Pointers & Dynamic Memory Allocation

Gaddis: Chapters 9, 11

CS 5301
Spring 2014
Jill Seaman

Pointers and Addresses

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

```cpp
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

```cpp
int *ptr;
```

• “ptr is a pointer to an int”. It can contain addresses of int variables.

```cpp
ptr = &x;
```

Dereferencing and initializing

• The unary operator * is the dereferencing operator.

• *ptr is an alias for the variable that ptr points to.

```cpp
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:

```cpp
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.

```cpp
//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.

```cpp
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
```

Output:
```
first: 10
first: 10
```

```
Pointers and Arrays

- pointer operations can be used with array variables.

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

```
Pointers and Arrays

- subscript operations can be used with pointers.

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

Output:
```
second: 20
```

```
Comparing Pointers

- pointers (addresses) maybe compared using the relational operators:

```
< <= > >= == !=
```

```
Examples:

```cpp
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
```

Note: array[index] is equivalent to *(array + index)
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 address of newly created anonymous 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 = arr;
    while (*ptr != 0) {
      ptr++;
    }
    return ptr;
  }
  ```
- The returned pointer must point to:
  - Dynamically allocated memory OR
  - An item passed in via an argument

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

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

## 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 Doe";
  ...
  delete [] sptr;
  ```

**Output**

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