CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 20
November 14, 2016
Templates

Casts and Conversions
Templates
Template overview

Templates are instructions for generating code.
Are type-safe replacement for C macros.
Can be applied to functions or classes.
Allow for type variability.

Example:
```cpp
template <class T>
class FlexArray { ... };
```
Later, can instantiate
```cpp
class RandString : FlexArray<const char*> { ... };
```
and use
```cpp
FlexArray<const char*>::put(store.put(s, len));
```
Template functions

Definition:

```cpp
template <class X> void swapargs(X& a, X& b) {
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

Use:

```cpp
int i,j;
double x,y;
char a, b;
swapargs(i,j);
swapargs(x,y);
swapargs(a,b);
```
Specialization

Definition:

```cpp
template <> void swapargs(int& a, int& b) {
   // different code
}
```

This overrides the template body for `int` arguments.
Template classes

Like functions, classes can be made into templates.

```cpp
template <class T>
class FlexArray {
    ...  
};
```

makes `FlexArray` into a template class.

When instantiated, it can be used just like any other class.

For a flex array of ints, the name is `FlexArray<int>`.  

No implicit instantiation, unlike functions.
Compilation issues

Remote (non-inline) template functions must be compiled and linked for each instantiation.

Two possible solutions:

1. Put all template function definitions in the .hpp file along with the class definition.

2. Put template function definitions in a .cpp file as usual but explicitly instantiate.
   
   E.g., \texttt{template class FlexArray<int>;} forces compilation of the \texttt{int} instantiation of \texttt{FlexArray}. 

CPSC 427, Lecture 20

8/19
Template parameters

Templates can have multiple parameters.

Example:
\texttt{template<class T, int size>} declares a template with two parameters, a type parameter \texttt{T} and an int parameter \texttt{size}.

Template parameters can also have default values.
Used when parameter is omitted.

Example:
\texttt{template<class T=int, int size=100> class A \{ ... \}.}

\texttt{A<double>} instantiates \texttt{A} to type \texttt{A<double, 100>}. \texttt{A<50>} instantiates \texttt{A} to type \texttt{A<int, 50>}. 
Templatizing a class

Demo 20a-BarGraph results from templatizing Row and Cell classes in 08-BarGraph.
Template parameter $T$ replaces uses of Item within Row.

Here is what was necessary to carry this out:

1. Fold the code from row.cpp into row.hpp.
2. Precede each class and function declaration (outside of class) with template<class T>.
3. Follow occurrences of Row with template argument <Item> in Graph.hpp and Graph.cpp.
4. Follow each use of Row with template argument <T> in row.hpp.
Using template classes

Demo 20b-Evaluate is a simple expression evaluator based on a precedence parser.

It uses templates and derivation together by deriving a template class Stack<T> from the template class FlexArray<T>, which is a simplified version of vector<T>.

The precedence parser makes uses of two instantiations of Stack<T>:

1. Stack<double> Ands;
2. Stack<Operator> Ators;
Casts and Conversions
Casts in C

A C cast changes an expression of one type into another.

Examples:

```c
int x;
unsigned u;
double d;
int* p;

(double)x; // type double; preserves semantics
(int)u; // type unsigned; possible loss of information
(unsigned)d; // type unsigned; big loss of information
(long int)p; // type long int; violates semantics
(double*)p; // preserves pointerness but violates semantics
```
Different kinds of casts

C uses the same syntax for different kinds of casts.

**Value casts** convert from one representation to another, partially preserving semantics. Often called *conversions*.

- `(double)x` converts integer `x` to equivalent `double` floating point representation.
- `(short int)x` converts integer `x` to equivalent `short int`, *if the integer falls within the range of a short int*.

**Pointer casts** leave representation alone but change interpretation of pointer.

- `(double*)p` treats bits at destination of `p` as the representation of a double.
C++ casts

C++ has four kinds of casts.

1. *Static cast* includes value casts of C. Tries to preserve semantics, but not always safe. Applied at compile time.

2. *Dynamic cast*. Applies only to pointers and references to objects. Preserves semantics. Applied at run time. [See demo 20c-Dynamic_cast.]


Explicit cast syntax

C++ supports three syntax patterns for explicit casts.

1. C-style: `(double)x`.

2. Functional notation: `double(x); myObject(10)`.
   (Note the similarity to a constructor call.)
   Only works for single-word type names.

3. Cast notation:
   ```
   int x; myBase* b; const int c;
   ▶ static_cast<double>(x);
   ▶ dynamic_cast<myDerived*>(b);
   ▶ reinterpret_cast<int*>(p);
   ▶ const_cast<int>(c);
   ```
Implicit casts

General rule for implicit casts: If a type \( A \) expression appears in a context where a type \( B \) expression is needed, use a semantically safe cast to convert from \( A \) to \( B \).

Examples:

- **Assignment:** `int x; double d; x=d; d=x;`
- **Pointer assignment:**
  ```cpp
  class A {
  ...
  };;
  class B : public A {
  ...
  };;
  A* ap; B* bp; ap = bp;
  ```
- **Initialization:**
  ```cpp
  A a=x; converts x to an A, then copies.
  ```
- **Construction:**
  ```cpp
  A a(x); calls A constructor, possibly casting x.
  ```
Ambiguity

Can be more than one way to cast from $B$ to $A$.

class B;
class A { public:
    A(){
    A(B& b) { cout<< "constructed A from B\n"; }
};
class B { public:
    A a;
    operator A() { cout<<"casting B to A\n"; return a; }
};
int main() {
    A a; B b;
    a=b;       // Triggers error comments
}

Comment from $g++$: conversion from 'B' to 'A' is ambiguous
Comment from $clang++$: error: reference initialization of type 'A &&' with initializer of type 'B' is ambiguous
**explicit keyword**

Not always desirable for constructor to be called implicitly.

Use `explicit` keyword to inhibit implicit calls.

Previous example compiles fine with use of `explicit`:
```cpp
class B;
class A {
public
  A(){}
  explicit A(B& b) { cout<< "constructed A from B\n"; } 
};
...
```

Question: Why was an explicit definition of the default constructor not needed?