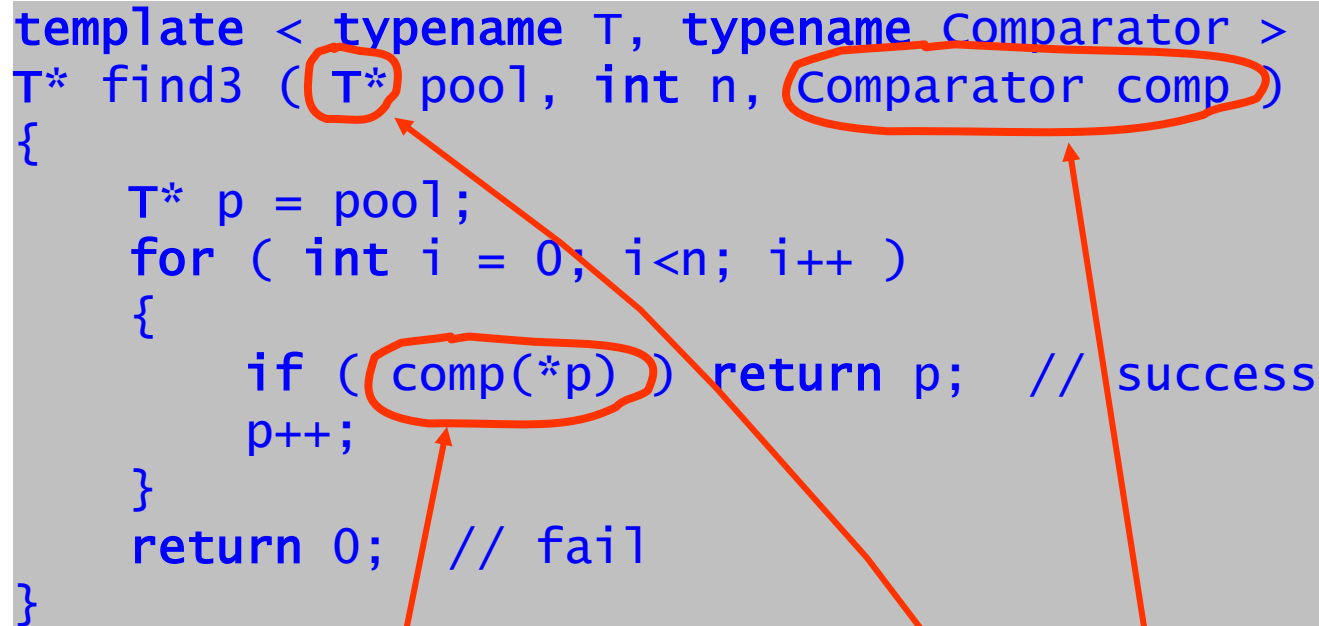
- The function searches integer arrays
- The function can only find values which are greater than 5

Conclusion: Generalize `find2()`!

# Functional Objects & Templates (9)

Generic find3() version:

```
template < typename T, typename Comparator >
T* find3 ( T* pool, int n, Comparator comp )
{
    T* p = pool;
    for ( int i = 0; i < n; i++ )
    {
        if ( comp(*p) ) return p; // success
        p++;
    }
    return 0; // fail
}
```



Advantages:

- The function searches arrays of any type (no requirements)
- The function can search values by any criteria
- It is more efficient than the original find2() version!

# Functional Objects & STL

Various *comparators* like...

Similar templates *exist*  
in the C++ Standard Library

```
template < typename T, T N >
class greater
{
public:
    bool operator()(T x) const { return x > N; }
};

template < typename T, T N >
class greater_equal
{
public:
    bool operator()(T x) const { return x >= N; }
};

template < typename T, T N >
class less_equal
{
public:
    bool operator()(T x) const { return x <= N; }
};
```

...all could be passed to the `find3()` function:

```
int* p = find3(A, 100, greater<int, 5>());
int* q = find3(A, 100, greater_equal<int, 10>());
int* r = find3(A, 100, less<int, 0>());
```

# Funct.Objects: Template Adaptors

Another way to organize a set of predicates like comparators:

```
template < typename T >
class compare
{
public:
    bool operator()(T x, T y) const { return x < y; }
};
```

The general comparator

```
template < typename T >
class positive {
public:
    bool operator()(T x) const { return
```

```
compare<T>() c;
return c(0,x);
```

Adaptor

```
    bool operator()(T x) const { return compare<T>()(0,x); }
};
```

```
template < typename T, T N >
class less
{
public:
    bool operator()(T x) const { return compare<T>(x,N); }
};
```

Adaptor