Egyption Multiplication

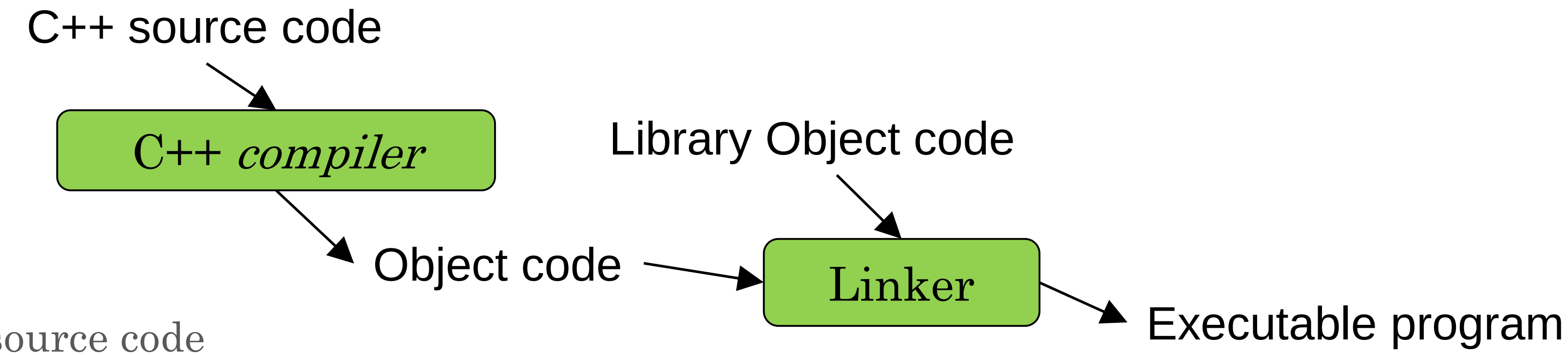
Lecture 2

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

A Word about Compilation

Compilation and Linking



- You write C++ source code
  - Source code is (in principle) human readable
- The compiler translates what you wrote into object code (sometimes called machine code)
  - Object code is simple enough for a computer to “understand”
- The linker links your code to system code needed to execute
  - E.g. input/output libraries, operating system code, and windowing code
- The result is an executable program
  - E.g. a .exe file on windows or an a.out file on Unix

See: [Decoding C++ Compilation Process: From Source Code to Binary]

CMake

- CMake is a family of tools
  - Building software
  - Testing software
  - Packaging software
- We will use it for building and testing
- CMake generates build systems files (Makefiles and or workspaces)
  - Those can be used to automatically build and test your code
- The user writes a single set of descriptive scripts
  - Define Targets and their inter-dependencies
- CMake is well integrated with many IDEs

Simplest Example

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(Demo1)
add_executable(Demo1 demo1.cpp)
```

In source build:

```
% cd demo1
% ls
CMakeLists.txt  demo1.cpp
```

Simplest Example

```
% cmake .
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile fe
-- Detecting C compile fe
-- Check for working CXX
-- Check for working CXX
-- Detecting CXX compiler
-- Detecting CXX compiler
-- Detecting CXX compile
-- Detecting CXX compile
-- Configuring done
-- Generating done
-- Build files have been
```

```
% make
Scanning dependencies of target Demo1
[ 50%] Building CXX object
CMakeFiles/Demo1.dir/demo1.cpp.o
[100%] Linking CXX executable Demo1
[100%] Built target Demo1

% ./Demo1
Hello, world!
```

Software Development Notes

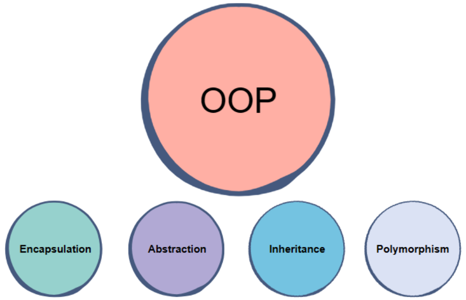
Challenges of Software Development

- Complexity
  - Software systems today are typically very large and very complex
- Longevity and Evolution
  - Systems are often in service for long periods of time
  - Being used for applications for which it was never intended
- High user expectations
  - Diversity of needs
  - Expectation for quality and security
  - Voodoo magic

Qualities of Software Systems

- Usefulness
  - Adequately address needs
- Timeliness
  - Quickly developed and deployed
  - Continuous integration/continuous deployment
- Reliability
  - Perform as expected
- Maintainability
  - Can easily make corrections, adaptations and extensions
  - Flexibility - Easily changeable
  - Simplicity - Anticipate and deal effectively with human error
  - Readability - Clarity and simplicity of design
- Reusability
  - Components can be repurposed for other applications
- User Friendliness
  - Intuitive use and access
- Efficiency
  - Efficient use of processing time, memory, and disk space

The Tenets of Object-Oriented (OO) Paradigm



- Abstraction
  - Hidden Data
    - Implementation of Abstract Data Type (ADT) is irrelevant
    - Interdependent class members are not accessed directly
      - This means no public class members
- Encapsulation
  - Data and methods on that data are bundled together
    - A class defines the data implementation, access to the data elements, and methods that act on the data
- Inheritance
  - A class (type) can take on the properties of another class
    - Creates the is-a relationship between the base class and the superclass
- Polymorphism
  - Derived objects (those of a class inherited from another) can behave differently
    - Interface of inherited methods remain the same, but may function differently

Structured Programming vs. OO Programming

- Structured Programming
  - Focus on logic and process flow
  - Defines operations on data manipulation
  - “Do something to data”

- OO Programming
  - Focus on data abstraction
  - Hides how data manipulation operations are performed
  - “Tell data to do something”

OO Development Activities

- Conceptualization
  - Establish a vision and core system requirements
- OO Analysis and Modeling
  - Build models the system's desired behavior
  - Unified Modeling Language (UML)
- OO Design
  - Create an architecture for implementation
- Implementation
  - Coding, debugging, and unit testing
  - Integration and integration testing
  - Regression and system testing
  - Deployment and deployment testing
- Maintenance
  - Fixing issues
  - Enhance functionality
  - Adapting the system to evolving needs and environments

Jia, Xiaoping. Object Oriented Software Development Using Java: Principles, Patterns, and Frameworks. Addison Wesley, 2003.

OO Analysis and Modeling

- Vague understanding of the problem is transformed into a precise description of the tasks that the software system needs to carry out
- The result is a detailed textual description, commonly called a *functional specification*, which:
  - Completely defines the tasks to be performed.
  - Is free from internal contradictions.
  - Is readable both by experts in the problem domain and by software developers.
  - Is reviewable by diverse interested parties.
  - Can be tested against reality.

OO Design Phase

- Structure the programming tasks into a set of interrelated types
  - Each type is specified precisely, listing both its responsibilities and its relationships
  - Usually language independent
- Result consists of a number of artifacts:
  - A textual description of the classes and their most important responsibilities
  - Diagrams of the relationships among the classes
  - Diagrams of important usage scenarios
  - State diagrams of objects whose behavior is highly state-dependent

Implementation Phase

- Types and methods are coded, tested, and deployed
- Often a prototype is built first
  - Reduced functionality
  - Helps verifying and correcting Analysis and Design
- Types (objects) are characterized by
  - State
  - Behavior
  - Identity

Egyptian Multiplication

Abstract

- Today, we'll will develop our first algorithm called 'Egyptian Multiplication'.
- An algorithm is a terminating sequence of steps for accomplishing a computational task.
- The first known algorithms have been documented 4000 years ago by the ancient Egyptian mathematicians.
- We will also talk about how to convert any recursive algorithm into an iterative one.

Multiplication

- We define multiplication as 'adding a number to itself a number of times'
- Formally:
  - Multiply something by one: $1a = a$ (1)
  - Multiply something one more time: $(n + 1)a = na + a$, i.e. by induction (2)
- We start with using a recursive implementation (both, n and a must be positive):

```
// version 0
int multiply0(int n, int a)
{
    if (n == 1) return a;                  // (1)
    return multiply0(n - 1, a) + a;        // (2)
}
```

Egyptian Multiplication

- Also known as 'Russian Peasant' Multiplication, first described by Ahmes
- It relies on the insight:

$$4a = \big((a + a) + a\big) + a$$
$$4a = (a + a) + (a + a)$$

- Depends on law of associativity:

$$a + (b + c) = (a + b) + c$$

- The idea is to keep halving n and doubling a while constructing a sum of power-of-2 multiples

Egyptian Multiplication

- Example $n = 41$ and $a = 59$:

$$41 \times 59 = (1 \times 59) + (8 \times 59) + (32 \times 59) = 2419$$

1	*	59
2		118
4		236
8	*	472
16		944
32	*	1888

- Each of the products is computed by doubling 59 the right number of times

Egyptian Multiplication

- Observation: coefficients needed for products represent the bits set in the binary representation of n:

2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	0	1

- Another observation: the algorithm relies on determining whether a number is odd or even:

$$n = \frac{n}{2} + \frac{n}{2} \Rightarrow even(n)$$

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \Rightarrow odd(n)$$

Egyptian Multiplication

```
// version 1
bool odd(int n) { return n & 1; }
int half(int n) { return n >> 1; }

int multiply1(int n, int a)
{
    if (n == 1) return a;
    int result = multiply1(half(n), a + a);
    if (odd(n)) result = result + a;
    return result;
}
```

Egyptian Multiplication

- Number of operations:
  - We half and recurse, so we need $\log_2 n$ additions
  - We need another addition sometimes (whenever the bit is set in the binary representation), i.e. the *pop* count: $\nu(n)$

$$N_+ = \lfloor \log_2 n \rfloor + (\nu(n) - 1)$$

Egyptian Multiplication

- Is our algorithm optimal?
  - As it turns out, no – not always:

$$N_+(15) = 3 + 4 - 1 = 6$$

  - But we can multiply by 15 using 5 additions:

```
// optimal multiply by 15
int multiply_by_15(int a)
{
    int b = (a + a) + a;    // 3 * a
    int c = b + b;          // 6 * a
    return (c + c) + b;     // 12 * a + 3 * a
}
```

  - This is called an 'addition chain', btw.

Exercise:

Find addition chains for all numbers $1 \leq n \leq 100$

Improving the Algorithm

Iterative Multiplication

- Algorithm does $\lfloor \log_2 n \rfloor$ recursion calls
- Let's convert the recursive version into an equivalent iterative version
  - Side note: any recursive algorithm can be converted into an equivalent iterative version using the technique that will be shown
- Note: *It's often easier to do more work rather than less*
- We're going to compute $r + na$, with r being the running result that accumulates the partial results na

Multiply-Accumulate

```
// Multiply-accumulate, version 0
int mult_acc0(int r, int n, int a)
{
    if (n == 1) return r + a;
    if (odd(n))
    {
        return mult_acc0(r + a, half(n), a + a);
    }
    return mult_acc0(r, half(n), a + a);
}
```

Multiply-Accumulate

- Simplify recursion: branches differ in first argument only

```
// Multiply-accumulate, version 1
int mult_acc1(int r, int n, int a)
{
    if (n == 1) return r + a;
    if (odd(n)) r = r + a;
    return mult_acc1(r, half(n), a + a);
}
```

- This makes the function *tail-recursive* (the recursion happens on return statement of the function)

Multiply-Accumulate (tail recursive)

- Further observations:
  - n is rarely equal to one
  - There is no point in checking for equality with one if n is even

```
// Multiply-accumulate, version 2
int mult_acc2(int r, int n, int a)
{
    if (odd(n))
    {
        r = r + a;
        if (n == 1) return r;
    }
    return mult_acc2(r, half(n), a + a);
}
```

- Don't assume that the compiler is performing this kind of optimizations for you!

Multiply-Accumulate

- A *strictly tail-recursive* function is one in which all the tail-recursive calls are done with the formal parameters of the function being the corresponding arguments

```
// Multiply-accumulate, version 3
int mult_acc3(int r, int n, int a)
{
    if (odd(n))
    {
        r = r + a;
        if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
    return mult_acc3(r, n, a);
}
```

Iterative Multiplication

- Now it's trivial to turn this into an iteration (replace tail recursion with `while(true)`):

```
// Multiply-accumulate, version 4
int mult_acc4(int r, int n, int a)
{
    while(true)
    {
        if (odd(n))
        {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Iterative Multiplication

- Use this for a new version of our multiply function:

```
// version 2
int multiply2(int n, int a)
{
    if (n == 1) return a;

    // Note: skip one iteration
    return mult_acc4(a, n - 1, a);
}
```

Further Improvements

Iterative Multiplication

- If n is a power of 2, then we first subtract one
  - This creates a number with all bits set in its binary representation
  - This is the worst case for our algorithm, so make sure n is odd

```
// version 3
int multiply3(int n, int a)
{
    while (!odd(n))
    {
        a = a + a;
        n = half(n);
    }

    if (n == 1) return a;
    return mult_acc4(a, n - 1, a);
}
```

Iterative Multiplication

- Last, but not least, we now have a superficial test for `odd(n)` in mult_acc4:

```
// final version 4
int multiply4(int n, int a)
{
    while (!odd(n))
    {
        a = a + a;
        n = half(n);
    }

    if (n == 1) return a;

    // even(n - 1) ==> n - 1 != 1
    return mult_acc4(a, half(n - 1), a + a);
}
```

Sieve of Eratosthenes

Sifting Primes

- Go through all numbers
  - Sifting out non-primes
  - Remaining numbers are prime
- Account for odd numbers only (starting at 3)

3	5	7	9	11	13	15	17	19	21	23	25	27
29	31	33	35	37	39	41	43	45	47	49	51	53

- In each iteration we take the first number (which is a prime) and cross out all multiples
  - Repeat as long as number is smaller than $\lfloor\sqrt{m}\rfloor$ (where m is largest considered number)

Sifting Primes

- Start with 3

3	5	7	9	11	13	15	17	19	21	23	25	27
29	31	33	35	37	39	41	43	45	47	49	51	53

- Next is 5:

3	5	7	9	11	13	15	17	19	21	23	25	27
29	31	33	35	37	39	41	43	45	47	49	51	53

- Last is 7:

3	5	7	9	11	13	15	17	19	21	23	25	27
29	31	33	35	37	39	41	43	45	47	49	51	53

Observations

index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
values:	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	

- Step size (number of entries between two crossed out numbers) is equal to factor
  - Number of index between two crossed out numbers is equal to factor
- Difference between two values is twice the value of the factor
- The first number crossed out is the square of the factor
  - All other multiples were already considered before

Implementing the Sieve

- One could assume that we need two arrays to implement the algorithm
  - Array of Booleans representing whether a number is a prime
  - Array of actual numbers
- However values don't need to be stored
- We can compute a value from an index:

$$value = 2 * index + 3$$

- Let's use an array of Booleans
  - `true` means prime
  - `false` means non-prime

Implementing the Sieve

```
template <typename I, typename N>
    requires(std::random_access_iterator<I> && std::integral<N>)
void mark_sieve(I first, I last, N factor)
{
    // precondition: range [first, last) is not empty
    // assert(first != last)

    *first = false;        // cross out first non-prime
    while (last - first > factor)
    {
        first = first + factor;
        *first = false;    // cross out next non-prime
    }
}
```

Implementing the Sieve

- Template

```
template <typename I, typename N>
```

- Use of concepts
  - `requires(std::random_access_iterator<I> && std::integral<N>)`
  - Defines requirements on type properties of arguments

- Iterators:
  - Think of them as ‘pointers’ to an element inside a sequence of values
  - ‘dereference’ iterator means “access the value of element it ‘points’ to”
    - `*first = false;`
  - ‘difference’ of iterators means “get the number of elements between them”
    - `while (last - first > factor)`
  - ‘adding’ (subtracting) N to iterator means “move to the Nth next (previous) element”
    - `first = first + factor;`

More Observations

- When sifting by p we will start at p^2
- When sifting numbers up to m stop at $p^2 \geq m$
- Value at index can be computed as:

$$value(i) = 2i + 3$$

- Index for value can be computed as:

$$index(v) = \frac{v - 3}{2}$$

- The index of the square of a value at i obviously is:

$$index(value(i)^2) = \frac{(2i + 3)^2 - 3}{2} = 2i^2 + 6i + 3$$

Implementing the Sieve

```
template <typename I, typename N>
    requires(std::random_access_iterator<I> && std::integral<N>)
void sift(I first, N n)
{
    std::fill(first, first + n, true);
    N i = 0, index_square = 3;
    while (index_square < n) {   // invariant: index_square = 2 * i^2 + 6*i + 3
        if (first[i]) {          // candidate is prime
            mark_sieve(first + index_square, first + n, i + i + 3);
        }
        ++i;
        index_square = 2 * i * (i + 3) + 3;
    }
}
```

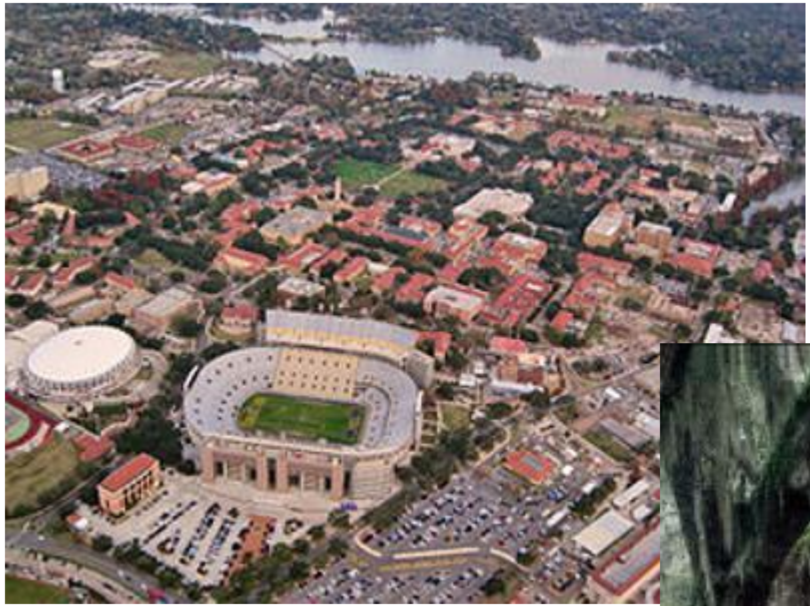
Exercise:

Improve further!

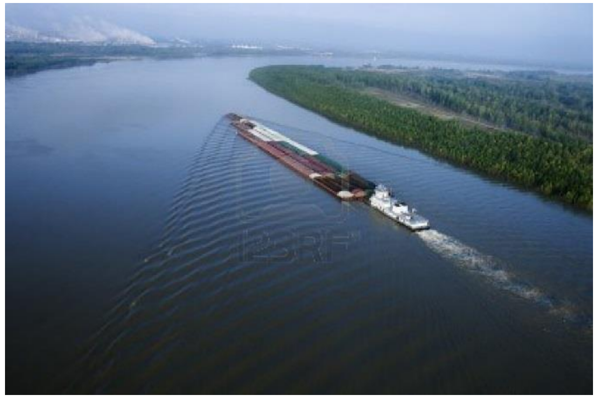
Conclusions

- Rewriting code is an important and constant process
  - Good code is never written in the first attempt
- Why is it important to pay attention to even smaller details?
  - This code in particular is used very widely in cryptography and other fields
- Doing many iterations enables deep understanding of the algorithm
  - This leads to more efficient implementations

- We repeatedly will come back to the Egyptian Multiplication algorithm over course of this semester



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP