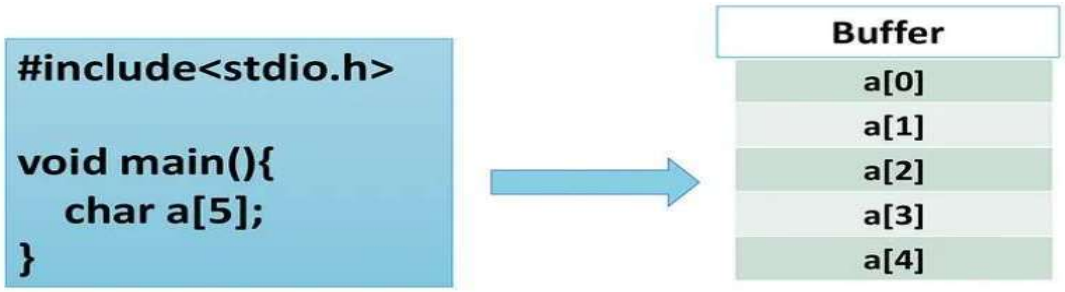
A Beginner's Guide to Buffer Overflow Vulnerability

Buffer

A `Buffer` is temporary storage usually present in the physical memory used to hold data.



Consider the most useless program ever made shown on the left image where a character buffer of length 5 is defined. In a big cluster of memory, a small memory of 5 bytes would be assigned to the buffer which looks like the image on the right.

Buffer Overflow

A `Buffer Overflow` occurs when more data is written to a specific length of memory such that adjacent memory addresses are overwritten.

DEMO (Controlling Local Variables):

Let's take an example of a basic authentication app which asks for a password and returns `Authenticated!` if the password is correct.

Without really knowing how the app works, let's enter a random password.

```
root@kali:~/Desktop/buffer-overflow/demo1# ./demo1
Enter password: randompassword
Authentication declined!
```

It says `Authentical Declined` since the password wasn't correct. To test, we need to enter large random data.

```
root@kali:~/Desktop/buffer-overflow/demo1# ./demo1
Enter password: verysuperbigabsolutelyrandompassword
Authenticated!
Segmentation fault
```

You must be wondering why it got authenticated and why there is a `Segmentation Fault!`. Let's see a more detailed version of the app.

```
root@kali:~/Desktop/buffer-overflow/demo1# ./demo2
Enter password: randompassword
usr_pass: randompassword
sys_pass: Secret
auth: 0
usr_pass   addr: 0xbffff25c
sys_pass   addr: 0xbffff26c
auth       addr: 0xbffff27c
Authentication declined!
```

As you can see, there are three variables: `auth`, `sys_pass` and `usr_pass`. The `auth` variable determines if the user is authenticated or not depending on the value(initially 0). The `usr_pass` stores the password that the user enters and the `sys_pass` variable is what the correct password is.

How the app works is if the `usr_pass` variable is equal to `sys_pass` then the `auth` variable becomes `1`. If the `auth` variable is not `0`, then the user is authenticated.

You may also see how the variables are stored in memory. Since the address is in `hexadecimal` and there is a difference of 1 therefore, `usr_pass` and `sys_pass` variables are buffers of length 16.

To test for Buffer Overflow, a long password is entered as shown.

```
root@kali:~/Desktop/buffer-overflow/demo1# ./demo2
Enter password: verysuperbigabsolutelyrandompassword
usr_pass: verysuperbigabsolutelyrandompassword
sys_pass: lutelyrandompassword
auth: 1685221239
usr_pass   addr: 0xbffff25c
sys_pass   addr: 0xbffff26c
auth       addr: 0xbffff27c
Authenticated!
Segmentation fault
```

As you can see the password entered in `usr_pass` variable overflows the `sys_pass` variable and then the `auth` variable.

Note: C functions like `strcpy()`, `strcmp()`, `strcat()` do not check the length of the variable and can overwrite later memory addresses which is what precisely buffer overflow is.

Refer to the code below for better understanding.

```
#include <stdio.h>

int main(void) {

    int auth = 0;
    char sys_pass[16] = "Secret";
    char usr_pass[16];

    printf("Enter password: ");
    scanf("%s", usr_pass);

    if (strcmp(sys_pass, usr_pass) == 0) {
        authorized = 1;
    }

    printf("usr_pass: %s\n", usr_pass);
    printf("sys_pass: %s\n", sys_pass);
    printf("auth: %d\n", authorized);
    printf("sys_pass   addr: %p\n", (void *)sys_pass);
    printf("auth       addr: %p\n", (void *)&authorized);

    if (auth) {
        printf("Authenticated!\n");
    }
    else{
        printf("Authentication declined!\n");
        }
}
```

Note: This might be the most unrealistic example and only meant for understanding purposes. You may not see such situations in real life.

Let's dive a little deeper into the concepts now.

Problem with gets()/scanf()

Try with this below example

```
#include <stdio.h>
#include<string.h>
char ch[5]="Seqw";
char uch[5]; // if you define this as a local variable there is a possibility of stack smashing
int auth=0;
int main()
{
  //char ch[5]="Seqw";
  //char uch[5];
  scanf("%s",uch);//input a string with 30 characters
  //gets(uch);
  //printf("\n%d",strlen(ch));
  if(strncmp(ch,uch,5)==0)
  {
     auth=1;
     printf("authvalue=%d\n",auth);
  }
  printf("auth=%p\t%d\n",(void*)&auth,auth);
  printf("ch=%p\t%s\n",(void*)ch,ch);
  printf("uch=%p\t%s\n",(void*)uch,uch);
  if(auth)
  printf("Auth Success\n");
  else
  printf("Auth Failed\n");
  return 0;
}
```

Strcmp Vs Strncmp

The only difference between **strncmp** and **strcmp** in C is that strncmp compares only the first N characters while strcmp continues until the termination conditions are met. The termination conditions for both strncmp and strcmp are:

- if two characters are different
- if null character (\0) is reached for any of the strings

Both strncmp and strcmp can be used to compare strings in C but it is **strongly advised to use strncmp** as on setting N correctly, it prevents buffer overflow which is a real problem in production systems and cause several security concerns.

Consider two null terminated strings S1 and S2.

When we use strcmp, we have to compare the entire string till the end (null character). In case of strncmp, we can customize this is compare only the first few characters. This is useful in reality as in production system, comparisons are limited to only a section of the data. Even in most platforms/ softwares, only the first character is used for comparison.

Comparing only limited number of characters defines an upper limit of comparison and makes the program behaviour deterministic and more efficient.

Let us take an example:

S1	O	P	E	N	G	E	N	U	S	null
S2	O	P	E	N	O	E	N	U	8	null
Difference	0	0	0	0	-8	0	0	0	27	-

In this, we are comparing "OPENGENUS" with "OPEN0ENU8". Carefully, observe the difference row.

Points to note:

- If we use strcmp, it will return -8 as it is the first mismatched difference.
- If we use strncmp over the entire string, it will return -8 as well.
- If our system matches only the first 4 characters, we can use strncmp with N=4 and it will return equal string. This is not directly possible with strcmp.

Now, consider the case with non null terminated strings.

S1	O	P	E	N	G	E	N	U	S	garbage
S2	O	P	E	N	G	E	N	U	S	garbage
Difference	0	0	0	0	0	0	0	0	0	can be anything

Both strings are same but does not have null character at the end. The last character is a garbage value that the value that was stored in the memory location in previous execution of any program.

It is likely to be different and will give different results for different system states. If it is same, the comparison will move to the next character/ memory location which will have garbage as well.

Thus, theoretically, all garbage value can be same and strcmp can end up comparing memory of the entire system. The major concern is that it is accessing memory outside of what it is required to which can lead to security issues.

Thus, in both ways, **performance and security**, strncmp should be preferred over strcmp.

Try this example…

```
#include <stdio.h>
#include <string.h>

int main()
{
   char *s1="opengenius";
   char *s2="openjeniu8";
   printf("%d\n",strcmp(s1,s2));
   printf("%d\n",strncmp(s1,s2,9));
   printf("%d\n",strncmp(s1,s2,4));
   char t1[]={'o','p','e'};
   char t2[]={'o','p','e'};
   printf("%d\n",strcmp(t1,t2));
   printf("%d\n",strncmp(t1,t2,3));
   printf("%d\n",strncmp(t1,t2,2));
```

```
    printf("%d\n",strncmp(t1,t2,4));
    return 0;
}
```

Strcat Vs Strncat

Functions like `strcpy()` or `strcat()` are able to cause a fatal failure on the software system such as an unintentional memory overwrite, when at least one of its arguments, which is expected to be a string of characters, is not properly terminated. In general, when the software crashes it is very hard to find the cause of this kind of failure because it does not manifest immediately.

Lets look at an example

Let's suppose the following code fragment, where frame is an array of 12 characters and `str` is a pointer to `char`. The length of `str` should be less than 4 characters, without including the terminating null character itself. This code calls `strcat()` to append `str`, which is retrieved from `Geo_getLatitude()`, to frame.

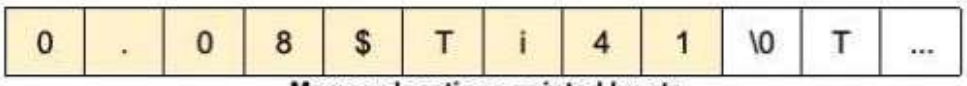
```
char frame[12], *str;
…
str = Geo_getLatitude(position);
frame = strcat(frame, str);
```

Before executing the function `strcat()`, frame contains the string "@078,".

@	0	7	8	,	\0	\0	\0	\0	\0	\0	\0

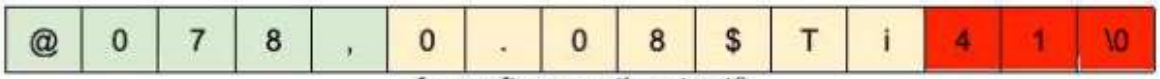
***frame* before executing strcat()**

The pointer `str` points to a memory location, an array of characters, that looks as shown below. Note the length of the string `str` is larger than expected, because it should contain the null character where the character '$' is located instead.



Memory locations pointed by str

In this context, executing `strcat()` causes a software failure in a quiet way, because the string "0.08$Ti4" was effectively appended to the variable frame but overwriting its consecutives memory locations, shaded in red, which had not been allocated for the variable frame.



***frame* after executing strcat()**

To avoid such situations we can use strncat…

It is recommended by many of the programmers that strncat() is safe as compared to strcat() because strcat() does not check for the size of the copied data, and copies until it gets to a null terminator, it might cause a buffer overflow while strncat() check for the size of the copied data, and will copy only 'n' bytes.

Try this example..

```
#include <stdio.h>
#include <string.h>
int main()
{
  // Take any two strings
  char src[10] = "Amrita";
  char dest1[50] = "CyberSecurity";
  char dest2[10] = "TIFAC";
  printf("Before strcat() function execution, ");
  printf("destination string : %s\n", dest1);
  // Appends the entire string of src to dest1
  strcat(dest1, src);
```

```
  // Prints the string
  printf("After strcat() function execution, ");
  printf("destination string : %s\n", dest1);
  printf("Before strncat() function execution, ");
  printf("destination string : %s\n", dest2);
  // Appends 3 characters from src to dest2
  strncat(dest2, src, 3);
  // Prints the string
  printf("After strncat() function execution, ");
  printf("destination string : %s\n", dest2);
  return 0;
}
```

Off by One error

An off-by-one condition is a logic error in size calculation when working with strings and arrays. It usually produces a boundary condition, which may lead to memory corruption or buffer overflow.

```
#include <stdio.h>
#include <string.h>
#define MAX_CHAR   20
```

#define MAX_VALUE 30 //the programmer has specified the wrong define directive variable to read the filename array. This mistake forces the code to read a few bytes beyond the bounds of the filename array; consequently, it prints random data to memory.

```
int main ()
{
      int x;
      int ite_loop = 0;
      char filename[MAX_CHAR] = "AAAAAAAAAAAAAAAAAA";
      printf("Length of filename array: %d\n",strlen(filename));
      for (x = 0; x <= MAX_VALUE; x++) {
            printf("%c",filename[x]);
            ite_loop += 1;
```

```
}
printf("\nIterations: %d\n",ite_loop);
char buf2[8]="buffer:";
char buf1[8]="buffer:";
char input[2]="H";
printf("%d",sizeof(buf2));
printf("%d",strlen(buf2));
printf("\n%s",buf2);
printf("\n%c",buf2[7]);
if(!strcmp(buf1,buf2))
{
   printf("\nAuthenticated");
}
else
{
   printf("\nNot Authenticated");
}
strncat(buf2, input, sizeof(buf2)-strlen(buf2));//The improper usage of the strncat function (third argument) produces an off-by-one condition
printf("%d",sizeof(buf2));
printf("%d",strlen(buf2));
printf("\n%s",buf2);
printf("\n%c",buf2[7]);
if(!strcmp(buf1,buf2))
{
   printf("\nAuthenticated");
}
else
{
   printf("\nNot Authenticated");
}
```

```
    return 0;
}
```

Some other string functions

strcpy() transfers the source string (including its null-terminator) into the destination buffer without regard for the size of either buffer. This may result in a buffer overflow if the destination buffer is smaller than the source string.

strncpy() enables specifying a maximum number of characters to copy from the source string, preventing exceeding the destination buffer's capacity. However, it might not append a null-terminator if there isn't enough space left after copying. This may cause Faulty Null Termination

memcpy() function transfers a specified number of bytes from the source buffer to the destination buffer, irrespective of any null-terminating characters present within the copied data. This outperforms strcpy(), because no need to check for null-terminating characters during copying.

strdup() duplicates a string to a location that will be decided by the function itself. Function will copy the contents of string to certain memory location and returns the address to that location. 'strdup' sounds like short form of **"string duplicate".**

Strdup is equivalent to

```
ptr2 = malloc(strlen(ptr1)+1);
strcpy(ptr2,ptr1);
```

Example for strdup

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
char myname[10];
int main() {
 strcpy(myname,"Amrita Vishwa Vidyapeetham");
 char* name;
 name = strdup(myname);
 printf("\nname=%p\t%s\n",(void*)name,name);
 printf("\nmyname=%p\t%s\n",(void*)myname,myname);
 free(name);
 return 0;
}
```

Some possible mitigations from overflows… are the following

1) Use snprintf() which always appends null character to the end of string but if there is no space output will be truncated.

```
Example of null-terminating snprintf

#include<stdlib.h>

#include<string.h>

#include<stdio.h>

int main(){

char mystr[5];

snprintf(mystr, sizeof(mystr),"%s","apple");

printf("%s",mystr);

return 0;

        }
```

2) The strlcpy() and strlcat() functions copy and concatenate strings with the same input parameters and output result as snprintf() means guarantee NUL-termination if there is room.
3) Always ensure input strings are within expected lengths before processing them.
4) Consider using dynamic memory allocation (eg: malloc())
5) Use safe functions like strcat_s, strcpy_s(), strncat_s, strncpy_s etc..(But GCC (or rather, glibc) does not support this. MSVC includes those functions also safe C libraries).
6) In case of strcpy_s(), function copies characters from a source string to a destination character array up to and including the terminating null character. The strcpy_s() function succeeds only when the source string can be fully copied to the destination without overflowing the destination buffer. The function returns 0 on success, Otherwise, a nonzero value is returned. Good than strlcpy() because no truncation…
7) Below Figure 1 shows the build error (in MSVC) while building the program with strcpy() and recommend using strcat_s(). In later versions of Visual Studio, additional security checks (CERT) are enabled by default on new projects, which makes the warning you see be treated as an error. But in Figure 2 no build error (in MSVC) because strcpy() is commented. Figure 3 shows while running Line number 17 no room to store in the destination. You can disable the CERT warnings you saw by changing the properties as follows. Right-click on Project-> select Properties. In the Property pages go to C/C++-> Preprocessor -> add _CRT_SECURE_NO_WARNINGS to the preprocessor definitions. Figure 4 shows the settings before adding the definition. Figure 5 and 6 show settings after adding the definition and the output without any build error while using strcpy() respectively.

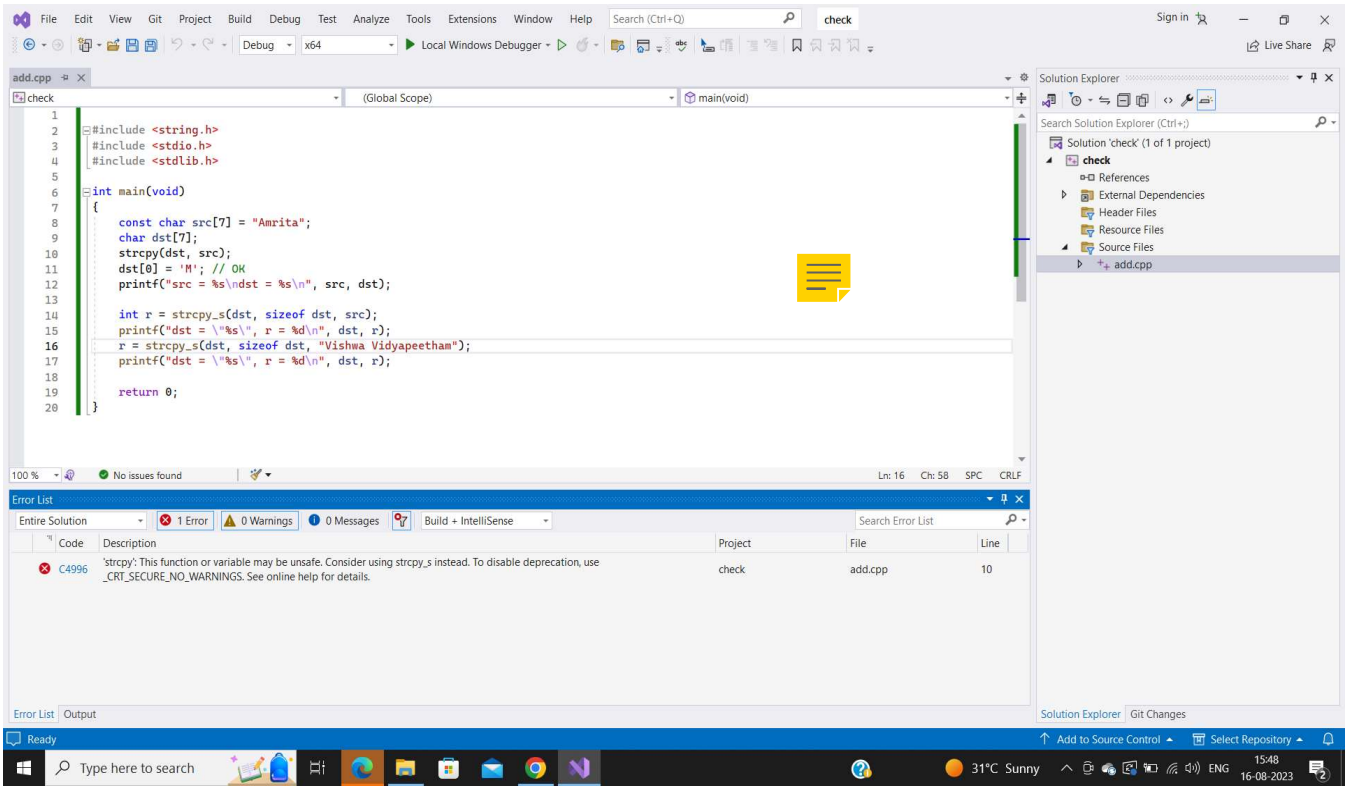


Figure 1

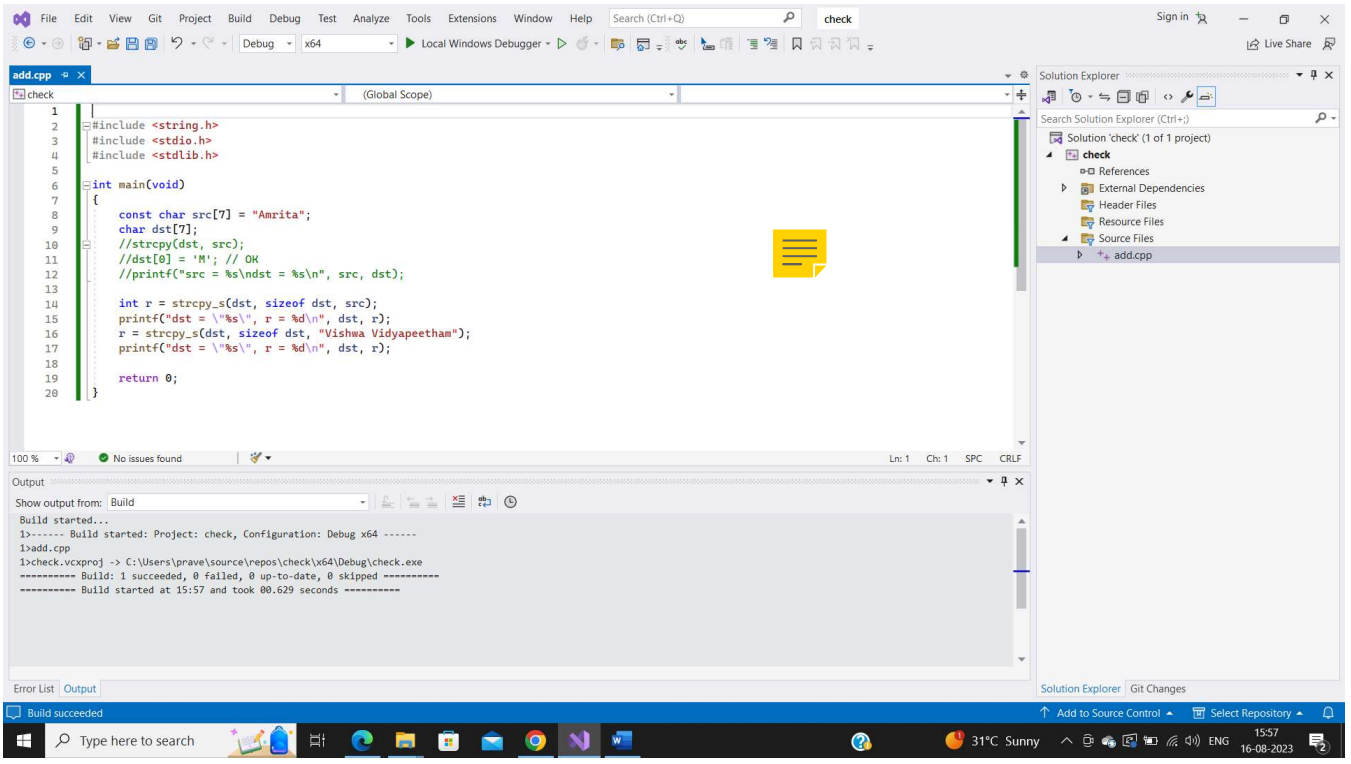


Figure 2

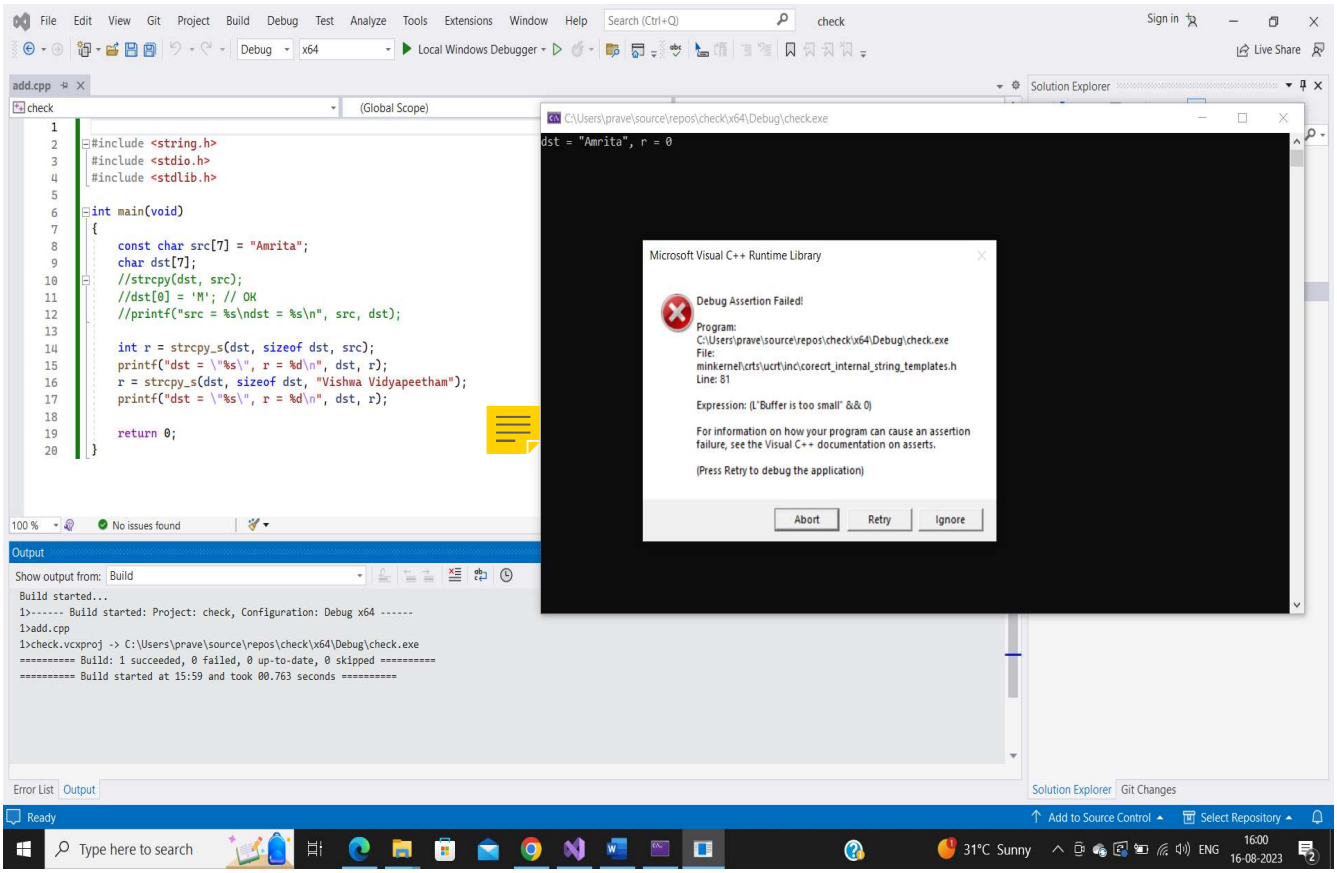


Figure 3

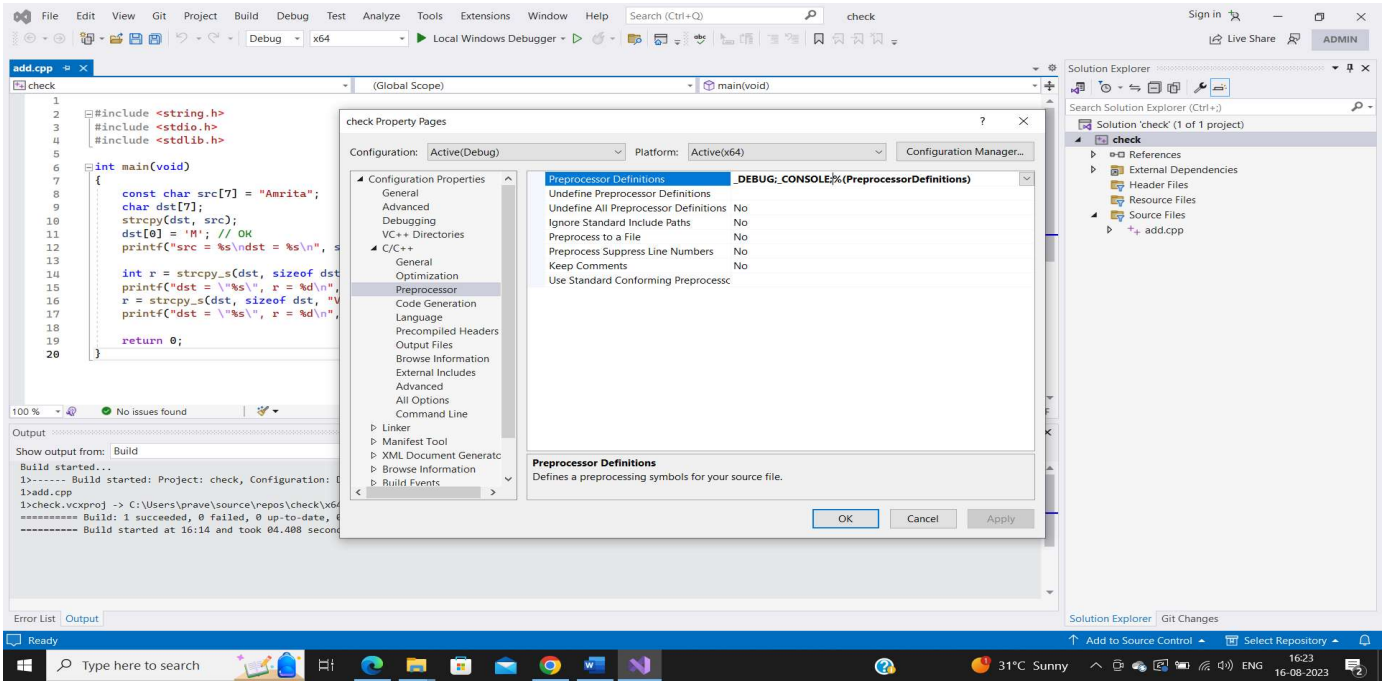


Figure 4

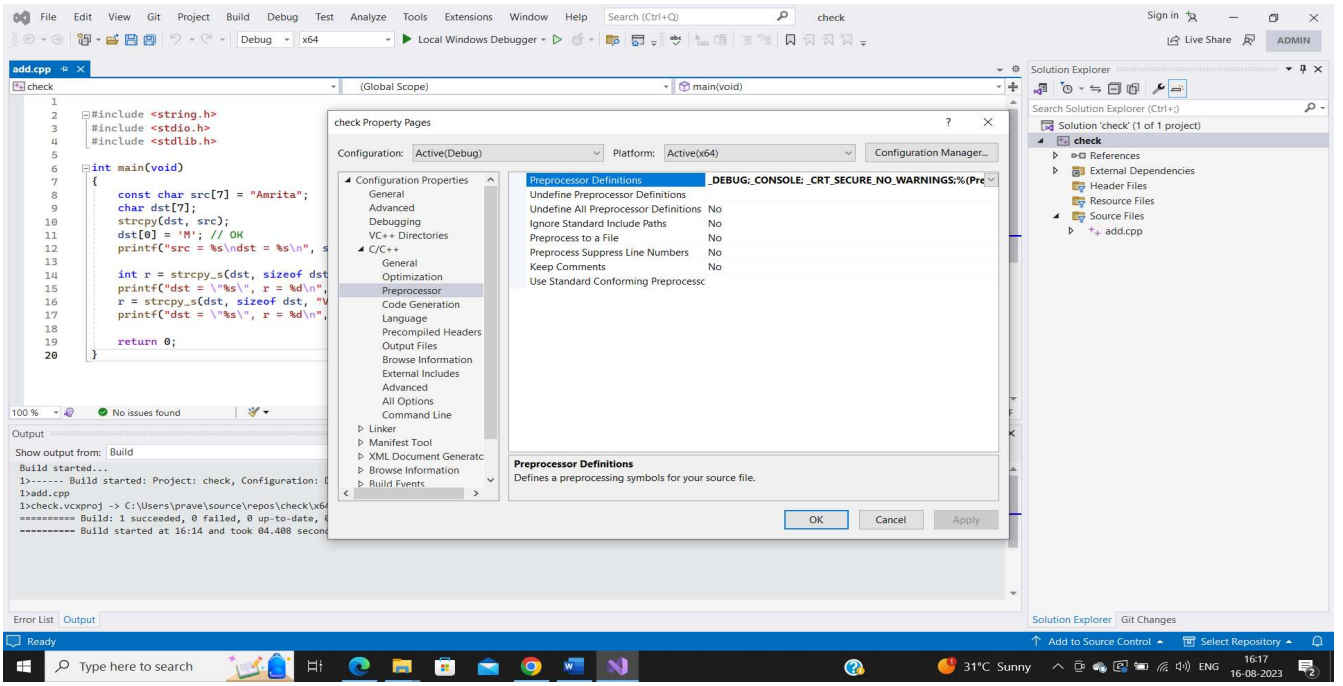


Figure 5

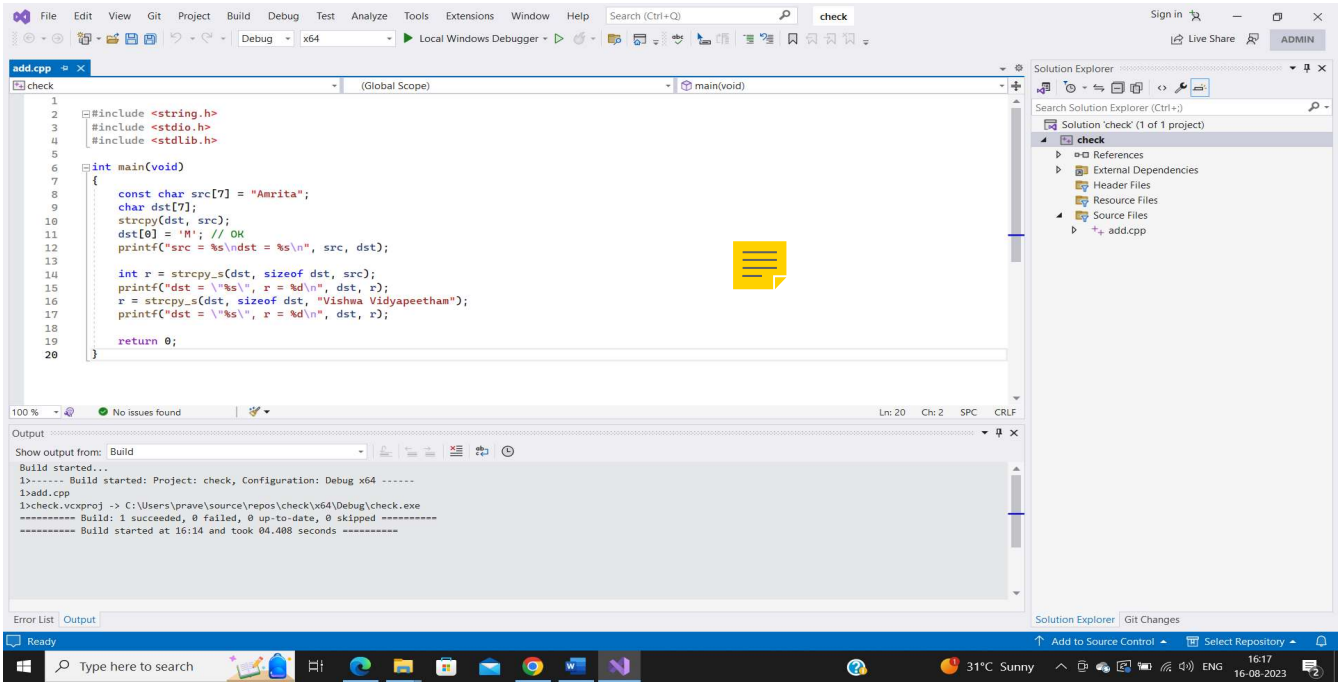


Figure 6

Wide characters

Wide characters are like character data. The main difference is that char takes 1-byte space, but a wide character takes 2-bytes (sometimes 4-byte depending on the compiler) of space in memory. For 2-byte space wide characters can hold 64K (65536) different characters. So the wide char can hold UNICODE characters. The UNICODE values are international

standard that allows for the encoding of characters virtually for any character of any language.

```
#include<iostream>

#include<string>

using namespace std;

int main() {

  wchar_t ch1 = L'a';

  wcout << "The wide characters and its size: " << endl;

  wcout << ch1 << endl<< sizeof(ch1)<< endl;

  wstring msg = L"Hello world!";

  wcout << msg<< endl<< sizeof(msg)<<endl;

  size_t const N{3};

  wstring wstr[N];

    wstr[0] = L"Latin alphabet (\""

              L"\\u0041 \\u0042 . . . \\u0059 \\u005a"

              L"\"): "

              L" \u0041 \u0042 . . . \u0059 \u005a";

    wstr[1] = L"Croatian letters (\""

              L"\\u01c4 \\u01c5 \\u01c6 \\u01c7 \\u01c8 \\u01c9 \\u01ca \\u01cb \\u01cc"

              L"\"): "

              L" \u01c4 \u01c5 \u01c6 \u01c7 \u01c8 \u01c9 \u01ca \u01cb \u01cc";

    wstr[2] = L"Romanian letters (\""

              L"\\u0218 \\u0219 \\u021a \\u021b"

              L"\"): "

              L" \u0218 \u0219 \u021a \u021b";

    for ( size_t i = 0; i < N; ++i ) {
```

```
            wcout << wstr[i] << endl<< sizeof(wstr[i])<< endl;
    }
}
```

wsncopy

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C wide string (which is signaled by a *null wide character*) is found before *num* characters have been copied, *destination* is padded with additional *null wide characters* until a total of *num* characters have been written to it. No *null wide character* is implicitly appended at the end of *destination* if *source* is longer than *num* (thus, in this case, *destination* may not be a null terminated C wide string).

```
#include <wchar.h>

int main ()

{

  wchar_t wcs1[] = L"To be or not to be";

  wchar_t wcs2[40];

  wchar_t wcs3[40];

  /* copy to sized buffer (overflow safe): */

  wcsncpy ( wcs2, wcs1, 40 );

  /* partial copy (only 5 characters): */

  wcsncpy ( wcs3, wcs2, 5 );

  wcs3[5] = L'\0';   /* null character manually added */

  wprintf (L"%ls\n%ls\n%ls\n",wcs1,wcs2,wcs3);

  return 0;

}
```

Explore more on strings……