Recursion

- Recursive function:
  - Function defined in terms of itself.
- Example:

  $f(x) = 2f(x-1) + x^2, \quad x \geq 1$

  $f(0) = 0$
- Not all mathematically recursive functions are efficiently implemented by C++ simulation of recursion.

A Simple Recursive Function

```
int f( int x )
{
	if( x == 0 )
			return 0;	/* base case */
	else
			return 2 * f( x - 1 ) + x * x;  /*recursive call*/
}
```

Recursive Calls

f(4)

58

*return 2*f(3) + 4*4*

21

*return 2*f(2) + 3*3*

6

*return 2*f(1)+2*2*

1

*return 2*f(0)+1*1*

0

return 0

A Simple Recursive Function

```
int f( int x )
{
    if( x == 0 )
            return 0;     /* base case */
    else
            return 2 * f( x - 1 ) + x * x;  /*recursive call*/
}
```

Try to Evaluate `f(-1)`

Recursive Calls

f(-1)

*return 2*f(-2)+(-1)*(-1)*

*return 2*f(-3)+(-2)*(-2)*

*return 2*f(-4)+(-3)*(-3)*

*return 2*f(-5)+(-4)*(-4)*

Infinite Recursion

```
int bad( int n )
{
      if( n == 0 )
            return 0;
      else
            return bad( n / 3 + 1 ) + n - 1;
}
```

Repeated calls to bad(1)!

Two Fundamental Rules of Recursion

1. Base cases:

   You must always have some base cases, which can be solved without recursion

2. Making progress:

   The recursive call must always be to a case that makes progress toward a base case.

Example: Printing Out Numbers

```
void printOut( int n ) // Print nonnegative n
{
        if( n >= 10 )
                printOut( n / 10 );
       printDigit( n % 10 );
}
```

Base case: printDigit(n % 10) if 0 <= n < 10

Recursion and Induction

- Theorem: The recursive number-printing algorithm is correct for n >=0
- Proof: by induction
  - Base case: if one digit number, call to printDigit.
  - Inductive hypothesis: Assume printOut works correctly for all numbers of k or less digits.

    A k+1 digit number is expressed as a k digit number followed by it least significant digit.

    |n/10| is the k digit number correctly printed by assumption
    and the last digit is n mod 10.

=> the program prints any (k+1) digit number correctly.

Four Basic Rules of Recursion

1. Base cases:

   You must always have some base cases, which can be solved without recursion

2. Making progress:

   The recursive call must always be to a case that makes progress toward a base case.

3. Design rule:

   Assume that all the recursive calls work. (see induction proof!)

4. Compound interest rule:

   Never duplicate work by solving the same instance of a problem in separate recursive calls.

Recursion

1. Indication of good use of recursion: difficult to trace down the sequence of recursive calls.
2. Gives cleaner code but has high cost.
3. Never be used as a substitute for a simple ‘for’ loop.
4. Bad idea to use it to evaluate simple mathematical functions.

C++ Classes

```
class IntCell
{
  public:
        IntCell( )
              { storedValue = 0; }

        IntCell( int initialValue )
              { storedValue = initialValue; }

        int read( )
              { return storedValue; }

        void write( int x )
              { storedValue = x; }

  private:
              int storedValue;
};
```

Extra Constructor Syntax and Accessors

```
class IntCell
{
  public:                                          - explicit constructor (avoids type conversions)
        explicit IntCell( int initialValue = 0 )   - default parameter
              : storedValue( initialValue ) {}     - initialization list


        int read( ) const                          - accessor (constant member function)
              { return storedValue; }

        void write( int x )             - mutator (implicit)
              { storedValue = x; }

  private:
              int storedValue;
};
```

C++11: use braces instead of parentheses for initialization list

```
              : storedValue{ initialValue } {}
```

Separation of Interface and Implementation

```
#ifndef  IntCell_H
#define IntCell_H

class IntCell
{

  public:
            explicit IntCell( int initialValue = 0 );

            int read( ) const;

            void write( int x );

  private:
                    int storedValue;
};

#endif
```

IntCell.h

Implementation

```
#include "IntCell.h"

    IntCell::IntCell( int initialValue ) : storedValue( initialValue )
    {
    }

    int IntCell::read( ) const
    {
        return storedValue;
    }

    void IntCell::write( int x )
    {
        storedValue = x;
    }
```

IntCell.cpp

Main Program

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

int main( )
{
       IntCell m;

       m.write( 5 );
       cout << "Cell contents: " << m.read( ) << endl;

       return 0;
}
```

TestIntCell.cpp

Range for (C++11)

```
vector<int> squares( 100 );
....

int sum = 0;
for ( int i = 0; i < squares.size( ); i++ )
     sum += squares[ i ];
```

C++11 “range for” (accessing every element in a collection):

```
int sum = 0;
for ( int x : squares )
        sum += x;

int sum = 0;
for ( auto x : squares )  // auto: compiler automatically infers type
        sum += x;
```

Pointers

- **Pointer variable:** stores the address where another object resides.

- **Example:** used for linked lists

IntCell Dynamic allocation

```
int main( )
{
        IntCell *m;                     // m is a pointer variable

        m = new IntCell( 0 );           // dynamic object creation
              // m = new IntCell{ 0 };  //C++11

        m -> write( 5 );                // -> access operator

        cout << "Cell contents: " << m -> read( ) << endl;

        delete m;               // garbage collection
        return 0;
}
```

TestIntCell.cpp

Important operator: address-of operator & , returns the memory location where an object resides.

C++11: Lvalues, Rvalues

Major change in C++11: new reference type called rvalue reference in addition to the standard lvalue reference.

lvalue: expression that identifies a non-temporary object.

rvalue: expression that identifies a temporary object or is a value (e.g., literal constant) not associated with any object.

Example:

```
vector<string> arr ( 3 );
const int x = 2;
int y;
int z = x + y;
string str = “foo”;
vector<string> *ptr = &arr;
```

lvalues: arr, str, arr[x], &x, y, z, ptr, x

rvalues: 2, “foo”, x+y, str.substr(0,1)

C++11: Lvalue and Rvalue Reference

lvalue reference: declared by placing & after some type (becomes a synonym for the object it reference)

```
string  str = “hell”;
string & rstr = str;
rstr += 'o';
bool cond = (&str == &rstr);
string & bad1 = “hello”;           \\illegal: “hello” is not a modif. value
string & bad2 = str + “ ”;         \\illegal: str + “ “ is not an lvalue
string & sub = str.substr( 0, 4); \\illegal: str.substr( 0, 4) not an lvalue
```

rvalue reference: declared by placing a && after some type (same as lvalue but it can also reference an rvalue (i.e., a temporary))

```
string && bad1 = “hello”;            \\Legal
string && bad2 = str + “ ”;          \\Legal
string && sub = str.substr( 0, 4); \\Legal
```

Parameter Passing

double avg(const vector<int> & arr, int n, bool & errorFlag);

- Call by value: creates a local copy of the object; copying.
- Call by constant reference: if the value of the actual parameter cannot be changed; no copying.
- Call by reference: if the formal parameter should be able to change the value of the actual argument; no copying. (C++11: call by lvalue reference)

Parameter Passing Options

- Call by value: for small objects that should not be altered by the function.
- Call by constant reference: for large objects that should not be altered by the function.
- Call by reference: for all objects that may be altered by the function.

Return Passing

- Return by value: always safe to return by value.

  string findMax(...)

- Return by constant reference: if the object is a class type (may be better than return by value).

  - the object itself cannot be modified later on.

  const string & findMax(...)

- Return by reference: very rarely used.

  string & findMax(...)

Destructor, Copy Constructor, operator=

- Destructor, copy constructor and operator= are special functions provided by C++.

- Destructor: called whenever an object goes out of scope or is subject to a delete.

- Copy constructor: special constructor required to construct a new object, initialized to a copy of the same type of object.

  It is called in the following instances:

  - Declaration with initialization : IntCell B = C, but not B = C
  - An object passed using call by value.
  - An object returned by value.

- ‘operator=‘ (copy assignment) is called when = is applied to two objects after they were constructed.

Defaults

```
IntCell::~IntCell( )
{
        // Does nothing, since IntCell contains only an int data member. If
        // IntCell contained any class objects, their destructors will be called.
}

int IntCell::IntCell ( const IntCell & rhs ) : storedValue( rhs.storedValue )
{
}

const IntCell & IntCell::operator=( const IntCell & rhs )
{
              if ( this != &rhs )        //test to make sure we are not copying to ourselves
                    storedValue = rhs.storedValue;
            return *this;
}
```

If data members are pointers the defaults are not good! => pointers in the two classes will point to the same object i.e. shallow copy.

Defaults (problems with pointers)

```
class IntCell
{
 public:
        explicit IntCell( int initialValue = 0 )
                { storedValue = new int( initialValue ); }
        int read( ) const { return *storedValue; }
        void write( int x ) { *storedValue = x; }
 private:
        int *storedValue;
};

int f( )
{
        IntCell a( 2 );
        IntCell b = a;
        IntCell c;
        c = b;

        a.write( 4 );
        cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;  //prints three 4
        return 0;
}
```

Solution

The big three needs to be written:

```
IntCell::IntCell( int initialValue )
        { storedValue = new int( initialValue ); }

IntCell::IntCell( const IntCell & rhs )
        { storedValue = new int( *rhs.storedValue ); }

IntCell::~IntCell( )
        { delete storedValue; }

const IntCell & IntCell::operator=( const IntCell & rhs )
        {
            if( this != &rhs )
                *storedValue = *rhs.storedValue;
            return *this;
        }
```