[Much of the material in these notes is derived from the Hubbard and Deitel books.]

**Addresses:**

Every variable has: a *name*, a *type*, an *address* in memory, and (possibly) a *value*.

If we want to refer to the address of a particular variable we can use the address operator, &. [We have already used this op in passing by reference.]

The address operator, &, operates on the variable name to produce an address:

$$\begin{tikzcd} \text{variable name} \arrow[r] & \& \arrow[r] & \text{variable address} \end{tikzcd}$$

e.g.,

```
//ptr1.cpp
#include <iostream>
int main(){
   int n=33;
   cout << "n= " << n << endl;
   cout << "&n= " << &n << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
n= 33
&n= 0x0068fe00
```

The address output is a hexadecimal number. All hex numbers start with 0x.

$0068fe00_{(16)} = 6880768_{(10)}$

$$\begin{array}{cccccccc} \underline{0} & \underline{0} & \underline{6} & \underline{8} & \underline{f} & \underline{e} & \underline{0} & \underline{0} \\ 16^7 & 16^6 & 16^5 & 16^4 & 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

[$=6 \times (16)^5 + 8 \times (16)^4 + f \times (16)^3 + e \times (16)^2 + 0 \times (16)^1 + 0 \times (16)^0$

$=6 \times 1{,}048{,}576 + 8 \times 65{,}536 + 15 \times 4096 + 14 \times 256 + 0 + 0$]

## Reference Variables

A reference variable is an ***alias*** for another variable.

In the following example, n and r are always the same. They are just different names for the same variable. Even the same address prints for both. They have the same *value* and occupy the same *location*.

```
//ptr2.cpp
#include <iostream>
int main(){
   int n=33;
   int &r = n;
   cout << n << "\t\t" << r << endl;
   n--;
   cout << n << "\t\t" << r << endl;
   r*= 2;
   cout << n << "\t\t" << r << endl;
   cout << &n << '\t' << &r << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
33            33
32            32
64            64
0x0068fe00    0x0068fe00
```

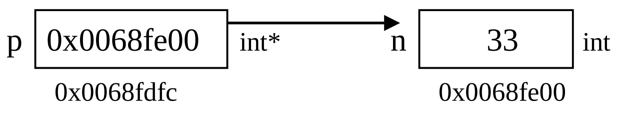
A reference parameter is also an alias, or synonym. A reference parameter for a function is just a reference variable whose scope is limited to the function.

**Pointers:**

If we take the address of a variable and store it in another variable, that's a ***pointer***.

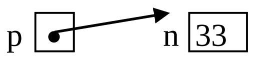
```
//ptr3.cpp
#include <iostream>
int main(){
   int n=33;
   int *p = &n;
   cout << n << "\t\t" << &n << '\n' << p << '\t' << &p << endl;
   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
33              0x0068fe00
0x0068fe00      0x0068fdfc
```



The pointer variable p and the expression &n have the same type (pointer to int) and the same value.

Since we don't care what the exact address of n is we might indicate a pointer as:



**Dereferencing a pointer:**

Since *p is an alias for n, we can use it to get the value of n.  This is called ***dereferencing*** a pointer.

```
//ptr3.cpp
#include <iostream>
int main(){
   int n=33;
   int *p = &n;  // p points to n
   cout << *p << endl;
   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
33
```

The & and * operators are inverses of each other.

| | |
|---|---|
| n = = *p | n = = *&n |
| p = = &n | p = = &*p |

We say that a (regular) variable *directly references* a value and a pointer *indirectly references* a value.

**Derived Types**:  Types derived from (based on) other types.

| | |
|---|---|
| int a[] = {33, 66}; | // type array of int |
| const int c = 33; | // type const int |
| int f() {return 33;} | // type function returns int |
| int & r = n; | // type reference to int |
| int * p = &n; | // type pointer to int |

These are all types derived from a single type.  A type that is derived from different types is called structured (e.g., struct, class).

## <u>**Initialization of Pointers:**</u>

Pointers can (and should) be initialized when they are declared: to 0, NULL, or an address. A pointer with value 0 or NULL is not pointing to anything. NULL is a symbolic constant defined in iostream.h (among others).

Even though the constant 0 has type int, it can be assigned to all the fundamental types, and takes on the appropriate value for each particular type:

```
char c = 0;          // initializes c to the char '\0' or NUL (ASCII 0)
float x = 0;         // initializes x to the float 0.0
char* p = 0;         // initializes p to NULL, no matter what type p is pointing to.
```

The NULL pointer cannot be dereferenced. So, *p = 22; is an invalid statement if p was initialized with int* p=0; We may wish to check before assigning a value to a dereferenced pointer:

```
        if (p) *p = 22;
```

Testing (p) is the same as testing the condition (p != NULL) because p = = 0 = = NULL.

## Passing by Reference:

The default parameter passing mechanism is, of course, pass by value.  We can pass by reference using either addresses (references) or pointers.  For example, the swap function works the same if we use addresses or pointers to pass function parameters:

```
//swap1.cpp
//swap function using address parameters
#include <iostream>
void swap (int&, int &);

int main(){             // main driver function
   int a=10, b=20;
   cout << a << '\t' << b << endl;
   swap (a, b);
   cout << a << '\t' << b << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

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

```
10    20
20    10
```

Same output, using pointer variables as parameters:

```
//swap2.cpp
//swap function using pointer parameters

#include <iostream>

void swap (int*, int *);

int main(){
   int a=10, b=20;
   cout << a << '\t' << b << endl;
   swap (&a,&b);
   cout << a << '\t' << b << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
void swap (int *xptr, int *yptr){
   int temp = *xptr;          //ptr added to names for clarity
   *xptr = *yptr;
   *yptr = temp;
   return;
}
```

Pointers can themselves be passed by reference, if we wish to change the addresses stored in them. If a pointer is not specifically passed by reference, the system will make a local copy of the pointer in the function, and any changes to the pointer are not carried back to the calling function.

```
//swap3.cpp
//swapping the pointers, not the values

#include <iostream>

void swap (int*&, int *&);

int main(){
   int a=10, b=20;
   int * aptr=&a, * bptr=&b;
   cout << *aptr << '\t' << *bptr << endl;
   swap (aptr,bptr);
   cout << *aptr << '\t' << *bptr << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

void swap (int *&xptr, int *&yptr){
   int *temp = xptr;
   xptr = yptr;
   yptr = temp;
   return;
}
```

**Lvalues**:

You have probably already come across the term ***lvalue*** in an error message. The term lvalue used to mean anything that could be on the left side of an assignment statement, e.g.: x = 4;   Now it is more general.

An lvalue is an expression referring to an object or function (i.e., a region of storage), anything whose address is accessible.

Names of objects (variables) are lvalues:
int n; n=127;

Literals are not lvalues:
44 = n;  //error

Symbolic constants are lvalues, even though they cannot be changed and so cannot appear on the left side of an assignment operation:

```
const int MAX = 999;        //OK
MAX=27;                     //error
```

In these examples, n is a *mutable lvalue* and MAX is an example of an *immutable lvalue*.

Other mutable lvalues:
subscripted variables a[5]=10;
Dereferenced pointers *p=25;

Immutable lvalues:
Arrays, functions, references.  (later)

To declare a reference variable the general syntax requirement is:
Type & refname = lvalue; e.g.,

```
int &r = n;                 //OK, n is an lvalue
int &r = 44;                //error
int &r = n++;               //error
int &r = cube(n);           //error
```

A literal or an expression is not an lvalue.

**<u>Constant vs. Non-constant Pointers</u>**:

There are four possibilities:

Non-constant pointer to non-constant data.
Non-constant pointer to constant data.
Constant pointer to non-constant data.
Constant pointer to constant data.

Example:

```
int x = 5;                  // x is an integer
int * const p = &x;         // p is a const pointer to x
                            // p can't be change but *p can be changed
const int * const p = &x;   // p is a const pointer to x, *p is also a const and can't be changed
```

This becomes important when passing by reference using pointers. We want to give the function as little access to the data as possible.

More examples (from Hubbard, p. 169):

```
int n = 44;                 //n is an int
int * p = &n;               // pointer to n
++(*p);                     //OK. Increments *p (an integer)
++p;                        //OK. Increments p (a pointer)
int * const cp = &n;        //a const pointer to n
++(*cp);                    //OK. Increments *cp
++cp;                       // error.  Pointer cp is a constant
const int k = 88;           //k is a const integer
const int * pc = &k;        //pc points to k
++(*pc);                    //error:  *pc is a constant (k)
++pc;                       //OK. Increments pointer pc
const int * const cpc = &k; //const pointer to a const integer
++(*cpc);                   //error: *cpc is a const
++cpc;                      //error: pointer cpc is a constant
```

**<u>Pointer Arithmetic</u>**:

Pointers contain addresses, not integers. But some arithmetic can be done on pointers. When you perform arithmetic on a pointer, it is pointing somewhere else. The change in address depends on the size of the type it is pointing to (i.e., when we increment a pointer, we are not simply adding 1).

We can use:

| | | |
|---|---|---|
| ++ | + | += |
| –– | – | –= |

Because pointer arithmetic depends on the size of the memory locations (for any particular type), pointer arithmetic is *machine dependent*.

## **Arrays and Pointers**:

We can use pointer arithmetic to traverse an array:

```
//arraytraversal.cpp
//modified from Hubbard ex. 6.9 p.163
//using pointer arithmetic to traverse an array

#include <iostream>

int main(){
   const short SIZE = 3;
   short a[SIZE] = {22, 33, 44};
   cout << "a = " << a << endl;   //a is the name of an array
   cout << "sizeof(short) = " << sizeof(short) << endl;
   short *end = a + SIZE;     // converts SIZE to offset
   short sum = 0;
   //short * p;
   for (short * p=a; p<end; p++){
      sum += *p;
      cout << "\t p = " << p;
      cout << "\t *p = " << *p;
      cout << "\t sum = " << sum << endl;
      }
   cout << "\n end = " << end << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
a = 0x0068fdfc
sizeof(short) = 2
         p = 0x0068fdfc  *p = 22        sum = 22
         p = 0x0068fdfe  *p = 33        sum = 55
         p = 0x0068fe00  *p = 44        sum = 99

 end = 0x0068fe02
```

Arrays as parameters:
We know that when we pass an array name as an argument to a function it is passed by reference even if there is no &. This is because the array name *is* a reference to the address of the first array element array[0]. In other words, it is a pointer. The compiler automatically converts int a[] in the argument list to int * const a. The default for an array name is as a constant pointer to non-constant data. This means that the pointer always points to the same memory location, and the data at that location can be modified through the pointer.

For example, the bubble sort algorithm using reference parameters:

```
//bubble1.cpp
//modified from Deitel ex. 5.15, p. 300
//sorts an array in ascending order

#include <iostream>
#include <iomanip>

void bubblesort (int *, const int);

int main(){
   const int SIZE = 10;
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   cout << "Data items in original order:\n";
   for (int i=0; i<SIZE; i++)
                        cout << setw(4) << a[i];
   bubblesort (a, SIZE);
   cout << "\n\n Data items in ascending order:\n";
   for (int i=0; i<SIZE; i++)
                        cout << setw(4) << a[i];

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

void bubblesort (int * array, const int size) {
   void swap (int *, int *);
   for (int pass = 0; pass<size-1; pass++)
            for (int j=0; j<size-1; j++)
                 if (array[j] > array[j+1])
                       swap(&array[j], &array[j+1]);
}

void swap (int *ptr1, int *ptr2){
   int temp = *ptr1;
   *ptr1 = *ptr2;
   *ptr2 = temp;
}
```

```
Data items in original order:
   2   6   4   8  10  12  89  68  45  37

 Data items in ascending order:
   2   4   6   8  10  12  37  45  68  89
```

**Pointers to/from functions**:

Just like an array name is really the memory address of the first element of the array, a function name is the starting memory address of the function's code. So the function name is actually a const pointer.

So a function (or a pointer to a function) can be a parameter to another function. This allows us to define functions of functions.

The following example program will sum f(0) + f(2) + f(3) + … + f(n) for any function.

```
//sums.cpp
//from Hubbard, ex. 6.15, p. 171

#include<iostream>

int sum (int (*) (int), int);
int square (int);
int cube (int);

int main(){
   cout << sum(square, 4) << endl;  // 1+4+9+16
   cout << sum(cube, 4) << endl;                // 1+8+27+64

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

int sum (int (*pf) (int k), int n) {
   int s=0;
   for (int i=1; i<=n; i++) s+= (*pf)(i);
   return s;
}

int square (int k) {return k*k;}

int cube (int k) {return k*k*k;}
```

```
30
100
```

The following is a bubble sort algorithm that will sort in either ascending order or descending order, depending on the value of a function reference.

```
//bubble2.cpp
//modified from Deitel ex. 5.26, p. 318
//sorts an array in ascending OR descending order
// using function pointers

#include <iostream>
#include <iomanip>

void bubblesort (int [], const int, int (*)(int, int) );
int ascending (int, int);
int descending (int, int);

int main(){
   const int SIZE = 10;
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int order;
   cout << "Data items in original order:\n";
   for (int i=0; i<SIZE; i++)
      cout << setw(4) << a[i];

   cout << "\n\nEnter 1 to sort in ascending order,\n"
      << "Enter 2 to sort in descending order:  ";
   cin >> order;
   if (order ==1) {
      bubblesort (a, SIZE, ascending);
      cout << "\n\n Data items in ascending order:\n";
      }
      else {
         bubblesort (a, SIZE, descending);
         cout << "\n\n Data items in descending order:\n";
         }
   for (int i=0; i<SIZE; i++)
      cout << setw(4) << a[i];

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

void bubblesort (int array[], const int size, int (*compare)(int, int))
{
   void swap (int *, int *);
   for (int pass = 0; pass<size-1; pass++)
      for (int j=0; j<size-1; j++)
         if ((*compare) (array[j], array[j+1]) )
            swap(&array[j], &array[j+1]);
}

void swap (int *ptr1, int *ptr2){
   int temp = *ptr1;
   *ptr1 = *ptr2;
   *ptr2 = temp;
}
```

```
int ascending (int a, int b) {
   return b < a;        // test for (b<a) -- will swap if true
   }

int descending (int a, int b) {
   return b > a;        // test for (b>a) -- will swap if true
   }
```

Output1:

```
Data items in original order:
  2   6   4   8  10  12  89  68  45  37

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order:  1


 Data items in ascending order:
  2   4   6   8  10  12  37  45  68  89
```

Output2:

```
Data items in original order:
  2   6   4   8  10  12  89  68  45  37

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order:  2


 Data items in descending order:
 89  68  45  37  12  10   8   6   4   2
```

A function's return type may be a reference as long as the value returned is an *lvalue* that exists in the main (or calling) function.

In the following example, the maximum of two values will be changed.  The function will return a reference to the larger number.

```
//maxwhich.cpp
//from Hubbard ex 6.7, p. 162
// function returns a reference to update maximum value

#include <iostream>

int& max (int&, int&);

int main(){
   int m = 44, n = 22;
   cout << m << '\t' << n << '\t' << max(m,n) << endl;
   max(m,n) = 55;                              //updates m to 55
   cout << m << '\t' << n << '\t' << max(m,n) << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

int& max (int &x, int &y){
   return (x>y ? x : y);
}
```

Output:

```
44    22    44
55    22    55
```

The following example uses a function to determine which array element to access. The function returns a reference to the particular element. It allows the programmer to use the more "natural" method of indexing arrays, starting from 1 instead of the C++ method of starting from 0:

```
//arrayelement.cpp
//modified from Hubbard ex. 6.8, p. 162
//uses 1-based indexing for array rather than 0-based indexing

#include <iostream>
float& element (float*, int);

int main(){
   float v[4];
   for (int k=1; k<=4; k++)
      element(v,k) = 1.0/k;
   for (int k=0; k<4; k++)
      cout << "v[" << k << "] = " << v[k] << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

float& element (float *array, int i){
   return array[i-1];
}
```

Output:

```
v[0] = 1
v[1] = 0.5
v[2] = 0.333333
v[3] = 0.25
```

## Pointers and Strings:

Strings can be implemented in a variety of different ways:

| | |
|---|---|
| string str; | //uses the string class library, #include <string> |
| char str[15]; | // a character array of size 14 (one char for NUL \0) |
| char * str; | // char array |

Can initialize:

| | |
|---|---|
| char str[] = "hello"; | // char array of size 6 |
| char *str = "hello"; | // char array of size 6 |

```
//stringtest.cpp
#include <iostream>
#include <cstring>

int main(){
   char s1[10];
   //s1 = "string1";                         //error
   cout << "enter a string:  "; cin >> s1;   //OK
   char* s2;
   s2 = "string2";                           //OK
   char s3[] = "hello";
   cout << s1 << endl << s2 << endl << s3 << endl;
   char* s4 = "helloagain";
   cout << "s4= " << s4 << "  size " << strlen(s4) << endl;
   s4 = "hiAgain";        //OK: string literal is type pointer to char
   cout << "s4= " << s4 << "  size " << strlen(s4) << endl;
   //s4 = 'x';            //error: char literal is type char

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

```
enter a string:  mystring
mystring
string2
hello
s4= helloagain  size 10
s4= hiAgain  size 7
```

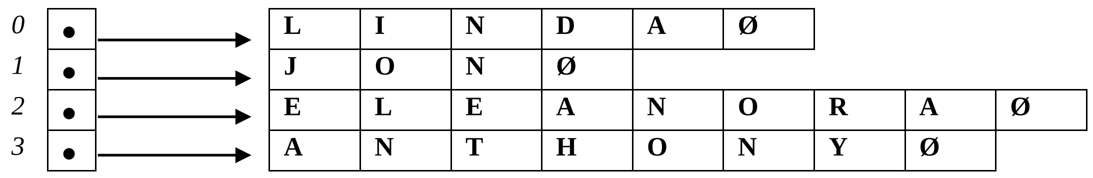
So, using the declaration char* str; provides us with access to strings of any length.  The string *and the string length* can change dynamically (during run time).

## Arrays of strings:

The declaration char name [4][10] will declare and allocate space for a 2-dimensional array of names:

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *0* | **L** | **I** | **N** | **D** | **A** | **Ø** | | | | |
| *1* | **J** | **O** | **N** | **Ø** | | | | | | |
| *2* | **E** | **L** | **E** | **A** | **N** | **O** | **R** | **A** | **Ø** | |
| *3* | **A** | **N** | **T** | **H** | **O** | **N** | **Y** | **Ø** | | |

There's a lot of empty, unused space. It would be more efficient (i.e., less wasted space) to allocate only the storage space needed -- a *ragged array*. For example, using pointers:



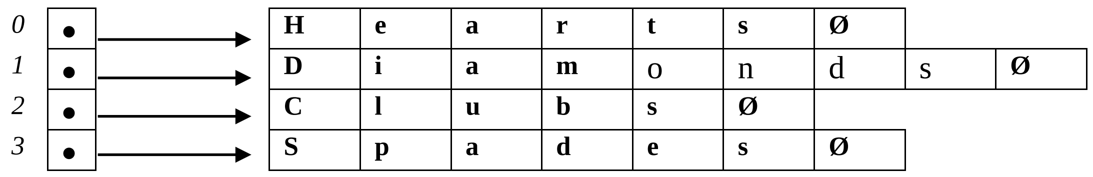
char* name[4];                    // array of pointers to char strings

```
#include <iostream>
int main(){
   char* name[]={"Linda","jon","eleanor","anthony"};
   for (int i=0; i<4; i++) cout << name[i] << endl;
   return 0;
}
```

```
Linda
jon
eleanor
anthony
```

A slightly more realistic example:

char * suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};



Even though the suit arrays are fixed in size, the pointers provide access to character strings of any length.

Results in a very flexible data structure.

**New and Delete**:

float * p; only *declares* the pointer variable. Since it is not initialized, it is not pointing to anything. Any attempt to access by dereferencing p will result in an error.
*p = 55.7;                    //error

One solution is to initialize pointers when they are declared. This means that the memory will be allocated at compile time. Another is to allocate memory for the pointer to point to, when it is needed. For this we use the dynamic operator, ***new***.

```
float *p;           //declares p pointer to float
p = new float;      //allocates memory for 1 float pointed to by p
*p = 55.7;          //OK
```

The new operator returns the address of 1 storage location of the right size for the type and the machine. This allocates memory at run time.
OR:   float *p = new float (55.7);

Also, this means that the pointer variable p points to a float location that has no name (other than *p, the value pointed to by p).

Just like the new operator allocates main memory dynamically, during run time, the ***delete*** operator frees up allocated memory and returns it to the collection of available storage locations.

```
//newdelete.cpp
#include <iostream>
int main(){
   float * p = new float (55.7);
   cout << *p << endl;        //OK. displays 55.7
   delete p;
   p = new float (17.3);
   cout << *p << endl;        //OK. displays 17.3
   delete p;
   *p = 10.0;                 //run-time error

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}
```

A deallocated pointer is like an uninitialized pointer: it doesn't point to anything.

Since constants can't be changed, a pointer to a constant can't be deleted:

```
const int *p = new int;
delete p;           //error
```

It can only be applied to pointers that were explicitly allocated with new.

**<u>Dynamic Arrays</u>**:

An array name is a const pointer allocated at compile time. In the following a is almost exactly the same as p.

```
float a[20];                        //static array
float * const p = new float[20];         //dynamic array
```

a is allocated at compile time and remains allocated throughout program execution; called *static binding*. p is allocated at run time; called *dynamic binding*.

This is a dynamic array. It can be deallocated and then reallocated (say, with a different size during execution. To deallocate a dynamic array:
delete [] p;

We need the brackets because ***p*** is an array, just like ***a*** (a const pointer to a[0]) is an array.

Strings can be implemented with dynamic arrays.

[See Schaum's Ex. 6.12, p. 168.]