eecs 281 Data Structures and Algorithms

Discussion 2: Week of Sep 14, 2011

MichiganEngineering

Contents

- Pointers and references
- valgrind
- Forward declarations
- Inheritance and polymorphism
- Virtual functions
- Constructor/destructor

MichiganEngineering

Pointers

- *A pointer is a variable that contains an address of some data object in memory (or sometimes, a function).*
- *Declare pointers with the * (star) operator. Examples:*

```
int* x;
int *x;
char* c;
char *c;
```

- Question: what is the difference between placing the star with the type, and with the variable?
- Answer: *the two are equivalent.* We could also separate the * from both: **int * x** is perfectly fine.

MichiganEngineering

Pointers - initialization

- We've *declared but not defined them. Right now, they're pointing at whatever happened to be there; i.e. some random memory location.* ***This is dangerous.***
- *We should initialize our pointers to something. If we don't yet have anything, it's good practice to initialize with NULL, (0). NULL means "this pointer isn't (yet) pointing to anything."*

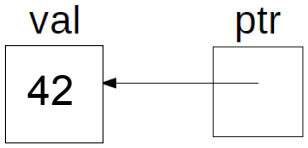
```
int* x = NULL;
char* c = NULL;
```

MichiganEngineering

Pointers – Address-of operator

- We need a way to retrieve the address of data objects (e.g. variables). Use the *address-of operator, denoted by &.*

```
int myInt = 42;
int* pMyInt = &myInt;
```



- The situation is a little bit different for arrays. Arrays already denote addresses, so no & is needed:

```
int myIntArray[ 5 ] = { 2, 4, 6, 8, 10 };
int* pMyIntArray = myIntArray;
```


Pointers – Dereferencing

- Now we need a way of obtaining the data from a given address. This is known as *dereferencing*. * is the *dereferencing operator*. You can think of * as kind of the inverse operator of &.
- **WARNING:** There may be confusion between the * for pointer declaration, and the * for dereferencing. *These are separate things, and can be differentiated by their contexts.*

```
int myInt = 42;
int* pMyInt = &myInt;
cout << *pMyInt << endl;
( *pMyInt ) *= 2;
cout << *pMyInt << endl;
cout << myInt << endl;
```

```
42
84
84
```


Pointers – Example 1

- What will the following function print to the console?

```
void pointerExample_1( void ) {
        int myInt0 = 100;
        int myInt1 = 200;
        int* pMyInt0 = &myInt0;
        int* pMyInt1 = pMyInt0;

        *pMyInt0 += 20;//deref +20
        *pMyInt1 += 40;//deref +40

        cout << myInt0 << endl;
        cout << myInt1 << endl;
        cout << *pMyInt0 << endl;//deref
        cout << *pMyInt1 << endl << endl;//deref

        pMyInt1 = &myInt1;
        *pMyInt0 *= 2;
        *pMyInt1 = myInt0 + *pMyInt0;

        cout << myInt0 << endl;
        cout << myInt1 << endl;
        cout << *pMyInt0 << endl;
        cout << *pMyInt1 << endl << endl;
}
```

```
160
200
160
160

320
640
320
640
```


Pointers – Example 2

- What will the following function print to the console?

```
void pointerExample_2( void ) {
        const int arraySize = 5;

        int myIntArray[ arraySize ] = { 2, 0, 6, 8, 10 };
        myIntArray[ 1 ] = myIntArray[ 0 ] * 2;
        int* myPtr = myIntArray; //no & needed (array is ptr)
        ++( *myPtr );//++ 1st elem of array
        ++myPtr;//++ ptr (to 2nd elem)
        *myPtr *= *( myPtr + 1 );// 2nd elem *= deref(3rd elem)
        myPtr += 2;//move ptr to 4th elem
        myPtr[ 1 ] *= myPtr[ 0 ];//elem right of 4th *= 4th elem
        *myPtr -= 1;//decrement 4th elem

        // display
        for ( unsigned int i = 0; i < arraySize; ++i ) {
                cout << myIntArray[ i ] << endl;
        }
}
```

```
3
24
6
7
80
```

Pointers – Array dereferencing, type independence

- Array subscript operator ([]) is a form of indexed dereferencing.
- C++ pointers are strictly typed; can't have an **int** pointer point to a **char** pointer, for example.

```
int myInt = 42;
int* pMyInt = &myInt;

char c = 'c'
char* pMyChar = &c;

pMyInt = pMyChar; //DOES NOT COMPILE!!!
```


Pointers – Typecasting

- We *can, however, cast one data type to another:*

```
int myInt = 42;
int* pMyInt = &myInt;

char c = 'c'
char* pMyChar = &c;

// pMyInt = pMyChar;
pMyInt = reinterpret_cast< int* >( pMyChar );
```

- There are several typecasts you can do. *reinterpret_cast* could be interpreted as the *most dangerous*. *static_cast* is safer, but there are situations in which it won't do.
- Old C-style casts like *pMyInt = (int*)pMyChar* will still work, though. These are like reinterpret_cast<>.


A few flavors of typecasting explained

- dynamic_cast <new_type> (expr)

//only for casting a derived class to a base class

- static_cast <new_type> (expr)

//casting in either direction between derived class and base class—does not ensure newly-casted object is complete

- reinterpret_cast <new_type> (expr)

//binary copy of value from from one ptr to another—zero typechecking whatsoever


Pointers and new

- Of course, pointers start to become really useful with *new, when we allocate memory on the heap:*

```
//ptr that points to 1st elem of new array
int* pMyIntAry = new int[ 1000 ];

//move ptr to index 25 and set value to 1001
pMyIntAry[ 25 ] = 1001;

//deallocate
delete[] pMyIntAry;
```


Pointers to pointers (to pointers...)

- We can also have *pointers to pointers:*

```
int myInt = 42;
int* pMyInt = &myInt;
int** ppMyInt = &pMyInt;
// int*** pppMyInt = &ppMyInt;
```

- Question: when might this be useful?
- Answer: *implementing dynamic multi-dimensional arrays, implementing pass-by-reference for pointers, arrays of strings, command-line options* e.g. *main (int argc, char **argv)*
- Question: could we do something like *int** ppMyInt = &&myInt* ? What about *int myInt2 = **ppMyInt* ?
- Answer: &&myInt will generate an error, because you can't take the address of an address. **ppMyInt *will* work, though. It evaluates to myInt. (What about **ppMyInt* ?)

```
int row=4,col=5;

int **a=new int*[row];
int i,j;
for (i=0;i<row;i++) {
a[i]=new int[col];
}
for (i=0;i<row;i++) {
for (j=0;j<col;j++) {
a[i][j]=i+j;
cout<<a[i][j]<<" ";
}
```

references

- A ***reference*** is an alternative name for a variable.
- A reference is not a variable.

```
int i = 37;
int& r = i;
```

defines one variable i, and one reference r.

Reference vs. Pointer

- Once created, a reference cannot be "reseated" to reference another object
- References cannot be made null (so must refer to something once created)
- Cannot be uninitialized
- & operator will yield a ptr to referenced obj

Example

```
//what does this output?
int i = 7;
int& ri = i;
cout << i << endl;
cout << ri << endl;

int* ptr_i = &ri;
cout << ptr_i << endl;
cout << *ptr_i << endl;
```

```
7
7
0x...
7
```

references

int i = 37; compare int i = 37;

int& r = i; reference Int* r = &i; address-of

- In C++ all references must be initialized.

```
int& k;
// compiler will complain: error: 'k' declared as reference but not initialized
```

(Most compilers will support a null reference without much complaint, crashing only if you try to use the reference in some way.)

- There are no operators that operate on references

MichiganEngineering

Reference-- usage

- One convenient application of references is in function parameter lists, where they allow passing of parameters used for output with no explicit address-taking by the caller.

```
void square (int x,  int& result) {
      result = x * x;
}
```

```
void square (int x,  int* result){
      *result = x * x;
}
Call: square(3,&y);
```

Call: square(3 , y); //passes read-write y

MichiganEngineering

Reference

- Challenge!

```
int& preinc(int& x) {
    return ++x;//preincrement
}
Call:
int y=1;
preinc(y) = 5; /*returns an
int reference that can then
be manipulated*/
```

```
//same as ++y, y = 5
```

MichiganEngineering

Valgrind

- Command line program
- Allows you to detect and locate memory leaks
- Usage:

valgrind –tool = memcheck –leak -check=yes [executable name]

MichiganEngineering

Using Valgrind

- Compile with –g option (just like for gdb)
- To run, just type "valgrind" before your normal program execution command
  - valgrind –leak -check=yes ./myprog (detailed memory checking)
- The errors memcheck flags *depend on the execution path*:
  - You need to run a suite of test cases to find bugs


Example Program

```
#include <stdlib.h>

void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0; // problem 1: heap block overrun
} // problem 2: memory leak -- x not freed

int main(void) {
    f();
    return 0;
}
```


Example Output: error 1

```
==19182== Invalid write of size 4
==19182== at 0x804838F: f (example.c:6)
==19182== by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after
   a block of size 40 alloc'd
==19182== at 0x1B8FF5CD: malloc
   (vg_replace_malloc.c:130)
==19182== by 0x8048385: f (example.c:5)
==19182== by 0x80483AB: main (example.c:11)
```


Example output: error 2

```
==19182== 40 bytes in 1 blocks are
   definitely lost in loss record 1 of 1
==19182== at 0x1B8FF5CD: malloc
   (vg_replace_malloc.c:130)
==19182== by 0x8048385: f (a.c:5)
==19182== by 0x80483AB: main (a.c:11)
```


Classes and Structs

- What are the differences between classes and structs in C++?
  - classes default to private access
  - structs default to public access
  - `struct`s are mostly used for backward compatibility with C

Forward Declarations

- Let the compiler know that the definition is coming later.
  - Must use pointers (size isn't known yet)

```
/*acknowledges
class B's
existence
but not impl*/
class B;
class A
{
  public:
  B*;
  myB;
  void f(B* b)
}
```

```
class A;

class B
{
  public:
  A*
  myA;
  void f(A* a)
}
```

Inheritance and Polymorphism

- Allow us to specify *relationships between types*
  - Abstraction, generalization, specification
  - The "is a" relationship
  - Examples?
- Why is this useful in programming?
  - Allows for code reuse
  - More intuitive/expressive code

Code Reuse

- General functionality can be written once and applied to **any** derived class
- Derived classes can have selective specialization by adding members or implementing virtual functions

Generic Programming

- Data-type independent way of programming
- Generic classes and functions are written as templates that can be instantiated with different data types to create specialized classes and functions of those data types
- Instantiations done statically at compile time
- How else to program in a data-type independent way?

MichiganEngineering

Dynamic Binding and Polymorphism

- Another data-type independent way of programming
- Multiple classes have different implementation of the same method with the same (overridden) name and interface
- At runtime, the system dynamically decides which method to call (bind to) based on object type, thus enabling polymorphism

MichiganEngineering

Virtual Function

- If a derived class is cast as its base class and the two classes share an overridden function, that of the base class will be called unless the function is virtual—then the derived class function is called. A virtual func is override-able

```
class Base
{
   public:
 int f();
 virtual int g();
};
class Derived: public Base
{
   public:
 int f();
 int g();
};
```

```
Derived obj;
Base *bp = &obj; //
cast to base
Derived *dp = &obj;

bp→f(); //base f()
dp→f(); //base f()
bp→g(); //base g()
dp→g(); //derived g()
```

MichiganEngineering

Virtual Functions

- A virtual function is a method in a base class that can be overridden by a derived class method.

```
class base
{ public:
        void virtual print() { cout << "calling base.print().\n";}
};

class derived : public base
{ public:
void print() { cout << "calling derived.print().\n";}
};

int main()
{
base A;  derived B;  base *pb;
A.print(); // calls base::print()
B.print(); // calls derived::print()
pb = &B;
pb->print(); // what does this call?
}
```

MichiganEngineering

Virtual Functions

- A virtual function is a method in a base class that can be overridden by a derived class method.

```
class base
{ public:
        void virtual print() { cout << "calling base.print().\n";}
};

class derived : public base
{ public:
void print() { cout << "calling derived.print().\n";}
};

int main()
{
base A;  derived B;  base *pb;
A.print(); // calls base::print()
B.print(); // calls derived::print()
pb = &B;
pb->print(); // what does this call?
}
```

Output looks like
calling base.print().
calling derived.print().
calling derived.print().


Template vs. Virtual Function

- Which is preferred? Why?
- Template is instantiated statically during compile time, more efficient, but code usuallly hard to read (and write)
- Virtual function is bound dynamically at runtime, less efficient, but code easier to maintain


Constructor/destructor

constructors - used to set up new instances

- •default - called for a basic instance
- •copy - called when a copy of an instance is made (assignment)
- •other - for other construction situations

destructors - used when an instance is removed


The Default Constructor

- A default constructor has no arguments, it is called when a new instance is created
  - C++ provides a default constructor that initializes all fields to 0
- To write your own, add following to your class:

```
class MyClass {
  public:
    …
    MyClass() {    // repeat class name, no
      code here    // return type
    }
}
```


Example Default Constructor

```
class Robot {
  public:
    static int numRobots = 0;
    Robot() {
      numRobots++;
      locX = 0.0;
      locY = 0.0;
      facing = 3.1415 / 2;
    }
  private:
    float locX;
    float locY;
    float facing;
}
```

MichiganEngineering

Destructor

- A destructor is normally not critical, but if your class allocates space on the heap, it is useful to deallocate that space before the object is destroyed
  - C++ provides a default destructor does nothing
  - can only have one destructor
- To write your own, add following to your class:

```
class MyClass {
  public:
    …
    ~MyClass() {
      code here
    }
}
```

MichiganEngineering

Example destructor

```
class Robot {
  public:
    char *robotName;
    Robot() {
      robotName = 0;
    }
    void setRobotName(char *name) {
      robotName = new char[strlen(name)+1];
      strcpy(robotName,name);
    }
    ~Robot() {
      delete [] robotName;
    }
}
```

MichiganEngineering

Creating Objects and Dynamic Arrays in C++

- `new` calls default constructor to create an object
- `new[]` calls default constructor for each object in an array
  - no constructor calls when dealing with basic types (`int`, `double`)
  - no initialization either
- `delete` invokes destructor to dispose of the object
- `delete[]` invokes destructor on each object in an array
  - no destructor calls when dealing with basic types (`int`, `double`)

MichiganEngineering

Array Destructor

For array of objects, the following suffices:

```
~Array()
{
    if (data != NULL)
    {
        delete[] data; // Object destructor will free
        data = NULL; // any memory pointed to by object
     }
}
```

But need "deep" destructor for array of pointers:

```
~Array()
{
   if (data != NULL)
   {
       for (int i =0; i < length; i++)
       {
          if (data[i])
          {
             delete data[i];
          }
        }
   delete[] data;
   data = NULL;
   }
}
```

MichiganEngineering

Question

Do we need to do "deep" destruct on both `arr1` and `arr2`?

```
class myObj {
  int len;
  int *data;
 public:
  myObj(int len=10) {
   if (len) {
    data = new int[len];
   }
  }
  ~myObj() { if (data) delete data; }
};

myObj *arr1 = new myObj[10];
myObj **arr2 = new myObj*[10];
```

MichiganEngineering

The Copy Constructor

- A copy constructor is used when we need a special method for making a copy of an instance
  - example, if one instance has a pointer to heap-allocated space, important to allocate its own copy (otherwise, both point to the same thing)
- To write your own, add following to your class:

```
class MyClass {
  public:
    …
    MyClass(const MyClass& obj) {
      code here
    }
}
```

MichiganEngineering

Example Copy constructor

```
class Robot {
  public:
    char *robotName;
    void setRobotName(char *name) {
      robotName = new char[strlen(name)+1];
      strcpy(robotName,name);
    }
    Robot(const Robot& obj) {
      robotName = new char[strlen(obj.robotName)+1];
      strcpy(robotName,obj.robotName);
    }
}
```

MichiganEngineering

Find the error

```
class Array
{
  public:
    int size;
    int* data;

    Array(int size): size(size), data(new int[size]) { }
    ~Array()
    {
        delete[] data;
    }
};
```

```
int main()
{
    Array first(20);
    first.data[0] = 25;
    {
        Array copy = first;
        std::cout << first.data[0] << " " << copy.data[0] <<std::endl;
    }   // (1)
    first.data[0] = 10;    // (2)
}
```

25 25
Segmentation fault



```
#include <iostream>
using namespace std;
class Array
{
  public:
    int size;
    int* data;

    Array(int size): size(size), data(new int[size])  { }

Array(const Array& copy)
    : size(copy.size), data(new int[copy.size])
{
    std::copy(copy.data, copy.data + copy.size, data);
}

    ~Array()
    {
        delete[] data;
    }
};

int main()
{
    Array first(20);
    first.data[0] = 25;
    {
        Array copy = first;
        std::cout << first.data[0] << " " << copy.data[0] <<std::endl;
    }   // (1)
    first.data[0] = 10;    // (2)
}
```


Array Class: Copy Constructor

```
Array(const Array& a) {
  length = a.getLength();
  data = new int[length];
  for (unsigned i =0; i< length; i++) {
    data[ i ] = a[ i ];
  }
}
```

Allows for:

```
Array a(2); // Array a is of length 2
Array b(a); // Array b is a copy of a
```

What's the big-Oh complexity of making a copy?


Array of Objects: Destructor

```
~Array(){
 if (data != NULL){
   delete[] data;
   // Object destructor will free
   // any memory pointed to by object
   data = NULL;
 }
}
```

← at most n times (assume destructor is $O(1)$)

← 1 step

Total: $O(n)$

What if data is of primitive type (e.g., `int`, `char`, `double`)?

Assume data are `int`s:

```
~Array() {
 if data){
   delete[] data ;  // data are ints
   data = NULL;
 }
}
```

An `int` has no destructor

← 1 step

← 1 step

Total: $O(1)$


Array of *Object: Delete

```
~Array(){
 if (data != NULL){
   for (int i =0; i < length; i++) {
      if (data[i]) delete data[i];
   }
   delete[] data;
   data = NULL;
 }
}
```

at most n times (assume destructor is $O(1)$)

1 step

1 step

Total: $O(n)$

MichiganEngineering

Other constructor

- It is often useful to provide constructors that allow the user to provide arguments in order to initialize arguments
- Form is similar to the copy constructor, except parameters are chosen by programmer:

```
class MyClass {
  public;
    …
    MyClass(parameters) {
      code here
    }
}
```

MichiganEngineering

Example constructor

```
class Robot {
  public:
    Robot(float x, float y, float face) {
      locX = x;
      locY = y;
      facing = face;
    }
}
```

calling:

```
  Robot r1(5.0,5.0,1.5);
  Robot r2(5.0,10.0,0.0);
  Robot* rptr;
  rptr = new Robot(10.0,5.0,-1.5);
```

MichiganEngineering

A Combination constructor

Can combine a constructor that requires arguments with the default constructor using default values:

```
class Robot {
  public:
    Robot(float x = 0.0, float y = 0.0,
          float face = 1.57075) {
      locX = x; locY = y; facing = face;
    }
}
```

calling:

```
  Robot r1;   // constructor called with default args
  Robot r2(); // constructor called with default args
  Robot r3(5.0); // constructor called with x = 5.0
  Robot r4(5.0,5.0; // constructor called with x,y = 5.0
  …
```

MichiganEngineering

Hiding the Default constructor

- Sometimes we want to make sure that the user gives initial values for some fields, and we don't want them to use the default constructor
- To accomplish this we declare an appropriate constructor to be used in creating instances of the class in the public area, then we put the default constructor in the private area (where it cannot be called)

Example constructor

```
class Robot {
  public:
    Robot(float x, float y, float face) {
      locX = x;
      locY = y;
      facing = face;
    }
  private:
    Robot() {}
}
```

calling:

```
  Robot r1(5.0,5.0,1.5);
  Robot r2;  // ERROR, attempts to call default constructor
```

Array Constructor

```
class Array {
  unsigned int length;      // array size
  int* data;                // array data

public:
  // Constructor:
  Array(unsigned len=0):length(len) {
    data = (len ? new char[len] : NULL);
  }
};
```

Usage: `Array a(10);`

Queue Constructor

```
class Queue {
  Array queue;  // array as queue

public:
  // Constructor:
  Queue(unsigned len=0) { ... }
};
```

Preferred usage: `Queue q(10);`
to declare a queue of 10 (initial) elements

How to write the constructor?

Queue Constructor: Bad Attempt

```
class Queue {
  Array queue;  // array as queue

public:
  // Constructor:
  Queue(unsigned len=0) {
    queue = Array(len);  // NOT (Why not?)
  }
};
```

Preferred usage: `Queue q(10);`
to declare a queue of 10 (initial) elements
How to write the constructor?

Queue Constructor: Bad Attempt

```
Queue(unsigned len=0) {
  queue = Array(len);  // NOT (Why not?)
}
```

Inefficient:

- a new array of length len is constructed and copied over to queue
- the new array is deconstructed

Potential bug:

- if data is an array of pointers instead of array of objects, the pointers will be freed when the new array is destructed, so the pointers copied to queue will be pointing to junk

Constructor Initialization List

```
class Queue {
  Array queue;  // array as queue

public:
  // Constructor:
  Queue(unsigned len=0) : queue(len) {}
};
```

Usage: `Queue q(10);`

Initialize without creating a copy