LECTURE 2

Pointers

A pointer contains an address in memory of a variable.

```c
int i = 2;
int *p; // p is a pointer to an int
p = &i; // p stores the address of i. "&i" means address of i.
int j = 3 + *p; // *p is the contents of what the pointer p points to.
    // In this case, *p is the same as i which equals 2.
p = &j; // Now p points to j. *p is now equal to j=5.
```

Think of computer memory as having addresses and values stored at those addresses.

<table>
<thead>
<tr>
<th>Address (p)</th>
<th>Value (*p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>234938530</td>
<td>address of i</td>
</tr>
<tr>
<td>395720335</td>
<td>address of j</td>
</tr>
</tbody>
</table>

So first p = 234938530, then later p = 395720335. If we write (*p)++, (*p) and therefore j is incremented from 5 to 6. * is called the indirection or dereferencing operator.

**Pointers and Arrays**

`float x[5]` is an array of 5 floating point numbers. `x[0]` is a float and `&x[0]` is the address of the first element of the array. Operators such as `<`, `==`, etc. work on pointers. Thus to set an array to zero, we can write

```c
float x[100];
float *px;
for(px=&x[0];px<=&x[99];px++) //px++ points to the next element of the array
    *px=0.0;
```

This is quite useful. Since `&x[0]` comes up a lot, it is often denoted by just `x`. So `px=x` is the same as `px=&x[0]`. `px=x+99` is the same as `px==&x[99]`.

**Pointer Arithmetic**

`px + int` is ok. For example, `px + 6` designates a place in memory that is 6 float places after `px`. The “6” is not in units of bytes; it is in units of the size of floats or whatever type `px` points to. Suppose we are given

```c
float *pf;
char *pc;
```

Then if `pf=20000` (address in bytes), `pf+6=20000 + 6*4` since each float takes 4 bytes. Similarly if `pc=10000`, `pc+6=10006` since each char takes 1 byte. The `sizeof` functions gives these sizes in bytes.
sizeof(float) = 4
sizeof(char) = 1

In general the sizeof(type) or sizeof(object) gives the size in bytes. px - pf gives an int that is the distance apart in floats, not in bytes.

**Example: String copy**

The function strcpy copies the string in array t into the array s.

```c
void strcpy(char *s, char *t)
{
    while ((s=*t)!='\0')
    {
        s++;
        t++;
    }
}

main()
{
    char t[]="hello";     // When an array is created, initialize its
    char s[100];         // size in memory. Don’t write char t[];
    strcpy(s,t);         // Notice that there is no "call" statement as in
                         // fortran.
    cout << s << endl;   // cout knows to print the whole string "hello"
}
```

Or we could write

```c
void strcpy(char *s, char *t)
{
    while (*s++=*t++)     // *s++ is the same as *(s++). First step:
                         // s[0]=t[0], i.e., *s*t. Then increment.
    ;                    // ";" is a blank statement.
}
```

Since the value of '\0' is 0 or false, the while statement will run until the end of the string is reached. (See page 31 of textbook for operator precedence.)

**Function Calling, Passing Arguments by Value**

In Fortran, suppose we have

```fortran
subroutine blah(x)
real*8 x
x=3.0
end
```
program main
real*8 y
y=4.0
call blah(y)       !blah replaces 4.0 by 3.0
write(*,*) y       !prints 3.0 as y’s value
stop
end

x and y are synonyms for the same address, i.e., the same location in memory. So if you change y, you change x.

What if you say

call blah(4.0d0)?

This causes big problems because 4.0d0 is now 3.0d0. Later on z=4.0d0 might give you 3.0d0 for z!

In C, you pass arguments to functions by value:

#include<iostream.h>

    void blah(double x)
    {
        x=3.0;
    }
    void main()
    {
        double y;
        y=4.0;
        blah(y);       //blah initially is passed the value 4.0 for x,
        cout << y << endl; //but then sets x=3.0. blah doesn’t change y.
        y=4.0;
        blah(y);       //gives 4.0 as y’s value
    }

x and y reside in different places in memory. y gets copied into x, the argument for blah. This is called “passing by value.” The value for y doesn’t actually change.
Fortran passes by address or reference. Both ways of passing variables are useful. C++ lets you do both with references.

```cpp
double x = 3.14;
double &y = x;  // A reference variable must be initialized when it is declared.
```

```
computer memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23778</td>
<td>4.0</td>
</tr>
<tr>
<td>12908</td>
<td>4.0</td>
</tr>
</tbody>
</table>
```

y is a reference to x which is a double variable. y is a synonym for x. x and y refer to the same place in memory. It acts exactly like the Fortran variable y in the subroutine blah.

```cpp
x = 2.0;  // now y = 2.0
y = 3.0;  // now x = 3.0
cout << x << y << endl;  // yields 3.0 3.0
```

(No * needed for references.)

**Why do we need references?**

1. It is a convenient shorthand. Suppose you need to do manipulations of \(x[18i+j][32k]\) where \(i, j, \) and \(k\) are fixed for a while. Then you could write

```cpp
double &rx = x[18i+j][32k];
```

Now use \(rx\) everywhere as a reference, until \(i\) or \(j\) or \(k\) changes. Then define a new reference. For example, to construct a unit matrix, we could write
double mat[100][100];
for(i=0;i<100;i++)
  for(j=0;j<100;j++)
    {
      double &m=mat[i][j];
      if(i==j)
        m=1.0;
      else
        m=0.0;
    }

2. References are very useful for function calls.

  void blah(double &x)
  {
    x=3.0;
  }

main()
{
  double y;
  y=4.0;
  blah(y);
  cout << y << endl; // yields y=3.0
}

This is really useful when you have “big” objects that you don’t want to copy and pass to a function. It’s more efficient to just tell the function where to find the object. But what if you say blah(4.0)? You’re hosed.

After a reference variable has been declared, you can do nothing to access the reference variable itself. That’s why a reference variable must be initialized when it is declared. Reference variables declared as arguments to functions are initialized when the function is called.

Constants

In Fortran we use parameter to set constant values. For example,

  parameter (pi=3.14159)

C++ has const. For example,

  const double pi=3.14159;

This line could go anywhere, whereas in Fortran, parameter statements are at the beginning of the program. If later on we write pi=7.0; we get a syntax error.

Constant Pointers

There are
1. pointers to const floats.

   ```c
   const float pi=3.14159;    // A const variable must be initialized when
   // it is defined (except if it is declared
   // with an extern statement, in which
   // case it is defined elsewhere).
   
   const float *ppi=&pi;
   *ppi=7.0;    //ERROR
   ```

2. “const pointers” to floats. Const pointers can’t change the place they point to in memory.

   ```c
   float a;
   float* const pa=&a;
   *pa=7.0;    //OK to do this
   float b;
   pa=&b;      //ERROR!
   ```

3. “const pointers” to “const floats”

   ```c
   const float * const p = &pi;
   *p = 7.0;    //ERROR!
   p=&b;        //ERROR!
   ```

   You can’t change anything in this case.

The rule is that the “const ness” applies to the thing immediately to the right of the `const` keyword. You can also read the declaration backward. For example, `float* const pa` can be read “`pa` is a constant pointer to a float.”

**Const in Function calls**

```c
void func(const double &x)
 {x=3.0;}    //ERROR
```

`const` should appear everywhere possible in function arguments. It’s more efficient if the compiler knows that those arguments won’t be changed, and it prevents bugs. For example, writing

```c
void strcpy(char* s, const char* t);
```

prevents bugs.
double abs(double x)  //Ok. Pass by value.
{....}

double abs(double &x)  //BAD! because you could change x.

double abs(const double &x)  //Good way to do things.

Function Prototypes (declarations versus definitions)
If the program is in separate files, or if the function is defined later in the file, you need to declare the function. A declaration tells you the function is there but doesn’t define it and tell you how it works. An example of a declaration is

double abs(const double &x);  // The ";" replaces {}.
  // The "x" is optional.

You can put this declaration anywhere in the program before abs is used.

New and Delete
In Fortran we dimension arrays as follows:

    parameter(maxsize=1000)
    real*8 vec(maxsize)

This means you have to recompile every time you have a different size for vec. Or you just make maxsize bigger than any conceivable size that vec could need. But this wastes memory.

In C++, we still have to dimension arrays and allocate memory, but we can do it efficiently by using new as in this example:

    double *vec;  //pointer to a vector
    cin >> n;
    vec = new double[n];

This allocates 8n bytes of memory for vec and gives you a pointer to it. The next call to new will give you a pointer to a new location in memory.

delete is used to give back the memory once the program is finished with it. So once the program is finished with vec, it executes

    delete [] vec;

and returns the allocated memory to the heap. This works not just for arrays, but also for single objects:

    double *pa;
    pa=new double;
    delete pa;
Objects created by `new` don’t go away until you call `delete`. For example,

```cpp
void badfunction()
{
    double *px=new double[1000];
}
```

Once the program is finished executing `badfunction`, it throws away `px` but the system has still set aside the memory and you can never get it back! This is called a “memory leak”; it’s a C++ programmer’s worst nightmare.

Once you’ve allocated the pointer, you can treat it like an array. It works exactly like an array. For example,

```cpp
float *a;
    a=new float[100];
    for(i=0;i<100;i++)
    a[i]=1.0;
```

When do you use `delete`? The rule of thumb is to use `delete` to deallocate memory if you have used `new` to allocate it. If you haven’t used `new`, don’t use `delete`. For example,

```cpp
float *a = new float[100];
float b[100];
//stuff
delete []a;   //use delete for array a because you used new
// don’t need delete for array b because you didn’t use new
```

### Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored as arrays just as other variables can. For example, consider the beginning of a program that sorts lines of text alphabetically:

```cpp
#define MAXLINES 5000    //max number of lines to be sorted
char *lineptr[MAXLINES];   //pointers to text lines
```

**Aside:** The `define` statement is a preprocessor command that replaces all subsequent occurrences of `MAXLINES` with 5000 (except if `MAXLINES` appear in quotes "MAXLINES"). In general `define` statements have the form

```cpp
#define name replacement text
```

The preprocessor will replace all subsequent occurrences of `name` with `replacement text` in the source file being compiled. `define` statements are often in header (.h) files that are included in .cc source files. The `replacement text` can be an expression or a statement. For example,
```c
#define forever for(;;)     //infinite loop

#define max(A,B) ((A) > (B) ? (A) : (B))

Names may be undefined with #undef.

Back to arrays of pointers: The declaration for lineptr

    char *lineptr[MAXLINES];     //pointers to text lines

says that lineptr is an array of MAXLINES elements, each element of which is a pointer
to a char. That is, lineptr[i] is a character pointer, and *lineptr[i] is the character
it points to, the first character of the i\textsuperscript{th} saved text line. Since lineptr is itself the name
of an array, it can be treated as a pointer. Thus it is a pointer to pointers. (How's that
for confusing?) We can initialize a pointer array as follows:

    char *name[] ={"Dopey", "Sleepy", "Doc"};

Notice that each string is a different length. name goes from name[0] to name[2], and
each element of name is a pointer to a string.

Another use for a pointer to an array is two dimensional arrays. Given the definitions

    int a[10][20];
    int *b[10];

then a[3][4] and b[3][4] are both legal references to a single int. a is a true two
dimensional array; each row of a has 20 int-sized elements. However, the important
advantage of the pointer array is that the rows of the array may be of different lengths.
That is, each element of b need not point to a twenty-element vector; some may point
to two elements, some to fifty, and some to none at all. So b doesn't waste memory.

If a multi-dimensional array is passed to a function, the function declaration typically
looks like

    func(int a[]); or func(int *a);     //for a 1D array

    func(int a[][1]); or func(int **a);    //for a 2D array

    func(int a[][1][]); or func(int ***a); //for a 3D array

These arrays are dimensioned elsewhere in the program and passed to the function. Here
is an example of how to use new and delete for a two dimensional array:

    #include <iostream.h>

    void func(int **p);
    void main()
    {
```
//Create a 2D array of integers p[5][2]

int** p;
p = new int*[5]; //create array of 5 pointers to integers
for(int i = 0; i < 5; i++)
    p[i] = new int[2]; //create row with 2 integers

func(p); //pass name of array without **

for(int i = 0; i < 5; i++)
    delete []p[i]; //delete in reverse order of creation

delete []p;
}

void func(int **p){
    int n = 0;
    for(int i = 0; i < 5; i++)
        for(int j = 0; j < 2; j++)
            {p[i][j]=n++;
             cout << i << " " << j << " " << p[i][j] << endl;}
}

In C and C++, elements of arrays are stored by row, so that the rightmost subscript, or column, varies fastest as elements are accessed in storage order. In Fortran, elements of arrays are stored by column so that the first subscript, or row, varies fastest as elements are accessed in storage order.

Pointers to Functions

In C and C++, a function is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. A function name alone refers to its address. The following does not declare a pointer to a function:

    int *pFn();

Instead, this declares a function that returns a pointer to an integer. The () takes effect first because it has a higher precedence than *. To get the desired effect, parentheses are required:

    int (*pFn)(float);

Now the * takes effect first. This is read as “pFn is a pointer to a function that takes a float argument and returns an int.” In use, its syntax is similar to that of an array:
int anotherFn(float f);  //Some function, defined somewhere
void myFunc(float x)
{
    int (*pFn)(float);  //Declare a pointer to a function
    pFn = anotherFn;    //A function name alone refers to its address.
    (*pFn)(x);          //Call the function pointed at by pFn
}

In the last line the extra parentheses are necessary when calling *pFn because the precedence of ( ) is higher than that of *. If we had just written *pFn(x), the program would interpret pFn as a function of x that returns a pointer and *pFn(x) is the value pointed at by pFn(x).

Pointers to functions are used a lot in Numerical Recipes. (Numerical Recipes in C: The Art of Scientific Computing by W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (Cambridge Univ. Press, 1992) is an excellent text on scientific computing techniques. They have a homepage at http://www.nr.com where you can download the text of the book.) For example suppose you want to integrate a function func(x) over x from a to b using Simpson’s rule. The declaration found in Numerical Recipes is

    float qsimp(float (*func)(float), float a, float b);

Here a pointer to func is passed to the integration routine qsimp. qsimp returns the value of the integral as a float. To use qsimp, we would write something like:

    float function(float);
    float integral = qsimp(function, a, b);

Notice that the argument of qsimp is just function, not &function since the name of the function by itself refers to the address of the function.