LECTURE 14A

Pointers-II

14.5 Pointers and functions

Whenever a portion of the program invokes a function with formal arguments, control will be transferred from the main to calling function and the value of actual argument is copied to the function. Withi n the function, the actual valued copied from the calling portion of the program may be altered or changed. When the control is transferred back from the function to calling portion of the program, the altered value are not transferred back. This type of passing formal arguments to a function is technically known as a call by value. Let us rite a program to understand this statement.

PROGRAM 14.4

```
/* A program to display the contents of a variable before and after the function is
invoked using a call by value*/

#include <iostream>

using namespace std;
void main(void)
   {
      int i;
      int function (int);

      i=20;
      cout<<"value of I before function call ="<<i<<endl;
      function (i);
      cout<<endl;
      cout<<"value of I after function call ="<<i<<endl;
      }
int function(int i)
   {
     int x;
     x = 5*i;
      cout<<"Inside the function, value of i =";
     cout<, x<< endl;
     return (x);
}
```

Output of the above program:

```
value of I before function call =20
```

```
Inside the function, value of i =100
value of I after function call=20
```

The above program shows that changed value of i is not transferred back to calling function main( ). Let us rite one more program to swap two numbers.

Program 14.5 :

```
/* A program to exchange the contents of two variables using call by value*/

#include <iostream>

using namespace std;
void main(void)
   {
      int x,y;
      void swap (int,int);
       x = 100;
      y = 20;
      cout<<"value before swap call ="<<endl;
      cout <<"x="<<x<<"y="<<y<<endl;
      swap(x,y); //call by value
       cout<<"value after swap() ="<<endl;
       cout <<"x="<<x<<"y="<<y<<endl;

      }
void swap(int x, int y)
   {
     int temp;
     temp = x;
      x = y;
     y = temp;
   }
```

```
Out put of the Program;
value before swap()
x =100 and y =20
value after swap()
x =100 and y =20
```

We see that parameters x and y which are passed to function swap are not returned as swapped numbers. Let us see how pointers help us to achieve, what we could not get by swapping numbers by passing parameters by value.

Program 14.6 :

```
/* A program to exchange the contents of two variables using call by reference*/

#include <iostream>

using namespace std;
void main(void)
   {
      int x,y;
      void swap (&int,&int);
       x = 100;
      y = 20;
      cout<<”value before swap()  =”<<endl;
      cout <<”x=”<<x<<”y=”<<y<<endl;
      swap(x,y); //call by value
       cout<<”value after swap() =”<<endl;
       cout <<”x=”<<x<<”y=”<<y<<endl;

      }
void swap(int &x, int &y)// Reference of x and y is passed
   {
     int temp;
     temp = x;
      x = y;
     y = temp;
   }
```

```
Out put of the Program;
value before swap()
x =100 and y =20
value after swap()
x =20 and y = 100
```

14.6 Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

The general syntax of a pointer to a function is,

*return_type (*variables)(list of parameters);*

Let us understand this concept by a program, which gives average of three numbers using pointer to function method. First we will write this program without using pointers and then with pointers. Same program without using Pointers

Program 14.7

```
//Without using Pointer to function
#include <iostream>
using namespace std;
void main(void)
    {
      double average (double, double,double);
       //function declaration
      double a, b, c, avg;
    //  double (*ptrf)(double, double,double);
// Pointer declaration
  //    ptrf = &average;
     cout<<"Enter three numbers\n";
      cin>>a;
        cin>>b;
        cin>>c;
//      avg = (*ptrf)(a,b,c); //function calling using pointer
      cout<<"a="<<a<<endl;
        cout<<"b="<<b<<endl;
       cout<<"c="<<c <<endl;
         avg=average(a,b,c);
       cout<<"Average="<<avg <<endl;
      }

   double average (double x, double y, double z)
      {
       double temp;
        temp = (x + y+ z)/3.0;
         return (temp);
        }
```

Program 14.7a

```
//Pointer to function
#include <iostream>
using namespace std;
void main(void)
    {
      double average (double, double,double);
       //function declaration
      double a, b, c, avg;
      double (*ptrf)(double, double,double);
// Pointer declaration
      ptrf = &average;
     cout<<"Enter three numbers\n";
      cin>>a;
        cin>>b;
        cin>>c;
      avg = (*ptrf)(a,b,c); //function calling using pointer
      cout<<"a="<<a<<endl;
        cout<<"b="<<b<<endl;
       cout<<"c="<<c <<endl;
       cout<<"Average="<<avg <<endl;
      }

   double average (double x, double y, double z)
      {
       double temp;
        temp = (x + y+ z)/3.0;
         return (temp);
        }Output of the above program
```

```
 Enter three numbers
 5   3  10
  a = 5
   b = 3
  c = 10
 Average =6
```

14.7 Passing a function to another function

In this section, how a function can be passed as a formal argument to another function is described using a pointer technique. C++ allows a pointer to pass one function to another function as an argument. The general syntax of passing one function to another function is:

return_type function_name (pointer_to_function(other arguments));

A program to demonstrate, that a function can be passed to another function as an argument, is given below. This program performs addition, subtraction and other operations.

Program 14.8.

```
// Passing a function to another function
#include <iostream>
using namespace std;

void main(void)
{
      float add (float, float);//Function declaration
      float sub (float, float);
      float action (float(*) (float, float),float, float);
      float (*ptrf) (float, float);
      //pointer to function declaration
      float a, b, value;
      char ch;
      cout<<"Passing a function to another function\n";
      cout<<"Enter any two numbers\n";
      cin>>a>>b;
      cout<<"a->addition"<<endl;
      cout<<"s->subtraction"<<endl;
```

```
        cout<<"option, please?\n";
        cin>>ch;
        if(ch=='a')
                ptrf =&add;
        else
              ptrf =&sub;
        cout<<"a="<<a<<endl;
        cout<<"b="<<b<<endl;
        value = action (ptrf, a,b);
        cout<< "Answer=" <<value <<endl;
}
float add (float x, float y)
{
        float ans;
        ans = x+y;
        return (ans);
}
float sub (float x, float y)
{
        float ans;
        ans = x-y;
        return (ans);
}
float action (float (*ptrf)(float, float), float x, float y)
{
        float answer;
        answer = (*ptrf)(x,y);
        return (answer);
}
```

1.8 Pointers and arrays

The concept of arrays is very much bound to the one of pointers. In fact, the *identifier of an array is equivalent to the address of its first element*, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations:

int numbers [20];
*int * p;*

The following assignment operation would be valid:

p = numbers;

After that, p and numbers would be equivalent and would have the same properties. The only difference is that we could change the value of pointer p by another one, whereas numbers will always point to the first of the 20 elements of type int with which it was defined. Therefore, unlike p, which is an ordinary pointer, numbers is an array, and an array can be considered a *constant pointer*. Therefore, the following allocation would not be valid:

numbers = p;

Because numbers is an array, so it operates as a constant pointer, and we cannot assign values to constants.

Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
 int numbers[5];
 int * p;
 p = numbers;  *p = 10;          10, 20, 30, 40, 50,
 p++;  *p = 20;
 p = &numbers[2];  *p = 30;
 p = numbers + 3;  *p = 40;
 p = numbers;  *(p+4) = 50;
 for (int n=0; n<5; n++)
   cout << numbers[n] << ", ";
 return 0;
}
```

In the chapter about arrays we used brackets ([ ]) several times in order to specify the index of an element of the array to which we wanted to refer. Well, these bracket sign operators [] are also a dereference operator known as *offset operator*. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
```

These two expressions are equivalent and valid both if a is a pointer or if a is an array.

1.5 Pointer initialization

When declaring pointers we may want to explicitly specify which variable we want them to point to:

```
int number;
int *tommy = &number;
```

The behavior of this code is equivalent to:

```
int number;
int *tommy;
tommy = &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points (tommy), never the value being pointed (*tommy). You must consider that at the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: *). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous code with:

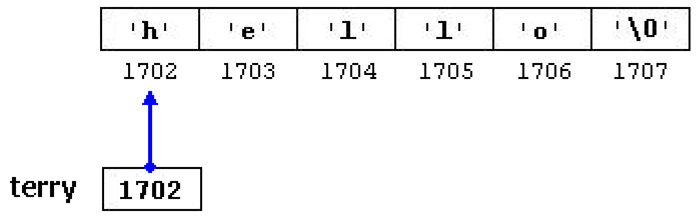
```
int number;
int *tommy;
*tommy = &number;
```

that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```
char * terry = "hello";
```

In this case, memory space is reserved to contain "hello" and then a pointer to the first character of this memory block is assigned to terry. If we imagine that "hello" is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:



It is important to indicate that terry contains the value 1702, and not 'h' nor "hello", although 1702 indeed is the address of both of these.

The pointer terry points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer). For example, we can access the fifth element of the array with any of these two expression:

```
*(terry+4)
```

```
terry[4]
```

Both expressions have a value of 'o' (the fifth element of the array).

Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4.

Suppose that we define three pointers in this compiler:

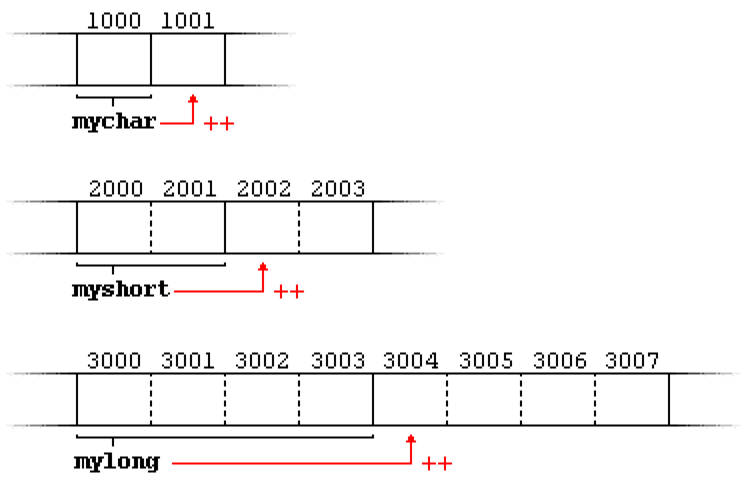
```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively.

So if we write:

```
mychar++;
myshort++;
mylong++;
```

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer.



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Both the increase (++) and decrease (--) operators have greater operator precedence than the dereference operator (*), but both have a special behavior when used as suffix (the expression is evaluated with the value it had before being increased). Therefore, the following expression may lead to confusion:

```
*p++
```

Because ++ has greater precedence than *, this expression is equivalent to *(p++). Therefore, what it does is to increase the value of p (so it now points to the next element), but because ++ is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with:

```
(*p)++
```

Here, the expression would have been evaluated as the value pointed by p increased by one. The value of p (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer).

If we write:

```
*p++ = *q++;
```

Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are increased. And then both are increased. It would be roughly equivalent to:

```
*p = *q;
++p;
++q;
```

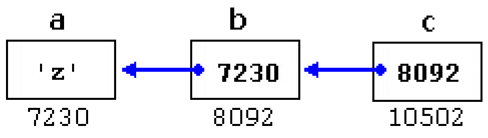
Like always, I recommend you to use parentheses () in order to avoid unexpected results and to give more legibility to the code.

Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would correspond to a different value:

- c has type char** and a value of 8092
- *c has type char* and a value of 7230
- **c has type char and a value of 'z'

void pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to change the type of the void pointer to some other pointer type that points to a concrete data type before dereferencing it. This is done by performing type-castings.

One of its uses may be to pass generic parameters to a function:

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

void increase (void* data, int size)
{
  switch (size)
  {
    case sizeof(char) : (*((char*)data))++; break;
    case sizeof(int) : (*((int*)data))++; break;
  }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;
}
```

```
y, 1603
```

sizeof is an operator integrated in the C++ language that returns the size in bytes of its parameter. For non-dynamic data types this value is a constant. Therefore, for example, sizeof(char) is 1, because char type is one byte long.

Null pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;
p = 0;     // p has a null pointer value
```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.