More Power to Functions

Function Overloading

Applicability rather wide/general

- The ***more widely/generally applicable way*** to give more power to a function is to
  - define ***separate function definitions***, as if they are meant to be functions entirely different one from another, and
  - use the ***same function name*** for all those definitions
- In other words, we simply define a set of ***overloaded functions***, which are functions different from the usual functions in that
  - they all ***share the same name***
- We do have to ensure that all of the separate function definitions have ***distinct signatures***
  - otherwise the compiler will complain and not compile

More Power to Functions

Function Overloading

Recap

- Allows programmer to define
  - ***different meanings*** for the ***same function name***
- Requires that each of the different meanings
  - has a ***signature*** that is ***distinct from those of all the others***

Function Signature

Recap

- In addition to its name, a function's signature is determined by
  - the ***number***, ***type*** and ***order*** of the ***function's parameter(s)***
- A function's ***return type***
  - ***does not*** play a role in defining the function's signature

More Power to Functions

Function Overloading

Swap example recap

```
#include <iostream>
#include <cstdlib>
using namespace std;

void Swap(char&, char&);
void Swap(int&, int&);
void Swap(double&, double&);

int main()
{
  char c1 = 'x', c2 = 'y';
  int i1 = 11, i2 = 22;
  double d1 = 11.11, d2 = 22.22;

  Swap(c1, c2);
  cout << "\nc1 = " << c1
       << "; c2 = " << c2 << endl;
  Swap(i1, i2);
  cout << "\ni1 = " << i1
       << "; i2 = " << i2 << endl;
  Swap(d1, d2);
  cout << "\nd1 = " << d1
       << "; d2 = " << d2 << endl;

  return(EXIT_SUCCESS);
}
```

```
void Swap(char& p1, char& p2)
{
  char temp = p1;

  p1 = p2;
  p2 = temp;
}

void Swap(int& p1, int& p2)
{
  int temp = p1;

  p1 = p2;
  p2 = temp;
}

void Swap(double& p1, double& p2)
{
  double temp = p1;

  p1 = p2;
  p2 = temp;
}
```


More Power to Functions

Function w/ Default Arguments

For some situations only
A "better" way if applicable

- When we have the situation where
  - all the overloaded functions have signatures that differ only in the ***number of parameters***
  - all the overloaded functions have (or can be made to have) the ***same operations and program logic***
  - the overloaded functions that have lesser number of parameters are simply letting the missing parameters take on ***default values***

  we can effect function overloading by taking advantage of C++'s support for ***function with default argument(s)***
- This is particularly useful for compactly providing a **`class`** with a set of overloaded constructors
  - as we have seen in the **`Date`** example previously discussed

More Power to Functions

Function w/ Default Arguments *E.g.*

Used in our `Date` class
Repeated as example

The following set of overloaded constructors

```
Date();              //default constructor
Date(int dd);             //1-parameter constructor
Date(int dd, char *mm);    //2-parameter constructor
Date(int dd, char *mm, int yy); //3-parameter constructor
```

can be compactly provided by

```
Date(int dd = 1, char *mm = 0, int yy = 1);
```

assuming (*1*, *null address*, *1*) are the appropriate default values for (`day`, `month`, `year`), respectively

More Power to Functions

Function Templates

For some situations only
Yet another "better" way if applicable

- When we have the situation where
  - all the overloaded functions have signatures that differ only in the ***types of parameters***
  - all the overloaded functions have the ***same operations and program logic***

  we can effect function overloading by taking advantage of C++'s support for ***function templates***
- Function templates essentially allow us to parameterize
  - not only the ***values*** of function arguments
  - but also the ***data types*** of function arguments

✶ Normal functions only allow us to parameterize
  - the ***values*** of function arguments

More Power to Functions

Function Templates

E.g. 1

```
#include <iostream>
#include <cstdlib>
using namespace std;
template <class T>
void Swap(T& p1, T& p2)
{ T temp = p1; p1 = p2; p2 = temp; }
int main()
{
  char c1 = 'x', c2 = 'y';
  int i1 = 11, i2 = 22;
  double d1 = 11.11, d2 = 22.22;
  Swap(c1, c2);
  cout << "\nc1 = " << c1 << "; c2 = " << c2 << endl;
  Swap(i1, i2);
  cout << "\ni1 = " << i1 << "; i2 = " << i2 << endl;
  Swap(d1, d2);
  cout << "\nd1 = " << d1 << "; d2 = " << d2 << endl;
  return(EXIT_SUCCESS);
}
```

***template prefix* (`T` *is called the template parameter*)**

Compare this with the version that uses overloaded functions

More Power to Functions

Function Templates

E.g. 2

```
#include <iostream>
#include <cstdlib>
using namespace std;
template <class T>
T FindMax(T p1, T p2)
{
  if (p1 > p2) return p1;
  else return p2;
}
int main()
{
  char c1 = 'x', c2 = 'y';
  int i1 = 11, i2 = 22;
  double d1 = 11.11, d2 = 22.22;
  cout << "Larger of c1 and c2 is " << FindMax(c1, c2) << endl;
  cout << "Larger of i1 and i2 is " << FindMax(i1, i2) << endl;
  cout << "Larger of d1 and d2 is " << FindMax(d1, d2) << endl;
  return(EXIT_SUCCESS);
}
```

More Power to Functions

Function Templates

E.g. 3

```
#include <iostream>
#include <cstdlib>
#include <cassert>
using namespace std;

template <class T1, class T2> T2 FindMaxIndex(const T1 data[], T2 size)
{
  T2 i, answer = 0; assert(size > 0);
  for (i = 1; i < size; i++)
    if (data[answer] < data[i]) answer = i;
  return answer;
}

int main()
{
  const size_t SIZE = 3;
  int a1[SIZE] = {3, 9, 2}, a2[4] = {7, 0, 1, 4};

  cout << "Max-value index of a1: " << FindMaxIndex(a1, SIZE) << endl;
  cout << "Max-value index of a2: " << FindMaxIndex(a1, 4) << endl;

  return(EXIT_SUCCESS);
}
```

***We want to parameterize the data type of* `size` *instead of using straight* `int` *because some compilers don't know how to covert from* `size_t` *to* `int`**

More Power to Functions

Function Templates

E.g. 4 (This won't work. Why?)

```
#include <iostream>
#include <cstdlib>

template <class T>
T GetValue()
{
  T value;

  cout << "Enter value: ";
  cin >> value;

  return value;
}
```

```
using namespace std;

int main()
{
  int intValue;
  double dblValue;
  char charValue;

  intValue = GetValue();
  cout << "Value entered is "
       << intValue << endl;
  dblValue = GetValue();
  cout << "Value entered is "
       << dblValue << endl;
  charValue = GetValue();
  cout << "Value entered is "
       << charValue << endl;

  return(EXIT_SUCCESS);
}
```

More Power to Functions

Function Templates

E.g. 4 (This will work. Why?)

```
#include <iostream>
#include <cstdlib>

template <class T>
void GetValue(T& value)
{
  cout << "Enter value: ";
  cin >> value;
}
```

```
using namespace std;

int main()
{
  int intValue;
  double dblValue;
  char charValue;

  GetValue(intValue);
  cout << "Value entered is "
       << intValue << endl;
  GetValue(dblValue);
  cout << "Value entered is "
       << dblValue << endl;
  GetValue(charValue);
  cout << "Value entered is "
       << charValue << endl;

  return(EXIT_SUCCESS);
}
```


More Power to Functions

Function Templates

E.g. 4 (can do **explicit specification** now)

```
#include <iostream>
#include <cstdlib>

template <class T>
T GetValue()
{
  T value;

  cout << "Enter value: ";
  cin >> value;

  return value;
}
```

```
using namespace std;

int main()
{
  int intValue;
  double dblValue;
  char charValue;

  intValue = GetValue<int>();
  cout << "Value entered is "
       << intValue << endl;
  dblValue = GetValue<double>();
  cout << "Value entered is "
       << dblValue << endl;
  charValue = GetValue<char>();
  cout << "Value entered is "
       << charValue << endl;

  return(EXIT_SUCCESS);
}
```

More Power to Functions

Function Templates

E.g. 4 Extra

```
#include <iostream>
#include <typeinfo>
#include <string>
#include <cstdlib>

using namespace std;

template <class T>
void GetValue(T& value)
{
  int iDummy;
  double dDummy;
  char cDummy;
  string int_name ( typeid(iDummy).name() );
  string double_name ( typeid(dDummy).name() );
  string char_name ( typeid(cDummy).name() );

  string type( typeid(value).name() );
  if ( type == int_name )
    cout << "Enter integer value: ";
  else if ( type == double_name)
    cout << "Enter double value: ";
  else if ( type == char_name )
    cout << "Enter character: ";
  cin >> value;
}
```

```
int main()
{
  int intValue;
  double dblValue;
  char charValue;

  GetValue(intValue);
  cout << "Value entered is "
       << intValue << endl;
  GetValue(dblValue);
  cout << "Value entered is "
       << dblValue << endl;
  GetValue(charValue);
  cout << "Value entered is "
       << charValue << endl;

  return(EXIT_SUCCESS);
}
```


Class Templates

Extending the template concept

- The concept of data type parameterization has also been extended to apply to **`class`** in C++
- When we have the situation where there is a need for
  - a related group of **`class`**es that differ in their construction only in some or all of the ***component data types***
  - *e.g.*, we may be thinking about creating a group of array classes (to avoid the limitations of C++'s built-in array) that we can use to create arrays of different data types (**`int`**, **`char`**, *etc.*)

  we can define *one generic* ***class template*** instead of defining *several specific* **`class`**es
- (Does container classes ring any bell?) STL
  - "class templates + algorithms + iterators" for various containers
- The syntax for defining and using **`class`** templates is
  - unfortunately, more complex than that for function templates

Syntax for a Class Template

The tough part

- Class definition is preceded by the template prefix (such as **`template <class T>`**), like in function templates
- For all functions associated with the class template that appear ***outside of the class template definition***
  - the template prefix must be placed immediately before each function prototype and implementation
  - *i.e.*, each of the functions is a function template
- ***Template class name*** must be used to refer to the class
  - *e.g.*, **`Array<T>`**
- Best way to learn the syntax is to study some examples and learn through making some mistakes

Class Template

`pairs.h`
(interface)

Ordered-pair data types the **non-template** way

```
#ifndef PAIRS_H
#define PAIRS_H

#include <iostream>

class IntPair
{
  private:    int x, y;
  public:     IntPair(int xx = 0, int yy = 0);
              void SetPair(int xx, int yy);
              void ShowPair(std::ostream& outs) const;
};

class DoublePair
{
  private:    double x, y;
  public:     DoublePair(double xx = 0, double yy = 0);
              void SetPair(double xx, double yy);
              void ShowPair(std::ostream& outs) const;
};

#endif
```

Class Template

pairs.cpp
(implementation)

Ordered-pair data types the **non-template** way

```
#include "pairs.h"
#include <iostream>
using namespace std;

IntPair::IntPair(int xx, int yy) : x(xx), y(yy) { }
void IntPair::SetPair(int xx, int yy) {x = xx; y = yy;}
void IntPair::ShowPair(ostream& outs) const
{ outs << '(' << x << ", " << y << ')'; }

DoublePair::DoublePair(double xx, double yy) : x(xx), y(yy) { }
void DoublePair::SetPair(double xx, double yy) {x = xx; y = yy;}
void DoublePair::ShowPair(ostream& outs) const
{ outs << '(' << x << ", " << y << ')'; }
```


Class Template

pairsapp.cpp
(application)

Ordered-pair data types the **non-template** way

```
#include "pairs.h"
#include <cstdlib>
#include <iostream>
using namespace stad;

int main()
{
  IntPair intObj;
  DoublePair doubleObj;

  intObj.SetPair(3, 5);
  doubleObj.SetPair(3.3, 5.5);

  cout << "\nOrdered pair of integers: ";
  intObj.ShowPair(cout);
  cout << "\nOrdered pair of doubles: ";
  doubleObj.ShowPair(cout);
  cout << endl;

  return (EXIT_SUCCESS);
}
```

Class Template

Ordered-pair data types the **template** way

pairs.h
(interface)

```
#ifndef PAIRS_H
#define PAIRS_H

#include <iostream>

template <class T>
class OrderedPair
{
  private:   T x, y;
  public:    OrderedPair(T xx = 0, T yy = 0);
             void SetPair(T xx, T yy);
             void ShowPair(std::ostream& outs) const;
};

#include "pairs.cpp"

#endif
```

?

Class Template

Ordered-pair data types the template way

pairs.cpp
(implementation)

?

```
template <class T>
OrderedPair<T>::OrderedPair(T xx, T yy) : x(xx), y(yy) { }

template <class T>
void OrderedPair<T>::SetPair(T xx, T yy)
{
  x = xx;
  y = yy;
}

template <class T>
void OrderedPair<T>::ShowPair(std::ostream& outs)
{
  out << '(' << x << ", " << y << ')';
}
```

Class Template

pairsapp.cpp
(application)

Ordered-pair data types the template way

```
#include <stdlib>
#include "pairs.h"
#include <iostream>
using namespace std;

int main()
{
  OrderedPair<int> intObj;
  OrderedPair<double> doubleObj;

  intObj.SetPair(3, 5);
  doubleObj.SetPair(3.3, 5.5);

  cout << "\nOrdered pair of integers: ";
  intObj.ShowPair(cout);
  cout << "\nOrdered pair of doubles: ";
  doubleObj.ShowPair(cout);
  cout << endl;

  return (EXIT_SUCCESS);
}
```


Templating Container Class

Guidelines adapted from Textbook

(assuming **`Item`** is *typename parameter* involved)

- *Template prefix* precedes each function prototype/implementation
- Outside of class definition (*e.g.*: in implementation file), append **`<Item>`** to each *class* name ***that refers to the class***
  - (NOT to names of *constructors*, although they *match the class name*)
  - *E.g.*: **`bag<Item>`**
- Use **`<Item>`** instead of **`value_type`**
- Outside of member functions and class definition, add **`typename`** before any use of one of class' type names
  - *E.g.*: **`typename bag<Item>::size_type`**
- Name implementation file with **`.template`** extension and `#include` it at bottom of header file
- Don't use *using directives* in implementation file
  - Write **`std::`** in front of any Standard Library functions instead
- For *functions with default arguments*, may need to specify default values in both function prototype and function implementation
  - Compiler dependent

Textbook Readings

- Chapter 6
  - Section 6.1
  - Section 6.2