MARUDHAR KESARI JAIN COLLEGE FOR WOMEN,VANIYAMBADI

PG AND RESEARCH DEPARTMENT OF COMPUTER SCIENCE

CLASS: I BSC COMPUTER SCIENCE

SUBJECT CODE: 23UCS11

SUBJECT NAME: Object Oriented Programming Concepts using C++

SYLLABUS

UNIT V

Files – File stream classes – file modes – Sequential Read / Write operations – Binary and ASCII Files – Random Access Operation –Templates – Exception Handling - String – Declaring and Initializing string objects – String Attributes – Miscellaneous functions.

C++ Files and Streams

In C++ programming we are using the **iostream** standard library, it provides **cin** and **cout** methods for reading from input and writing to output respectively.

To read and write from a file we are using the standard C++ library called **fstream**

Data Type	Description
fstream	It is used to create files, write information to files, and read information from files.
ifstream	It is used to read information from files.
ofstream	It is used to create files and write information to the file

simple example of writing to a text file testout.txt **using C++ FileStream programming.**

```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main () {
5.   ofstream filestream("testout.txt");
6.   if (filestream.is_open())
7.   {
8.     filestream << "Welcome to javaTpoint.\n";
9.     filestream << "C++ Tutorial.\n";
10.    filestream.close();
11.  }
12.  else cout <<"File opening is fail.";
13.  return 0;
14. }
```

15. In C++, for every file operation, exists a specific *file mode*. These file modes allow us to *create, read, write, append or modify* a file. The file modes are defined in the class **ios**. Let's see all these different **modes** in which we could open a file on disk.

File Modes	Description
ios::in	Searches for the file and opens it in the **read** mode only(*if the file is found*).
ios::out	Searches for the file and opens it in the **write** mode. If the file is found, its content is overwritten. If the file is not found, a new file is created. *Allows you to **write** to the file.*
ios::app	Searches for the file and opens it in the **append** mode i.e. this mode allows you to **append** new data to the end of a file. If the file is not found, a new file is created.
"ios::binary"	Searches for the file and opens the file(if the file is found) in a binary mode to perform binary input/output file operations.
ios::ate	Searches for the file, opens it and positions the pointer at the end of the file. This mode when used with **ios::binary, ios::in** and **ios::out** modes, *allows you to **modify** the content of a file.*

"ios::trunc"	Searches for the file and opens it to truncate or deletes all of its content(*if the file is found*.
"ios::nocreate"	Searches for the file and if the file is not found, a new file will not be created.

This tutorial has introduced you to some important disk input/output operations that we can perform on a file and modes in which we can open a file on disk. In the next tutorial, we are going to explain how to read an existing file on the disk .

C++ Reading a file on disk

In our last article, we have introduced you to some important file input/output operations we could perform. In this tutorial, we are going to explain how to read the content of an existing file. But before we read the content of an existing file present on the disk, we must open it. For this, the C++ language provides us stream classes used to perform file input/read operations.

In order to perform file input/read operation, C++ provides us a few *file stream classes*, such ifstream, to perform the file input operations.

- **fstream**, to perform the any file input and output operations.

These file stream classes provides us a function named *open()*, using which we could provide location of the file stored on the disk to depending on the mode in which we open this file. Let's read the file mode to use to *read* the content of a file.

Syntax of open() function

```
file-stream-object("filename", mode);
```

- ***file-stream-object***, is object of a file stream class used to perform a file input/read operation.
- ***filename***, is the name of file on which we are going to perform file operations.
- ***mode***, is *single or multiple file modes* in which we are going to open a file.

Reading a file

Let's look at the mode required to read a file.

File Mode	Description
ios::in	Searches for the file and opens it in **read** mode only(*if the file is found*).

 Using getline() function to read a file

We can read the content of a file using an in-built function **getline()** and this function is available to the stream classes **fstream** and **ifstream**. Let's see the syntax of **getline()** function -

```
getline (char* arr, int length);
```

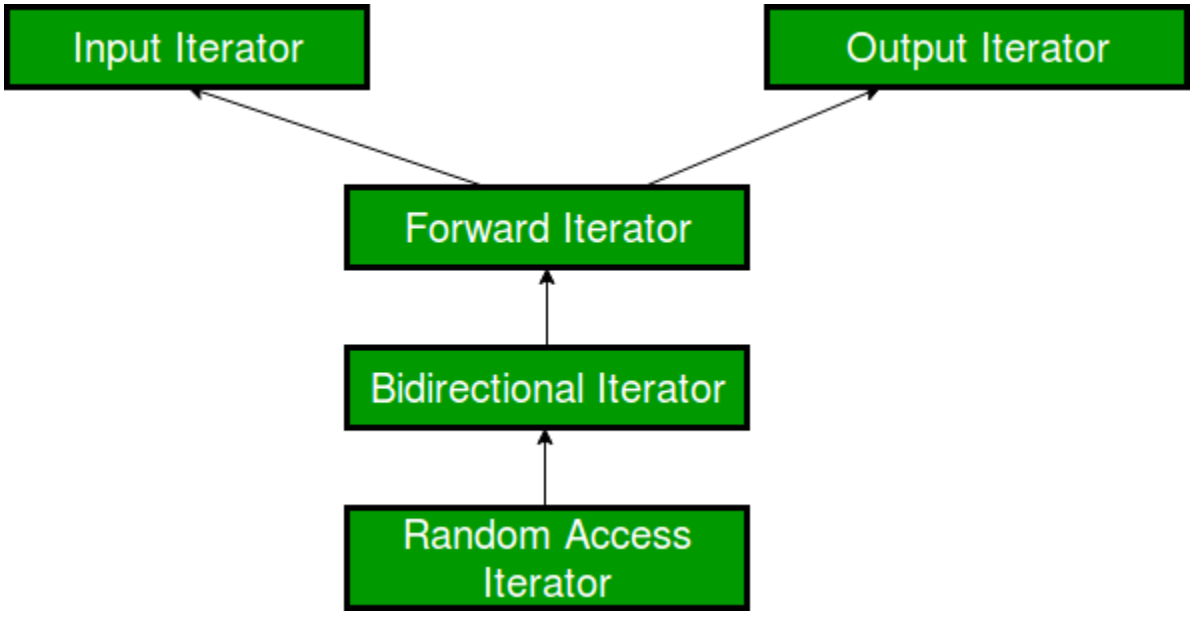
The **getline()** function pulls the characters from the file and stores them into s as a c-string)*char array*) with the maximum *length*, until either the extracted character is the delimiting character i.e. *a newline character('\n')* or the EOF(*end of file*) is reached.

Random Access Iterators in C++

After going through the template definition of various STL algorithms like you must have found their template definition consisting of objects of type **Random-access Iterator**. So what are they and why are they used?

Random-access iterators are one of the five main types of iterators present in C++ Standard Library, others being:

- **Input iterators**
- **Output iterator**
- **Forward iterator**
- **Bidirectional iterator**
    - Also, note that **Iterator algorithms do not depend on the container type**. As the iterators provide common usage for all of them, the internal structure of a container does not matter.



- So, from the above hierarchy, it can be said that random-access iterators are the strongest of all iterator types.

- **Salient Features:**

- **1) Usability:** Random-access iterators can be **used in multi-pass algorithms**, i.e., an algorithm which involves processing the container several times in various passes.
- **2) Equality/Inequality Comparison:** A Random-access iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not.
- So, the following two expressions are valid if A and B are Random-access iterators:
- A == B // Checking for equality
- A != B // Checking for inequality
- **Relational Operators:** Although, Bidirectional iterators cannot be used with relational operators like =, random-access iterators being higher in hierarchy support all these relational operators.
- If A and B are Random-access iterators, then
- A == B // Allowed
- A <= B // Allowed
- **7) Arithmetic Operators:** Similar to relational operators, they also can be used with arithmetic operators like +, – and so on. This means that Random-access iterators can move in both the direction and that too randomly.
- If A and B are Random-access iterators, then
- A + 1 // Allowed
- B - 2 // Allowed

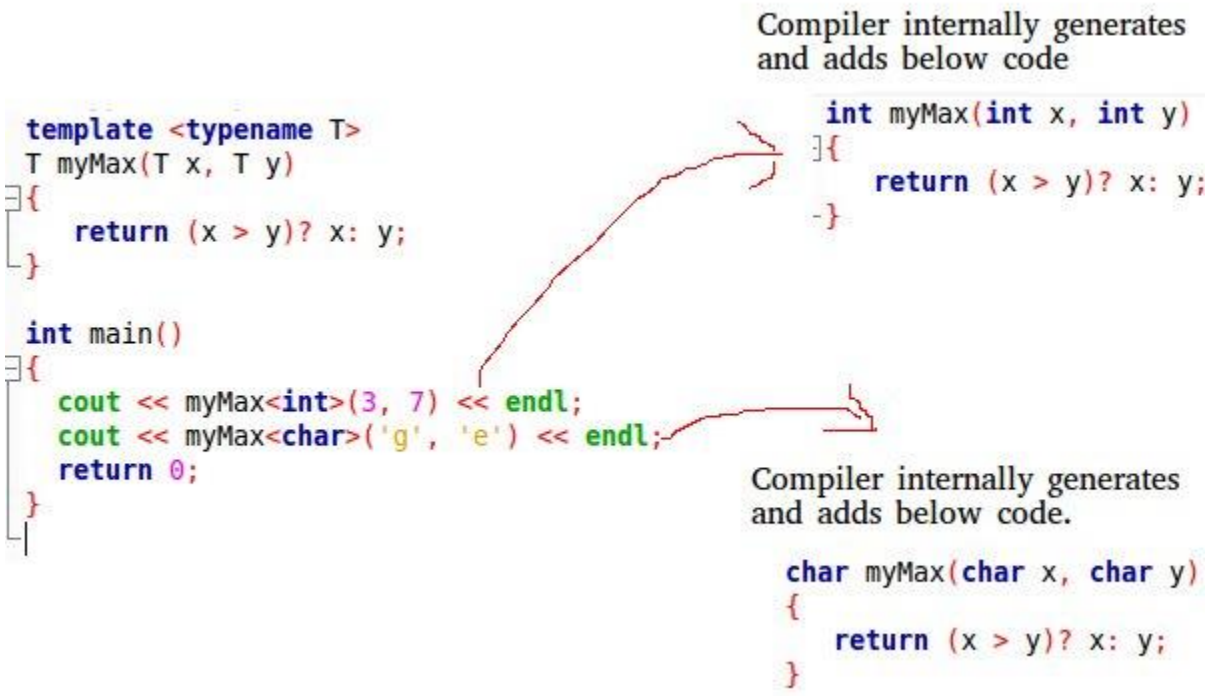
Templates

A **template** is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.

C++ adds two new keywords to support templates: ***'template'*** and ***'type name'***. The second keyword can always be replaced by the keyword **'class'**.

How Do Templates Work?

Templates are expanded at compiler time. This is like macros. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



Class Templates

Class templates like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.

What happens when there is a static member in a template class/function?

Each instance of a template contains its own static variable. See Templates and Static variables for more details.

What is template specialization?

Template specialization allows us to have different codes for a particular data type. See Template Specialization for more details.

Can we pass non-type parameters to templates?

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, that they must be const. The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes for a specified non-type value at compile time. In the below program, if we replace 10000 or 25 with a variable, we get a compiler error.

Exception Handling

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.). C++ provides the following specialized keywords for this purpose:
try: Represents a block of code that can throw an exception.
catch: Represents a block of code that is executed when a particular exception is thrown.
throw: Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2) *Functions/Methods can handle only the exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

C++ try and catch:

Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable “age” is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that’s why we are throwing an exception of type int in the try block (age), we can pass “int myNum” as the parameter to the catch statement, where the variable “myNum” is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

Exception Handling in C++

1) The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

- CPP

```
#include <iostream>

using namespace std;

int main()

{

  int x = -1;// Some code

 cout << "Before try \n";

  try {

    cout << "Inside try \n";

    if (x < 0)
```

```
    {
      throw x;
      cout << "After throw (Never executed) \n";   }}
  catch (int x ) {
    cout << "Exception Caught \n";
  }
  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Output:

Before try

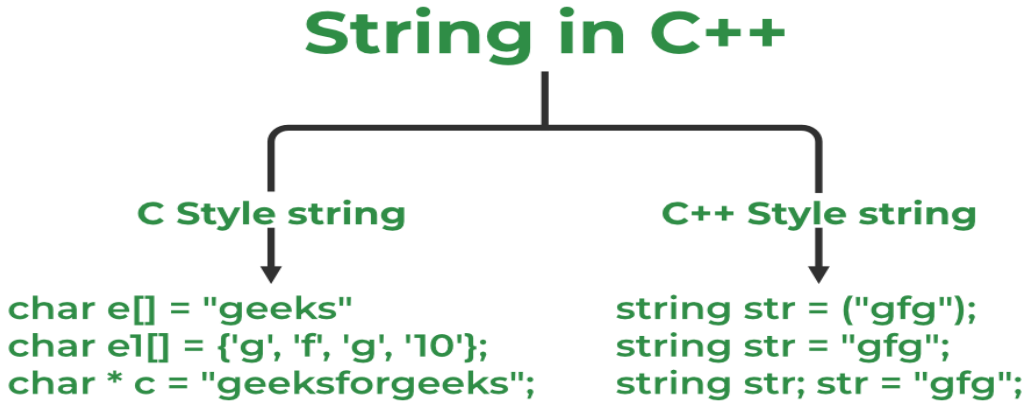
Inside try

Exception Caught

After catch (Will be executed)

Strings in C++

C++ strings are sequences of characters stored in a char array. Strings are used to store words and text. They are also used to store data, such as numbers and other types of information. Strings in C++ can be defined either using the **std::string class** or the **C-style character arrays.**



1. C Style Strings

These strings are stored as the plain old array of characters terminated by a **null character '\0'**. They are the type of strings that C++ inherited from C language.

Syntax:

char str[] = "GeeksforGeeks";

2. std::string Class

These are the new types of strings that are introduced in C++ as std::string class defined inside <string> header file. This provides many advantages over conventional C-style strings such as dynamic size, member functions, etc.

Syntax:

std::string str("GeeksforGeeks");

Ways to Define a String in C++

Strings can be defined in several ways in C++. Strings can be accessed from the standard library using the string class. Character arrays can also be used to define strings. String provides a rich set of features, such as searching and manipulating, which are commonly used methods. Despite being less advanced than the string class, this method is still widely used, as it is more efficient and easier to use. Ways to define a string in C++ are:

- Using String keyword

- Using C-style strings

1. Using string Keyword

It is more convenient to define a string with the string keyword instead of using the array keyword because it is easy to write and understand.

Syntax:

string s = "GeeksforGeeks";

string s("GeeksforGeeks");

2. Using C-style strings

Using C-style string libraries functions such as strcpy(), strcmp(), and strcat() to define strings. This method is more complex and not as widely used as the other two, but it can be useful when dealing with legacy code or when you need performance.

char s[] = {'g', 'f', 'g', '\0'};

char s[4] = {'g', 'f', 'g', '\0'};

char s[4] = "gfg";

char s[] = "gfg";

Example:

- C++

```
// C++ Program to demonstrate C-style string declaration

#include <iostream>

using namespace std;
```

```
int main()

{

    char s1[] = { 'g', 'f', 'g', '\0' };

    char s2[4] = { 'g', 'f', 'g', '\0' };

    char s3[4] = "gfg";

    char s4[] = "gfg";

    cout << "s1 = " << s1 << endl;

    cout << "s2 = " << s2 << endl;

    cout << "s3 = " << s3 << endl;

    cout << "s4 = " << s4 << endl;

    return 0;

}
```

Output

s1 = gfg

s2 = gfg

s3 = gfg

s4 = gfg

How to Take String Input in C++

String input means accepting a string from a user. In C++. We have different types of taking input from the user which depend on the string. The most common way is to take input with **cin** keyword with the extraction operator (>>) in C++. Methods to take a string as input are:

- cin
- getline
- stringstream

1. Using Cin

The simplest way to take string input is to use the **cin** command along with the stream extraction operator (>>).

Syntax:

cin>>s;

2. Using getline

The getline() function in C++ is used to read a string from an input stream. It is declared in the <string> header file.

Syntax:

getline(cin,s);

How to Pass Strings to Functions?

In the same way that we pass an array to a function, strings in C++ can be passed to functions as character arrays. Here is an example program:

Example:

- C++

```
// C++ Program to print string using function

#include <iostream>

using namespace std;

void print_string(string s)

{

cout << "Passed String is: " << s << endl;

 return;

}

int main()

{

    string s = "GeeksforGeeks";

    print_string(s);

     return 0;

}
```

Output

Passed String is: GeeksforGeeks

Difference between String and Character array in C++

The main difference between a string and a character array is that strings are immutable, while character arrays are not.

String	Character Array
Strings define objects that can be represented as string streams.	The null character terminates a character array of characters.
No Array decay occurs in strings as strings are represented as objects.	The threat of array decay is present in the case of the character array
A string class provides numerous functions for manipulating strings.	Character arrays do not offer inbuilt functions to manipulate strings.
Memory is allocated dynamically.	The size of the character array has to be allocated statically.

Know more about the difference between strings and character arrays in C++

C++ String Functions

C++ provides some inbuilt functions which are used for string manipulation, such as the strcpy() and strcat() functions for copying and concatenating strings. Some of them are:

Function	Description
length()	This function returns the length of the string.

Function	Description
swap()	This function is used to swap the values of 2 strings.
size()	Used to find the size of string
resize()	This function is used to resize the length of the string up to the given number of characters.
find()	Used to find the string which is passed in parameters
push_back()	This function is used to push the passed character at the end of the string
pop_back()	This function is used to pop the last character from the string
clear()	This function is used to remove all the elements of the string.
strncmp()	This function compares at most the first num bytes of both passed strings.
strncpy()	This function is similar to strcpy() function, except that at most n bytes of src are copied
strrchr()	This function locates the last occurrence of a character in the string.

Function	Description
strcat()	This function appends a copy of the source string to the end of the destination string
find()	This function is used to search for a certain substring inside a string and returns the position of the first character of the substring.
replace()	This function is used to replace each element in the range [first, last) that is equal to old value with new value.
substr()	This function is used to create a substring from a given string.
compare()	This function is used to compare two strings and returns the result in the form of an integer.
erase()	This function is used to remove a certain part of a string.

C++ Strings iterator functions

In C++ inbuilt string iterator functions provide the programmer with an easy way to modify and traverse string elements. These functions are:

Functions	Description
begin()	This function returns an iterator pointing to the beginning of the string.

Functions	Description
end()	This function returns an iterator that points to the end of the string.
rfind()	This function is used to find the string's last occurrence.
rbegin()	This function returns a reverse iterator pointing to the end of the string.
rend()	This function returns a reverse iterator pointing to the beginning of the string.
cbegin()	This function returns a const_iterator pointing to the beginning of the string.
cend()	This function returns a const_iterator pointing to the end of the string.
crbegin()	This function returns a const_reverse_iterator pointing to the end of the string.
crend()	This function returns a const_reverse_iterator pointing to the beginning of the string.

Example:

```
// C++ Program to demonstrate string iterator functions

#include <iostream>
```

```
using namespace std;

int main()

{

    // declaring an iterator

    string::iterator itr;// declaring a reverse iterator

    string::reverse_iterator rit;

    string s = "GeeksforGeeks";

    itr = s.begin();

    cout << "Pointing to the start of the string: " << *itr<< endl;

    itr = s.end() - 1;

   cout << "Pointing to the end of the string: " << *itr << endl;

    rit = s.rbegin();

    cout << "Pointing to the last character of the string: " << *rit << endl;

    rit = s.rend() - 1;

    cout << "Pointing to the first character of the string: " << *rit << endl;
```

```
    return 0;

}
```

Output

Pointing to the start of the string: G

Pointing to the end of the string: s

Pointing to the last character of the string: s

Pointing to the first character of the string: G

String Capacity Functions

In C++, string capacity functions are used to manage string size and capacity. Primary functions of capacity include:

Function	Description
length()	This function is used to return the size of the string
capacity()	This function returns the capacity which is allocated to the string by the compiler
resize()	This function allows us to increase or decrease the string size
shrink_to_fit()	This function decreases the capacity and makes it equal to the minimum.

Example:

- C++

```
#include <iostream>

using namespace std;

int main()

{

    string s = "GeeksforGeeks";

       // length function is used to print the length of the string

    cout << "The length of the string is " << s.length() << endl;

    // capacity function is used to print the capacity of the string

    cout << "The capacity of string is " << s.capacity()<< endl;

     // the string.resize() function is used to resize the string to 10 characters

      s.resize(10);

     cout << "The string after using resize function is " << s << endl;

     s.resize(20);

  cout << "The capacity of string before using shrink_to_fit function is "<< s.capacity() <<
endl;
```

```
    // shrink to fit function is used to reduce the capacity of the container

   s.shrink_to_fit();

  cout << "The capacity of string after using shrink_to_fit function is "<< s.capacity() <<
endl;

return 0;}
```

Output

The length of the string is 13

The capacity of string is 15

The string after using resize function is GeeksforGe

The capacity of string before using shrink_to_fit function is 30

The capacity of string...

String Attribute

Use string attributes to store text values.

For example, strings are used for customer information such as First Name and Postal Code, order information such as Order ID and Shipping Method, or site information such as Site Section or Site Language.

The string attribute is available in the following scopes: Event, Visit, Visitor.

Size limits

EventStream string attributes have a maximum length of 1,500 characters. Strings are truncated to 1,500 characters if they exceed this limit.

AudienceStream string attributes values have a maximum length of 1,000 characters. If an enrichment results in a string larger than 1,000 characters, the value is not saved.

String enrichments

The following enrichments are available to string attributes.

Set string

This enrichment sets the value of a string attribute, either from a constant value you provide or from the value of another string attribute. The source values can only be a string, iQ variable, or Omnichannel attribute.

Attribute Name: customer_type

- **Starting Value**: ""
- **Enriched With**: "unknown"
- **Resulting Value**: "unknown"

Split string

This enrichment allows you to set multiple values based on a distribution percentage. Each value you set also has a percentage setting. There can be multiple value/percentage entries, but the distribution must total 100%. The distribution is based on a random number generator, so smaller samples might not match the distribution, but as more values are assigned the distribution ratio will become more accurate. The source values can only be a string, iQ variable, or Omnichannel attribute.

In this example an attribute named test_group is used to segment users into two equal groups (50/50), one named “GroupA” and the other “GroupB”. This attribute can then be used to identify activity associated with each group.

Attribute Name: test_group

- **Starting Value**: ""
- **Resulting Value**: Set string’s value to "GroupA" for 50% of string population
  Set string’s value to "GroupB" for 50% of string popularion

Remove string

This enrichment removes the entire value from the attribute.

- **Starting Value**: "Jane Smith"
- **Resulting Value**: ""

Lowercase string

This enrichment will lowercase the current value of the string attribute.

Attribute Name: email_address

- **Starting Value**: "First.Last@Example.com"
- **Resulting Value**: "first.last@example.com"

Join attributes

This enrichment joins multiple values with a delimiter to form a single text value. The delimiter can be one or more characters. For example, you could create a page hierarchy value by combining site_section, page_category, and other page level attributes.

Attribute Name: page_path

- **Starting Value**: ""
- **Enriched With**:
  Attribute 1: site_region="en-us"

Attribute 2: site_section="Electronics"

Attribute 3: category_name="Tablets"

Delimiter: ":"

- **Resulting Value**: "en-us:Electronics:Tablets"

Set string to date

This enrichment converts the value of a string attribute to a date and allows for custom formatting (learn more about the date formatter). The source values can be a date, string, iQ variable, or file import attribute.

Attribute Name: last_purchase_date

- **Starting Value**: ""
- **Enriched With**: Last Login Date with format "yyy-MM-dd"
- **Resulting Value**: "2019-12-31"

Miscellaneous functions.

Inline functions

- Default Arguments
- Function Overloading
- Function Templates
- Typical usage:– Member function of a class
- Very simple, short, fast operations
- Especially in inner loops
- Especially *set* and *get* functions
- Saves the (non-trivial) overhead of invoking a method, creating activation record, etc.
- Encourages cleaner, more readable code

        - **inline** qualifier is strictly advisory

- Typical usage:– Member function of a class
        - Very simple, short, fast operations
        - Especially in inner loops
        - Especially *set* and *get* functions
        - Saves the (non-trivial) overhead of invoking a method, creating activation record, etc.
        - Encourages cleaner, more readable code
        - **inline** qualifier is strictly advisory