Organizing Projects in C++

3081 Program Design and Development

Organizing Projects in C++

Part of the Software Development Process

- Organizing Code
- .h Header Files and .cpp Source Files
- Compiling and Linking
- Declaration versus Definition
- Makefile
- Namespaces

Organizing Code for Team Development

MODULARIZE

Manage A LOT of code with good organization of both classes and files.

REUSE

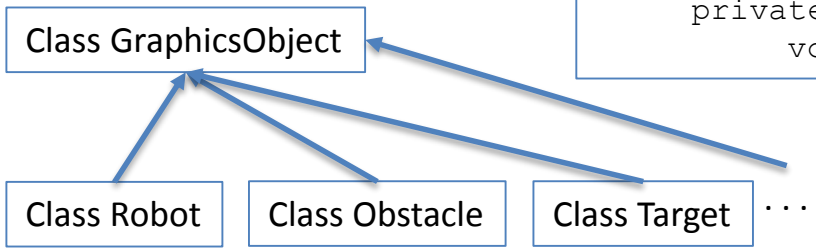
Don’t reinvent the wheel – share within and across projects.

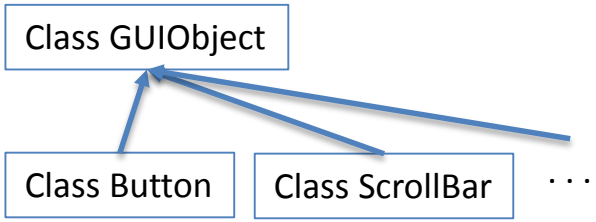
INTEGRATE

Modularized code needs an easy way to come together.

Doing these well and easily, helps you to modify code.

```
class Robot:public GraphicsObject {
     public: …
     private:
          void moveUp() …
```

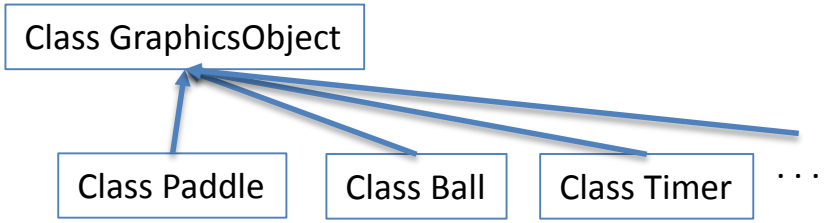


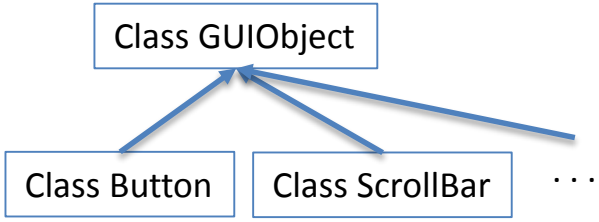


Robot Simulator

```
// Robot Simulator



Void main() {
   Robot robot;
   UI menu;
   …
}

UI setUpUI() {
   Button up;
   Button down;
   ScrollBar barHorizontal;
   …
}
```

Pong






```
// The Game of Pong


Void main() {
   Paddle player1
   Paddle player2
   Ball ball
   UI menu
   …
}

UI setUpUI() {
   Button speedUp;
   Button speedDown;
   ScrollBar barHorizontal;
      …
}
```

Declaration VS Definition Examples

- Declaration:

```
int myfunc(float p1, int p2);
```

- Definition:

```
int myfunc(float p1, int p2) {
    return (int) ( p1/(float)p2 );
}
```

- Declaration and Definition:

```
float myvar;
```

- Declaration:

```
extern float myvar;
```

Declaration VS Definition

Declaration : A statement that tells the compiler this function, variable or class* looks like "this" and is defined somewhere. *Eckel*

- It can be defined in any target or library to which the file will be linked.

Definition : A statement that tells the compiler to allocate memory for the variable, function, or class* and to store it in that memory. *Eckel*

- RULE: You must declare a thing before you reference it or call it (but you need not define it).
- RULE: You cannot define anything more than once (but you can declare it more than once).

* *class* is a little tricky because it contains both variables and functions with a mix of *what looks like* declarations and definitions. However, you want to separate the two conceptually, so that you know which parts to put in .h and .cpp.

```
/*************************
 * This defines a class DataClass that is a database of data.
 * Each entry has 3 elements.
 * The first is degrees, entered by the user.
 * The second is the equivalent in radians.
 * The last is the sin.
 *************************/

#include <math.h>
#include <iostream>

class DataClass {
public:
    DataClass();
    int add(int degree);
    void print();
private:
    int dataCount;
    struct dataStruct {
        int degree;
        float radian;
        float sin;
    };
    dataStruct theData[25];
};
```

DECLARATION
(or *public interface*)
in DataClass.h

```cpp
/*************************
 * This defines a class DataClass that is a database of data.
 * Each entry has 3 elements.
 * The first is degrees, entered by the user.
 * The second is the equivalent in radians.
 * The last is the sin.
 *************************/

#include <math.h>
#include <iostream>

class DataClass {
public:
    DataClass();
    int add(int degree);
    void print();
private:
    int dataCount;
    struct dataStruct {
        int degree;
        float radian;
        float sin;
    };
    dataStruct theData[25];
};
```

DECLARATION
(or *public interface*)
in DataClass.h

```cpp
#include "DataClass.h"

using namespace std;

DataClass::DataClass() {
    dataCount = 0;
}

int DataClass::add(int degree) {
    if (dataCount >= 25)
        return 0;
    int i = dataCount;
    theData[i].degree = degree;
    theData[i].radian = (float) degree * (3.14 / 180.0);
    theData[i].sin = sin(theData[i].radian);
    ++dataCount;
    return dataCount;
}

void DataClass::print() {
    for (int i=0;i<dataCount; i++) {
        cout << theData[i].degree << " " << theData[i].radian << " " << theData[i].sin << endl;
    }
}
```

DEFINITION
(or *private implementation*)
in DataClass.cpp

```cpp
/*****************************
 * This program creates a database of data.
 * It uses the DataClass defined in DataClass.cpp.
 *****************************/

#include "DataClass.h"

int main()
{
    DataClass myData;

    myData.add(90);
    myData.add(270);

    myData.print();

    return 1;
}
```

```cpp
/*************************
 * This defines a class DataClass that is a database of data.
 * Each entry has 3 elements.
 * The first is degrees, entered by the user.
 * The second is the equivalent in radians.
 * The last is the sin.
 *************************/

#include <math.h>
#include <iostream>

class DataClass {
public:
    DataClass();
    int add(int degree);
    void print();
private:
    int dataCount;
    struct dataStruct {
        int degree;
        float radian;
        float sin;
    };
    dataStruct theData[25];
};
```

DECLARATION
(or *public interface*)
in DataClass.h

```cpp
#include "DataClass.h"

using namespace std;

DataClass::DataClass() {
    dataCount = 0;
}

int DataClass::add(int degree) {
    if (dataCount >= 25)
        return 0;
    int i = dataCount;
    theData[i].degree = degree;
    theData[i].radian = (float) degree * (3.14 / 180.0);
    theData[i].sin = sin(theData[i].radian);
    ++dataCount;
    return dataCount;
}

void DataClass::print() {
    for (int i=0;i<dataCount; i++) {
        cout << theData[i].degree << " " << theData[i].radian << " " << theData[i].sin << endl;
    }
}
```

DEFINITION
(or *private implementation*)
in DataClass.cpp

Putting It All Together

To create a main() or user-defined object in separate file:

1. Identify all dependencies.
2. #include C/C++ libraries and headers using <>
3. #include user-defined header files using " ".
4. Put public interface in .h header file.
5. Put private implementation in .cpp source file.
6. Compile each .cpp file (not link) to create an object file.
7. Link main.o with other object files to create executable.

Makefiles

make is a utility that uses a *script* stored in the *makefile* to automatically compile and link project files.

- Resources :
  - http://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html
  - Eckart, Ch. 3 Section **Make: managing separate compilation**

- Why Useful?
  - Shortcuts : less typing.
  - Efficient Compiliation : only compiles what has changed.
  - Good Management : shows dependencies in one place.
  - Automatic : it “knows” how to compile and link.

Makefile Components

- A “Target” Entry

  <target>: [<dependency >]*

  [<TAB> <command> <endl>]+

```
file1.o : file1.cpp file1.h file2.h
    g++ -c file1.cpp -o file1.o
```

```
file1.exe : file1.o file2.o file3.o file4.o
    g++ file1.o file2.o file3.o file4.o -o file1.exe
```

```
file1.exe : file2.o file3.o file4.o
    g++ -c file1.cpp -o file1.o
    g++ file1.o file2.o file3.o file4.o -o file1.exe
```

- Macro : string replacement, just like #define.

  OBJS = file1.o file2.o

  $(OBJS)

```
OBJS = file1.o file2.o file3.o file4.o

file1.exe : $(OBJS)
    g++ $(OBJS) -o file1.exe
```

- Common Macros:
  - OBJS : the target object files
  - CC : the compiler (e.g. g++ or c)
  - DEBUG : -g (compile to use gdb debugger)
  - CFLAGS : compiler flags (e.g. -Wall –c)
  - LFLAGS : linker flags (e.g. –Wall)

```
OBJS = file1.o file2.o file3.o file4.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

all: $(OBJS)
    $(CC) $(LFLAGS) $(OBJS) -o link_example
```

Makefile Dummy Targets

Dummy Targets : a way to “call” a certain command within makefile.

```
OBJS = main.o DataClass.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)
```

- all:

```
all: $(OBJS)
    $(CC) $(LFLAGS) $(OBJS) -o main.exe
```

- clean:

```
clean:
    \rm *.o main.exe
```

- tar:

```
SOURCES = main.cpp DataClass.cpp
HEADERS = DataClass.h

tar:
    tar cfv example.tar makefile $(SOURCES) $(HEADERS) $(OBJS)
```