Introduction to C, C++, and Unix/Linux

CS 60

Wednesday June 1, 2005

Today

→ C and C++ together

→ Exception handling

→ Reading [KR] Chapters 1-7

→ Read [So] chapters 1, 3, 4 (Boolean), 9, 13, 14 & 18.

Notes

Questions?

Combining C++ and C

- We often need to use C libraries (or object files) in C++ programs
- But C functions are defined differently, and they do not link with C++ programs
- But we can fix that...

func.c

```
int func(int x, int y)
{
  return(x+y);
}
```

main.cpp

```
int func(int, int);

int main()
{
  int z = func(2, 3);
  return z;
}
```

```
$ gcc –c func.c
$ g++ -o main main.cpp func.o
/tmp/ccAUxg9t.o(.text+0x1b): In function `main':
: undefined reference to `func(int, int)'
collect2: ld returned 1 exit status
```

func.c

```
int func(int x, int y)
{
  return(x+y);
}
```

main.cpp

```
extern “C” {
int func(int, int);
}

int main()
{
  int z = func(2, 3);
  return z;
}
```

```
$ gcc –c func.c
$ g++ -o main main.cpp func.o
$ main
$ echo $status
5
```

func.c

```
int func(int x, int y)
{
  return(x+y);
}
```

func.h

```
#if defined(__cplusplus)
extern "C" {
#endif
int func(int, int);
#if defined(__cplusplus)
}
#endif
```

main.cpp

```
#include "func.h"

int main()
{
  int z = func(2, 3);
  return z;
}
```

Now func.h can be included in both C and C++ source files, and func.o (or libfunc.a) can be used with both C and C++ programs

Try it!

Exception handling

- Exceptions are emergency procedures – run-time program anomalies
  - Division by zero, arithmetic or array overflow, exhaustion of free memory, illegal parameter, etc.
- What to do when such an anomaly occurs?
  - Ignore, segmentation fault, program abort, program exit
- Exception handling provides a standard way of defining and responding to such anomalies

Assert

`-DNDEBUG` flag ignores asserts
The whole **`assert(expr)`** is ignored!

- The **`assert`** function checks to see if a condition is true.
  If it is not, the program is aborted with an error message, including the file name and line number of the assert
- This is useful to the programmer, but not to the end user

```
#include <cassert>  // or <assert.h>

int MyArray::store(int index, int value)
{
    assert(index >= 0);
    assert(index < MAX_I_ARRAY_SZ);
    m_iarray[index] = value;
}
```

Error handling

- if/else/else constructs distract from the core functionality of the program and cause spaghetti code (intermixing of the algorithm and the error handling)
- Asserts are drastic – they abort the program
- Exceptions allow us to continue and explicitly handle the error

```
int *arr = new int[1000];
if (arr == 0) {
  cerr << “No space\n”;
  exit(1);
}
```

```
int *arr = new int[1000];
assert(arr);
```



```
prog: prog.cpp:38: int main(int, char**):
   Assertion ‘arr’ failed.
Abort (core dumped)
```

```
int *arr, len=1000;
try {
  int *arr = new int[len];
  if (arr == 0) throw arr;
}
catch (int* str) {
  cerr << “Smaller...\n”;
  len = 500;
  arr = new int[len];
}
```



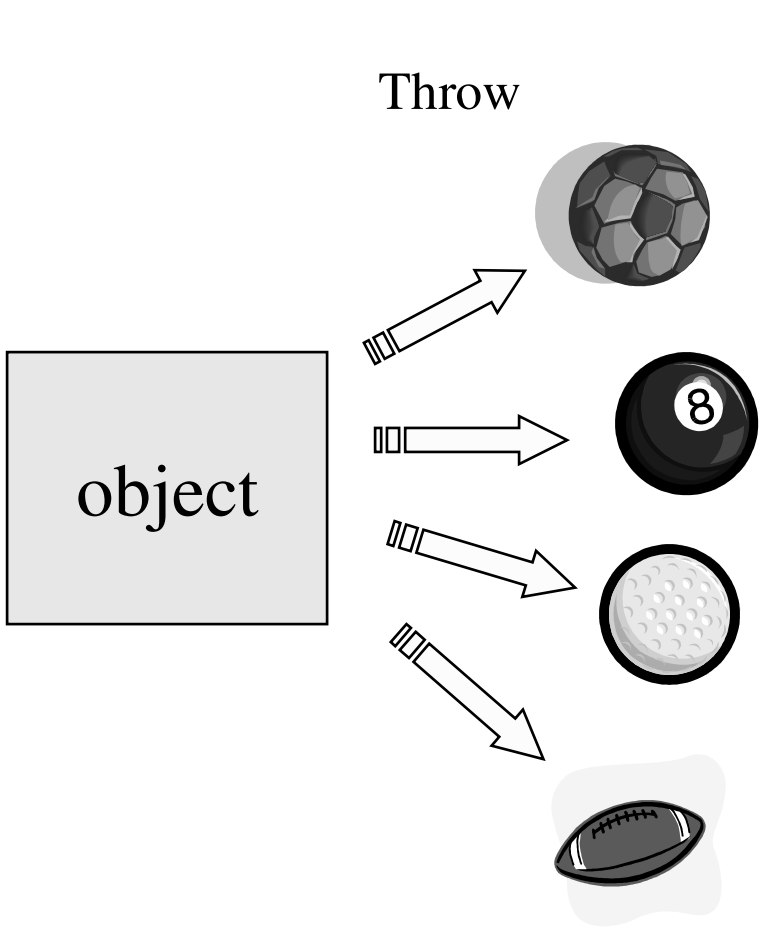
```
Smaller...
```

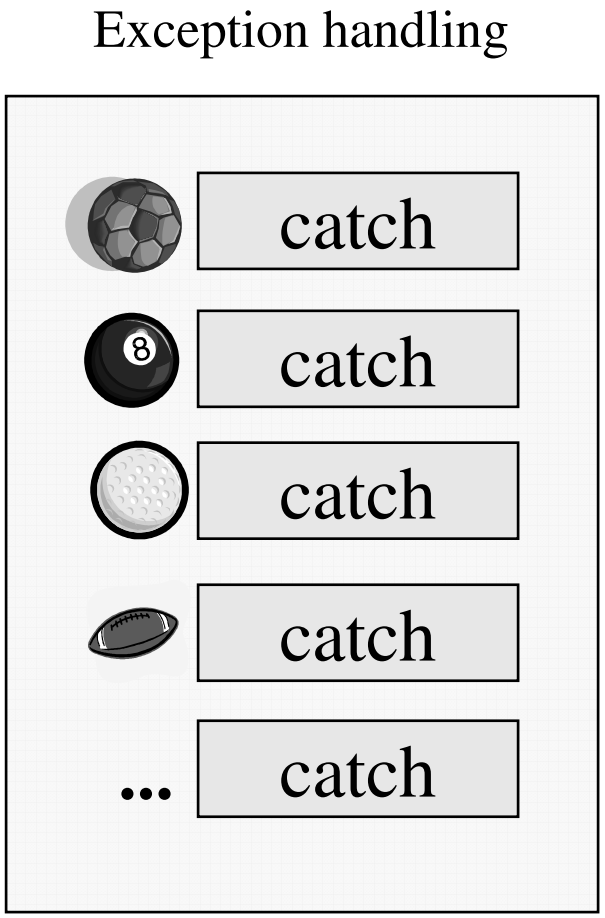
- Typically the exceptions would be thrown at a deeper level (in some class functions)
- “throw / catch” allows for the error handling to be defined separate from the main code
  - Your class (or library) doesn’t have to handle every exception – just throw exceptions for the calling program to handle
  - For example stack empty or stack full

assert vs. exceptions

- **`assert`** is good for checking conditions that should never happen
- **`assert`** is very handy during program development and debugging
- Exceptions are good for handling conditions that are rare but possible, and don't necessarily require program termination

Exception handling





Exceptions: `try`, `throw`, and `catch`

1. A description of a possible problem – what type of exceptions will we handle?
2. A section of code in which the exception may occur, enclosed in a `try` block
3. Something that causes an exception and triggers the emergency procedures through a `throw` statement
4. Exception handling code inside a `catch` block

Problem description

- Define objects that can describe the problems

```
class fire_emergency {
    public:
        // Which engine is on fire
        int engine;

        // Other information
        // about the fire
};
```

Where problems may occur

- Use the **`try`** statement to define a section of code in which an exception may occur

```
try {
  fly_from_point_a_to_point_b();
}
```

Uses class objects that may throw exceptions

Triggering an exception

- Something that causes an exception and triggers the emergency procedures through a **`throw`** statement.

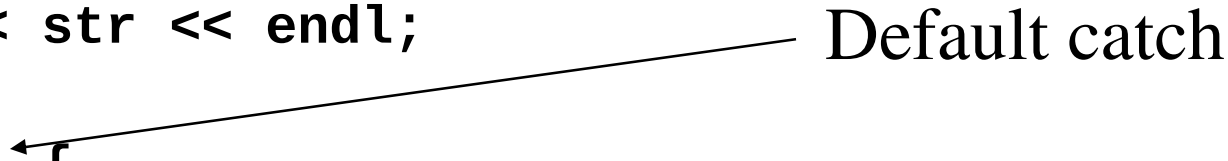
```
// Watch for fire in engine #2
void Engine::Sensor_2(void) {
    while (engine_running()) {
        if (engine_on_fire()) {
            fire_emergency fire_info;
            fire_info.engine = 2;
            throw(fire_info);
        }
    }
}
```

Handling the exception

- Catch (handle) the exception based on its type, via a **`catch`** block

```
try {
     fly_from_point_a_to_point_b();
}
catch (fire_emergency &fire_info) {
    active_extinguisher(fire_info.engine);
    turn_off(fire_info.engine);
    land_at_next_airport();
}
```

```
try {
     fly_from_point_a_to_point_b();
}
catch (fire_emergency& fire_info) {
    //...
    land_at_next_airport();
}
catch (food_emergency& food_info) {
    replenish(food_info);
}
catch (string& str) {
    cout << str << endl;
}
catch (...) {
    cout << “Something’s wrong! << endl;
}
```



Throw – Catch

- Throws and Catches are matched up by comparing the type of the object thrown with the types of the catch handlers
- A match is made if any of these holds:
  - Both types are exactly the same
  - The catch handler type is a public base class from which the thrown object is derived
  - The catch handler type is a pointer, and the object thrown can be converted to that pointer type by a standard pointer conversion

- Only one **`catch`** is executed, and order matters
- The program continues after the **`catch`** blocks
- **`catch(...)`** catches anything – usually listed last as the default
- A **`throw`** with no **`catch`** causes the program to abort
- Throwing an exception during a catch block causes the program to abort

Class destructors

- It is fine to throw exceptions in a class constructor
- Don’t throw an exception in a class destructor!
- Reason:
  - An exception is not handled in the current procedure, so first that procedure is exited – and its variables are destroyed
  - If the destructor for one of these variables causes a second exception to be thrown, the program will abort (without doing the exception handling you intended).

One use:

```
float z;
try {
  int x, y;
  cin >> x;
  cin >> y;
  if (y == 0) throw “DIV 0”;
  z = x/y;
}
catch (string str) {
  cout << “Error: ” << str;
}
```

More typical use:

```
MyApp app;
try {
  app.Setup();
  app.Run();
}
catch (DivZero dz) {
...
}
catch (InsufMemory mem) {
...
}
catch (string str) {
  cout << “Error: ”;
  cout << str << endl;
}
catch (...) {
  cout << “Random error\n”;
}
```

Exception class

- Often there will be a special class defined just for exceptions (e.g., **`CException`**)

```
throw CException("Error #3", "engine",
                 true, true, 42);

class CException {
  string Message;
  string Module;
  bool Urgent;
  int AnswerToLife;
public:
  CException(string Message, ...);
};
```

Bottom line...

- Main advantages of C++ exception handling:
  - Separates error handling code from normal program code
  - Encourages uniform and thorough error handling
- The exception handling mechanism is particularly useful in large projects, where clarity of code is vital and thorough exception handling is important
- **`assert(expr)`** is still quite useful, though!