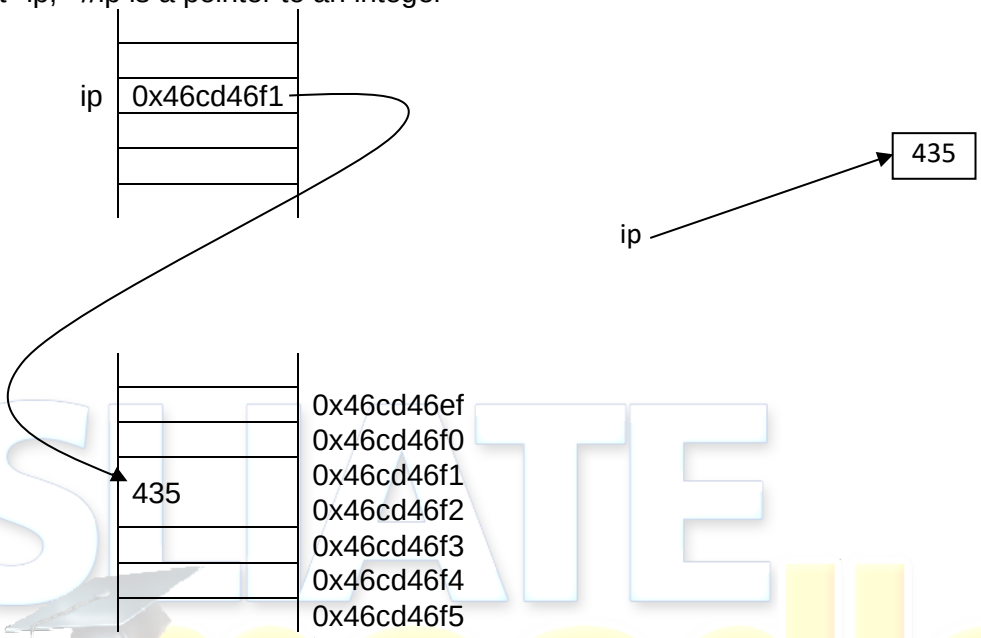
Week 11

Pointers

A pointer points to a memory location which contains data of a particular type. Therefore, the content of a pointer is not data but the address of some data. A pointer is declared by the use of an '*' after the data type.

E.g.: int* ip; //ip is a pointer to an integer



This typing of data is important since a pointer is a variable which can undergo certain arithmetic operations such as '++' which moves a pointer to the next item of data.

Since different data types occupy different sizes of memory block, the data type of the pointer defines how many bytes the pointer must be incremented.

Unlike arrays, pointers may be used as the return type of a function.

One very common application of both arrays and pointers is the representation of strings of characters.

Declaring Multiple Pointers:

When declaring pointers, all the following three statements would have the same effect:

```
char* t;   char *t;   char * t;
```

If you declare more than one variable in the same statement, it might cause confusion.

E.g.: char x, y, z; //three variables of type char

```
char* x, y, z;  //x is char pointer, y and z are variables of type char
char *x, *y, *z;  //three char pointers
```

De-referencing Numeric Pointers:

We can access the data which a pointer is referencing by preceding the pointer name with an asterisk (*).

If we display a pointer (of a numeric data type) then we will see the memory address, not the data in that address.

The address of any variable can be obtained by using '&' (*address of* operator) before the variable name.

Example:

```
#include <iostream.h>
void main()
{
        int x=4; int* y=&x;
        cout<<"Variable is "<<x<<endl;
        cout<<"Pointer is "<<y<<endl;
        cout<<"De-referenced pointer is "<<*y<<endl;
}
```

Output:

```
Variable is 4
Pointer is 0x6317241a
De-referenced pointer is 4
```

De-referencing Character Pointers:

Pointers of type *char* (i.e. strings) behave in a rather different manner to numeric types.

Firstly, *cout* will display the string rather than the address.

Secondly, if you de-reference a char pointer which you are using to represent a string you will get the single character which the pointer is addressing.

Example:

```
#include <iostream.h>
void main()
{
        char* x="C++";
        cout<<"String is "<<x<<endl;
        cout<<"De-referenced char pointer is "<<*x<<endl;
        x++;
```

```
        cout<<"String after incrementing is "<<x<<endl;
        cout<<"De-referenced char pointer is now "<<*x<<endl;
}
```

Output:

String is C++

De-referenced char pointer is C

String after incrementing is ++

De-referenced char pointer is now +

Strings

Unlike some other languages, there is no 'string' data type in C++.

We can only represent strings of characters in two rather indirect ways:

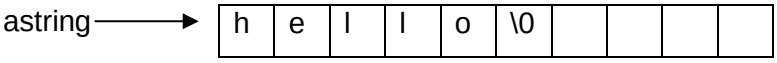
1. As arrays of type char
2. By pointers of type char

An Array of Type char:

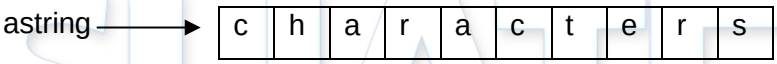
An array of 10 characters would be declared as *char astring[10];*

In fact the maximum string which we could contain in this array would be 9 characters long, since all strings must be terminated by the escape sequence character '\0'.

The following figure shows the contents of *astring* when it contains the string 'hello'.



If *astring* was to contain the string 'characters' then the terminating '\0' would be lost.



Any attempt to display or copy this string would cause problems since the string has an undefined length, and the program would simply display or copy the contents of all memory locations after the string until a '\0' was encountered.

To initialize a character array with a string literal, we can use the assignment operator (=) and {}.

E.g.: char astring[]={"hello"};

We cannot assign a string literal to a character array after it has been declared - this can only be done one character at a time.

Example:

```
#include <iostream.h>
void main()
{
        char astring[4]={"abc"};
        char bstring[4];
        //bstring={"xyz"} will not compile
        bstring[0]='x';
        bstring[1]='y';
        bstring[2]='z';
        bstring[3]='\0';
        cout<<astring<<endl;
```

```
        cout<<bstring<<endl;
}
```

Output:

abc

xyz

Likewise, if we want to make one array of characters equal to another, then unfortunately the assignment operator cannot be used.

E.g.: *bstring* = *astring* is not allowed

So, to copy the contents of *astring* into *bstring*, we can use *strcpy(destination, source)* and *strncpy(destination, source, num_chars)* of the header file *string.h*

E.g.: strcpy(bstring,astring);

strncpy(bstring,astring,9);

Example:

```
#include <iostream.h>
#include <string.h>
void main()
{
        char astring[20];
        char bstring[10];
        cout<<"Enter a string (upto 19 characters): ";
        cin>>astring;
        cout<<astring<<endl;
        strncpy(bstring,astring,9);
        bstring[9]='\0';   //This is important
        cout<<bstring<<endl;
}
```

Output:

Enter a string (upto 19 characters): Superconductivity

Superconductivity

Supercond

Using Pointers to Strings:

Arrays are difficult to manipulate and cannot be used as the return type of a function.

A complementary approach is to use a pointer to reference a string of characters.

To declare a pointer to a string it must be of type *char*.

E.g.: char* str_ptr;

Here, the pointer *str_ptr* is able to point to the first character of a string of characters.

We can use a char pointer to reference an array of characters quite easily.

Example:

```
#include <iostream.h>
void main()
{
        char str[11]={"char array"};
        char* str_ptr=str;
        cout<<str_ptr;
}
```

Output:

```
char array
```

It is easy to assign literals at declaration, requiring only " ".

E.g.: char* str_ptr="A string";

However, this is not a particularly safe way to create strings, since any attempt to assign a longer string to the same pointer can cause other data to be overwritten.

The assignment operator may be used at any time after declaration to make one char pointer equal to another.

Example:

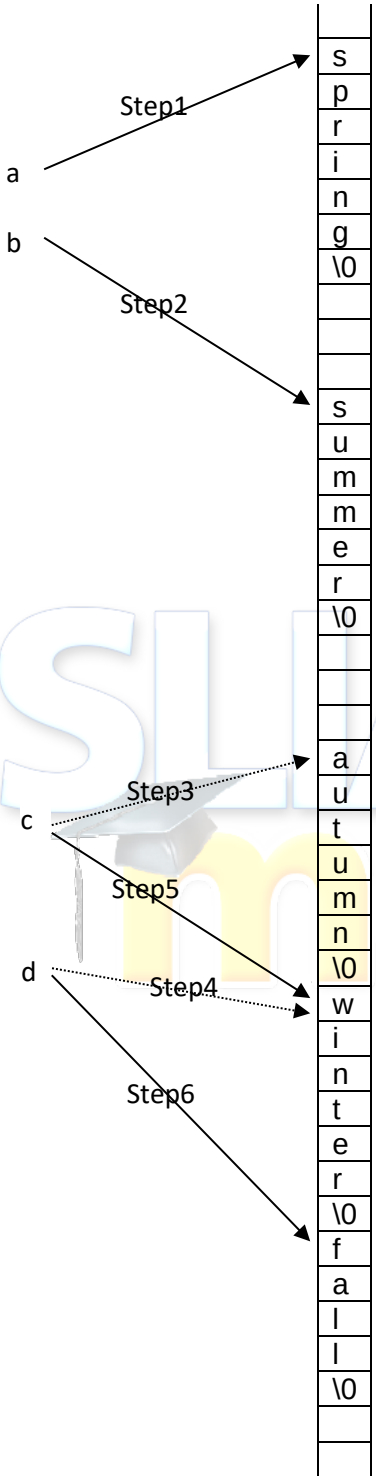
```
#include <iostream.h>
void main()
{
        char a[10]={"spring"};   //Step1
        char b[10]={"summer"};   // Step2
        char* c="autumn";   // Step3
        char* d="winter";  // Step4
        //a=b;   will not compile
        //a={"printemps"}   will not compile
        c=d;   // Step5
        cout<<a<<endl;
        cout<<b<<endl;
        cout<<c<<endl;
        cout<<d<<endl;
```

```
        d="fall";   // Step6
        cout<<c<<endl;
        cout<<d<<endl;
}
```

Output:

```
spring
summer
winter
winter
winter
fall
```

The following figure shows what happens when we execute the above program.



Using the assignment operator with char pointers should only be used for short term process.

For safe storage of strings arrays are more reliable.

It is possible to use *strcpy* or *strncpy* with character pointers to copy data rather than address.

However, this is not a recommended approach since we may overwrite data in subsequent memory locations.

It will happen if we declare a char pointer to point to a string of one length and then copy a longer string to it.

Since char pointer only points to the address of one character, what may be in the subsequent memory locations is unpredictable.

Example:

```
#include <iostream.h>
#include <string.h>
void main()
{
        char* a="fall";
        char* b="winter";
        char* c="summer";
        cout<<a<<endl;
        cout<<b<<endl;
        cout<<c<<endl<<endl;
        strcpy(b,a);
        cout<<a<<endl;
        cout<<b<<endl;
        cout<<c<<endl<<endl;
        strcpy(a,c);
        cout<<a<<endl;
        cout<<b<<endl;
        cout<<c<<endl<<endl;
}
```

Output:

```
fall
winter
summer

fall
fall
summer

summer
```

r

summer

A Strategy for Strings:

In practice, we can see that although pointers provide a flexible means for string manipulation, they can lead to ‘unsafe’ practices in some contexts.

Therefore we are better served by using arrays to store strings, but may use pointers to manipulate them.