Week 4

Pointers & Structs

Gaddis: Chapters 9, 11

CS 5301
Spring 2016

Jill Seaman

Pointers and Addresses

- The address operator (&) returns the address of a variable.

```
int x;
cout << &x << endl;    // 0xbffffb0c
```

- Pointer: a variable that stores the address of another variable, providing indirect access to it.
- An asterisk is used to define a pointer variable

```
int *ptr;
```

- "ptr is a pointer to an int". It can contain addresses of int variables.

```
ptr = &x;
```


Dereferencing and initializing

- The unary operator * is the dereferencing operator.
- *ptr is an alias for the variable that ptr points to.

```
int x = 10;
int *ptr;   //declaration, NOT dereferencing
ptr = &x;   //ptr gets the address of x
*ptr = 7;   //the thing ptr pts to gets 7
```

- Initialization:

```
int x = 10;
int *ptr = &x; //declaration, NOT dereferencing
```

- ptr is a pointer to an int, and it is initialized to the address of x.

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 and Arrays

- You can treat an array variable as if it were a pointer to its first element.

```
int numbers[] = {10, 20, 30, 40, 50};

cout << "first: " << numbers[0] << endl;
cout << "first: " << *numbers << endl;

cout << &(numbers[0]) << endl;
cout << numbers << endl;
```

Output:

```
first: 10
first: 10
0xbffffb00
0xbffffb00
```


Pointer Arithmetic

- When you **add a value n to a pointer**, you are actually adding n times the size of the data type being referenced by the pointer.

```
int numbers[] = {10, 20, 30, 40, 50};

// sizeof(int) is 4.
// Let us assume numbers is stored at 0xbffffb00
// Then numbers+1 is really 0xbffffb00 + 1*4, or 0xbffffb04
// And numbers+2 is really  0xbffffb00 + 2*4, or 0xbffffb08
// And numbers+3 is really  0xbffffb00 + 3*4, or 0xbffffb0c

cout << "second: " << numbers[1] << endl;
cout << "second: " << *(numbers+1) << endl;

cout << "size: " << sizeof(int) << endl;
cout << numbers << endl;
cout << numbers+1 << endl;
```

Output:

```
second: 20
second: 20
size: 4
0xbffffb00
0xbffffb04
```


- **Note: array[index] is equivalent to *(array + index)**

Pointers and Arrays

- pointer operations * + can be used with array variables.

```
int list[10];
cin >> *(list+3);
```

- subscript operations: [ ] can be used with pointers.

```
int list[] = {1,2,3};
int *ptr = list;
cout << ptr[2];
```


Comparing Pointers

- pointers (addresses) maybe compared using the 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`

Dynamic Memory Allocation

- When a function is called, memory for local variables is automatically allocated.
- When a 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 your program 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 <u>address</u> of newly created <u>anonymous</u> variable.
- use dereferencing operator to access it:

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


Dynamically allocated 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;
}
```

- Program will throw an exception and terminate if not enough memory available to allocate

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;  // note [] required for dynamic arrays!
```

- 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:

```
int * findZero (int arr[]) {
   int *ptr;
   ptr = arr;
   while (*ptr != 0)
      ptr++;
   return ptr;
}
```

NOTE: the return type of this function is (int *) or pointer to an int.

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

NOTE: if the function returns dynamically allocated memory, then it is the responsibility of the calling function to delete it.

Returning Pointers from Functions: duplicateArray

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

   int *newArray;
   if (size <= 0)          //size must be positive
      return NULL;         //NULL is 0, an invalid address

   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;  //caller deletes mem
```

Output

```
0 ok
1 ok
2 ok
3 ok
4 ok
```


Structures

- A structure stores a collection of objects of **various** types
- Each element in the structure is a member, and is accessed using the dot member operator.

```
struct Student {
   int idNumber;
   string name;
   int age;
   string major;
};
```

Defines a new data type

```
Student student1, student2;
student1.name = "John Smith";
Student student3 = {123456,"Ann Page",22,"Math"};
```

Defines new variables

Structures: operations

- Valid operations over entire structs:
  - assignment: `student1 = student2;`
  - function call: `myFunc(gradStudent,x);`

```
void myFunc(Student, int); //prototype
```

- **Invalid** operations over structs:
  - comparison: `student1 == student2`
  - output: `cout << student1;`
  - input: `cin >> student2;`
  - Must do these member by member

Arrays of Structures

- You can store values of structure types in arrays.

```
Student roster[40];  //holds 40 Student structs
```

- Each student is accessible via the subscript notation.

```
roster[0] = student1; //copy student1 into 1st position
```

- Members of structure accessible via dot notation

```
cout << roster[0].name << endl;
```


Arrays of Structures: initialization

- To initialize an array of structs:

```
struct Student {
    int idNumber;
    string name;
    int age;
    string major;
};

int main()
{
    Student roster[] = {
        {123456,"Ann Page",22,"Math"},
        {111222,"Jack Spade",18,"Physics"}
    };

}
```


Arrays of Structures

- Arrays of structures processed in loops:

```
Student roster[40];

//input
for (int i=0; i<40; i++) {
  cout << "Enter the name, age, idNumber and "
       << "major of the next student: \n";
  cin >> roster[i].name >> roster[i].age
      >> roster[i].idNumber >> roster[i].major;
}

//output all the id numbers and names
for (int i=0; i<40; i++) {
  cout << roster[i].idNumber << endl;
  cout << roster[i].name << endl;
}
```


Passing structures to functions

- Structure variables may be passed as arguments to functions:

```
void getStudent(Student &s) {  // pass by reference
  cout << "Enter the name, age, idNumber and "
       << "major of the student: \n";
  cin >> s.name >> s.age >> s.idNumber >> s.major;
}

void showStudent(Student x) {
   cout << x.idNumber << endl;
   cout << x.name << endl;
   cout << x.age << endl;
   cout << x.major << endl;
}

// in main:
Student student1;
getStudent(student1);
showStudent(student1);
```

Pointers to structures

- We can define pointers to structures

```
Student s1 = {12345,"Jane Doe", 18, "Math"};
Student *ptr = &s1;
```

- To access the members via the pointer:

```
cout << *ptr.name << end;    // ERROR: *(ptr.name)
```

- dot operator has higher precedence, so use ():

```
cout << (*ptr).name << end;
```

- or equivalently, use ->:

```
cout << ptr->name << end;
```


Dynamically Allocating Structures

- Structures can be dynamically allocated with new:

```
Student *sptr;
sptr = new Student;

sptr->name = "Jane Doe";
sptr->idNum = 12345;
...
delete sptr;
```

- Arrays of structures can also be dynamically allocated:

```
Student *sptr;
sptr = new Student[100];
sptr[0].name = "John Deer";
...
delete [] sptr;
```

No arrows (->) necessary.
It's just an array of Student