Programming C++ Lecture 2

Howest, Fall 2014

Instructor: Dr. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

Arrays and Pointers

```
void myPrint(const char *);

int main() {
    char *phrasey = “C++Fun”;
    myPrint(phrasey);
    return 0;
}
```

```
void myPrint(const char * s1) {
    while ((*s1) != ‘\0’) {
        cout << *s1;
        s1++;
    }
}
```

Arrays and Pointers

```
void copy1(char*, const char *);

int main() {
    char phrase1[10];
    char *phrase2 = “Hello”;
    copy1(phrase1, phrase2);
    cout << phrase1 <<  endl;
    return 0;

}
```

```
void copy1(char * s1,

              const char * s2) {
    for(int i =0; s2[i] != ‘\0’; i++) {
         s1[i] = s2[i];
    }

}
```

Arrays and Pointers

```
void copy2(char*, const char *);

int main() {
    char phrase3[10];
    char *phrase4 = “GBye”;
    copy2(phrase3, phrase4);
    cout << phrase3 <<  endl;
    return 0;

}
```

```
void copy2(char * s1,

             const char * s2) {
  for(; *s2 != ‘\0’;
        s1++, s2++) {
        *s1 = *s2;
  }

}
```

const with Pointers and Data

- 4 types
  - nonconstant pointer to nonconstant data
  - nonconstant pointer to constant data
  - constant pointer to nonconstant data
  - constant pointer to constant data

Nonconst Ptr, Nonconst Data

```
void convertToUpperCase(char *);

int main() {
    char phrase[] = “Hello world”;
    convertToUpperCase(phrase);
    cout << phrase << endl;
    return 0;

}
```

```
void convertToUpperCase(
                    char * sPtr) {
  while (*sPtr != ‘\0’) {
     if ((*sPtr) == ‘o’) {
        *sPtr = ‘O’;
     } sPtr++;
  }

}
```

Nonconst Ptr, Const Data

```
void printChars(const char *);

int main() {
    const char phrase[] = "Hello world";
    printChars(phrase);
    cout << endl;
    return 0;
}
```

```
void printChars (
        const char * sPtr) {
    for( ; *sPtr != '\0'; sPtr++) {
        cout << *sPtr;
    }
}
```

Const Ptr, Nonconst Data

```
void printChars(const char *);

int main() {
    const char phrase[] = "Hello world";
    printChars(phrase);
    cout << endl;
    return 0;

}
```

```
void printChars (
        char * const sPtr) {
  //can we change array elems?
  //can we do sPtr++?
  for( ; *sPtr != '\0'; sPtr++) {
        *sPtr = toupper(*sPtr);
        cout << *sPtr;
  }
}
```

Const Ptr, Nonconst Data

```
void printChars(const char *);

int main() {
    const char phrase[] = "Hello world";
    printChars(phrase);
    cout << endl;
    return 0;
}
```

```
void printChars (
        char * const sPtr) {
    //can we change array elems? YES
    //can we do sPtr++? NO
    for(int i =0; (sPtr[i]) != '\0'; i++) {
        sPtr[i] = toupper(sPtr[i]);
        cout << sPtr[i];
        //or *(sPtr + i)
    }
}
```

Const Ptr, Nonconst Data

```
int main() {
    int x, y;
    int * const ptr = &x;  //const pointer has to be initialized
    *ptr = 7;  //modifies x – no problem
    ptr = &y; //compiler ERROR – const ptr cannot be reassigned
    return 0;

}
```

- Arrays are constant pointers to nonconstant data

Const Ptr, Const Data

```
int main() {
    int x = 5, y;
    const int * const ptr = &x;  //const pointer has to be initialized
    cout << *ptr << endl;  //no problems – nothing modified
    *ptr = 7;  //compiler ERROR – const data cannot be changed
    ptr = &y; //compiler ERROR – const ptr cannot be reassigned
    x++; //is this ok?
    return 0;

}
```

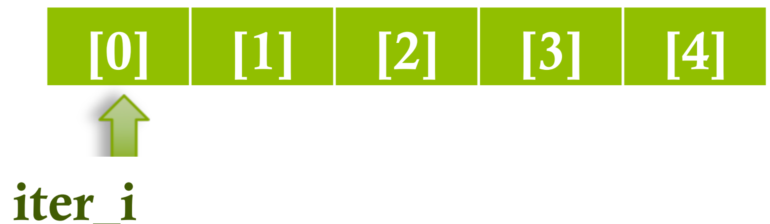
Vector

```
#include <vector>

using std::vector;

vector<int> integers1(5);  //already initialized to zero

cout << integers1.size() << endl;  //type is actually size_t

integers1[3] = 89;   //will NOT check bounds, but .at(3) will

vector<int> integers2(integers1);  //copies 1 into 2
```

Vector and Iterators


```
#include <vector>

using std::vector;

vector<int> integers1(5);  //already initialized to zero

vector<int>::iterator iter_i;  //pointer into vector

for(iter_i = integers1.begin(); iter_i != integers1.end(); iter_i++) {
    cout << (*iter_i) << “ “;

} cout << endl;
```

Vector and Iterators




```
#include <vector>

using std::vector;

vector<int> integers1(5);  //already initialized to zero

vector<int>::iterator iter_i;  //pointer into vector

for(iter_i = integers1.begin(); iter_i != integers1.end(); iter_i++) {
    cout << ++(*iter_i) << “ “; //you can use iterator to modify elements!

} cout << endl;
```

2D Vector and Iterators

```
#include <vector>

using std::vector;

vector< vector<int> > matrix(numRows, vector<int>(numCols));

for (vector< vector<int> >::const_iterator row =matrix.begin(); row != matrix.end(); row++) {
    for (vector<int>::const_iterator col = (*row).begin(); col != (*row).end(); col++) {
        //do something with *col
        cout << *col << " ";
    }
}
```

Dynamic Memory Management

- Like Java, puts things on the heap instead of the stack (so can be returned from functions!)
- Unlike Java, you manage memory yourself – no garbage collection
- Helps create dynamic structures, arrays of correct size
- Use **new** and **delete**
- **new** finds memory of correct size, returns pointer to it

Dynamic Allocation

```
double *pi = new double(3.14159);

int *num = new int();   *num = 9;

int *grades = new int[40];
```

- Finds space for 40 integers, returns address of first element to int pointer grades
- Memory allocated for array NOT initialized (unknown)
- Remember array name is a constant pointer to 0^{th} element of array

Dynamic Deallocation

```
double *pi = new double(3.14159);

int *num = new int();       *num = 9;

int *grades = new int[40];  //finds space for 40 integers, returns
    address of first element to int pointer grades

delete pi;

delete num;

delete [] grades;  //NEED [] when deleting an array!
```

Dynamic Deallocation

int *grades = **new** int[40]; //finds space for 40 integers, returns address of first element to int pointer grades

delete [] grades; //NEED [] when deleting an array!

grades = NULL; //good to null so no dangling pointers

- You MUST pair every **new** with a **delete**
- Not releasing dynamically allocated memory back to the heap can cause memory leaks. This is BAD.

Pointers to Pointers

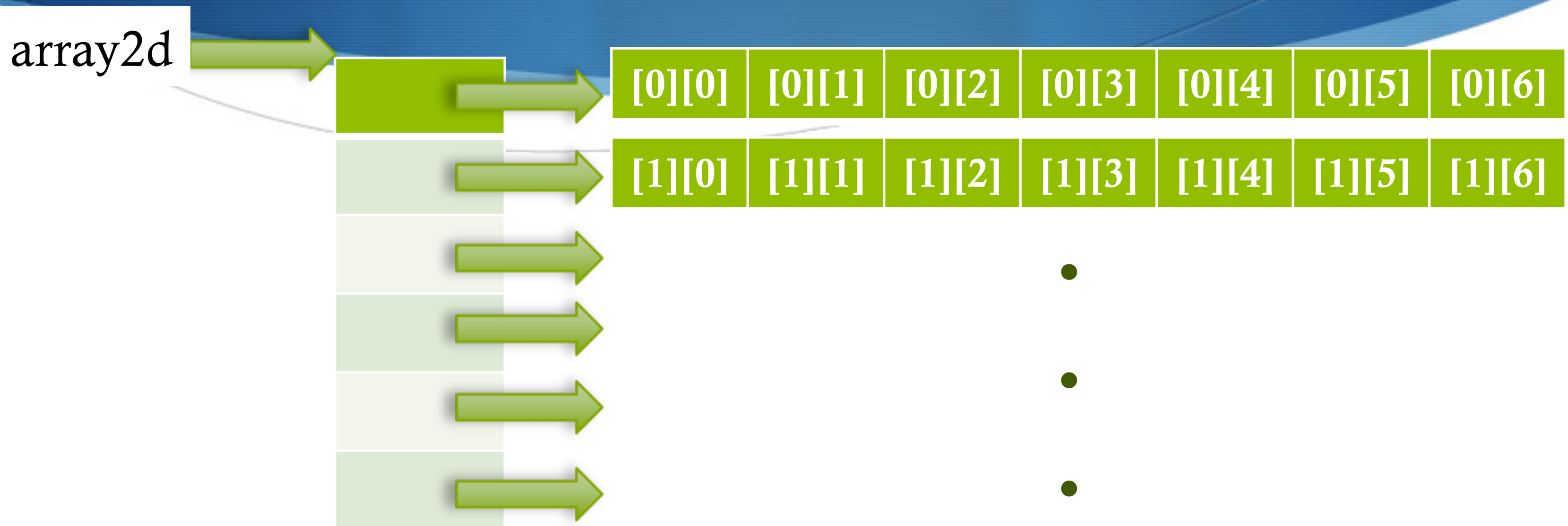
```
int **array2d = new int*[6];
```

- array2d is a pointer to a pointer, or a pointer to an array of pointers

```
for (int i = 0; i < 6; i++) {
    array2d[i] = new int[7];  //initialize each row to array of ints
} array2d[0][0] = 8;
```

- Dynamically allocated 2d arrays are NOT contiguous in memory

Pointers to Pointers




```
int **array2d = new int*[6];

for (int i = 0; i < 6; i++) {
    array2d[i] = new int[7];
} //Dynamically allocated 2d arrays NOT contiguous in
    memory (each new is contiguous)
```

Dealloc Pointers to Pointers

```
int **array2d = new int*[6];

for (int i = 0; i < 6; i++) {
    array2d[i] = new int[7];  //initialize each row to array of ints

}  array2d[0][0] = 8;

 for (int i = 0; i < 6; i++) {
    delete [] array2d[i];

}

delete [] array2d; //You MUST pair every new with a delete
```

Function Templates

- We can do function overloading

```
int boxVolume(int side) {
    return side * side * side;
}
double boxVolume(double side) {
    return side * side * side;
}
```

- Why define 2 functions that look identical, but have different types?
- Overloading that is more compact and convenient = function templates. Only write it once!

Function Templates

- Template

  template <class T> //or template <typename T>

  T boxVolume(T side) {

  return side * side * side;

  }

- C++ compiler automatically generates separate function template specializations for each type the function is called with.

- T is placeholder for actual data type

- int result = boxVolume(3); double result = boxVolume(6.2);

Inputting from a File

```
#include <fstream>
#include <stdio>
#include <stdlib>
using namespace std;
int main() {
     ifstream inputFile(“file.in”, ios::in);
     if (!inputFile) {
       cerr << "File could not be opened" << endl;
       exit(1);  //or return -1
    }
    int numPpl;
    inputFile >> numPpl;
    cout << "Num people is " << numPpl << endl;
    return 0;
}
```

Classes!

- Classes encapsulate objects
- Member variables
- Member functions
  - Getter and setter functions
  - Constructors and destructors
- Access specifiers
  - Public versus private

A Class

```
#include <iostream>
#include <string>
using namespace std;
class  Course {
public:
    void setCourseName(string name) {
        courseName = name;
    }
    string getCourseName() {
        return courseName;
    }
private:
    string courseName;
};
```

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

```
#include <iostream>
#include <string>
using namespace std;
class  Course {
public:
    void setCourseName(string name) {
        courseName = name;
    }
    string getCourseName() {
        return courseName;
    }
private:
    string courseName;
};
```

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << “Enter course name: “;
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

```
#include <iostream>
#include <string>
using namespace std;
class  Course {
public:
     void setCourseName(string name) {
          courseName = name;
     }
     string getCourseName() {
          return courseName;
     }
private:
     string courseName;
};
```

```
int main() {
     string nameOfCourse;
     Course myCourse;
     cout << myCourse.getCourseName();
     cout << endl;
     cout << “Enter course name: “;
     getline( cin, nameOfCourse );
     myCourse.setCourseName(
                         nameOfCourse);
     cout << myCourse.getCourseName();
     cout << endl;
     return 0;
}
```

A Class

```
#include <iostream>
#include <string>
using namespace std;
class  Course {
public:
    void setCourseName(string name) {
        courseName = name;
    }
    string getCourseName() {
        return courseName;
    }
private:
    string courseName;
};
```

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << “Enter course name: “;
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                   nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

```
#include <iostream>
#include <string>
using namespace std;
class  Course {
public:
    void setCourseName(string name) {
        courseName = name;
    }
    string getCourseName() {
        return courseName;
    }
private:
    string courseName;
};
```

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

Objects with Pointers, References

```
Course myCourse;  myCourse.getCourseName();

Course &courseRef = myCourse;  courseRef.getCourseName();

Course *coursePtr = &myCourse;  coursePtr->getCourseName();

Course* myCourse1 = new Course( );

myCourse1->getCourseName();
```

- Inside your class, don’t return a reference or pointer to private data members! BAD style.

A Class with a Constructor

```
class  Course {
public:
     Course( string name ) {
          setCourseName(name);
     }
     void setCourseName(string name) {
          courseName = name;
     }
     string getCourseName() {
          return courseName;
     }
private:
     string courseName;
};
```

```
int main() {
     Course myCourse1("C++
          Programming" );
     string nameOfCourse;
     cout << "Enter course name: ";
     getline( cin, nameOfCourse );
     Course myCourse2(nameOfCourse);
     cout << myCourse1.getCourseName();
     cout << endl;
     cout << myCourse2.getCourseName();
     cout << endl;
     return 0;
}
```

Constructors

- Special class methods with class name
- Cannot return anything
- Initialize state of object when created
- Usually public
- Called implicitly for every object creation
- Default constructor: no parameters, automatically created by compiler if no constructor
  - Calls default constructor of object data members of class

Destructors

- Similar to constructor: tilde followed by class name (~Course( ) { … })
- Receives no parameters, cannot return value.
- Only 1 destructor and must be public.
- Called implicitly when object destroyed (goes out of scope)
  - Does NOT release object's memory, but performs housekeeping.
- Compiler implicitly creates empty one if none exists.

Destructors

- To see order in which constructors/destructors called, see

http://users.elis.ugent.be/~jsartor/howest/constructorDestructor.cpp

- When in particular are they useful?
  - When you need to deallocate memory (call delete) because you called new in your class (probably for a pointer member variable)

Interface vs. Implementation

- Interface defines and standardizes way to interact – says what services are available and how to request them.
- Implementation – how services are carried out.
- Separate them: interface = *.h, implementation = *.cpp
- *.h includes function prototypes and data members
- *.cpp defines member functions (use :: binary scope resolution operator to tie functions to class definition)

A Class

```
//Course.h
#include <string>
using namespace std;
class  Course {
public:
     Course( string name );
     void setCourseName(
             string name);
     string getCourseName();
private:
     string courseName;
};
```

```
//Course.cpp
#include "Course.h"

Course::Course( string name ) {
   setCourseName(name);
}
void Course::setCourseName(string name) {
   courseName = name;
}
string Course::getCourseName() {
   return courseName;
}
```

Test Program

```
//test program can be in another file – testCourse.cpp
#include <iostream>
using namespace std;
#include "Course.h"
int main() {
    Course myCourse1( "CS105: Programming in C++" );
    string nameOfCourse;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    Course myCourse2(nameOfCourse);
    cout << myCourse1.getCourseName();
    cout << endl;
    cout << myCourse2.getCourseName();
    cout << endl;
    return 0;
}
```

Preprocessor Wrapper

```
//Course.h
#include <string>
using namespace std;
#ifndef COURSE_H
#define COURSE_H
class  Course {
public:
     Course( string name );
     void setCourseName(
              string name);
     string getCourseName();
private:
     string courseName;
};
#endif
```

- Prevent header code from being included into same source code file more than once
- Use uppercase, usually file name with "." replaced by "_"

Abstraction and Encapsulation

- Abstraction = creation of a well-defined interface for object
- Encapsulation = keep implementation details private
  - Data members and helper functions private
- Promotes software reusability
- Can change class data representation and/or implementation without changing code that uses class
- Good software engineering

Compilation and Linking

- Compiler uses included interface .h files to compile .cpp file into object code
  - Course.h + testCourse.cpp -> testCourse.o
  - Course.h + Course.cpp -> Course.o
- Linker takes object code of testCourse.cpp and Course.cpp and STL and puts it together into an executable.
  - testCourse.o + Course.o + stl.o -> testC.exe