C++ at Velocity, Part 3

Ben Yuan
CS 002 - WI 2014
January 10, 2014

Last time...

- Pointers
- Pointer arithmetic
- Structs
- Classes

Templates

- Recall that C++ is a **statically typed** language.
- Recall our linked-list example from earlier.
  - Linked list of integers
- What if we want a linked list of **`Vector2`**?
- Don't want to copy a whole bunch of code!
  - Where possible, don't repeat yourself
  - What if we copy an implementation with bugs?
  - What if we need to make changes later?

(C++ 6.19)

Templates

- Templates allow us to apply one code pattern to many data types.
- Templates take one or more types as "parameters".

```
template <class T>
T sum(T a, T b)
{
    T result = a + b;
    return result;
}
...
printf("%d\n", sum<int>(8, 11));
```

(C++ 6.20-31)

Templates

- Classes and structs can also be templated.

```
template <class T>
struct node
{
   T data;
   node<T> * next;
};
...
node<double> n;
n.data = 3.14159;
```

(C++ 6.20-31)

Templates

- N.B.: As a general rule, template *classes* must be completely defined in header file!

```
template <class T>
class Vector2
{
...
public:
    Vector2(T x, T y)
    {
        this->x = x;
        this->y = y;
    }
...
}
```

Tools - gdb

- gdb is a **debugger**
- Allows systematic, careful examination of program execution and state
- Command-line based
- Use when you want to step through a program line-by-line
- Use when a program crashes

(C 6.2 (I'm basically mirroring Mike's slides))

Tools - **gdb**

- Before using **gdb**:
  - compile with **-g**
  - this includes source code information with program
- Then run **gdb ./programname**
- You're now in a **gdb** terminal

```
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
...
(gdb)
```

(C 6.3)

`gdb` - Running a Program

- **`run`** starts a program running
  - If all goes well, the program runs normally
  - If something goes wrong, GDB usually tells you, then stops
- Command-line arguments are given to **`run`**
  - e.g. **`run 1 2 3`**
- **`start`** starts the program, then stops at the first line
  - Useful if you need to step through from the beginning

(C 6.4-5)

gdb - Breakpoints

- To make sure the debugger stops somewhere, set a breakpoint with **`break`**
  - **`break foo.c: 100`**
  - stops when execution reaches the specified line
- To clear a breakpoint, use **`clear`** or **`delete`**
  - **`clear foo.c: 100`** (by line number)
  - **`delete 1`** (by breakpoint number)

(C 6.4-5)

gdb - Moving Around

- **continue** - runs until the next breakpoint
- **next** - runs the next source code line
- **step** - runs the next source code line, tracing into function calls
- **finish** - traces out of the current function

(C 6.9)

gdb - Gathering Information

- **`print`** - prints arbitrary expressions
  - can even call functions!

```
(gdb) print i
$1 = 42
(gdb) print square(i)
$2 = 1764
```

- **`display`** - prints a value each time the program stops

```
(gdb) display *p
```

(C 6.10)

gdb - Gathering Information

- **`backtrace`** - prints call stack
  - what functions are waiting to be resolved?

```
(gdb) bt
#0  divide (a=15625, b=37) at debugging2.cpp:31
#1  0x0000000000400566 in main (argc=1, argv=0x7fffffffe288)
    at debugging2.cpp:74
```

  - topmost = innermost function
- Things to look for
  - Accessing arrays out of bounds
  - Dereferencing invalid pointers (segmentation fault)
  - Freeing memory that was never allocated
  - Freeing memory that was already freed

(C 6.6-8)

Arithmetic on objects?

- Recall that we can perform arithmetic on primitive types.
```
int a = 3, b = 4, c;
c = a + b;
```

- What if we want to perform arithmetic on things that are not primitive types?
```
Vector2 a(3, 4), b(5, 12), c;
c = a + b; // ???
```

(C++ 6.20-37)

Operator overloading

- **Operator overloading** is the mechanism that lets us do this.
- Whenever we use an operator `+`, C++ calls some function `operator+(...)`.
- Most operators can be overloaded.

(C++ 6.20-37)

Operator overloading

- Suppose we want vectors to support < (compare by norm).

```
class Vector2
{
    ...
    friend bool operator< (Vector2 & v1, Vector2 & v2);
};

bool operator< (Vector2 & v1, Vector2 & v2)
{
    return (v1.Length() < v2.Length());
}
```

(C++ 6.20-37)

Operator overloading

- Suppose we want vectors to support < (compare by norm).

```
Vector2 a(5, 6), b(8, 113);

if (a < b) // true!
{
   ...
}
```

(C++ 6.20-37)

Class hierarchy

- We've mentioned "subclass", "derived class" in previous lectures.
- **Inheritance** is the mechanism by which subclasses work.

(C++ 7.2-8)

Class hierarchy

- Suppose you have a class for polygons:

```
class Polygon
{
protected:
   double w, h;
public:
   double set_dim(double a, double b) {...}
};
```

(C++ 7.9)

Class hierarchy

- A rectangle is a type of polygon:

```
class Rectangle : public Polygon
{
public:
   double area()
   { return w * h; }
};
```

(C++ 7.10)

Class hierarchy

- So is a triangle:

```
class Triangle : public Polygon
{
public:
    double area()
    { return w * h / 2.0; }
};
```

(C++ 7.10)

Class hierarchy

- Rectangles and triangles inherit **`Polygon::set_dim()`**

```
Rectangle a;
Triangle b;

a.set_dim(5.0, 5.0);
b.set_dim(6.0, 4.0);

printf("%f\n", a.area()); // prints 25.0
printf("%f\n", b.area()); // prints 12.0
```

Virtual functions

- Suppose we defined **`Polygon::area()`**.
- Notice that base-class pointers can point to subclass instances as well!
  - e.g. **`Polygon *`** pointers can point to **`Rectangle`** instances.
- We want to be able to do the following...

  ```
  Polygon * p = new Triangle(...);
  printf("%f\n", p->area());
  ```

  and have it just work.
- **Virtual functions** let us do this.

(C++ 7.20-23)

Virtual functions

- A **virtual function** is redefinable by subclasses.
- Calls using base-class pointers automatically invoke the subclass version if applicable.
  - This is the **polymorphism** mechanism.
- Functions must explicitly be marked virtual (unlike Python, Java, etc.)

(C++ 7.20-23)

Virtual functions

```
class Polygon {
public:
   virtual double area();
};

class Triangle {
public:
   double area() { return w * h / 2; }
};
```

(C++ 7.20-23)

Virtual functions

```
Polygon * p1 = new Rectangle(6.0, 4.0);
Polygon * p2 = new Triangle(6.0, 4.0);

printf("%f\n", p1->area()); // prints 24
printf("%f\n", p2->area()); // prints 12
```

(C++ 7.20-23)

Abstract base classes

- It's possible to define a virtual function with no given implementation:

```
virtual double foo() = 0;
```

- This is a **pure virtual** function.
- Any class with at least one pure virtual function is an **abstract base class**.
  - Cannot be instantiated!
  - Can only be used as a base class.

(C++ 7.31-34)

The C++ Standard Template Library

- Provides a large basket of built-in algorithms and data structures
- Use for your own projects
- Template library
  - can be used with arbitrary data types, including your own
- You'll encounter the STL in more depth from Week 4 onwards

The C++ Standard Template Library

- Data structures
  - sequence types (list, vector, deque)
  - collection types (set, multiset)
  - mapping types (map, unordered_map)
  - common operations for each
  - iterator objects
- Strings
  - rewritable, <u>resizable</u>
- Other features (including C++11 features)
  - algorithms, random numbers, regexes, ...

Namespaces

- Many of the C++ STL constructs are defined in the `std` namespace.
- **Namespaces** are used to separate functions and classes with similar names but different origins.
- To use a member of a namespace:

```
std::cout << "fish";
```

or

```
using namespace std;
cout << "fish";
```

Further reading

- The CS11 C and C++ lecture slides:
  - C: http://courses.cms.caltech.edu/cs11/material/c/mike/
  - C++: http://courses.cms.caltech.edu/cs11/material/cpp/donnie/
- External resources:
  - cplusplus.com Tutorial:
    - http://www.cplusplus.com/doc/tutorial/
  - cplusplus.com Library Reference:
    - http://www.cplusplus.com/reference/