CIS 190: C/C++ Programming

Vectors, Enumeration, and Overloading

Outline

- Vectors
- Enumeration
- “Print” functions
- Function Overloading
- New/Delete
- Destructors

Vectors

- similar to arrays, but much more flexible

- provided by the C++ Standard Template Library (STL)
  - must `#include <vector>` to use

Declaring a Vector

```
vector <int> intA;
```

– empty integer vector, called intA

```
vector <int> intB (10);
```

– integer vector with 10 integers, initialized to zero

```
vector <int> intC (10, -1);
```

– integer vector with 10 integers, initialized to -1

Copying Vectors

- can assign one vector to another
  - even if they're different sizes
  - as long as they're the same type

```
intA = intB;
```

- can create a copy of an existing vector when declaring a new vector

```
vector <int> intA (intB);
```

Accessing Vector Members

- two different methods

- square brackets

```
intB[2] = 7;
```

- at() operation

```
intB.at(2) = 7;
```

Accessing Vector Members with []

- square brackets function as they did with arrays in C

- no bounds checking
  - sometimes it works (C is being “nice)
  - sometimes it doesn’t work

Accessing Vector Members with .at()

- `.at()` operator uses bounds checking

- will throw an exception when out of bounds
  - causes program to terminate
  - we can handle it (with try-catch blocks)

- slower than `[]`, but safer

Passing by Reference

- by default, vectors are passed by value
  - a copy is made, and that copy is passed to the function; changes made do not show in main()

- can explicitly pass by reference if necessary

```
// function prototype
void ModifyV (vector <int> &ref);

// function call
ModifyV (refVector);
```

Multi-dimensional Vectors

- multi-dimensional vectors are basically "a vector of vectors"

```
vector < vector <char> > charVec (10);
```

- size at end (here, 10), is optional
  - without it, creates an empty vector

Multi-dimensional Vectors

- multi-dimensional vectors are basically "a vector of vectors"

```
vector < vector <char> > charVec (10);
```

this space in between the two closing '>' characters is required by many implementations of C++

resize()

```
void resize (n, val);
```

- resize function used to resize vectors

- **`n`** is new size of vector
  - if larger than current, vector size is expanded
  - if smaller than current, vector is reduced to first *n* elements
- **`val`** is optional value to place in new elements
  - if not specified, default constructor is used

using resize()

- if we declare an empty vec (**emptyVec**) we can change it to the size **NUM_ROWS** by **NUM_COLS**

```
// resize rows first
emptyVec.resize(NUM_ROWS);

for (int i = 0; i < NUM_ROWS; i++)
{
  // resize each row to new column size
  emptyVec[i].resize(NUM_COLS);
}
```

push_back()

- add a new element at the end of a vector

```
void push_back (val);
```

- **val** is the value to be assigned to the new element of the vector that is added

```
charVec.push_back('a');
```

resize() vs push_back()

- **`resize()`** is best used when you know the exact size a vector needs to be
  - like when you know the total number of possible destinations for HW6, for example

- **`push_back()`** is best used when elements are added one by one
  - like when you are reading in TrainCars from a file, and need to put them in the appropriate city row

size()

- unlike arrays in C, vectors in C++ “know” their size (due to C++ managing the memory of a vector for you)

- size() returns the number of elements in the vector it is called on

```
int cSize = charVec.size();
```

Outline

- Vectors
- Enumeration
- “Print” functions
- Function Overloading
- New/Delete
- Destructors

Enumeration

- type of variable used to set up collections of named integer constants

- useful for “lists” of values that are tedious to implement using **`#define`** or **`const`**

```
#define WINTER 0
#define SPRING 1
#define SUMMER 2
#define FALL   3
```

Enumeration Types

- two types of **`enum`** declarations

- named type

```
enum seasons {WINTER, SPRING,
              SUMMER, FALL};
```

- unnamed type

```
enum {WINTER, SPRING,
      SUMMER, FALL};
```

Enumeration Types

- named types allow you to create variables of that type, and use it in function args, etc.

```
enum seasons CurrentSemester;
currentSemester = SPRING;
```

- unnamed types are useful for naming constants, but when you don't intend to declare variables, etc.

Enumeration Benefits

- named enumeration types allow you to restrict valid values
  - a ‘seasons’ variable cannot have a value other than the four seasons in the enum declaration

- unnamed types allow simpler management of a large list of constants

Outline

- Vectors
- Enumeration
- “Print” functions
- Function Overloading
- New/Delete
- Destructors

“Print” functions

- function returns a string
  - call function within a **`cout`** statement

```
string PrintName (int studentNum);
```

- function performs its own printing
  - call function separately from a **`cout`** statement

```
void PrintName (int studentNum);
```

Outline

- Vectors
- Enumeration
- “Print” functions
- Function Overloading
- New/Delete
- Destructors

Function Overloading

- last class, covered overloading constructors

```
Date::Date (int m, int d, int y);
Date::Date (int m, int d);
Date::Date ();
```

- functions in C++ are uniquely identified by both their names and their parameters
  - **but NOT their return type!**
  - we can overload any kind of function

Overloading Example

```
void PrintMessage (void) {
   cout << "Hello World!" << endl;
}

void PrintMessage (string msg) {
   cout << msg << endl;
}
```

Overloading Details

- can use default values, like with constructors

```
void PrintMessage
        (string msg = "Hello World!") {
    cout << msg << endl;
}
```

- need to be careful about accidentally passing ambiguous arguments

Operator Overloading

- given variable types have predefined behavior for operators like **+**, **−**, **==**, etc.

- might be nice to have these operators work for user-defined variables, like Classes
  - often best to have them as member functions
  - allows access to private member data and functions

Overloading Restrictions

- cannot overload `::`, `.`, `*`, `or` `?` `and` `:`
- cannot create new operators
- overload-able operators include

  `=`, `>>`, `<<`, `++`, `--`, `+=`, `+`,
  `<`, `>`, `<=`, `>=`, `==`, `!=`, `[]`

Operator Overloading Example

- any arguments passed in must be **`const`**, and must be passed in by reference

```
Complex Complex::operator+
          (const Complex &num2)
{
  double r_real = real + num2.real;
  double r_imag = imag + num2.imag;
  return Complex(r_real, r_imag );
}
```

Outline

- Vectors
- Enumeration
- “Print” functions
- Function Overloading
- New/Delete
- Destructors

new and **delete**

- replace the **malloc()** and **free()** functions from C

```
Date *datePtr1, *datePtr2;
datePtr1 = new Date;
datePtr2 = new Date(7,4);

delete datePtr1;
delete datePtr2;
```

Managing Memory in C++

- just as important as managing memory in C

- keep track of what memory “you” have
- think carefully about
  - “losing” pointers
  - memory leaks
  - when memory should be deleted (freed)

Outline

- Vectors
- Enumeration
- “Print” functions
- Function Overloading
- New/Delete
- Destructors

Destructors

- opposite of constructors
- used when memory of a user-created Class type is deleted
- compiler automatically provides for you
  - does not take into account dynamic memory

Destructor Example

- let's say we have a new member variable of our **`Date`** class called '**`m_calendar`**' that is a dynamically allocated array of characters
  - dynamically allocated in our constructor

- we must create a destructor to handle this

```
Date::~Date() {
   delete m_calendar;
}
```

Homework 6

- Classy Trains
  - last homework!!!
- practice with implementing a C++ class
- more emphasis on:
  - error checking
  - code style and choices

Project

- proposal due April 2nd; project due the day of the presentation (April 24th at earliest)
  - **can't use late days for project deadlines**

- think about what you want to do
- think about who you want to work with
  - work must be done in pairs
  - post on Piazza to find teammates

- details will be up before next class