Pointers and Arrays

For : COP 3330.

Object oriented Programming (Using C++)

http://www.compgeom.com/~piyush/teach/3330

Piyush Kumar

Introduction To Pointers

- A pointer in C++ holds the value of a memory address
- A pointer's type is said to be a pointer to whatever type should be in the memory address it is pointing to
  - Just saying that a variable is a pointer is not enough information!
- Generic syntax for declaring a pointer:

```
dataType *pointerVarName;
```

- Specific examples

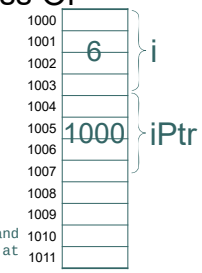
```
int *iPtr;   //Declares a pointer called "iPtr" that will
             //point to a memory location holding an
             //integer value
float *fPtr; //Declares a pointer called "fPtr" that will
             //contain an address, which is an address of
             //a memory location containing a float value
```

The Operator &

- There is an operator, &
  - When this symbol is used as a unary operator on a variable it has a different meaning than as a binary operator or type specifier
  - It is unrelated to the "logical and", which is && or "bitwise and", &
  - It is unrelated to the use of & in regards to reference parameters
- The operator & is usually called the "address of" operator
- It returns the memory address that the variable it operates on is stored at in memory
- Since the result is an address, it can be assigned to a pointer

Using The "Address Of" Operator

```
int  i = 6; //Declares an int, stored
            //in memory somewhere
            //In this example, it is
            //stored at address 1000

int *iPtr;  //Declares a pointer.  The
            //contents of this variable
            //will point to an integer
            //value.  The pointer itself
            //must be stored in memory, and
            //in this example, is stored at
            //memory location 1004

iPtr = &i;  //Sets the iPtr variable to
            //contain the address of the
            //variable i in memory
```

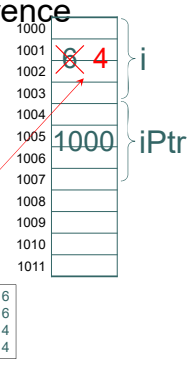


The Operator *

- Like the &, the * operator has another meaning as well
- The * operator is usually referred to as the "dereference operator"
- The * operator operates on a pointer value
  - The meaning is *"Use the value that this pointer points to, rather than the value contained in the pointer itself"*
- If a pointer is of type "int *" and the dereference operator operated on the pointer, the result is a value of type "int"
- Dereferenced pointers can be used as L-values or R-values
  - When used as an L-value (on the left side of an assignment), the pointer is unaffected, but the memory that it points to is changed
- When a pointer that is pointing to memory you are not allowed to access is dereferenced, the result is a program crash via a "segmentation fault"

Using The Dereference Operator

```
int  i = 6; //Declares integer called i
int *iPtr;  //Declares a pointer to an int

iPtr = &i;  //Sets the iPtr variable to
            //contain the "address of" the
            //variable i in memory

cout << "i: " << i << endl;
cout << "i: " << *iPtr << endl;

*iPtr = 4;  //Changes the memory being
            //pointed to by iPtr to contain
            //the value 4

cout << "i: " << i << endl;
cout << "i: " << *iPtr << endl;
```

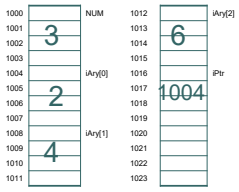


Arrays And Pointers

- The name of an array variable in C++, without the use of the [ ] operator, represents the starting address of the array
- This address can be stored in a pointer variable
  - Since array values are guaranteed to be in contiguous memory, you can access array values using this one pointer
  - Examples of this will come later, after discussing "pointer arithmetic"

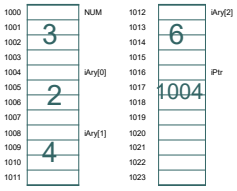
```
const int NUM = 3;
int   iAry[NUM] = { 2, 4, 6 };
int  *iPtr;

iPtr = iAry; //Assigns iPtr to point
             //to the first integer
             //in the iAry array

//This cout prints the value of the
//value stored in the location iPtr
//points to (the first int in the
//iAry, in this case)
cout << "val: " << *iPtr << endl;
```

val: 2



Pointer Arithmetic, Motivation

- Incrementing the contents of an "int *" variable by one doesn't make sense
  - Integers require 4 bytes of storage
  - Incrementing iPtr by 1, results in the pointer pointing to location 1005
  - However, location 1005 is not an address that is the starting address of an integer
- Dereferencing a pointer that contains an invalid memory location, such as 1005 **may** result in a ***Bus Error***
  - When your program results in a bus error, the program crashes immediately (Segmentation fault).

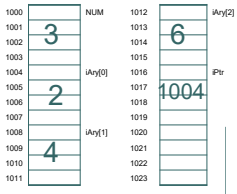


Pointer Arithmetic, Description

- Recall that a pointer type specifies the type of value it is pointing to
  - C++ can determine the size of the value being pointed to
  - When arithmetic is performed on a pointer, it is done using this knowledge to ensure the pointer doesn't point to intermediate memory locations
  - If an int requires 4 bytes, and iPtr is a variable of type "int *", then the statement "iPtr++;" actually increments the pointer value by 4
  - Similarly, "iPtr2 = iPtr + 5;" stores the address "five integers worth" past iPtr in iPtr2
    - If iPtr was 1000, then iPtr2 would contain the address 1020
    - 1020 = 1000 + 5 * 4
- Pointer arithmetic is performed automatically when arithmetic is done on pointers
  - No special syntax is required to get this behavior!

Using Pointer Arithmetic

```
const int NUM = 3;
int   iAry[NUM] = { 2, 4, 6 };
int  *iPtr;
int  *iPtr2;
int  *iPtr3;
int   i;

iPtr = iAry; //Assigns iPtr to point
             //to the first integer
             //in the iAry array

iPtr3 = iAry;
for (i = 0; i < NUM; i++)
{
  iPtr2 = iPtr + i;
  cout << i << " " <<
          iAry[i] << " " <<
          *iPtr2 << " " <<
          *iPtr3 << " " <<
          *(iPtr + i) << endl;
  iPtr3++;
}
```



0 2 2 2 2
1 4 4 4 4
2 6 6 6 6

Other Variables

Static Allocation Of Arrays

- All arrays discussed or used thus far in the course have been "statically allocated"
  - The array size was specified using a constant or literal in the code
  - When the array comes into scope, the entire size of the array can be allocated, because it was specified
- You won't always know the array sizes when writing source code
  - Consider a program that modifies an image
  - As the developer, you won't know what image size the user will use
  - One solution: Declare the image array to be 5000 rows by 5000 columns
    - Problem #1: This likely wastes a lot of memory – if the user uses an image that is 250x250, then there are 24,937,500 unused pixels. If each pixel requires 4 bytes, this is almost 100 MB (megabytes!) of wasted space
    - Problem #2: What if the user needs to edit an image that is 6000x6000? Your program will fail, and likely result in a crash

Dynamic Allocation Of Arrays

- If an array is "dynamically allocated", then space is not reserved for the array until the size is determined
  - This may not be until the middle of a function body, using a value that is not constant or literal
  - The size may be input by the user, read from a file, computed from other variables, etc.
- As memory is "claimed" using dynamic allocation, the starting address is provided, allowing it to be stored in a pointer variable
- Since pointers can be used to access array elements, arrays can be dynamically allocated in this way
- Dynamically allocated memory is claimed from the heap, as opposed to the stack

The "new" Operator

- A new operator is used to perform dynamic allocation
  - The operator is the "new" operator
- The new operator:
  - Attempts to find the amount of space requested from the heap
  - "Claims" the memory when an appropriately sized available chunk of the heap is found
  - Returns the address of the chunk that was claimed
- "new" can be used to allocated individual variables:

```
iPtr = new int; //allocates an int variable
```

- "new" can also be used to allocated arrays of variables:

```
iPtr = new int[5]; //allocates an array of 5 integers
```

- Array elements can be accessed using pointer arithmetic and dereferencing, or via the well-know [ ] operator, indexing an array

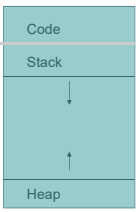
Static Vs Dynamic Allocation

Stack allocation

```
int intArray[10];
intArray[0] = 6837;
```

Heap allocation

```
int *intArray;
intArray = new int[10];
intArray[0] = 6837;

...

delete[] intArray;
```



Dynamic Allocation Of Arrays, Example

- This fragment lets the user decide how big of an array is needed

```
int i;      //Loop variable
int *iary;  //This will be our array - an int pointer
int num;    //Length of the array (input from user)

cout << "Enter length of array: ";
cin >> num;
iary = new int[num]; //Dynamically declare an ary.  Get
                     //necessary mem, assign address to iary
for (i = 0; i < num; i++)
{
  cout << "Enter int num " << i << ":";
  cin >> iary[i];  //use iary as if it were an array!
}

for (i = 0; i < num; i++)
{
  cout << "Index " << i << ": " << iary[i] << endl;
}
```

Outputs Of Dynamic Allocation Example

```
Enter length of array: 7
Enter int num 0:3
Enter int num 1:1
Enter int num 2:6
Enter int num 3:8
Enter int num 4:3
Enter int num 5:2
Enter int num 6:1
Index 0: 3
Index 1: 1
Index 2: 6
Index 3: 8
Index 4: 3
Index 5: 2
Index 6: 1
```

```
Enter length of array: 3
Enter int num 0:8
Enter int num 1:4
Enter int num 2:1
Index 0: 8
Index 1: 4
Index 2: 1
```

Note: In the left example, the array required 28 bytes of memory (7 * 4). Exactly 28 bytes was allocated for the array.

In the right example, the array required only 12 bytes (3 * 4). Exactly 12 bytes was allocated for the array, and no extra memory was unused and wasted.

Another Dynamic Allocation Example

- What is the likely result of the following program fragment?

```
int i;      //Loop variable
int *iary;  //This will be our array - an int pointer
int num;    //Length of the array (input from user)

for (i = 0; i < 100000; i++)
{
  num = 50000;

  iary = new int[num];

  //Call a function to randomly fill the array
  //Do some sort of processing on the 50000 element ary
  //Do it again and again and again, accumulating stats.
}
```

Example Problem Description

- The likely result would be that the program would be a failure
  - The reason is that the new operator claims the memory requested each iteration of the loop
  - There is only a finite amount of memory, though, and the amount requested is likely beyond the amount available
- The problem is that while the memory is claimed, it is never released, or "freed", or "deleted"
- If you don't free the memory, but you do change the pointer pointing at it to point to a different address, then:
  - The original memory is still claimed
  - There is no way to access the original memory, since no pointers are pointing to it
  - The chunk of memory is wasted throughout the entire execution of the program
  - This is referred to as a "memory leak", and should be avoided

Using The "delete" Operator

- Dynamically allocated memory can be released back into the available memory store using the "delete" operator
- The delete operator operates on a pointer and frees the memory being pointed to
  - Recall – a pointer may be pointing to a single value, or an array of values
  - Due to this, the delete operator is used differently to delete single values and arrays
- Deleting a single value being pointed to:

```
delete iPtr;
```

- Deleting an array of values being pointed to:

```
delete [] iPtr;
```

- Using the delete operator on a null pointer has no effect
- Using the delete operator on a pointer pointing to memory that is not currently claimed by your program will cause a segmentation fault
  - Initialize all pointers to 0 (zero)
  - Set all pointers to 0 after using the delete operator on them

Fixing The Memory Leak

```
int i;      //Loop variable
int *iary;  //This will be our array - an int pointer
int num;    //Length of the array (input from user)

for (i = 0; i < 100000; i++)
{
  num = 50000;

  iary = new int[num];

  //Call a function to randomly fill the array
  //Do some sort of processing on the 50000 element ary
  //Do it again and again and again, accumulating stats.

  delete [] iary; //No need to tell delete the size of
                  //the array.  This only frees up the
                  //memory that iary is pointing to. It
                  //does NOT delete the pointer in any way
}
```

Dynamically Allocating Objects

- The arrow operator is another operator needed for working with pointers
  - The arrow operator is a dash and a greater than symbol: ->
  - It is used to access public member variables or functions of an object that is being pointed to by a pointer
  - It is used the same way the dot operator is used on an actual object, but the arrow is used on a pointer variable instead
- The arrow is used for convenience
  - Alternatively, you could deference the pointer and use the dot operator
- Since the arrow operator implies a dereference, using the arrow operator on a pointer that doesn't point to claimed memory results in a segmentation fault!

Using The Arrow Operator ->

```
class CircleClass
{
  public:
    float x;
    float y;
    float z;
    float radius;
};

int main()
{
  CircleClass myObj;
  CircleClass *myPtr;
  myPtr = &myObj;

  myObj.x = 5;
  myPtr->y = 9;
  myObj.z = 15;
  myPtr->radius = 56.4;

  ...
```

I access the same memory location using both the actual object and a pointer to that object.

The dot operator is used with the object

The arrow operator is used with the pointer

Dynamically Allocating Objects

```
class TempClass
{
  public:
    int    ival;
    double dval;
};

int main()
{
  TempClass *temp;      //4 bytes (or sizeof(tempClass*)
  temp = new TempClass; //Claims enough space for all
                        //members of a tempClass object

  temp->ival = 16;      //Since temp is a pointer,
  temp->dval = 4.5;     //the arrow operator is used

  ...
```

Note: The actual object that is allocated (the memory location) never gets a name! It is *only* pointed to by the temp pointer!

Using Constructors With Dynamic Allocation

- Remember – a constructor is used whenever an object is allocated, whether statically or dynamically

```
class IntClass
{
  public:
    int val;

    IntClass() //Default ctor sets val to 0
    {
      val = 0;
    }
    IntClass(int inVal) //Initializes val to value passed in
    {
      val = inVal;
    }
};

IntClass ic;    //sets ic.val to 0                          } Uses the
IntClass *icPtr = new IntClass;      //sets icPtr->val to 0 } default ctor

IntClass ic2(6); //sets ic2.val = 6                         } Uses the
IntClass *icPtr2 = new IntClass(10); //sets icPtr->val to 10} value ctor
```

The sizeof Operator

- Often, you need to know how many bytes of memory a variable or type requires.
- Different architectures use different sizes.
- Use the sizeof operator to determine the current architecture's size of a var or type

```
int num;    //Length of the array (input from user)
cout << "sizeof(int): " << sizeof(int) << endl;
cout << "sizeof(float): " << sizeof(float) << endl;
cout << "sizeof(double): " << sizeof(double) << endl;
cout << "sizeof(char): " << sizeof(char) << endl;
cout << "sizeof(num): " << sizeof(num) << endl;
```

```
sizeof(int): 4
sizeof(float): 4
sizeof(double): 8
sizeof(char): 1
sizeof(num): 4
```

Result may vary on different machines! (These results are common)

Dynamically Alloc Mem in C

- Operators "new" and "delete" don't exist in C, but C programmers still need dynamic allocation.
- Three important functions for C dynamic allocation

```
//malloc takes one parameter, size, which is simply
//the number of bytes you are requesting..  Returns a
//void *, which is a generic pointer to any type.
void *malloc(size_t size);

//calloc initializes each ary element to 0.  nelem is
//the number of elements you are requesting, and
//elsize is the number of bytes each element requires.
void *calloc(size_t nelem, size_t elsize);

//free takes one param, which is a pointer to memory
//that was previously allocated using malloc or calloc
void free(void *ptr);
```

Dynamically Alloc Mem in C Example

```
#include <stdlib.h>
//---
int *iary;  //This will be our array - an int pointer
int *iary2; //Another integer array.
int num;    //Length of the array (input from user)

cout << "Enter length of ary: ";
cin >> num;

iary = (int *)malloc(num * sizeof(int)); //not init.
iary2 = (int *)calloc(num, sizeof(int)); //init to 0

//Something useful happens here..

//Free up the memory now!
free(iary);
free(iary2);
```

Prefer C++ to C

```
string *stringarray1 = static_cast<string *> (malloc(10*sizeof(sting)) );
string *stringarray2 = new string[10];

…

free (stringarray1); // no destructors called
// What happened if string object reallocated stuff? Enlarged itself?
delete[] stringarray2;
```

Use same form of new/delete

```
string *stringarray2 = new string[100];

…

delete stringarray2;
// Program behavior undefined
// At least the 99 strings are still in the memory somewhere → Memory LEAK!
```

new/delete in constructors/destructors

- For a class with dynamically allocated memory, Initialize pointers in constructors to 0.
- If unknown size, make them null.
  - Deleting a null pointer is always safe.
- Make sure you delete them all in the destructor.
- Useful when you need to implement a class that manages a resource. (Never manage multiple resources in a single class, this will only lead to pain.)

Big Law of three

- if a class defines one (or more) of the following it should probably explicitly define all three
  - Destructor
  - Copy constructor
    - Point2d second(first);
  - Assignment operator
    - first = second;

Example

```
class person
{
    char* name;
    int age;

public:

    // the constructor acquires a resource:
    // in this case, dynamic memory obtained via new[]
    person(const char* the_name, int the_age)
    {
        name = new char[strlen(the_name) + 1];
        strcpy(name, the_name);
        age = the_age;
    }

    // the destructor must release this resource via delete[]
    ~person()
    {
        delete[] name;
    }
};
```

Rule of three: Missing Copy Constructor/Assignment operator

Example

```
// 1. copy constructor
person(const person& that)
{
    name = new char[strlen(that.name) + 1];
    strcpy(name, that.name);
    age = that.age;
}

// 2. copy assignment operator
person& operator=(const person& that)
{
    if (this != &that)
    {
        delete[] name;
        // This is a dangerous point in the flow of execution!
        // We have temporarily invalidated the class invariants,
        // and the next statement might throw an exception,
        // leaving the object in an invalid state :(
        name = new char[strlen(that.name) + 1];
        strcpy(name, that.name);
        age = that.age;
    }
    return *this;
}
```

Advice

- Most of the time, you do not need to manage a resource yourself because an existing class can do that for you.

```
class person
{
    std::string name;
    int age;

public:

    person(const std::string& name, int age) : name(name), age(age)
    {
    }
};
```

Out of memory errors

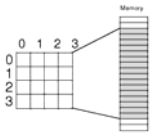
```
#include <iostream>

using namespace std;

void OutOfMemory(){
        cerr << "No more memory\n";
    abort();
}


int main() {
        set_new_handler(OutOfMemory);
        double *pbigarray = new double[200000000];
        cout << "I am here\n";
        pbigarray[9999999] = 123;
        cout << pbigarray[999999] << endl;

        return 0;
}
```

Multidimensional Arrays

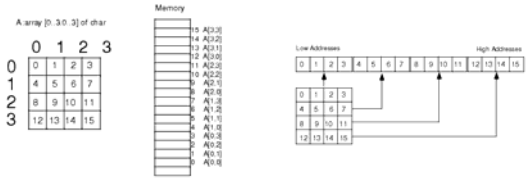
- Definition
  `Type MDarray[size_1][size_2] ... [size_k]`
- What it means
  - k - dimensional array
  - **`MDarray`**: array identifier
  - **`size_i`**: a positive constant expression
  - **`Type`**: standard type or a previously defined user type and is the base type of the array elements
- Semantics
  - **`MDarray`** is an object whose elements are indexed by a sequence of **`k`** subscripts
  - the **`i`**-th subscript is in the range **`0 ... size_i-1`**

Multi-dimensional Arrays

- Multidimensional arrays are laid out in row-major order
- Consider
  `int M[2][4];`
- `M` is two-dimensional array that consists of 2 subarrays each with 4 elements.
  - 2 rows of 4 elements
- The array is assigned to a contiguous section of memory
  - The first row occupies the first portion
  - The second row occupies the second portion



Multi-dimensional Arrays



Multi-dimensional arrays

- Example:
  int myImage[NROWS][NCOLS];

Can be used as parameter in a function prototype. Example:

```
void  process_matrix( int in[ ][4], int out[ ][4], int nrows)
void  process_matrix( int in[4][4], int out[4][4], int nrows)

//Invalid
void  process_matrix( int in[][], int out[][], int nrows)
```

Multi-dimensional arrays

- Really an array of arrays.

00	01	02	03
10	11	12	13
20	21	22	23

```
int main() {
        int x [3][4];
        int (*ip)[4]; // pointer to an array of 4 integers
        int *oip = &(x[0][0]);

        ip = x;
        for (int i = 0 ; i < 3; ++i)
        for (int j = 0; j < 4; ++j)
                x[i][j] = i*10 + j;

        cout << (*ip)[0] << "\t" << (* (++ip) )[0] << endl;
        cout << * (++oip) << endl;
        return 0;
}
```

Recommended exercises: 5.9, 5.30, 5.14, 5.23

Multi-dimensional Arrays

- int (*matrix)[10];
  - Pointer to an array of 10 integers
- int *matrix[10];
  - Array of 10 integer pointers.
  - Example:
    - int main(int argc , char *argv[])
    - argv[0] = “prgram name”
    - argv[1] = “../data/filename”
    - argv[2] = “2”

Buffer Overflow problems

Let A be a string and B be an integer.
Changing A could change B!

A	A	A	A	A	A	A	A	B	B
0	0	0	0	0	0	0	0	0	3

```
/* overflow.c - demonstrates a buffer overflow */
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
        char buffer[10];
        if (argc < 2) {
          fprintf(stderr, "USAGE: %s string\n", argv[0]);
          return 1;
        }

        strcpy(buffer, argv[1]);
        return 0;
}
```

Buffer Overflow problems

Let A be a string and B be an integer.
Changing A could change B!


```
/* better.c - demonstrates one method of fixing the problem */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
        char buffer[10];
        if (argc < 2) {
          fprintf(stderr, "USAGE: %s string\n", argv[0]);
          return 1;
        }

        strncpy(buffer, argv[1], sizeof(buffer));
        buffer[sizeof(buffer) - 1] = '\0';
        return 0;
}
```

Type conversions

- C-Style casts
  - `float average = (float) sum / items;`
  - `float average = (float) (sum/items);`
- C++ Style
  - static_cast< *type* >(*identifier*)
  - `float average = static_cast< float >( sum ) / items;`
  - `float average = sum / static_cast< float >( items );`

Type conversions

x=(float) i;	**cast in C++ - C notation**
x=float(i);	**cast in C++, functional notation**
x=static_cast<float>(i);	**ANSI C++ - recommended**
i=reinterpret_cast<int>(&x)	**ANSI C++, not portable and system dependent**
func(const_cast<int>(c_var))	**where C_var is a const variable Used for removing "const-ness" when invoking func. Use with care.**

static_cast

- static_cast<T>(expression)
  The static_cast<>() is used to cast between the integer types.
  'eg' char->long, int->short etc.
- Static cast is also used to cast pointers to related types, for example casting void* to the appropriate type.

```
BaseClass_Employee* a = new DerivedClass_Manager();
static_cast<DerivedClass_Manager>(a)->derivedClassMethod();
```

reinterpret_cast

```
float f = 2.5f;
double * pd = reinterpret_cast<double*>(&f);

cout << f << endl << *pd  << endl;
```

Outputs (!!!):

```
$ ./a.exe
2.5
3.50861e+159
```

Reinterpret cast simply casts one type bitwise to another. Any pointer or integral type can be casted to any other with reinterpret_cast, easily allowing for misuse. static_cast will not be allowed in this case.

const_cast

- const_cast<T>(expression)
  The const_cast<>() is used to add/remove const(ness) of a variable.

```
class A {public: void func() {} };
void f(const A& a)
{
        A& b = const_cast<A&>(a);
        b.func();
}
```

dynamic_cast

- Dynamic cast is used to convert pointers and references at run-time, generally for the purpose of casting cast a pointer or reference up or down an inheritance chain (inheritance hierachy).

```
class Employee { ... };
class Manager : public Employee { ... };

void f(Employee* a) {
        Manager* b = dynamic_cast<Manager*>(a);
}
```

Type conversions

- All Pointers can be converted to void *
- An explicit cast is required in C++ when you want to convert a void * to another pointer type.

```
char *char_p;
void *generic_p;
. . .
generic_p=char_p;                          // OK, char* va in void*
char_p=generic_p;                          // OK in C, illegal in C++
char_p=static_cast<char *> (generic_p);    // The C++ way.
```

Implicit conversions

- **char, short and bool are promoted to int**
- **Integer types which cannot be represented with an int are promoted to unsigned**
- **In an expression with mixed type, lower order operand are promoted to the upper order, with the following rule:**
  - **int < unsigned < long < unsigned long < float < double < long double**
- **bool is an integer type, true is promoted to 1 and false to 0**