ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

The Standard: C++17g (A Developed Future Perspective)

Gaurav Kumar Roy [1]

DIS (Diploma in Security), BCA, C|EH, CCNA, CH|FI, MCA from Department of Computer Application, Lovely Professional University, Phagwara, Punjab, India[1]

ABSTRACT: This research paper is mainly written as a tribute to Sir Bjarne Stroustrup and a proposal standard C++ developer community. This research paper gives a modern view of the rise of programming paradigm and the way C++ can rule the language paradigm world. With the implementation of these proposed changes to C++ language, the C++ programmers can get benefit and can move the development to a new level. The **uncaught_exception dynamic return-type overloading** concept, **iterated-initialization** can make C++ programming more innovative.

KEYWORDS: Programming, coding, program, code-snippet, C++, compiler, multi-conceptual, logic-oriented, overloading, exception, multithreading, polymorphism, parsing, lexeme, keywords, paradigm

I. INTRODUCTION

C++ was designed to be a systems programming language and has been used for embedded systems programming and other resource-constrained types of programming since the earliest days. C++ is used in essentially every application areas which includes scientific calculations and projections, developing compilers, developing kernels of operating systems, device drivers for different hardware architecture, games and VRs, cryptographic algorithms, distributed systems infrastructure, animation and design, telecommunications, embedded systems applications (e.g. mars rover autonomous driving), aeronautics software development, CAD/CAM systems, ordinary business applications, graphics, e-commerce sites also and large web applications (such as airline reservation). How does C++ support such an enormous range of applications? Sir Bjarne Stroustrup in one of his research paper told the basic answer which is: *"by good use of hardware and effective abstraction"*.

With all the standard updates provided by C++ since its creation was phenomenal. Developers and programmers are bringing out the best in C++ from the rests to compete in the upcoming fifth generation of programming. A fifth-generation programming language can be shortened as 5GL is a new upgraded level of programming language(s) and they mostly follow constraints oriented problem solving mechanism given to the program, rather than implementing an algorithm written by any programmer. Most constraint-based, *multi-concept integrating statement based languages*, *compound logic-oriented* programming languages and some other declarative languages comes under the category of fifth-generation languages. While fourth-generation programming languages are intended to build specific programs, fifth-generation languages are premeditated to make the computer solve a given problem without the programmer. Thinking this way from the users' point of view, users only needs to worry about what problems need to be solved and what conditions need to be met and without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in artificial intelligence research. Mercury, Prolog, Python, OPS5, Ruby, Lisp etc are some examples of future-level programming languages. Now it's time to emerge out the *Standard C++17g* as 5th generation language. The work in this research paper (other than introduction and conclusion) is divided into two sections –

a) Uncaught Exception Handling
b) Dynamic Function Overloading
c) Single Lined Initialization of Variables

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

II. THE UNCAUGHT-EXCEPTION

We already know the way the standard C++11 provided and integrated the concept of multithreading and exception handling mechanisms. Exception handling is considered better than that of traditional return-an-error-code way since exceptions cannot be ignored and moreover, the error handling logic is alienated from the problem at hand. Uncaught exceptions are those exceptions where an exception is thrown or re-thrown and but have not so far entered a matching catch clause programmers have to either terminate that section of code using some predefined functions like abort() of C language or the C++ need to provide the intelligence to terminate that block of uncaught-exception declaring it as an unexpected exception which ultimately exists the program. Or programmers' can install their own terminate( ) kind of function using the standard set_terminate( ) function that will return a pointer value to the terminate( ) function. Moreover, uncaught_exception will identify if stack unwinding is currently in progress or not. This can be done using a new predefined function name uncaught_exception( ) which will be preceded by either bool (boolean) to grab value either as TRUE or FALSE, or this pre-defined function can also have data type as int (integer) for returning integer value. This can be set as an overloaded function based on the change in return type. Boolean based returning will tell whether uncaught_exception( ) found any un-handled and un-caught exceptions within the program unit by indicating using TRUE or FALSE. Uncaught_exception( ) will results to TRUE if such exceptions are not found or if stack unwinding is currently in progress in this thread. If exception is found, then it is handled and terminated. On the other hand, returning integer value, the function will prompt how many uncaught exceptions have been detected, and returns the number of uncaught exception objects found or detected in the current thread.

The basic structure of uncaught exception handling can be something like this –

```
try {
 // protected code
}catch(...) {
   // code to handle any exception
<data_type> uncaught_exception() noexcept
}
```

This function will keep on looking for how many exceptions within an existing thread have been thrown or re-thrown and not yet handled by any matching catch statement. It is recommended to sometimes to throw an exception even when uncaught_exception( ) results to TRUE. So using this uncaught exception handling concept, program execution will be much safer and implementation of associated logic by any programmer will be specific. In another research paper, I will provide the code snippet that will be required to design the terminate ( ) function.

III. THE DYNAMIC FUNCTION OVERLOADING

As we all know that overloading is one of the most striking feature C++ provide its programmers'. To state more than one meaning for a single function name or an operator within that specific scope which is called function overloading (while dealing with functions) and operator overloading (while dealing with operators) and in combination termed as overloading. An overloaded assertion is a declaration that had been declared keeping the name same as a previously declared assertion or lists of assertions resides within the same scope, except that both declarations need to have dissimilar argument type, or number of arguments should be different, return-type can also be different and obviously all of these differences in different definition. In this section we will deal with function overloading only in an upgraded format. C++17g can also provide new way + variation of overloading any function. With my new approach to think of an overloading a function, here comes two sub categories of function overloading. These can be coined as –

- Static function overloading
- Dynamic function overloading

Among these two types, static function overloading is what we usually do like this –

```
#include<conio>
```

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

```
#include<iostream>
int add(int, int);
float add(float, float);
void main()
{
  int g, s ;
  float d, k, y, x;
  clrscr();
  cout << "Enter two integers:\n";
  cin >> g >> s;
  x = add(g, s);
  cout << "Total of Two integers: " << x << endl;
  cout << "Enter two floating point numbers:\n";
  cin >> d >> k;
  y = add(d, k);
  cout << "Total of two floats: " << y << endl;
  getch();
}

float add(float x, float y)
{
  float tot;
  tot = x + y;
  return tot;
}
int add(int x, int y)
{
  float tot;
  tot = x + y;
  return tot;
}
```

This is static function overloading as it will select any one of these two functions based on the type of value given, though the name of function is same, but programmer is aware of the fact that this program will take wither of int or float value only.

But there may arise some situation where neither the return type is known (which may exceed from general floating point range to long long or something like this exceeding any range after addition or multiplication of two or more variables). In that case either a runtime error will occur or the precision of output will not be accurate and precise. Again if we take long float, long int or some large structural data-type for handling such issues, then static memory wastage will also occur, which is unnecessary incrementation of variable size and internally the compiler will assign large memory for that / those variables. So, for mitigating this type of problem, I have introduced the new concept of dynamic function overloading.

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

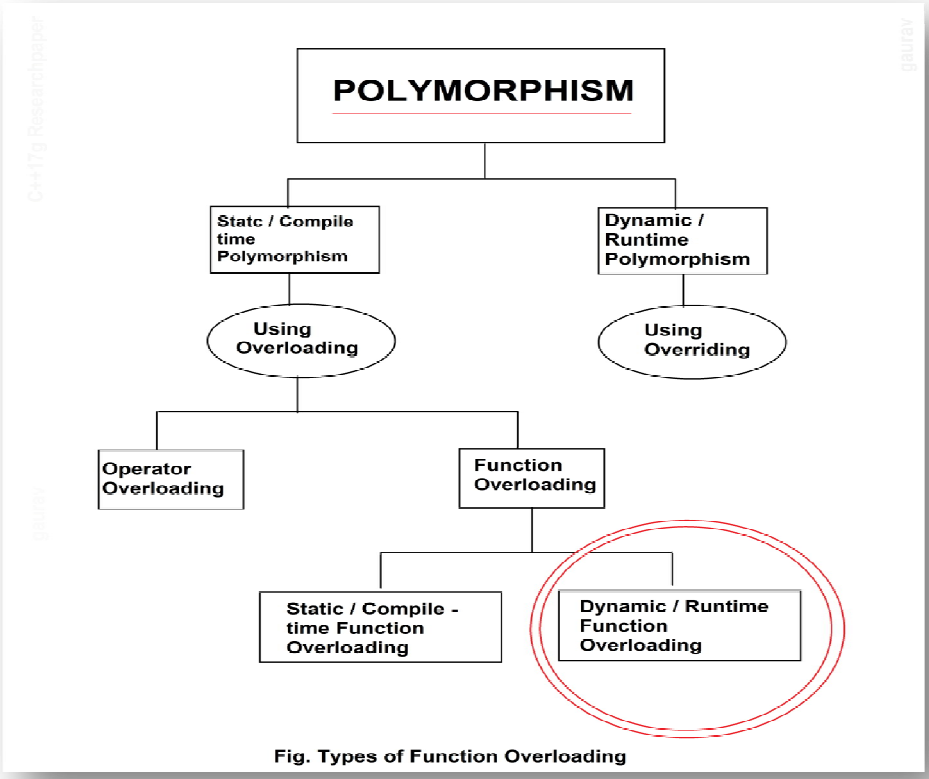

Fig. Types of Function Overloading

Lets' discuss about this in details. Here the programmers do not have to write any details about the return-type of the function which the programmers will overload. Moreover, the programmers do not have to write the type of parameter(s), but only need to mention the number of parameters that will be passed. A new operator will be used here at the beginning of any function which is named as "grave accent" which is represented in the keyboard by (`). (This operator will be found under the escape button in association with ~ (tilde) button.) Using this grave-accent – which will work as an operator and will be identified by the new C++ compiler as it will be designed and this new special – operator will get stored under operator-class of the compiler. Within a C++ compiler, the category of arranging all valid tokens at the time of lexical analysis is done based on classification of tokens used in the code where each kind of token is put to their respective classes. These classes are categorized as –

- ✓ Constants and Literal class
- ✓ Operator class
- ✓ Symbol class
- ✓ Keyword class
- ✓ Identifier class

So, during the working of lexical analysis, the lexical analyser of C++ compiler will check for all the tokens, and as the grave-accent encountered, it will put that in the 'operator class' category. Hence, programmers just have to put it in conjunction with the function name (at the beginning, before function name) to make the return type or in case of parameters to make argument type dynamic. Dynamic in the sense that the compiler – while encountering this grave-accent symbol will construct an image of return type or argument type based on the values that are passed to the argument (i.e. if there are two arguments *g and h*. Now for 1st argument let suppose it is passed with value 6.2, this grave-accent will dynamically assign *g* with type *float* and let for 2nd value *h*, value passed is 4, then grave-accent will dynamically assign *h* as type *int*) which may vary from return-types like –

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

- ✓ int
- ✓ float
- ✓ void
- ✓ double
- ✓ long
- ✓ char
- ✓ bool

Similarly, programmers can use this grave-accent with argument names also that will be declared within function parenthesis. The syntax will look something like this –

```
auto <parameter_counter> = 0;
        <grave-accent operator> <function_name> ( <parameter_counter>: <accent-grave>arg1,
        <accent-grave>arg2, .... , <accent-grave>argN)
//with the end of execution of this above line, <parameter_couner> value changes from intial value 0 to N ...
        //... is having value as 3
        {
        // Body of the function;
        // return <var>; //Can return any type of variable.
        }
```

Returning any type of variable after calculation won't affect the code as the grave-accent operator will be designed and instructed in such a way that it will first identify the type of value entered by user at run-time and at run-time only when the turn comes to return a value back to the function after calculation, the calculated value will be predicted to be of that specific type. Also, we have noticed that using the storage class 'auto' we have to declare a parameter counting variable, and need to initially set its value to 0 (zero). The keyword auto will automatically assign this variable a data-type based on the initial value i.e. 0 (which is integer). This feature was integrated and added with C++11 that programmers can declare a variable or an object without specifying its specific type by using this keyword 'auto'. Then at runtime the parameter-counter will re-assign it with a value (by dynamic incremented counting mechanism) based on the number of arguments set at the time of constructing this overloaded function.

Let us have a view of the structure of a program of how it will look like –

```
auto param=0;
` add(param:` x, ` y, `z) //with the end of execution of this line, variable param is ...
                          //... having value as 3
  {
    ` tot;
    tot = x * y * z;
    return tot;
  }
```

In this case, programmers do not have to write separate add() once for returning integer, second time for float and so on for arbitrary data-types. But an error will get generated of the number of parameters passed is more than the number of arguments declared inside parenthesis with the accent-grave operator. This means, If there is one parameter passed to this function, then it is okay, if two parameters passed, then also okay, three parameters passed to the above code snippet, then also okay. But in case the programmer passes more than three values to the above program, it will generate error as it can't handle three parameters (since not instructed by the programmer at the time of programming). When one or two values will be passed, the compiler will automatically assign that value to the first argument assuming the program structure like this –

```
auto <param_counter>=0;
<return-type> add(<param_counter>: <data type> x)
        //with the end of execution of the above line, variable param is ...
        //... having value as 1
  {
    <data-type> tot;
```

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

```
    tot = x * 0 * 0; //By default other arguments will be having value as 0;
    return tot;
  }
```

In case, a programmer wants to incorporate multiple tasks within the function which will get triggered or changed based on number of parameter passed, then the programmer can use conditions also like this –

```
auto param=0;
` add(param:` x, ` y, `z)  // variable 'param' is having value 3 now
  {
    auto tot=0;    // initialize tot as 0, so variable 'tot' will be recognized as type int
    if (param==1) //condition one: for single argument passing
        {
                cout<< "Single Parameter <<x;
                cout<< "No calculation;
        }
    else if (param == 2) //condition two: when passing two arguments

        {
                cout<< "Total is:" <<x+y;
        }
    else if         //condition three: when passing three arguments
        {
                tot = x * y * z;
                 return tot;
        }
    else
        {
                try{
                        cout<< " Argument number mismatched";
                        cout<<param<<parameters only;
                }catch(exception& e)
                        {}
        }
  }
```

In this program the argument number mismatch or number of argument exceed exception will get generated if number of value passed is more than the number of arguments declared. In this way, using if-else we can eliminate the runtime errors also. So overall we can say that this new technique gives us so many facilities including dynamic error handling technique with or without exception handling mechanism. Plus, it allows programmers to reduce the code to be written many times for each type of return type and / or for performing overloading based on the number of parameters.

IV. SINGLE-LINED INITIALIZATION

Another new feature can be introduced for initializing multiple values in one single shot, one single line. As we all know that in C++, for initializing variables we have to follow this procedure –

- <data_type> <var_name>;
  <var_name> = <value>;

- <data_type> <var_name> = <value>;

- <data_type> <var_name1> = <value1> , <var_name2> = <value2>, ... , <var_nameN>= <valueN>;

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

- <data_type> <var_name1>;
  <var_name1> = <value1>;
  <data_type> <var_name2>;
  <var_name2> = <value2>; and so on....

For single variable assigning, it might look simple, but for assigning multiple variables with their respective values using assignment operator is quiet tedious task. So, this would be better if my proposed concept get come into action in the next standard C++ version. I have proposed two different types of variable initialization mechanism in this research thesis. These are named as –

- **Multi-assignment statement (using = ,)** : In this way of approach of initialization of variables, programmers just have to follow the syntax mentioned below –

  CASE 1:
  <data_type> <var1>, <var2>, ..., <varN> = <value1>, <value2>, ..., <valueN>;

  In this case, the programmer should keep in mind that the number of variables along with the number of values remains the same. And the values assigned should be of that particular type only. Again another syntax can play significant role when on a single line programmers have to assign multiple variables and each of them is having different data type, then the thing that programmer can do will look something like this –

  CASE 2:
  auto <var1>, <var2>, ..., <varN> = <value1>, <value2>, ..., <valueN>;

  Here, the keyword 'auto' (which is a standard concept came to existence since C++11) will automatically assumes the type of variable based on the assigned values. This makes a C++ program more flexible and simpler from programmer's point of view.

- **Implicit-iterated Array-Initialization:** This will be another significant approach to grab multiple lines of dynamic array initialization into one single line. Usually, programmers will dynamically initialize values to each array location in this way –

```
#include<conio.h>
#include<iostream.h>
void main()
{
        int ary[6], i;
        clrscr();
        for(i=0; i<6;i++)
                {
                ary[i] = i;
                }
        for(i=0; i<6;i++)
                {
                cout<<ary[i]<<endl;
                }
                getch();
}
```

  But with me proposed technique, programmers can write the initialization of array (dynamically) on a single line. This will look something like this –

```
#include<conio.h>
#include<iostream.h>
void main()
```

ISSN(Online) : 2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 12, December 2016

```
{
int no_of_var;
cin>> no_of_var;
int arr[no_of_var] = {for(int i=0; i<no_of_var; i++) arr[i]=i; }; //normal for loop
}
```

Here the user will just give the array size which will get dynamically fitted in the array & the initialization will also be done within the curly braces by using the for loop implicitly, all of them accommodated on a single line. Also programmers can use the range-based for loop that has been introduced in C++11 standard and using the range-based for loop, programmer can assign values to all the array variables in one shot using my proposed implicit-iterated array-initialization technique like this –

```
#include<conio.h>
#include<iostream.h>
void main()
    {
        int arr[6] = {for(int i:  {2, 3, 4, 5, 6, 8})}; //range-based for loop
    }
```

Here, the range-based for loop is implemented implicitly within the curly braces of array initialization which initializes *arr[0]* with value *2*, *arr[1]* with value *3* and so on.

V. CONCLUSION

Here, I have implemented and portray a detailed view of what ideas and concepts I have got for the next generation and next version of C++ programming. I hope Sir Bjarne Stroustrup will like my proposed concepts which are although complex but at the same time more effective if some-one understands the true essence of writing dynamic, programmer-independent level of code. I'm honoured that Sir Bjarne Stroustrup wanted to see my proposed thesis and willing to implement these via his C++ community developer scholars.

REFERENCES

[1] Abstraction and the C++ Machine Model, by Bjarne Stroustrup.
[2] Bjarne Stroustrup The C++ Programming Language, Person Publication
[3] The Standard C++ Library, by Nicolai M. Josuttis
[4] Kyle Loudon, C++ Pocket Reference 1st Edition, O'reilly Publication
[5] Wang, P. S. (1994). C++ with object oriented programming. Internation Thomson Publishing
[6] B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007
[7] B. Stroustrup: Learning Standard C++ as a New Language. C/C++ Users Journal. pp 43-54. May 1999 3
[8] J. Daniel Garcia and B. Stroustrup: Improving performance and maintainability through refactoring in C++11. Isocpp.org. August 2015.