Program : **B.Tech**

Subject Name: **Object Oriented Programming & Methodology**

Subject Code: **CS-305**

Semester: **8th**

Object Oriented Programming Systems
(CS-304)
Class Notes

Unit-V

String

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class)
- C-strings (C-style Strings)

C-strings

In C programming, the collection of characters is stored in the form of arrays, this is also supported in C++ programming. Hence it's called C-strings.
C-strings are arrays of type char terminated with null character, that is, \0 (ASCII value of null character is 0).

How to define a C-string?

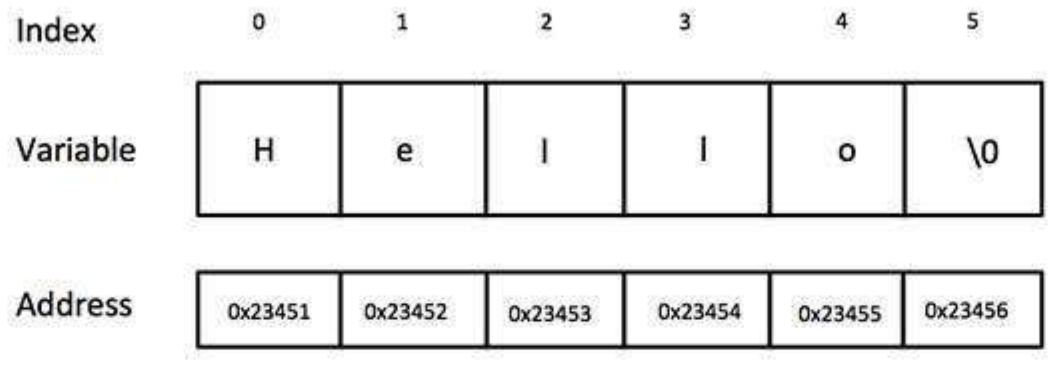
char str[] = "C++";
In the above code, str is a string and it holds 4 characters.
Although, "C++" has 3 character, the null character \0 is added to the end of the string automatically.

Alternative ways of defining a string

char str[4] = "C++";
char str[] = {'C','+','+','\0'};
char str[4] = {'C','+','+','\0'};
Like arrays, it is not necessary to use all the space allocated for the string. For example:
char str[100] = "C++";

char greeting[] = "Hello";

Following is the memory presentation of above defined string in C/C++ −



Example 1: C++ String to read a word

C++ program to display a string entered by user.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    char str[100];

    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: " << str << endl;
    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: "<<str<<endl;
    return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Example 2: C++ String to read a line of text

```
#include <iostream>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

String Object

In C++, you can also create a string object for holding strings.
Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

Example 3: C++ string using string data type

```
#include <iostream>
using namespace std;

int main()
```

```
{
   // Declaring a string object
   string str;
   cout << "Enter a string: ";
   getline(cin, str);

   cout << "You entered: " << str << endl;
   return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

```
#include <iostream>
using namespace std;

void display(char s[]);

int main()
{
   char str[100];
   string str1;
   cout << "Enter a string: ";
   cin.get(str, 100);
   cout << "Enter another string: ";
   getline(cin, str1);
   display(str);
   display(str1);
   return 0;
}
void display(char s[])
{
   cout << "You entered char array: " << s <<;
}
void display(string s)
{
   cout << "You entered string: " << s << endl;
}
```

Output

```
Enter a string: Programming is fun.
Enter another string: Programming is fun.
You entered char array: Programming is fun.
```

You entered string: Programming is fun.

C++ supports a wide range of functions that manipulate null-terminated strings −

Sr.No	Function & Purpose
1	**strcpy(s1, s2);**Copies string s2 into string s1.
2	**strcat(s1, s2);**Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**Returns the length of string s1.
4	**strcmp(s1, s2);**Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**Returns a pointer to the first occurrence of string s2 in string s1.

```
#include <iostream>
#include <cstring>

using namespace std;

int main () {
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;
   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;
   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;
   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

Output

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

```
#include <iostream>
```

```
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

When Should I Use std::string?

The advantages to using std::string:

- Ability to utilize SBRM design patterns
- The interfaces are much more intuitive to use, leading to less chances of messing up argument order
- Better searching, replacement, and string manipulation functions (c.f. the cstring library)
- The size/length functions are constant time (c.f. the linear time strlen function)
- Reduced boilerplate by abstracting memory management and buffer resizing
- Reduced risk of segmentation faults by utilizing iterators and the at() function
- Compatible with STL algorithms

Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

Run Time Errors - They are also known as exceptions. An exception caught during run time creates serious issues.

Why Exception Handling?

Following are 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 any 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 caller. If the caller chooses not to catch them, then the exceptions are handled by 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 exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling techniques in our code.

In C++, Error handling is done by three keywords:-

- Try
- Catch
- Throw

Syntax

```
Try
{
//code
throw parameter;
}
catch(exceptionname ex)
{
//code to handle exception
}
```

Try

Try block is intended to throw exceptions, which is followed by catch blocks. Only one try block.

Catch

Catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks.

Throw

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.

Example of Exception

below program compiles successful but the program fails during run time.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int a=10,b=0,c;
        c=a/b;
        return 0;
}
```

Implementation of try-catch, throw statement

Below program contains single catch statement to handle errors.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int a=10,b=0,c;
        try //try block activates exception handling
        {
                if(b==0)
                throw "Division by zero not possible";//throws exception
                c=a/b;
}
catch(char* ex)//catches exception
{
        cout<<ex;
}
        getch();
        return 0;
}
```

Output

```
0
```

Example for multiple catch statement

Below program contains multiple catch statements to handle exception.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int x[3]={-1,2,};
        for(int i=0;i<2;i++)
        {
                int ex=x[i];
                try {
                        if (ex > 0)
                                throw ex;
                        else
                                throw 'ex';
                } catch (int ex) {
                                cout << " Integer exception;
                }
                catch (char ex)
                {
                                cout << " Character exception" ;
                }
        }
}
```

Output

```
Integer exception Character exception
```

Example for generalized catch statement

Below program contains generalized catch statement to catch uncaught errors. Catch(…) takes care of all type of exceptions.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int x[3]={-1,2,};
        for(int i=0;i<2;i++)
        {
                int ex=x[i];
                try
                {
                if (ex > 0)
                        throw x;
                else
                        throw 'ex';
                }
                catch (int ex)
                {
                        cout << " Integer exception" ;
                }
                catch (char ex)
                {
                        cout << " Character exception" ;
                }catch (...)
                {
                        cout << "Special exception";
                }
        }
                return 0;
}
```

Output

```
Integer exception Special exception
```

Standard Exceptions in C++

There are standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:

- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic_error** - Exception happens in the internal logical of a program.
    - **domain_error** - Exception due to use of invalid domain.
    - **invalid argument** - Exception due to invalid argument.
    - **out_of_range** - Exception due to out of range i.e. size requirement exceeds allocation.
    - **length_error** - Exception due to length error.
- **runtime_error** - Exception happens during runtime.
    - **range_error** - Exception due to range errors in internal computations.
    - **overflow_error** - Exception due to arithmetic overflow errors.
    - **underflow_error** - Exception due to arithmetic underflow errors
- **bad_alloc** - Exception happens when memory allocation with new() fails.
- **bad_cast** - Exception happens when dynamic cast fails.
- **bad_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
- **bad_typeid** - Exception thrown by typeid.

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way –

```
#include <iostream>
#include <exception>
using namespace std;

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

This would produce the following result −

```
MyException caught
C++ Exception
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Multithreading in C++

Multithreading support was introduced in C+11. Prior to C++11, we had to use POSIX threads or p threads library in C. While this library did the job the lack of any standard language provided feature-set caused serious portability issues. C++ 11 did away with all that and gave us **std::thread**. The thread classes and related functions are defined in the **thread** header file.

std::thread is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.

A callable can be either of the three

- A function pointer

- A function object
- A lambda expression

After defining callable, pass it to the constructor.

What is multi-threaded programming?

Single-threaded programs execute one line of code at a time, then move onto the next line in sequential order (except for branches, function calls etc.). This is generally the default behaviour when you write code. Multi-threaded programs are executing from two or more locations in your program at the same time (or at least, with the illusion of running at the same time). For example, suppose you want to perform a long file download from the internet, but don't want to keep the user waiting while this happens. Imagine how inconvenient it would be if we couldn't browse other web pages while waiting for files to download! So, we create a new thread (in the browser program for example) to do the download. In the meantime, the main original thread keeps processing mouse clicks and keyboard input so you can continue using the browser. When the file download completes, the main thread is signaled so it knows about it and can notify the user visually, and the thread performing the download closes down.

How does it work in practice?

Most programs start off running in a single thread and it is up to the developer to decide when to spin up (create) and tear down (destroy) other threads. The general idea is:

1. Application calls the system to request the creation of a new thread, along with the thread priority (how much processing time it is allowed to consume) and the starting point in the application that the thread should execute from (this is nearly always a function which you have defined in your application).
2. Main application thread and secondary thread (plus any other created threads) run concurrently
3. When the main thread's work is done, or if at any point it needs to wait for the result of a secondary thread, it waits for the relevant thread(s) to finish. This is (misleadingly) called a join operation.
4. Secondary threads finish their work and may optionally signal to the main thread that their work is done (either by setting a flag variable, or calling a function on the main thread)
5. Secondary threads close down
6. If the main thread was waiting on a join operation (see step 3), the termination of the secondary threads will cause the join to succeed and execution of the main thread will now resume

Threads

A **Thread** is a part of our software (Code Segment) that can run independently of the rest of the process. For example, one process with two threads will share the Code Segment, Data Segment and Heap. Much of the Process State can be shared as well; however, each thread will need some thread-specific state information held aside. In addition, each thread needs its own Stack space to keep track of the function and method calls.

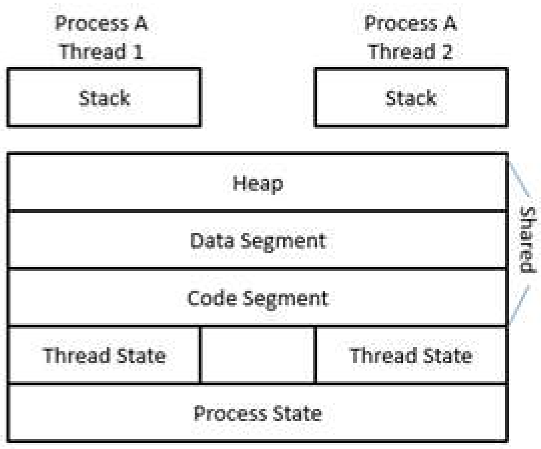


Fig. 5.1 Thread

Why create threads?

There are a few main reasons:

- You need to perform a task which takes a long time to complete but don't want to make the user wait for it to finish. This is called task parallelism and the threads which perform the long-running task are usually called worker threads. The purpose of creating the worker thread or threads is to ensure the application maintains responsiveness in the user interface, rather than to actually make the task run faster. Importantly, the task must be able to run largely independently of the main thread for this design pattern to be useful.
- You have a complex task which can gain a performance advantage by being split up into chunks. Here you create several worker threads, each dedicated to one piece of the task. When all the pieces have completed, the main thread aggregates the sub-results into a final result. This pattern is called the parallel aggregation pattern. See notes about multi-core processors below. For this to work, portions of the total result of the task or calculation must be able to be calculated independently – the second part of the result cannot rely on the first part etc.
- You want to serve possibly many users who need related but different tasks executing at the same time, but the occurrence time of these tasks is unpredictable. Classic examples are a web server serving many web pages to different users, or a database server servicing different queries. In this case, your application creates a set of threads when it starts and assigns tasks to them as and when they are needed. The reason for using a fixed set of threads is that creating threads is computationally expensive (time-consuming), so if you are serving many requests per second, it is better to avoid the overhead of creating threads every time a request comes in. This pattern is called *thread pooling* and the set of threads is called the thread pool.

This simple example code creates 5 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit().

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;
```

```
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
  long tid;
  tid = (long)threadid;
  cout << "Hello World! Thread ID, " << tid << endl;
  pthread_exit(NULL);
}

int main () {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;

  for( i = 0; i < NUM_THREADS; i++ ) {
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);

    if (rc) {
      cout << "Error:unable to create thread," << rc << endl;
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Data collection

A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. They typically represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).