

Argument Arrays & String Tokenization

Sanjaya Kumar Jena

ITER, Bhubanewar



Kay A. Robbins, & Steve Robbins

Unix™ Systems Programming
Communications, concurrency, and Treads
Pearson Education



Brain W. Kernighan, & Rob Pike

The Unix Programming Environment
PHI

Agenda

- 👉 Command line arguments and Argument arrays
- 👉 `strtok()` & `strtok_r()` library functions for string tokenization
- 👉 Argument array creation (version-I, version-II, version-III)

Command line arguments and Argument arrays

☞ **Token** is a string of characters containing no white space unless quotation marks are used to group tokens.

Command line arguments and Argument arrays

- **Token** is a string of characters containing no white space unless quotation marks are used to group tokens.
- A **command line** consists of tokens (the arguments) that are separated by white space: blanks, tabs or a backslash (\) at the end of a line.

Command line arguments and Argument arrays

- ☞ **Token** is a string of characters containing no white space unless quotation marks are used to group tokens.
- ☞ A **command line** consists of tokens (the arguments) that are separated by white space: blanks, tabs or a backslash (\) at the end of a line.
- ☞ When a user enters a command line corresponding to a C executable program, the **shell** parses the command line into tokens and passes the result to the program in the form of an argument array.

Command line arguments and Argument arrays

- ☞ **Token** is a string of characters containing no white space unless quotation marks are used to group tokens.
- ☞ A **command line** consists of tokens (the arguments) that are separated by white space: blanks, tabs or a backslash (\) at the end of a line.
- ☞ When a user enters a command line corresponding to a C executable program, the **shell** parses the command line into tokens and passes the result to the program in the form of an argument array.
- ☞ An **argument array** is an array of pointers to strings. The end of the array is marked by an entry containing a `NULL` pointer.

Command line arguments and Argument arrays

- ✎ **Token** is a string of characters containing no white space unless quotation marks are used to group tokens.
- ✎ A **command line** consists of tokens (the arguments) that are separated by white space: blanks, tabs or a backslash (\) at the end of a line.
- ✎ When a user enters a command line corresponding to a C executable program, the **shell** parses the command line into tokens and passes the result to the program in the form of an argument array.
- ✎ An **argument array** is an array of pointers to strings. The end of the array is marked by an entry containing a `NULL` pointer.

Example

Consider the command line input: `$ mine -c 10 2.0`

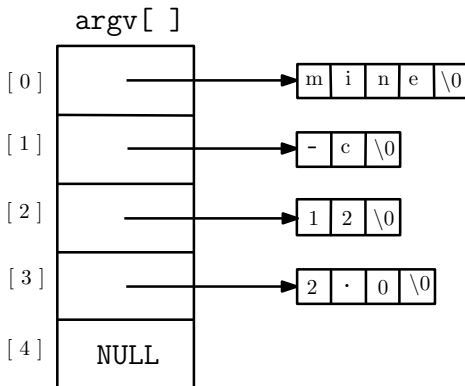
- ✎ Four tokens: (1) `mine`, (2) `-c`, (3) `10`, (4) `2.0`
- ✎ First token is the name of the **command** or **executable**
- ✎ The `mine` program might start with the following line

```
int main(int argc, char *argv[])
```
- ✎ `argc` parameter contains the number of command-line tokens or arguments and `argv` is an array of pointers to the command-line tokens.

The `argv` is an example of an **argument array**

Argument array structure

An argument array is an array of pointers to strings. The end of the array is marked by an entry containing a `NULL` pointer.



The `argv` array for the call `mine -c 10 2.0`

Argument arrays are also useful for handling a variable number of arguments in calls to `execvp` and for handling environment variables.

Printing all command line arguments to standard output

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for(i=0;i<argc;i++)
        printf("argv[%d]: %s\n",i,argv[i]);
    return 0;
}
```

Alternative code for the argument processing loop;

Printing all command line arguments to standard output

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for(i=0;i<argc;i++)
        printf("argv[%d]: %s\n",i,argv[i]);
    return 0;
}
```

Alternative code for the argument processing loop;

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for(i=0;argv[i]!=NULL;i++)
        printf("argv[%d]: %s\n",i,argv[i]);
    return 0;
}
```

C Library Function strtok()

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

```
char *strtok(char *str, const char *delim);
```

- 📖 `strtok` splits a string into tokens.
- 📖 On the first call to `strtok()` the string to be parsed should be specified in `str`.
- 📖 In each subsequent call that should parse the same string, `str` must be NULL. (i.e. the first call to `strtok` is different from subsequent calls)
- 📖 The second argument to `strtok` is a string of allowed token delimiters.

A delimiter is one or more characters that separate text strings. Common delimiters are commas (,), semicolon (;), colon (:), quotes (", '), braces ({}), pipes (|), hyphen (-), or slashes (/ \) etc.

- 📖 Each successive call to `strtok` returns the start of the next token and inserts a '\0' at the end of the token being returned. The `strtok` function returns NULL when it reaches the end of the string, `str`.
- 📖 `strtok` does not allocate new space for the tokens, but rather it tokenizes `str` in place. Thus, if you need to access the original `str` after calling `strtok`, you should save a copy of the string.

Example: strtok() function

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[] ="ITER-IBCS-SHM-SUM-IDS";
    char *token;
    token=strtok(str, "-");
    while (token!=NULL) {
        printf("Token=%s\n", token);
        token=strtok(NULL, "-");
    }
    return 0;
}
```

Example: strtok () to Count Tokens

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[]="ITER-IBCS-SHM-SUM-IDS";
    char *delimiters="-";
    int numtokens;
    /* count the number of tokens in str */
    if (strtok(str, delimiters) != NULL)
        for(numtokens = 1; strtok(NULL, delimiters) != NULL;
            numtokens++) ;

    printf("Number of tokens=%d\n", numtokens);
    return 0;
}
```

A case with strtok () Function

What will be the output of the given code sample?

```
int main()
{
    char str[] = "ITER-IBCS-SOA-SUM-ids";
    char ptr[] = "iter-ibcs-soa-sum-ids-CSE";
    char *token, *ptoken;
    token=strtok(str, "-");
    ptoken=strtok(ptr, "-");
    while (token!=NULL) {
        printf("Token=%s\n", token);
        token=strtok(NULL, "-");
    }
    return 0;
}
```


A case with strtok () Function

What will be the output of the given code sample?

```
int main()
{
    char str[] = "ITER-IBCS-SOA-SUM-ids";
    char ptr[] = "iter-ibcs-soa-sum-ids-CSE";
    char *token, *ptoken;
    token=strtok(str, "-");
    ptoken=strtok(ptr, "-");
    while (token!=NULL) {
        printf("Token=%s\n", token);
        token=strtok(NULL, "-");
    }
    return 0;
}
```

The behavior that causes the above code to fail also prevents **strtok** from being used safely in programs with multiple threads. If one thread is in the process of using **strtok** and a second thread calls **strtok**, subsequent calls may not behave properly. POSIX defines a thread-safe function, **strtok_r**, to be used in place of **strtok**. The **_r** stands for reentrant, indicating the function can be reentered (called again) before a previous call finishes.

Reason For Above Behavior of `strtok()`

- ☞ Because of `strtok` definition, it must use an internal static variable to keep track of the current location of the next token to parse within the string.
- ☞ However, when calls to `strtok` with different parse strings occur in the same program, the parsing of the respective strings may interfere because there is only one variable for the location.

C Library Function `strtok_r()`

```
#include <string.h>

char *strtok_r(char *str, const char *delim, char **saveptr
);
```

The `strtok()` and `strtok_r()` functions return a pointer to the next token, or `NULL` if there are no more tokens.

- 🔗 `strtok_r` splits a string into tokens. The `strtok_r` function is a reentrant version of `strtok`.
- 🔗 The `saveptr` argument is a pointer to a `char` variable that is used internally by `strtok_r()` in order to maintain context between successive calls that parse the same string.
- 🔗 Different strings may be parsed concurrently using sequences of calls to `strtok_r()` that specify different `saveptr` arguments.

Example: strtok_r() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[] = "Lesson-plan-USP-DOS-FML-PLC";
    printf("Entered strin::");
    puts(str);
    char *token;
    char *last;
    token = strtok_r(str, "-", &last);
    while (token!=NULL) {
        printf("Token:%s\n", token);
        printf("\t\tRemaining part of the string:%s\n",last);
        token = strtok_r(NULL, "-", &last);
    }
    return (0);
}
```

strtok() VS strtok_r()

You know the behavior of `strtok()`. An incorrect use of `strtok` to determine the next tokens in the string.

```
int main() {
    char str[] = "ITER-IBCS-SOA-SUM-ids";
    char ptr[] = "iter-ibcs-soa-sum-ids-CSE";
    char *token, *ptoken;
    token=strtok(str, "-");
    ptoken=strtok(ptr, "-");
    while (token!=NULL) {
        printf("Token=%s\n", token);
        token=strtok(NULL, "-");
    }    return 0; }
```

Resolving the issues using `strtok_r()`

```
int main() {
    char str[] = "ITER-IBCS-SOA-SUM-ids";
    char ptr[] = "iter-ibcs-soa-sum-ids-CSE";
    char *token, *ptoken, *sptr1, *sptr2;
    token=strtok_r(str, "-", &sptr1);
    ptoken=strtok_r(ptr, "-", &sptr2);
    while (token!=NULL) {
        printf("Token=%s\n", token);
        token=strtok_r(NULL, "-", &sptr1);
    }    return 0; }
```

Exercise: `strtok()` vs `strtok_r()`

Develop a program to determine the average number of words per line.

Exercise: strtok() VS strtok_r()

Develop a program to determine the average number of words per line.

Sample input to the program: a string of the form This is a line of text\n It is the second line \n then next line". Here, \n is the newline character and *space*(" ") is the word separator.

Exercise: strtok() VS strtok_r()

Develop a program to determine the average number of words per line.

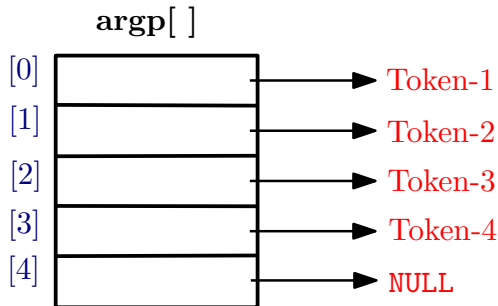
Sample input to the program: a string of the form This is a line of text\n It is the second line \n then next line". Here, \n is the newline character and space(" ") is the word separator.

```
#define LINE_DELIMITERS "\n"
#define WORD_DELIMITERS " "
static int wordcount(char *s) {
    .....
return count;}
double wordaverage(char *s) {
int linecount = 1;
char *nextline;
    .....
words = wordcount(nextline);
    .....
return (double)words/linecount;
}
int main(){
    char str[]="This is a line of text\n It is the second line \n
        then next line";
    double wordavg=wordaverage(str);
    printf("Word average=%f\n",wordavg);
    return 0;}
```


Argument Array Creation

Procedure

- Tokenize the input string using `strtok` or `strtok_r` or by own logic
- Put the tokens into an array. (i.e array of pointers to string)
- Put the last array pointer pointing to NULL as shown as;

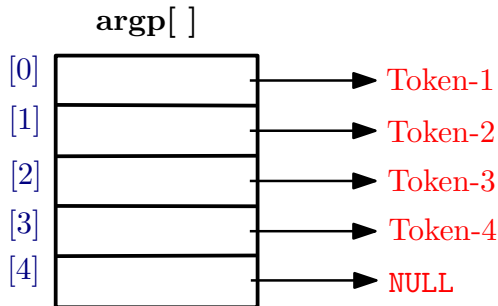


- Display the array

Argument Array Creation

Procedure

- 🔗 Tokenize the input string using `strtok` or `strtok_r` or by own logic
- 🔗 Put the tokens into an array. (i.e array of pointers to string)
- 🔗 Put the last array pointer pointing to NULL as shown as;



- ➡ Display the array

How to Develop an argument array?

Argument Array Creation Version-I

Development of a function `makeargv()` to create an argument array from a string of tokens.

`makeargv` function prototype

```
char **makeargv(char *s);
```

Argument Array Creation Version-I

Development of a function `makeargv()` to create an argument array from a string of tokens.

makeargv function prototype

```
char **makeargv(char *s);
```

main part to invoke makeargv function

```
char **makeargv(char *);  
int main()  
{  
    char s[]="This is a string";  
    char **myargv;  
    myargv=makeargv(s);  
    if(myargv==NULL)  
        fprintf(stderr,"Failed to construct an argument array\n");  
    else{  
        for(int i=0;myargv[i]!=NULL;i++)  
            printf("Myargv[%d]: %s\n",i,myargv[i]);  
    }  
    return 0;  
}
```

makeargv Function Definition for Version I

```
char **makeargv(char *s)
{
    int ntokens,i;
    char *t,**argvp;
    argvp=NULL;
    t=(char *)malloc(sizeof(char)*(strlen(s)+1));
    strcpy(t,s);
    if(strtok(s," ")!=NULL){ /* count the number of tokens in s */
        for(ntokens=1;strtok(NULL," ")!=NULL;ntokens++);
    }
    printf("number of tokens=%d\n",ntokens);
    argvp=(char **)malloc((ntokens+1)*sizeof(char *));
    /* insert pointers to tokens into the argument array */
    *argvp=strtok(t," ");
    for(i=1;i<ntokens;i++){
        *(argvp+i)=strtok(NULL," ");
    }
    *(argvp+ntokens)=NULL; /* put in final NULL pointer */
    return argvp;
}
```

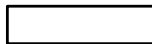
HOW argv Works in myargv Function ?

Declaration

`**argvp`

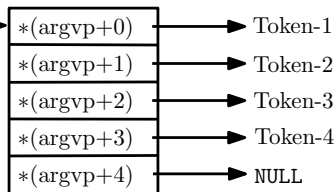
Dynamic memory allocation

Dereference



`argvp`

`*argvp`



In general `* (argvp+i)`

Argument Array Creation Version-II

The following alternative prototype specifies that **makeargv** should pass the argument array as a parameter. This alternative version of **makeargv** returns an integer giving the number of tokens in the input string. In this case, **makeargv** returns -1 to indicate an error.

makeargv function prototype

```
int makeargv(char *s, char ***argvp);
```

main Part for Version II Argument Array

main part to invoke `makeargv` function in version II.

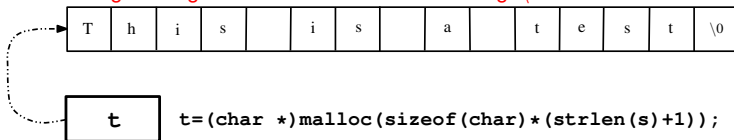
```
int makeargv(char *, char ***); /* Function prototype
    for makeargv */
int main()
{
    char mytest[]="This is a test";
    char **myargv;
    int numtokens,i;
    numtokens=makeargv(mytest, &myargv);
    if(numtokens== -1)
        fprintf(stderr, "Failed to construct an argument
            array\n");
    else{
        for (i = 0; i < numtokens; i++)
            printf("myargv[%d]: %s\n", i, myargv[i]);
    }
    return 0;
}
```


Implementation Strategy of `makeargv`

- ✍ Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.

Implementation Strategy of `makeargv`

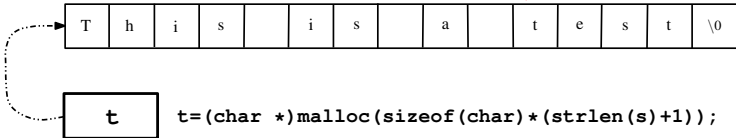
- ✎ Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.



The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.

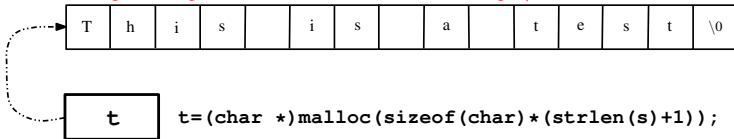


The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

- Copy `s` into `t`. `strcpy(t,s);`

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.

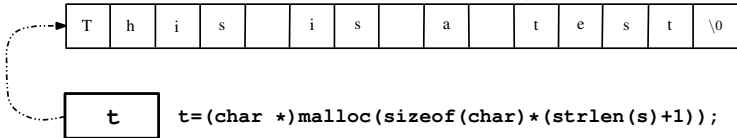


The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

- Copy `s` into `t`. `strcpy(t,s);`
- Make a pass through the string `t`, using `strtok` to count the tokens.

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.



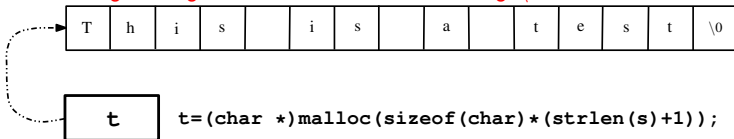
The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

- Copy `s` into `t`. `strcpy(t,s);`
- Make a pass through the string `t`, using `strtok` to count the tokens.

```
if(strtok(t," ")!=NULL){  
    for(numtokens=1; strtok(NULL," ")!=NULL; numtokens++);  
}
```

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.



The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

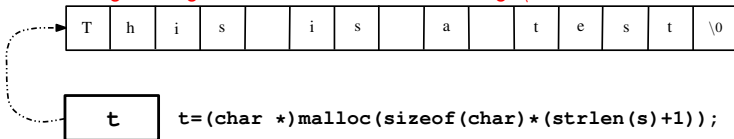
- Copy `s` into `t`. `strcpy(t,s);`
- Make a pass through the string `t`, using `strtok` to count the tokens.

```
if(strtok(t," ")!=NULL){  
    for(numtokens=1; strtok(NULL," ")!=NULL; numtokens++);  
}
```

- Use the count (`numtokens`) to allocate an `argv` array.

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.



The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

- Copy `s` into `t`. `strcpy(t,s);`
- Make a pass through the string `t`, using `strtok` to count the tokens.

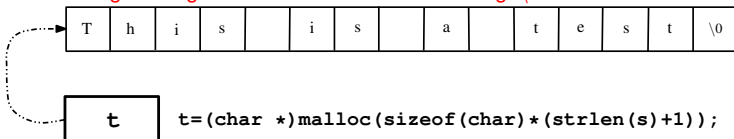
```
if(strtok(t, " ") != NULL) {  
    for (numtokens=1; strtok(NULL, " ") != NULL; numtokens++);  
}
```

- Use the count (`numtokens`) to allocate an `argv` array.

```
*argvp=(char **)malloc((numtokens+1)*sizeof(char *));
```

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.



The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

- Copy `s` into `t`. `strcpy(t,s);`
- Make a pass through the string `t`, using `strtok` to count the tokens.

```
if(strtok(t," ")!=NULL){  
    for(numtokens=1;strtok(NULL," ")!=NULL;numtokens++);  
}
```

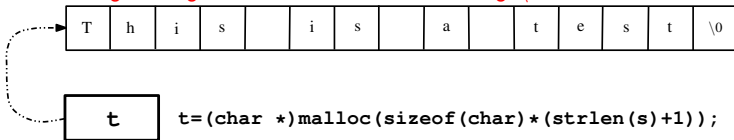
- Use the count (`numtokens`) to allocate an `argv` array.

```
*argvp=(char **)malloc((numtokens+1)*sizeof(char *));
```

- Copy `s` into `t` again. `strcpy(t,s);`

Implementation Strategy of `makeargv`

- Use `malloc` to allocate a buffer `t` for parsing the string in place. The `t` buffer must be large enough to contain `s` and its terminating `'\0'`.



The `makeargv` function makes a working copy of the string `s` in the buffer `t`.

- Copy `s` into `t`. `strcpy(t,s);`
- Make a pass through the string `t`, using `strtok` to count the tokens.

```
if(strtok(t," ")!=NULL){  
    for(numtokens=1;strtok(NULL," ")!=NULL;numtokens++);  
}
```

- Use the count (`numtokens`) to allocate an `argv` array.

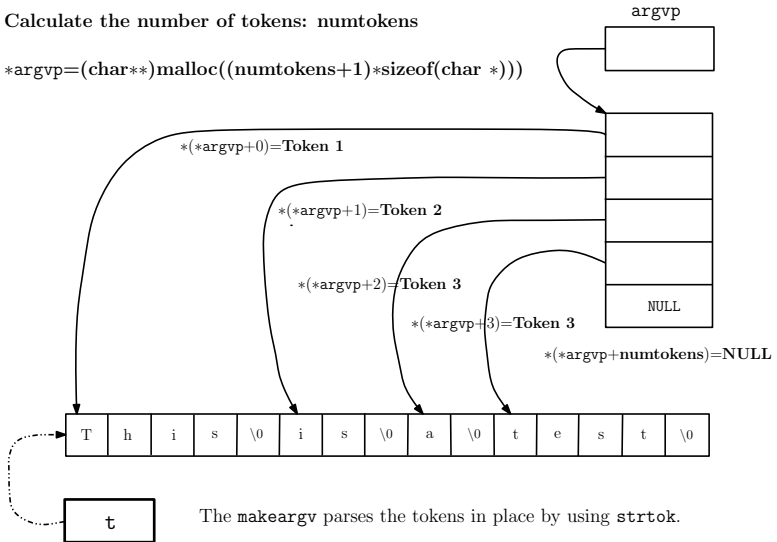
```
*argvp=(char **)malloc((numtokens+1)*sizeof(char *));
```

- Copy `s` into `t` again. `strcpy(t,s);`
- Use `strtok` to obtain pointers to the individual tokens, modifying `t` and effectively parsing `t` in place.

Creating Argument Array for ptrs (argv) to the Tokens

Calculate the number of tokens: numtokens

```
*argv=(char**)malloc((numtokens+1)*sizeof(char *))
```



The makeargv parses the tokens in place by using strtok.

Library Function: `strspn()`

`strspn()` - get length of a prefix substring

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

```
size_t strspn(const char *s, const char *  
              accept);
```

RETURN VALUE:

The `strspn()` function returns the number of bytes in the initial segment of `s` which consist only of bytes from `accept`.

The `strspn()` function calculates the length (in bytes) of the initial segment of `s` which consists entirely of bytes in `accept`.

Example: strstrpn()

strstrpn() - get length of a prefix substring

```
#include<stdio.h>
#include<string.h>
int main()
{
    char *s="--#?This # is a String - ?";
    char *snew;
    size_t len;
    char *accept="----#?";
    printf("Origina string s=%s\n",s);
    len=strspn(s,accept);
    printf("Length of the initial segment of s which
           consists entirely of bytes in accept=%ld\n",len);
    snew=s+len; //shifting base addr of s to len
    printf("Now snew=%s\n",snew);
    return 0;
}
```

makeargv Function Definition for Version II

```
int makeargv(char *s, char ***argvp)
{
    int numtokens=-1,i;
    char *t,*snew;
    *argvp=NULL;
    printf("Original String Length=%ld\n",strlen(s));
    /* To skip over leading delimiters use strspn() */
    /* For more on strspn() refer $ man strspn */
    snew=s+strspn(s, " ");
    printf("Skipping leading delimiters(If any): String Length=%ld\n",strlen(snew));
    t=(char *)malloc(sizeof(char)*(strlen(snew)+1));
    strcpy(t,snew);
    if(strtok(t, " ")!=NULL){
        for(numtokens=1;strtok(NULL, " ")!=NULL;numtokens++);
    }
    *argvp=(char **)malloc((numtokens+1)*sizeof(char *));
    strcpy(t,snew);
    **argvp=strtok(t, " ");
    for(i=1;i<numtokens;i++)
        *(*argvp+i)=strtok(NULL, " ");
    *(*argvp+numtokens)=NULL;
    return numtokens;
}
```

Argument Array Creation Version-III

The following alternative prototype specifies that **makeargv** should pass the argument array and a **delimiter set** as parameters. This third version of **makeargv** returns an integer giving the number of tokens in the input string. In this case, **makeargv** returns -1 to indicate an error.

makeargv function prototype Version III

```
int makeargv(const char *s, const char *  
delimiters, char ***argvp);
```

The **const** qualifier means that the function does not modify the memory pointed to by the first two parameters.

main Part for Version III Argument Array

```
int makeargv(const char *s, const char *delimiters, char ***
    argvp);
int main(int argc, char *argv[]) {
    char delim[] = " \t";
    int i; int numtokens;
    char **myargv;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s string\n", argv[0]);
        return 1;
    }
    if ((numtokens = makeargv(argv[1], delim, &myargv)) == -1) {
        fprintf(stderr, "Failed to construct an argument array for %
            s\n", argv[1]);
        return 1;
    }
    printf("The argument array contains:\n");
    for (i = 0; i < numtokens; i++)
        printf("%d:%s\n", i, myargv[i]);
    return 0;
}
```

The above program that takes a single string as its command-line argument and calls `makeargv` to create an argument array. **TO RUN::** `$/a.out "This is a test"`

Implementation of `makeargv`

Implementation of `makeargv` based on the prototype as;

```
int makeargv(const char *s, const char *  
            delimiters, char ***argvp);
```

The `makeargv` function creates an argument array pointed to by `argvp` from the string `s`, using the delimiters specified by `delimiters`. If successful, `makeargv` returns the number of tokens. If unsuccessful, `makeargv` returns -1.

makeargv Function Definition for Version III

```
int makeargv(const char *s, const char *delimiters, char ***
    argvp)
{
    int numtokens=-1,i;
    char *t;
    const char *snew;
    if ((s == NULL) || (delimiters == NULL) || (argvp == NULL)) {
        return -1;
    }
    *argvp=NULL;
    snew=s+strspn(s,"delimiters");
    t=(char *)malloc(sizeof(char)*(strlen(snew)+1));
    strcpy(t,snew);
    if(strtok(t,"delimiters")!=NULL){
        for(numtokens=1;strtok(NULL," ")!=NULL;numtokens++);
    }
    *argvp=(char **)malloc((numtokens+1)*sizeof(char *));
    strcpy(t,snew);
    **argvp=strtok(t,"delimiters");
    for(i=1;i<numtokens;i++)
        *(*argvp+i)=strtok(NULL," ");
    *(*argvp+numtokens)=NULL;
    return numtokens;
}
```

Summary: Argument Array Version I, II, III

makeargv function prototype Version I

```
char **makeargv(char *s);
```

Summary: Argument Array Version I, II, III

makeargv function prototype Version I

```
char **makeargv(char *s);
```

makeargv function prototype Version II

```
int makeargv(char *s, char ***argvp);
```

Summary: Argument Array Version I, II, III

makeargv function prototype Version I

```
char **makeargv(char *s);
```

makeargv function prototype Version II

```
int makeargv(char *s, char ***argvp);
```

makeargv function prototype Version III

```
int makeargv(const char *s, const char *  
delimiters, char ***argvp);
```