Crash Course for C++

Oct 2

What is C++?

- is a better C (Procedural Programming)
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Hello World

```
// helloworld.cpp

#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

Comments

```
// helloworld.cpp


/* this is a multiline
  * C-style comment. The
  * compiler will ignore
  * it.
*/
```

Preprocessor Directives

- Building C++ Program takes 3 steps
  1. preprocessor
     - recognize meta-information about code
  2. compile
     - translate code into machine language objects
  3. linking
     - combine objects into one application

Preprocessor Directives

- #include
  - *Insert the file content here*
  - in C, #include <stdio.h>
  - in C++, #include <iostream>
    - ".h" is omitted for standard library
  - in C++, include C library
    - for <stdio.h>, do #include <cstdio>

Preprocessor Directives

- #define

```
#define MAX 10 // constant value, macro
if( x < MAX ) exit(1);

#define TESTING // option
#ifdef TESTING // include
    cout << “Testing now”;
#endif

#ifndef TESTING // exclude
    cout << “Not testing”;
#endif
```

Variables

- Declare anywhere, use within declared block or contained blocks

Type	Description	Usage
`int`	Positive and negative integers (range depends on compiler settings)	`int i = 7;`
`short`	Short integer (usually 2 bytes)	`short s = 13;`
`long`	Long integer (usually 4 bytes)	`long l = -7;`

`unsigned int`
`unsigned short`
`unsigned long`
	Limits the preceding types to values >= 0	`unsigned int i =2;`
		`unsigned short s = 23;`
		`unsigned long l = 5400;`

`float` `double`	Floating-point and double precision numbers	`float f = 7.2;` `double d = 7.2`
`char`	A single character	`char ch = 'm';`
`bool`	`true` or `false` (same as non-0 or 0)	`bool b = true;`

Variables

- Casting
  - Convert a variable's value into other compatible type
  - Three ways of casting

```
bool someBool = (bool)someInt;  // C-style
bool someBool = bool(someInt);
bool someBool = static_cast<bool>(someInt);
```

  - if some information is lost, compiler warns
    - int someint = somefloat;

Operator	Description	Usage

`=`	Binary operator to assign the value on the right to the variable on the left.	`int ;`
		`i = 3;`
		`int j;`
		`j = i;`

`!`	Unary operator to negate the true/false (non-0/0) status of a variable.	`bool b = !true;` `bool b2 = !b;`

`+`	Binary operator for addition.	`int i = 3 + 2;`
		`int j = i + 5;`
		`int k = i + j;`

`-`
`*`
`/`
	Binary operators for subtraction, multiplication, and division.	`int i = 5-1;`
		`int j = 5*2;`
		`int k = j / i;`

`%`	Binary operator for remainder of a division operation. Also referred to as the *mod* operator.	`int remainder = 5 % 2;`
`++`	Unary operator to increment a variable by 1. If the operator occurs before the variable, the result of the expression is the unincremented value. If the operator occurs after the variable, the result of the expression is the new value.	`i++;` `++i;`
`--`	Unary operator to decrement a variable by 1.	`i--;` `--i;`

`+=`	Shorthand syntax for `i = i + j`	`i += j;`

`-=`
`*=`
`/=`
`%=`
	Shorthand syntax for
	`i = i - j;`
	`i = i * j;`
	`i = i / j;`
	`i = i % j;`
		`i -= j;`
		`i *= j;`
		`i /= j;`
		`i %= j;`

`&` `&=`	Takes the raw bits of one variable and performs a bitwise "and" with the other variable.	`i = j & k;` `j &= k;`
`|`	Takes the raw bits of one variable and performs a bitwise "or" with the other variable.	`i = j | k;` `j |= k;`

Operator	Description	Usage

`<<`
`>>`
`<<=`
`>>=`
	Takes the raw bits of a variable and "shifts" each bit left (<<) or right (>>) the specified number of places.	`i = i << 1;`
		`i = i >> 4;`
		`i <<= 1;`
		`i >>= 4;`

`^` `^=`	Performs a bitwise "exclusive or" operation on the two arguments.	`i = i ^ j;` `i ^= j;`

Enumerator

```
typedef enum { kPieceTypeKing,
 kPieceTypeQueen, kPieceTypeRook,
kPieceTypePawn
} PieceT;
```

Exception Handling

- Exception
  - unexpected situation
  - Traditional way of indicating error
    - return a special value, NULL
  - C++ way
    - throws an exception object
    - The caller of the function catches it

```
#include <stdexcept>

double divideNumbers(double inNumerator, double inDenominator)
{
    if (inDenominator == 0) {
        throw std::exception();
    }

    return (inNumerator / inDenominator);
}
```

```
#include <iostream>
#include <stdexcept>

int main(int argc, char** argv)
{
    try {
        std::cout << divideNumbers(2.5, 0.5) << std::endl;
        std::cout << divideNumbers(2.3, 0) << std::endl;
    } catch (std::exception exception) {
        std::cout << "An exception was caught!" << std::endl;
    }
}
```

const

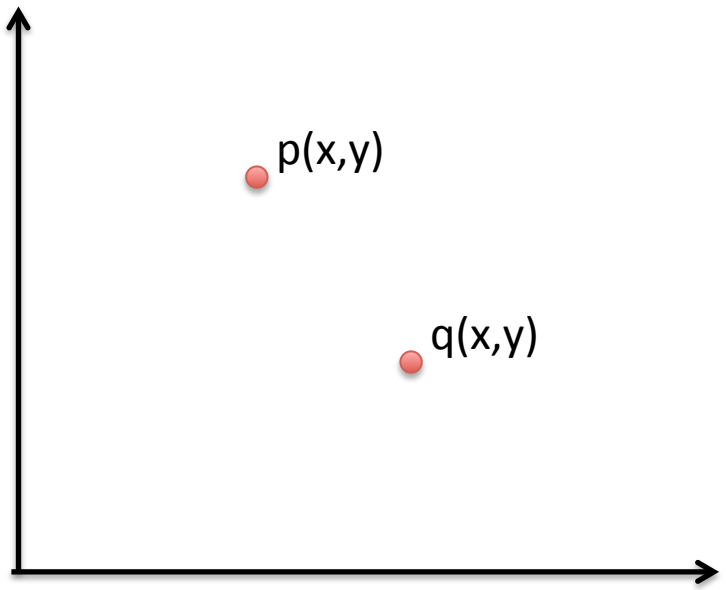
- Constants
  - Variable whose value never changes
  - const float pi = 3.14
- Protected variable
  - protect a variable from changes
  - void myfunc(const char* myString);

User-defined Types

- Design of a program is to
  - decide which types you want
  - determine operations needed for each type
- Example
  - We want a point type in 2D coordination
  - Type Point?
  - What state (data) it needs?
  - What operations it needs?

Type Point

- State
  - x, y coordinates
- Operations
  - constructors
  - destructors
  - setX(), setY()
  - vector operations (+, -, .)
  - comparisons (==, !=)
  - …



Class Point: Declaration

```
class Point {
    double x, y;
public:
    Point ();
    Point ( double, double );
    ~Point();
    void setX( double );
    void setY( double );
    friend Point operator+( Point, Point );
    friend Point operator-( Point, Point );
    friend bool operator==( Point, Point );
    friend bool operator!=( Point, Point );
}
```

Class Point: Definition

```
class Point {
        double x, y;
public:
        Point () { x=y=0; }
        Point ( double, double)
        ~Point() {}
        void setX( double ix );
        void setY( double iy ) { y = iy; }
        friend Point operator+( Point, Point );
        friend Point operator-( Point, Point );
        friend bool operator==( Point, Point );
        friend bool operator!=( Point, Point );
}
```

```
Point::Point(double ix, iy)
{
        x = ix;
        y = iy;
}

void Point::setX( double ix )
{
        x = ix;
}

Point operator+(Point a, b)
{
return Point(a.x+b.x, a.y+b.y);
}
```

Class Point: Use

```
int main() {
    Point a(1, 2);
    Point b(3, 4);
    Point c = a + b;
    Point d = a – b;
    if( c == d )
        out << “a=b=(0,0)” << endl;
}
```

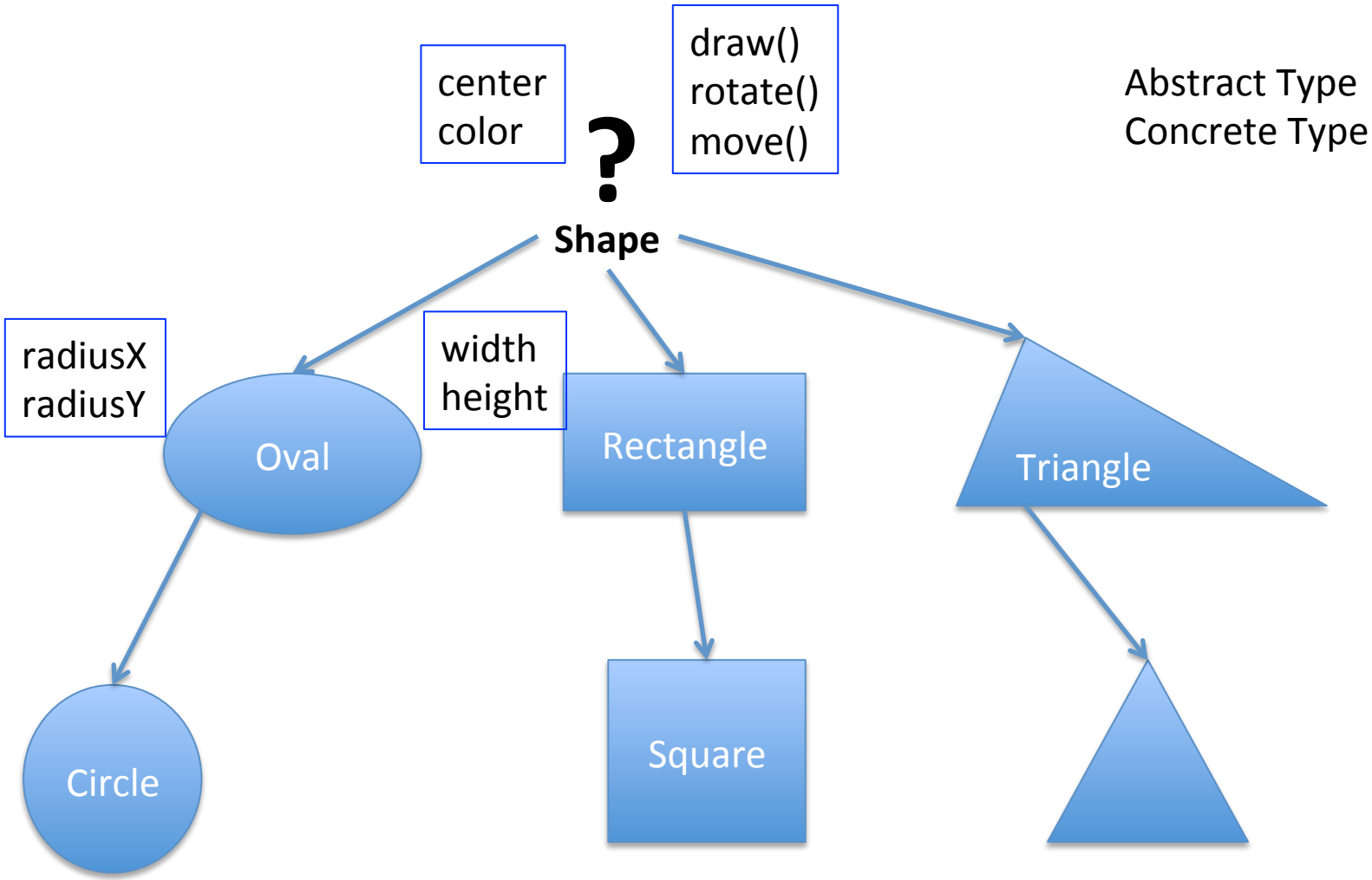
Class Shape

```
enum Kind {circle, triangle}

class Shape {
    Kind k;
    Point center;
    Color col;
    …
public:
    void draw();
    void rotate();
    …
}
```

- One class handles all related objects
- different kind needs different data
- draw() does different job for different kind
- What if we need to add new kind? rectangle?
- Very difficult to manage code

Class Hierarchy



Abstract Class

```
// Provides common interface to all shapes
class Shape{
    Point center;
    Color col;
public:
    Point where(){ return center; }
    void move(Point to) {center=to;  draw();}
    virtual void draw()=0;
    virtual void rotate(int angle)=0;
};
```

Sub class/Super class

```
class Oval: public Shape {
    int radiusX;
    int radiusY;
public:
    Oval ( Point p, int rX, rY );
    void draw(){/*...*/}
    void rotate(int) {/*...*/}
};
class Circle : public Oval {
public:
    Circle( Point p, int r) : Oval( p, r, r ) {}
    void rotate(int) {};
}
```

Benefit of abstract class

```
void draw_all (vector<Shape> v) {
    for(int i=0 ;i<v.size(); ++i)
        v[i]->draw();
}
```

Generic Programming

```
// Make a stack of any type
template<class T> class Stack {
    T* v;
    int max_size;
    int top;
public:
    Stack(int s);
    ~Stack();
    void push(T);
    T pop();
};
```

Classes

- Date
- Time
- Student
- Stack
- Queue
- Bank account
- Set
- Animal
- Anything…