CMSC 341
Lecture 2

Announcements

Tutors wanted

Office hrs: M 12:15-1:30, W 10-11

Project 1

Templates

Provide “parametric polymorphism”

Polymorphism: ability of variable/function/class to take more than one form

- ad-hoc polymorphism: function overloading
- true polymorphism: virtual member functions; dynamic binding
- parametric polymorphism:
  - controlled through a parameter
  - works at compile time

Two types

- function templates
- class templates

Why Function Templates?

Suppose you want to write a function of two args of same type which returns the largest.

Could write each:

```
int maxInt (int a, int b);
int maxFloat (float a, float b);
    …
const Foo &maxFoo(const Foo &a, const Foo &b);
```

With each code:

```
int maxInt (int a, int b) {
    if (a < b) return b;
    else return a;
    }
```

Why Function Templates? (cont)

Using templates, write one function to handle all:

```
template <class T>
T max (T a, T b) {
    if ( a < b) return b;
    else return a;
    }
```

For instance:

```
template <class Comparable>
Comparable max(Comparable a, Comparable b) {
    if (a < b) return b;
    else return a;
    }
```

Compiling Templates

When compiler sees calls such as:

```
int i1 = 1, i2 = 2;
float f1 = 1.1, f2 = 2.2;
const Foo &foo1 = initval1, &foo2 = initval2;
max (i1, i2);
max (f1, f2);
max (foo1, foo2);
```

Compiler creates different functions for each type of args that are used

```
int maxInt(int a, int b)
float maxFloat(float a, float b)
const Foo maxFoo(const Foo &a, const Foo&b)
```

What happens?

Suppose you have:

```
template <class Comparable>
  Comparable max(Comparable a, Comparable b) {
      if (a < b) return b;
      else return a;
      }
```

And you write:

```
  char *str1 = “abc”;
  char *str2 = “def”;
  max(str1, str2);
```

What gets compared?

What does happen.

Addresses compared, not contents.

Can work around by overloading max function:

```
  char* max(char *a, char *b) {
      if (strcmp(a,b) < 0) return b;
      else return a;
      }
```

MemCell.h

```
#ifndef _MEMCELL_H_
#define _MEMCELL_H_
template <class Object>
class MemCell {
public:
  explicit MemCell(const Object &initVal =
  Object());
  ~MemCell();
  MemCell(const MemCell &mc);
  const MemCell &operator=(const MemCell &rhs);
  const Object &read() const; // accessor
  void write (const Object &x);     // mutator
private:
  Object _storedVal;};
#endif
```

MemCell.C

```
#include “MemCell.H”

template <class Object>
MemCell<Object>::MemCell(const Object &initVal) :
  _storedVal(initVal) {}

template <class Object>
MemCell<Object>::MemCell(const MemCell &mc)
  {write(mc.read()); }

template <class Object>
MemCell <Object>::~MemCell() {}
```

MemCell.C (cont)

```
template <class Object>
const MemCell<Object> &MemCell<Object>
            ::operator=(const MemCell<Object>
  &rhs)
  { if (this != &rhs) write(rhs.read());
    return *this; }

template <class Object>
const Object &MemCell<Object>::read() const
  { return _storedVal; }

template <class Object>
void MemCell<Object>::write(const Object &x)
  { _storedVal = x; }
```

TestMemCell.C

```
#include “MemCell.H”
#include <iostream.h>  // for cout
#include “mystring.h”
#include <stdlib.h>  // for EXIT_SUCCESS
int main() {
  MemCell<int> m1;
  MemCell<string> m2(“hello”);

  m1.write(37);
  string str = m2.read();
  str += “ world”;
  m2.write(str);
  cout << m1.read() << endl;
  cout << m2.read() << endl;
  return (EXIT_SUCCESS);
  }
```

Compiling Templates (revisited)

Each instantiated template object is considered to be a different type by C++

Array<int> != Array<float>

Suppose:

```
#include “Array.H”
#include “Foo.H”
int main() {
    Array<int> ai;
    Foo<float> ff;
    …
}
```