Ch 9. Pointers
Part 2

CS 2308
Fall 2011

Jill Seaman

Lecture 5

Pointer Arithmetic

- Operations on pointers over data type d:
  - ptr + n where n is int: ptr+n*sizeof(d)
  - ptr – n where n is int: ptr-n*sizeof(d)
  - ++ and -- : ptr=ptr+1 and ptr=ptr-1
  - += and -=
  - subtraction: ptr1 – ptr 2
    result is number of values of type d between the two pointers.

Initializing Pointers

- Pointers can be initialized as they are defined.

```
int myValue;
int *pint = &myValue;

int ages[20];
int *pint1 = ages;

int *p1 = &myValue, *p2=ages, x=1;
```

- Note: pointers to data type d can be defined along with other variables of type d.

```
double x, y, *d, radius;
```


Comparing Pointers

- pointers maybe compared using relational operators:

  < <= > >= == !=

- Examples:

```
int arr[25];

cout << &arr[1] > &arr[0] << endl;
cout << arr == &arr[0] << endl;
cout << arr <= &arr[20] << endl;
cout << arr > arr+5 << endl;
```

- What is the difference?
  - `ptr1 < ptr2`
  - `*ptr1 < *ptr2`

Pointers as Function Parameters

- Use pointers to implement pass by reference.

```
//prototype: void changeVal(int *);

void changeVal (int *val) {
   *val = *val * 11;
}

int main() {

   int x;
   cout << "Enter an int " << endl;
   cin >> x;
   changeVal(&x);
   cout << x << endl;
}
```

- How is it different from using reference parameters?

Pointers as array parameter

- Pointer may be used as a parameter for array

```
double totalSales(double *arr, int size) {
   double sum = 0.0;
   for (int i=0; i<size; i++) {
      sum += *arr++;              //OR: sum += arr[i];
   }
}

int main() {
   double sales[4];
   // input data into sales here
   cout << "Total sales: " << totalSales(sales, 4) << endl;
}
```

- What? `sum += *arr++;`

```
sum = sum + *arr;
arr = arr+1;
```

Note: * and ++ have same precedence, but associate right to left: `*(arr++)` not: `(*arr)++`

Dynamic Memory Allocation

- When a function is called, memory for local variables is automatically allocated.
- When function exits, memory for local variables automatically disappears.
- Must know ahead of time the maximum number of variables you may need.
- Dynamic Memory allocation allows you to create variables on demand, during run-time.

The new operator

- “new” operator requests dynamically allocated memory for a certain data type:

```
int *iptr;
iptr = new int;
```

- new operator returns address of newly created anonymous variable.
- use dereferencing operator to access it:

```
*iptr = 11;
cin >> *iptr;
int value = *iptr / 3;
```

new with arrays

- dynamically allocate arrays with new:

```
int *iptr;  //for dynamically allocated array
int size;

cout << "Enter number of ints: ";
cin >> size;
iptr = new int[size];

for (int i=1; i<size; i++) {
   iptr[i] = i;
}
```


Make sure new succeeded

- new will fail if not enough memory available.
- new returns NULL (which is 0) if it fails.
- A pointer whose value is 0 is called a "Null pointer".

```
iptr = new int[10000];
if (iptr == NULL) {
   cout << "Error allocating memory." << endl;
   return EXIT_FAILURE;
}
```

delete!

- When you are finished using a variable created with new, use the delete operator to destroy it:

```
int *ptr;
double *array;

ptr = new int;
array = new double[25];
. . .
delete ptr;
delete [] array;
```

- Do not “delete” pointers whose values were NOT dynamically allocated using new!
- Do not forget to delete dynamically allocated variables (Memory Leaks!!).

Returning Pointers from Functions

- functions may return pointers:

```
char *findNull (char *str) {
   char *ptr;
   ptr = str;
   while (*ptr != '\0')
      ptr++;
   return ptr;
}
```

- The returned pointer must point to
  - dynamically allocated memory OR
  - an item passed in via an argument

Returning Pointers from Functions: `duplicateArray`

```
int *duplicateArray (int *arr, int size)
{
   int *newArray;

   if (size <= 0)    //size must be positive
      return NULL;

   newArray = new int [size];  //allocate new array

   for (int index = 0; index < size; index++)
      newArray[index] = arr[index];  //copy to new array

   return newArray;
}


int a [5] = {11, 22, 33, 44, 55};
int *b = duplicateArray(a, 5);
for (int i=0; i<5; i++)
   if (a[i] == b[i])
      cout << i << " ok" << endl;
delete [] b;
```


Problems returning pointers

- Bad:

```
char *getName() {
   char name[81];
   cout << "Enter your name: ";
   cin.getline(name, 81);
   return name;
}
```

  - what happens to name on function exit?

- Good:

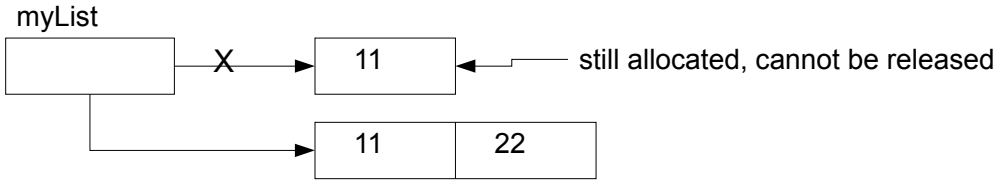
```
char *getName () {
   char *name;
   name = new char[81];
   cout << "Enter your name: ";
   cin.getline(name, 81);
   return name;
}
```

Memory Leak!

```
int *appendArray (int x, int *arr, int size)
{
   int *newArray;
   newArray = new int [size+1];  //allocate new array

   for (int index = 0; index < size; index++)
      newArray[index] = arr[index];  //copy to new array

   newArray[size] = x;               //add x to last spot

   return newArray;
}

// in main:
int *myList;
int size = 1;
myList = new int [1];
myList[0] = 11;

// inside menu case for add
int z;
cout << "Enter int for z: ";
cin >> z;
myList = appendArray(z, myList, size);  //MEMORY LEAK HERE!
```


Memory Leak

```
// inside case for add choice
int z;
cout << "Enter int for z: ";
cin >> z;
myList = appendArray(z, myList, size);  //MEMORY LEAK HERE!
```



- Called "Pointer Reassignment

Memory Leak: solved

```
// inside case for add choice
{ int z;
  cout << "Enter int for z: ";
  cin >> z;

  int *newList;     //temp variable, local to the case block
  newList = appendArray(z, myList, size);
  delete [] myList;
  myList = newList;
}
```