Generic programming

- Definition
- Function templates
- Operator overloading
- Class templates

Templates

- Function templates: for algorithm abstraction
  - Define a function independent of the data type \( T \) on which the function operates
  - \( \text{Swap<T>(a,b)} \)
- Class templates: for data abstraction
  - Define a class (an object) independent of the data type \( T \) of members—often used for containers
  - \( \text{Set<T>, List<T>, Stack<T>, ...} \)
Generic programming

- Definition
- Function templates

Problem: generic algorithms

- Define max(l,r) for every type?
- Waste of space (code bloat): every definition is almost identical
- COPY and PASTE may lead to errors and very bored programmers

What do we really want?

- Programmer should define general function definition which works for both int and double parameters (and others!)
- Compiler should generate the code for each data type
  - Compiler doesn’t get bored
  - Compiler doesn’t make copy-paste errors
  - Compiler can generate code for data types as needed—can reduce code bloat

Solution: Function templates

- A function template is a pattern used by the compiler to automatically generate a family of function definitions
  - This is a powerful feature of C++
  - To define a function template, use:
    
    ```
    template<class T>
    T func(T arg)
    ```

  - T will be replaced by the compiler with the data type for which the function code makes sense
  - The compiler will complain if it cannot!

Function templates: example

- A function template for the function max:
  ```
  template <class T>
  T myMax (T left, T right)
  {
  if (left < right)
  return right ;
  else
  return left ;
  }
  ```

  Here “class T” means “type T”
  In myMax(), the compiler must compare “<”
  - Hence, T can be any type for which the operator “<” is defined

  // returns the maximum
  // of 2 integers
  int max(int left, int right)
  {
  if (left < right)
  return right ;
  else
  return left ;
  }

  // returns the maximum
  // of 2 doubles
  double max(double left, double right)
  {
  if (left < right)
  return right ;
  else
  return left ;
  }

#include <iostream>
using namespace std;

.. place here the function template

```
int main()
{
  int integer1 = 4;
  int integer2 = 10;
  // Note: compiler can figure out "int" type here!
  int max1 = myMax(integer1, integer2);
  cout << "The maximum integer is " << max1 << endl;

  double double1 = 100.20;
  double double2 = 5.7;
  // Note: compiler can figure out "double" type here!
  double max2 = myMax<double>(double1, double2);
  cout << "The maximum double is " << max2 << endl;
  return 0;
  ```
Example output

- The output is:
  
  The maximum integer is: 10
  The maximum double is: 100.20

- Note: some older compilers do not accept separate compilation for function templates
  - In this case function templates should be in the same file with main()—not in a header file!

Template error example

- `myMax(integer1, double1)`
  - error: no matching function for call to `myMax(int, double)`
  - Compiler cannot infer type of T because both arguments are supposed to be T!
- `myMax<int>(integer1, double1)`
  - warning: passing ‘double’ for argument 2 to ‘T maxb(T, T) [with T = int]’
  - Compiler knows type, but “double1” is wrong

Generic programming

- Definition
- Function templates
- Operator overloading

Why can’t “T” be any type?

- Remember: T can be ANY type for which the code in the function definition makes sense (the compiler doesn’t complain!)!
- In `myMax()`, T can be any type for which the operator “<” is defined: int, double, float, char (only lower or only upper case)
  - What if we want to compare 2 arrays or 2 structs?
  - Write your own “<” operator for other types!

Operator overloading

- An overloaded operator has more than one definition
  - Can implement different algorithms for different types
  - A + B for matrices, etc.
  - Not allowed in Java (for reasons discussed later)

Operator overloading rules

- All operators can be overloaded except:
  - . (member access from an object)
  - :: (scope resolution)
  - .* (member object selector, don’t ask)
  - ?: (if-then-else expression)
- You cannot change the unary/binary nature of an operator
- You cannot override precedence rules
Operator overloading:
example “<<“

- The operator “<<“ is used for output basic type variables:
  
  ```
  cout << a << 'a' << 5.0/2
  ```

- How can we use “<<” to print larger variables (arrays, structures)?
  
  - Solution: overload the “<<” binary operator

Ex. 1: Overloading operator<<

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

struct Student {
    string name;
    int stud_nr;
    int grade;
};

// Overload operator << to output a structure
// of type Student with newline

ostream& operator<<(ostream& outs, const Student& the_object) {
    outs << the_object.name << " 
    " << the_object.stud_nr << "  
    " << the_object.grade << endl;
    return outs; // allow chaining "<<!
}
```

Ex. 1: Overloading operator<<

```c++
int main() {
    Student me, you;
    me.name = "Sheila";
    me.stud_nr = 14537780;
    me.grade = 8;
    you.name = "Bob";
    you.stud_nr = 14532180;
    you.grade = 4;
    cout << me << you;
    return 0;
}
```

Output:

```
Sheila  14537780  8
Bob  14532180  4
```

By default operator “<<” is bitwise left-shift

- `ostream` class overloads this operator by providing a function with the prototype:

  ```
  ostream& operator<<(ostream&, type)
  ```

Any function that overloads “<<” has to return a reference to the `ostream` object parameter

- This allows: `cout << "a chain" << " of " << "output"
- `ostream` can be `cout` or any other stream (including files)

Notes on operator<<()

Operator overloading:
example “<“

- The operator “<” is defined only for basic types (int, real, char)
- How can we compare two structs?
  - Solution: overload the “<” operator
  - Then, we can use `myMax()` for other types

Ex. 2: Overloading operator<

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

struct Student {
    string name;
    int stud_nr;
    int grade;
};

// overloads “<“ operator to compare 2 structs of type Student

bool operator<(const Student& student1, const Student& student2) {
    return (student1.grade < student2.grade);
}
```

Ex. 2: Overloading operator<
int main()
{
    Student me, you;
    me.name = "Sheila";
    me.stud_nr = 14537780;
    me.grade = 8;
    you.name = "Bob";
    you.stud_nr = 14532180;
    you.grade = 4;
    if (me < you)
        cout << you.name << " is more clever than " << me.name << endl;
    else
        cout << me.name << " is more clever than " << you.name << endl;
    return 0;
}

Ex. 2: Overloading operator<

Why not overload an operator?

- Can be wrong
  - int operator+(int a, int b) { return a - b; }
- Can be confusing
  - What order does student A < B use?
- This is why Java doesn't allow operator overloading
- In general, overload operator<< for printing
  - Use more descriptive function names instead
  - Will need to overload some operators to use libraries

Generic programming

- Definition
- Function templates
- Operator overloading
- Class templates

Problem: generic classes

What do we really want?

- Programmer should define general List class for lists of any type (doubles, char, classes)
  - Generic classes can be provided by a library
  - Compiler should generate type-specific code
    - As with function templates, compiler should do all the work!

Solution: class templates

- A class templates is a pattern used by the compiler to automatically generate class types
- To define a class template use:
  - template<class T> class myType {
- A container is a generic class
  - Examples: lists, sets, queues, etc.
  - See also: java.util.*
Class templates: example

// Generic Node
template<class T>
struct Node
{
    Node<T>* next;
    T data;
};

// Generic List container
template<class T>
class GenericList
{
public:
    void add(const T& data);
    List();
    ~List();
    friend ostream& operator<<(ostream& outs, GenericList<T>& list);
private:
    Node<T>* head;
};

int main()
{
    // Note: compiler needs to know type here!
    GenericList<int> first_list;
    first_list.add(1);
    first_list.add(2);
    cout << first_list;
    GenericList<char> second_list;
    second_list.add('A');
    second_list.add('B');
    cout << second_list;
    return 0;
}

Outline

- Generic programming
- Standard Template Library (STL)

Standard Template Library (STL)

"Don't reinvent the wheel; use libraries"
B. Stroustrup, The programming C++ language, 2000

- STL is a large collection of templates for useful classes and functions
  - Developed by A. Stepanov and M. Lee at Hewlett Packard Labs in 1994
  - STL is tested, debugged, offers high performance, and is free
  - Practically, STL now is part of C++ language
  - Similar to the way java.util.* is part of Java

More information

- Read chapter 16 in Prata's book

STL overview

- STL contains:
  - generic classes (class templates, or containers)
  - generic algorithms (function templates)
- Examples:
  - Containers: class templates for vector, list, stack, queue, map, set
  - Iterators: pointer-like objects to access container elements
  - Algorithms: function templates for operations on containers
STL containers

- Containers contain data of a single type and functions to operate on that data
- Some frequently used STL containers:
  - `vector`: for dynamic arrays (order by index)
  - `list`: for doubly-linked list
  - `stack`: for LIFO (last in first out) structures
  - `queue`: for FIFO (first in first out) structures
  - `deque`: for double-ended queue
  - `priority_queue`: for order by max value
  - `map`: for key-value pairs

Vectors:

- Vectors are like arrays that can grow and shrink while your program is running
- Elements of the vector can be of any type.

Example:

```cpp
#include <vector>
using namespace std;

int main() {
    // declare a vector of integers
    vector<int> my_vector;
    // declare a vector of objects of type my_class
    vector<my_class> my_class_vector;
    // ... other vector operations ...
    // return 0 to indicate success
    return 0;
}
```

How do you access elements?

- Each container type has its own `iterator` type to access its data
  - `vector<int>::iterator iterV;`
  - `list<double>::iterator iterL;`

Using iterators

- Iterators are similar to pointers:
  - `iter++`: advances to the next element in the container
  - `iter--`: goes back to the previous element
  - `*iter`: gives access to the data pointed by `p`.
  - `iter[2]` or `*(iter+2)`: point to the same element
  - `==` and `!=`: compares for equality or difference
- Note: Iterators use operator overloading!
- Each container has member functions `begin()` and `end()` to return an iterator to the first and the last element in the container

How to walk through a container `c`?

```cpp
vector<int>::iterator p; // declare p
for(p = c.begin(); p != c.end(); p++) {
    // process element
    do_stuff(*p);
}
```

How to work with a vector?

- Use `my_vector[i]` to read and change a vector element that already has a value
  - Example: `my_vector[2] = 5;`
  - Note: `[ ]` cannot be used for initialization!
- Use `push_back()` to add an element to the end of the vector
  - Example: `my_vector.push_back(10)`
- Use `pop_back()` to delete the last element
A vector container example

```cpp
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int> vec;
    vector<int>::iterator p;
    vec.push_back(10);
    vec.push_back(3);
    vec.push_back(7);
    vec.push_back(5);
    cout << "These are the contents of the vector:\n";
    for (p = vec.begin(); p != vec.end(); p++)
        cout << *p << " ";
    cout << endl;
    return 0;
}
```

Lists are the same!

```cpp
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> myList;
    list<int>::iterator p;
    for (int i = 1; i <= 10; i++)
        myList.push_back(i*2);
    cout << "These are the contents of the list:\n";
    for (p = myList.begin(); p != myList.end(); p++)
        cout << *p << " ";
    cout << endl;
    return 0;
}
```

Generic algorithms

```cpp
#include <algorithm>

STL provides more than 70 standard generic algorithms for the most basic and commonly used operations of containers
Each algorithm is implemented independently of the individual container type (that is, it is templated)
```

STL algorithms

- Some frequently used STL generic algorithms:
  - `find(x)` : finds x in a container
  - `count(x)` : counts the number of occurrences of x in the container
  - `max_element()` : returns the maximum element in a container
  - `sort(b,e)` : sorts a range of elements in a container
  - `swap(a,b)` : assigns the contents of a to b and the contents of b to a

Sorting a vector using STL

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    vector<int> vec;
    vector<int>::iterator p;
    vec.push_back(10);
    vec.push_back(3);
    vec.push_back(7);
    vec.push_back(5);
    cout << "Vector before sort:\n";
    for (p = vec.begin(); p != vec.end(); p++)
        cout << *p << " ";
    cout << endl;
    sort(vec.begin(), vec.end());
    cout << "Vector after sort:\n";
    for (p = vec.begin(); p != vec.end(); p++)
        cout << *p << " ";
    cout << endl;
    return 0;
}
```

Output:

```
Vector before sort:
10 3 7 5
Vector after sort:
3 5 7 10
```

More information

- Prata, Appendix G
- S. Lipmann, C++ primer
- Internet
Outline

- Generic programming
- Standard Template Library (STL)
- 2D data visualization

How to visualize the results?

- Write results to a file and:
  - use Microsoft Excel,
  - use gnuplot, or
  - pngwriter
- Many other options as well...

Microsoft Excel help

- Internet:

GNUplot

- Data and mathematical function plotting program
- http://www.gnuplot.info

Gnuplot example: plots

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

int main()
{
    // First, write data as x,y pairs (ex: "1 4.5")
    // to "myplotdata.dat"

    // To run gnuplot from your C++ program...
    system("d:/gnuplot/wgnuplot.exe d:/gnuplot/mycmdfile.txt");
    // or on kits (in directory with "mycmdfile.txt")...
    system("gnuplot mycmdfile.txt");

    return 0;
}
```

GNUplot command file

- Simple syntax to plot data as line graph:
  ```
  set data style lines
  plot "d:/gnuplot/myplotdata.dat" title 'line 1'
  pause -1
  ```

GNUplot help

- Use gnuplot help (in gnuplot program):
  kits$ gnuplot
  gnuplot> help
  gnuplot> quit
- Internet:
  - http://www.gnuplot.info/docs/node1.html

PNGwriter

- Easy-to-use open source graphics library that uses PNG (Portable Network Graphics) as output format
- The interface supports the following:
  - Plotting (no axes)
  - Basic shapes (lines, circles, rectangles)
  - Color (translucent, filled)
  - Scaling, interpolation

PNGwriter example

```cpp
#include "pngwriter.h"
int main() {
    int i;
    pngwriter png(300, 300, 0, "test.png"); // width and height, background & filename
    // Loop over line to draw...
    for (i = 1; i < 300; i++) {
        double y = 150 + 100*sin((double)i*9/300.0);
        png.plot(i, y, 0.0, 0.0, 1.0); // x, y coordinate of pixel
                                        // red, green, blue components
    }
    png.close();
    return 0;
}
```

PNGwriter help

- Internet:
  - http://pngwriter.sourceforge.net/
- Ask your grader for pointers to help

Self test exercises

1. Why should operator<< return the ostream argument?
   ```cpp```
   template<class T>
   ostream& operator<<(ostream& os, T obj) {
       return os;
   }
   ```cpp```
2. Why must operator<< be declared a friend of the class?
3. Declare a vector of type:
   ```cpp```
   class Student {
       int grade;
       string name;
       int id;
   }
   ```cpp```
4. Write a loop to find the student with a) the highest grade, then b) lowest id.
5. Can we use generic functions to find both?
Self test exercises

- 6. Why do we have to use `vector<int>` to declare a vector of int, but we can call `myMax(1, 29)` without specifying the `<int>`?
- 7. Why not use a template for every function?
- 8. When would you use a template instead of inheritance?

Questions?

- Thanks!
  - Good luck on the assignments