TEMPLATES

Today

- We will start to look at the fourth kind of polymorphism — *parametric polymorphism* — that we came across in lecture V.4.
- In C++ this is embodied in the form of *templates*
- Good references are:
  - `http://www.learncpp.com/cpp-tutorial`
  - `http://www.cplusplus.com/doc/tutorial`
  - `http://www.cppreference.com/index.html`

  and you can also consult Chapters 6 amd 7 in the textbook, but the textbook does not cover this stuff in as much detail as the online sources.

Function templates

- Imagine you want to write a function `mmax` that will return the bigger of two numbers.
- (We call it `mmax` to avoid a nameclash with a standard library function.)
- Trivial:

```
int mmax(int x, int y){
  if(x > y){
    return x;
  }
  else {
    return y;
  }
}
```

- But what if we also want a function that will do the same for two `doubles`?
- Well, obviously we could write:

```
double mmax(double x, double y){
  if(x > y){
    return x;
  }
  else {
    return y;
  }
}
```

- but wouldn't it be nicer to write a function that would take both `doubles` and `ints`
- It turns out that templates allow us to do *exactly* this.

- You can define the (initially odd) looking:

```
template<typename T>
T mmax(T x, T y){
  if(x > y){
    return x;
  }
  else {
    return y;
  }
}
```

- This has two new things.

- One is the first line:

  ```
  template<typename T>
  ```

- This can also be written:

  ```
  template<class T>
  ```

- Either way, it tells the compiler that the thing that follows will be a template, and that when the template is used, the `T` will be replaced with some type that the compiler understands.
- The second new thing is the use of the `T` in place of a type in the definition of the return type and the parameters.
- Again this is a kind of placeholder for a type that the compiler will be told about later.

- Now we can use the template.
- Here are three calls with different parameter types:

```
int    iValue = mmax(3, 7);
double dValue = mmax(2.3, 6.5);
char   cValue = mmax('a', '6');
```

- We can write templates with more than one parametric parameter type:

```
template<typename T, typename U>
bool gThan(T x, U y){
  if(x > y){
    return true;
  }
  else {
    return false;
  }
}
```

- Here we have to name two types since we use two.

- Note also that this example, like `mmax` relies on the overloading of the > operator.
- You will find that templates and operator overloading tend to go together.

How the compiler handles this

- You don't need to know this to use templates, but this is what is going on behind the scenes.
- Given the template version of `mmax`, when the compiler sees:

```
int     iValue = mmax(3, 7);
```

it builds and then compiles:

```
int mmax(int x, int y){
  if(x > y){
    return x;
  }
  else {
    return y;
  }
}
```

- And something analgous happens when it sees:

```
double dValue = mmax(2.3, 6.5);
```

- So you end up with the same code as if you had written lots of functions with different parameter types, but the compiler does most of the work.

- Here's another example:

```
template<class T>
T average(T *tArray, int number){
  T tSum = 0;
  for(int i = 0; i < number; i++){
    tSum += tArray[i];
  }

  tSum /= number;
  return tSum;
}
```

- Note that here we have a parameter of type `int` as well as one of type `T`.
- You can find this example in `templates1.cpp` on the class website.

Template classes

- We can also define classes where the types of some of the data members and function members are parametric.

```
template<class T>
class aPair{
private:
  T values[2];

public:
  aPair(T first, T second){
    values[0] = first;
    values[1] = second;
  }
};
```

- Just as for template functions, we announce that this is a template using `template<class T>` and then use `T` (or whatever name we want) to stand for the parametric type.
- When we declare an object of this template type we need to specify what type `T` is:

  ```
  aPair<int> p(1, 2)
  ```

- This example is in the sample program `templates2.cpp`.

Template specialization

- If you want a template class to do something different for a particular type of parameter, you can write a *specialization* of the template.

```
template<>
class aPair <char>{
private:
  char values[2];
public:
  aPair(char first, char second){
    values[0] = first;
    values[1] = second;
  }
  void print(){
    cout << values[0] << values [1] << endl;
  }
};
```

- Note the way we specify what kind of specialization we are defining:

```
template<>
class aPair <char>{
```

- And note that we have to define all the bits of the generic template that we want to keep (like the constructor).
- Again this example is in `templates2.cpp`.

- In the example, all the functions are defined inside the class definition.
- If we want to declare the member functions outside the class definition we need to be a bit more longwinded.
- We need to indicate in the function header that it belongs to a template class.
- For example:

```
aPair<char>::aPair(char first, char second){
  values[0] = first;
  values[1] = second;
}
```

  would work for the constructor of the template specialization.

- This:

```
template<class T>
T aPair<T>::getSecond(){
    return values[1];
  }
```

  would work for an access function for the more generic class.

Templates and multi-file projects

- As we have discussed before, it is common practice to split class definitions into two files.
- The `class` declaration in a `.h` file and the function code in a `.cpp` file.
- This may cause problems if you do it for a template class.
- Older compilers don't allow the template class declaration to be in a separate file from the function code.

Generic stack example

- Here is an example of a generic stack, using a template and a version of the stack class we defined earlier this term:

```
template <class TYPE>
class stack {
  public:
    explicit stack( int size=100 ) : max_len(size),
                                     top(EMPTY),
                                     s(new TYPE[size])
                                     {assert( s != 0 );}
    ~stack() { delete []s; }
    void reset() { top = EMPTY; }
    void push( TYPE c ) { s[++top] = c; }
    TYPE pop() { return s[top--]; }
    TYPE top_of() const { return s[top]; }
    bool empty() const { return( top == EMPTY ); }
    bool full() const { return( top == max_len - 1 ); }
```

```
  private:
    enum { EMPTY = -1 };
    TYPE *s;
    int max_len;
    int top;
};
```

- As we discussed above, the identifer TYPE is the generic template argument and is replaced when a variable of this type is declared, e.g.:

```
stack<char>   stk_ch;
stack<char *> stk_str(200);
stack<point>  stk_point(10);
```

- Again, the template saves writing essentially the same code to operate on data of different types

- Code snippet using stack template to reverse an array of strings:

```
void reverse( char *str[], int n ) {
  stack<char *> stk(n);
  int i;
  for ( i=0; i<n; ++i ) {
    stk.push( str[i] );
  }
  for ( i=0; i<n; ++i ) {
    str[i] = stk.pop();
  }
}
```

- Here's a `main()` to go with it:

```
int main( int argc, char *argv[] ) {
  int i;
  cout << "before:\n";
  for ( i=0; i<argc; i++ ) {
    cout << argv[i] << endl;
  }
  reverse( argv, argc );
  cout << "\nafter:\n";
  for ( i=0; i<argc; i++ ) {
    cout << argv[i] << endl;
  }
} // end of main()
```

- This code is all in `templates3.cpp`
- If you run the above example, you should enter command-line parameters; the program will print them out in the order they were entered, then run `reverse()` to invert the order of the parameters and print them again, using the new order

- For example:

```
unix-prompt> ./a.out abc def 123
before:
./a.out
abc
def
123

after:
123
def
abc
./a.out
```

- This example decalres all the member functions inline:

```
TYPE top_of() const { return s[top]; }
void push( TYPE c ) { s[++top] = c; }
bool empty() const { return( top==EMPTY ); }
```

- As noted above, external declaration gets a bit clunky.

- External examples for the same function definitions:

```
template<class TYPE> TYPE stack<TYPE>::top_of() const {
  return s[top];
}

template<class TYPE> void stack<TYPE>::push( TYPE c ) {
  s[++top] = c;
}

template<class TYPE> bool stack<TYPE>::empty() const {
  return( top==EMPTY );
}
```

Summary

- This lecture looked at templates.
- Templates are an important method for obtaining parametric polymorphism.
- We looked at:
  - Function templates.
  - Class templates.
- In the next class we will go on to look at the ready-made templates of the standard template library.