Course Context

CSE687 – Object Oriented Design

Jim Fawcett, January 19, 2010

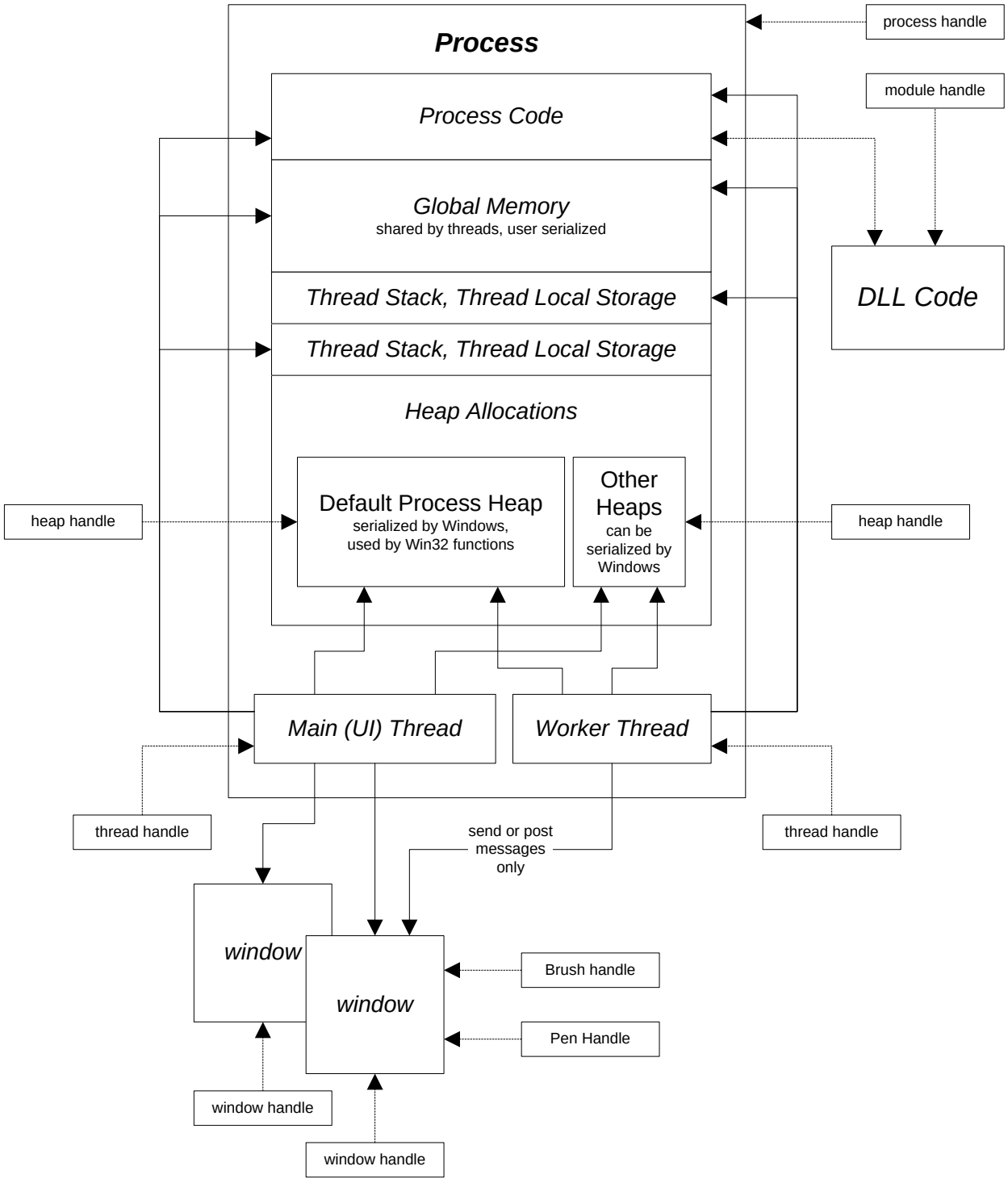
These notes layout a knowledge substrate on which the course is built. We will skim through each of these context areas at the beginning of the course to get you started. We will discuss the details throughout this course.

The purpose of CSE687 – OOD is to provide a sophisticated view of Software Design in general and Object Oriented design in particular. But software design does not happen in a vacuum. We need the surgical instruments that a good programming language provides to turn a design into an effective implementation. Moreover, a programming language shapes the design techniques we use and our ideas about design. We also need some background information about the platform:

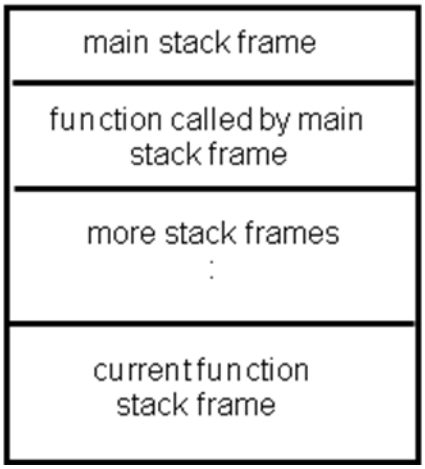
Machine Context:

1. Application = one or more processes
2. Execution images + loaded DLLs = process
3. Threads are basis of execution
    a. Each thread has its own stackframe and thread local storage
    b. All threads in a process share the same address space
4. Handles – references to data structures in the OS kernel. OS API has functions that create, manipulate, and destroy these data structures much like objects.
5. MMFs – memory mapped files used to share code and data between processes
    a. Two processes can share the same MMF and so share memory

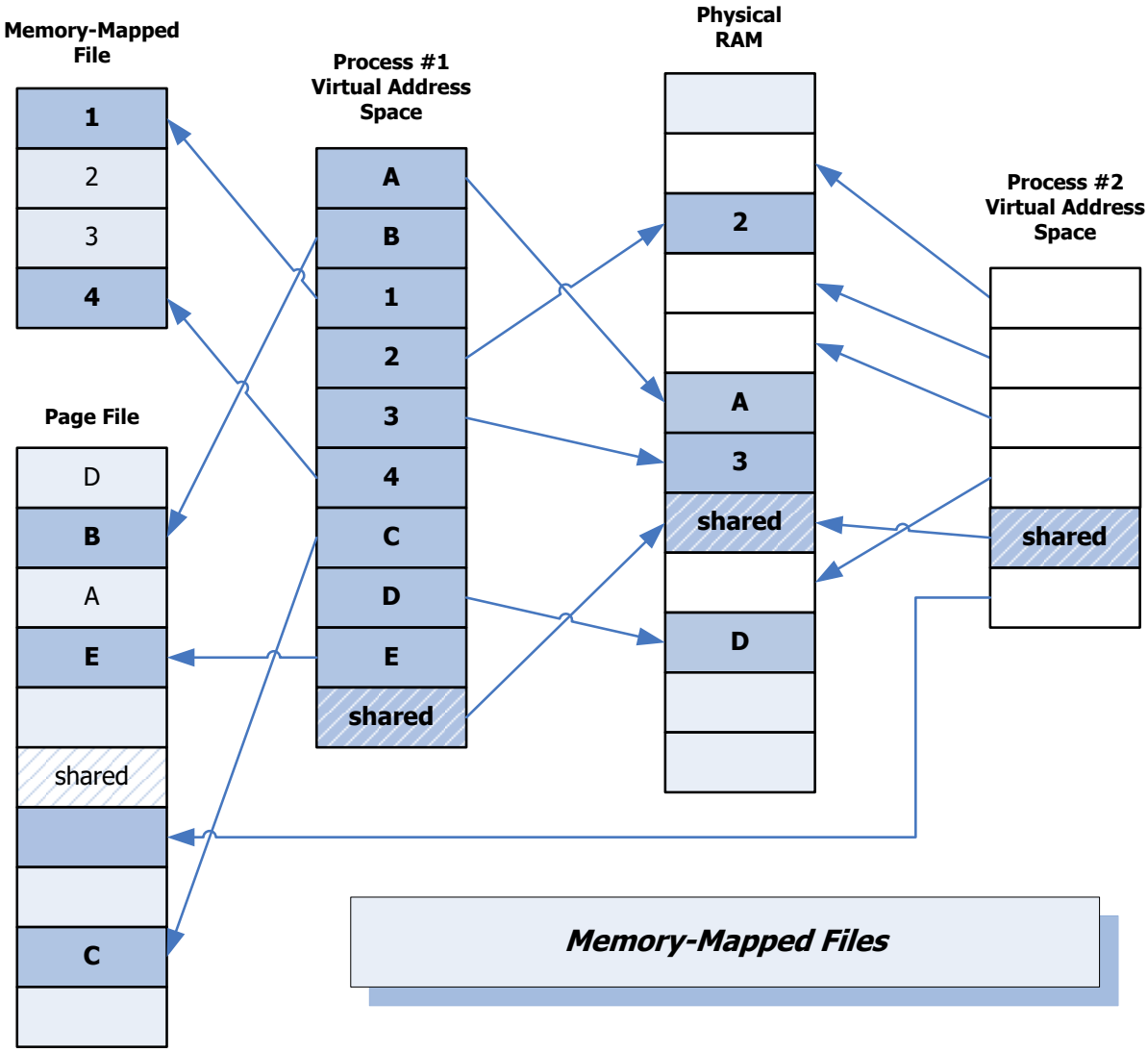
Layout of Processes and Threads



Stack memory: - temporary scratch pad



- defined only while computational thread passes through a function.
- holds input parameters, local data, and return values, used as scratch-pad memory
- guaranteed to be valid during the evaluation of a containing expression, won't be valid after
- expression evaluation starts with function evaluation first, then expression evaluation as algebraic combination of terms
- stack frame is destroyed when expression evaluation is complete


Memory-Mapped File
1
2
3
4
Page File
D
B
A
E
shared
C
Process #1 Virtual Address Space
A
B
1
2
3
4
C
D
E
shared
Physical RAM
2
A
3
shared
D
Process #2 Virtual Address Space
shared
Memory-Mapped Files

C++ Language Context:

1. Memory model
    a. Static
        i. Holds global data, static data, and code
        ii. Lifetime is life of process
    b. Stack – composed of stack frames, one for each thread in process
        i. Scratchpad memory for function invocations and block statements
        ii. Curly braces "{" and "}" define a scope. Each scope gets its own stack frame.
        iii. Lifetime is life of invocation
        iv. C++ code calls constructor at declaration and calls destructor when leaving stack frame scope
    c. Heap
        i. Memory resources used by application
        ii. Lifetime is time between calls to new and delete
2. Compilation model
    a. Preprocessor generates composite source
        i. C++ source for one implementation file (.cpp)
        ii. C++ source for each included header file (.h) placed at the location of the include statement
    b. Compiler generates one obj file for each compilation of an implementation file
    c. Linker builds execution image (exe), dynamic link library (dll), or static library (lib) from a set of objs. This is a process of code deposition and resolution of all inter package references.
    d. Binding – process of associating a memory address with a name in the code text
        i. Values: associates name of a type with memory location sized to hold instances
        ii. Functions: associates an invocation with memory location of code to execute
    e. Early binding
        i. Inline functions are partially compiled but code is not deposited in client code in another package
        ii. Template classes and functions are validated but no code is generated since the type is not known until a client declares an instance.
        iii. All other code is translated into object form. That has unresolved addresses for all references outside the package (compilation unit). References inside the package are fully resolved.
    f. Late binding:
        i. Object code for all inline functions is deposited at the site of the invocation.
        ii. Template code is compiled and deposited for all client declarations and invocations.
3. Computational Model

C/C++ Memory Model Contents

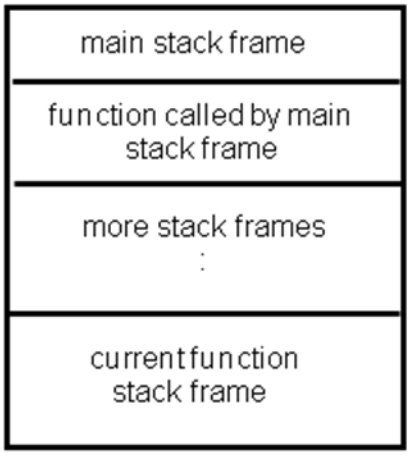
Static memory: - available for the lifetime of the program

defined outside any function (globals) and initialized before main is entered.

global data and functions are made private by qualifying as static, otherwise they are public

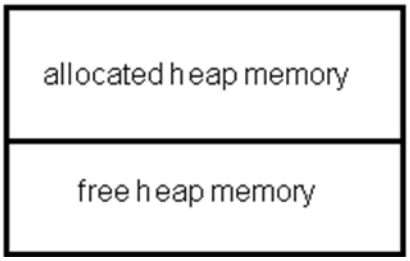
memory allocations local to a function, but qualified as static

Stack memory: - temporary scratch pad



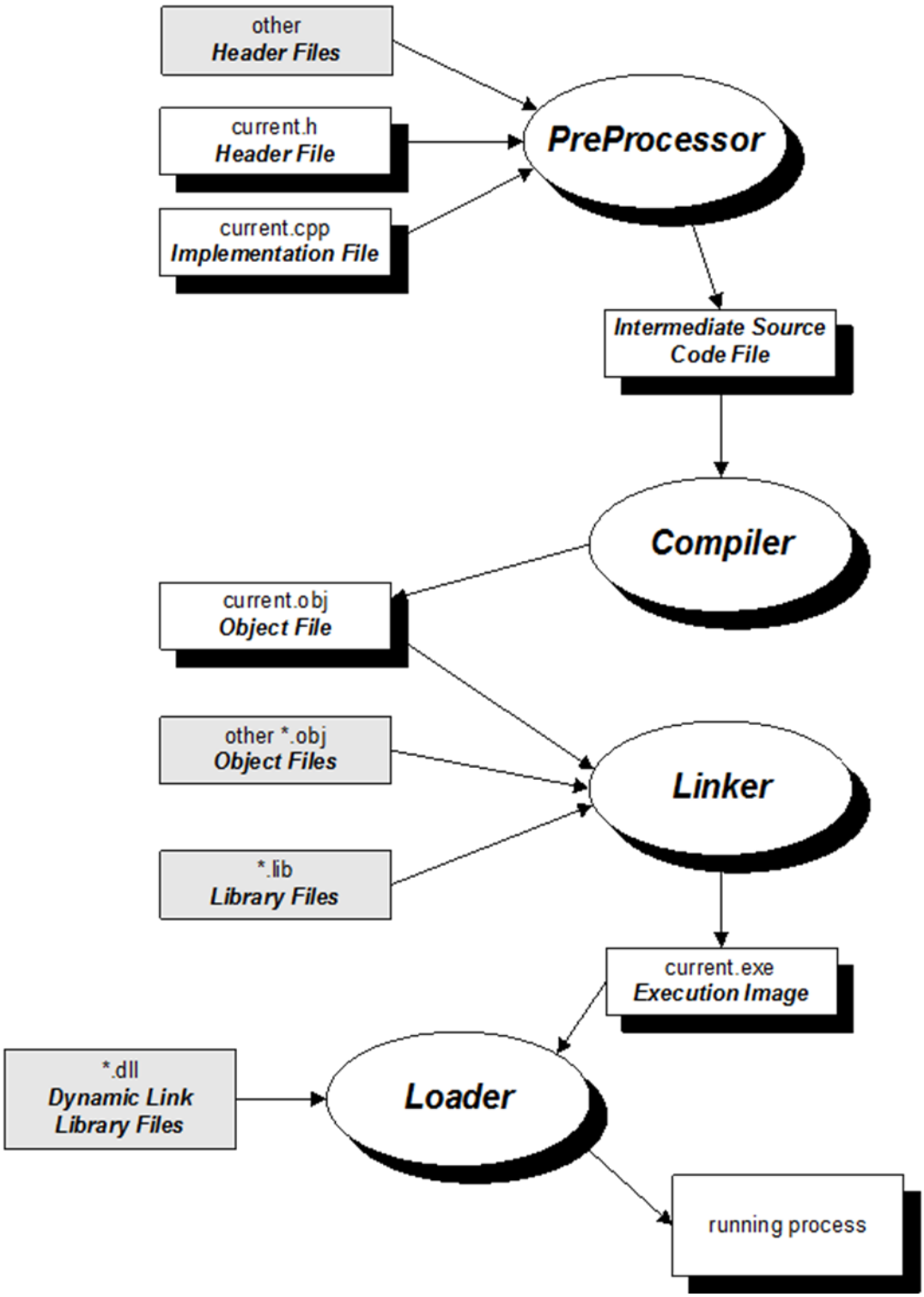
- defined only while computational thread passes through a function.
- holds input parameters, local data, and return values, used as scratch-pad memory
- guaranteed to be valid during the evaluation of a containing expression, won't be valid after
- expression evaluation starts with function evaluation first, then expression evaluation as algebraic combination of terms
- stack frame is destroyed when expression evaluation is complete

heap memory: - valid from the time of allocation to deallocation

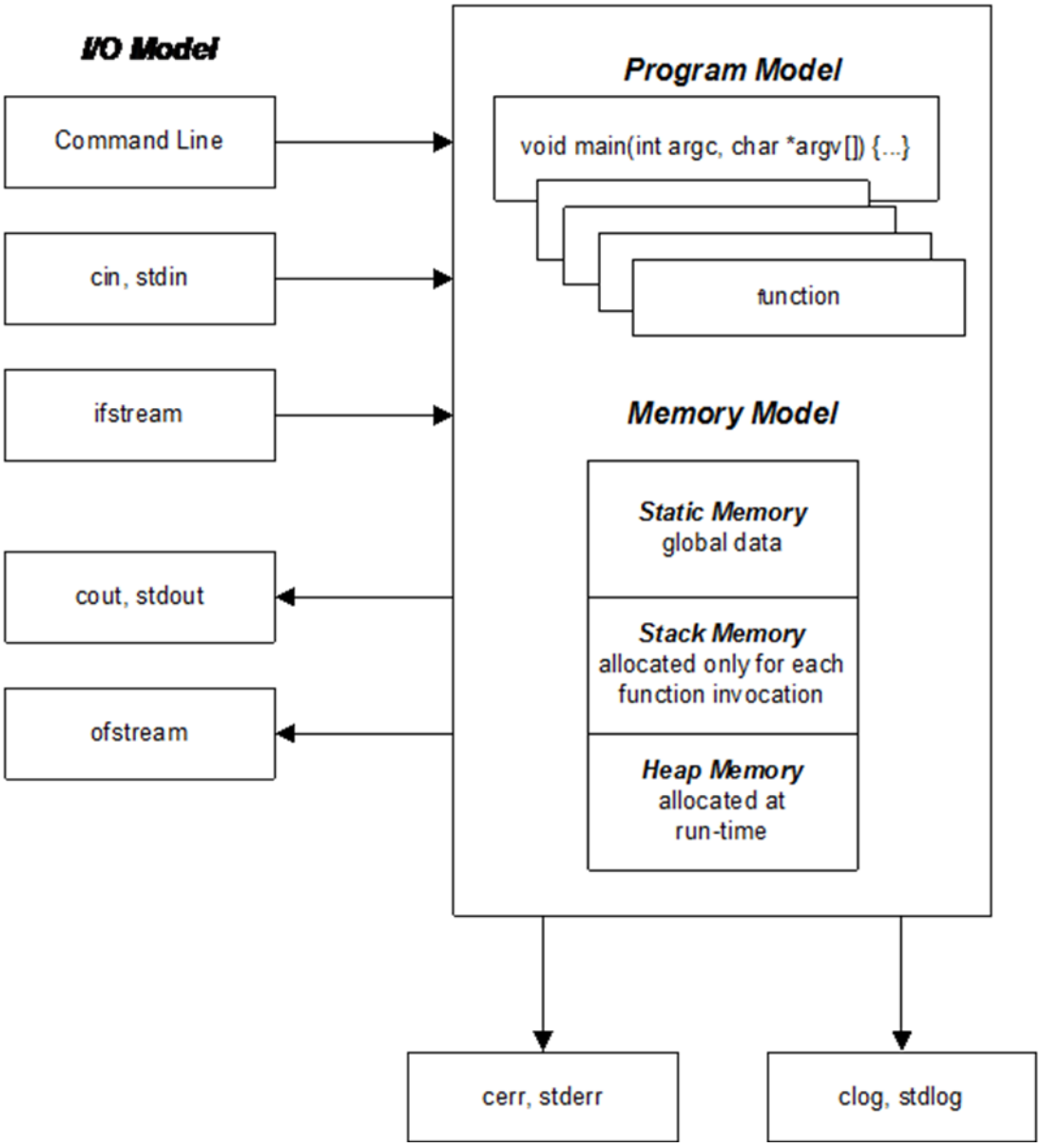


- allocated/deallocated at run time by invoking operators new/delete (or functions malloc/free)
- memory is available to anyone with a pointer to the allocated memory from the time of allocation until deallocated.

C/C++ Compilation Model Contents

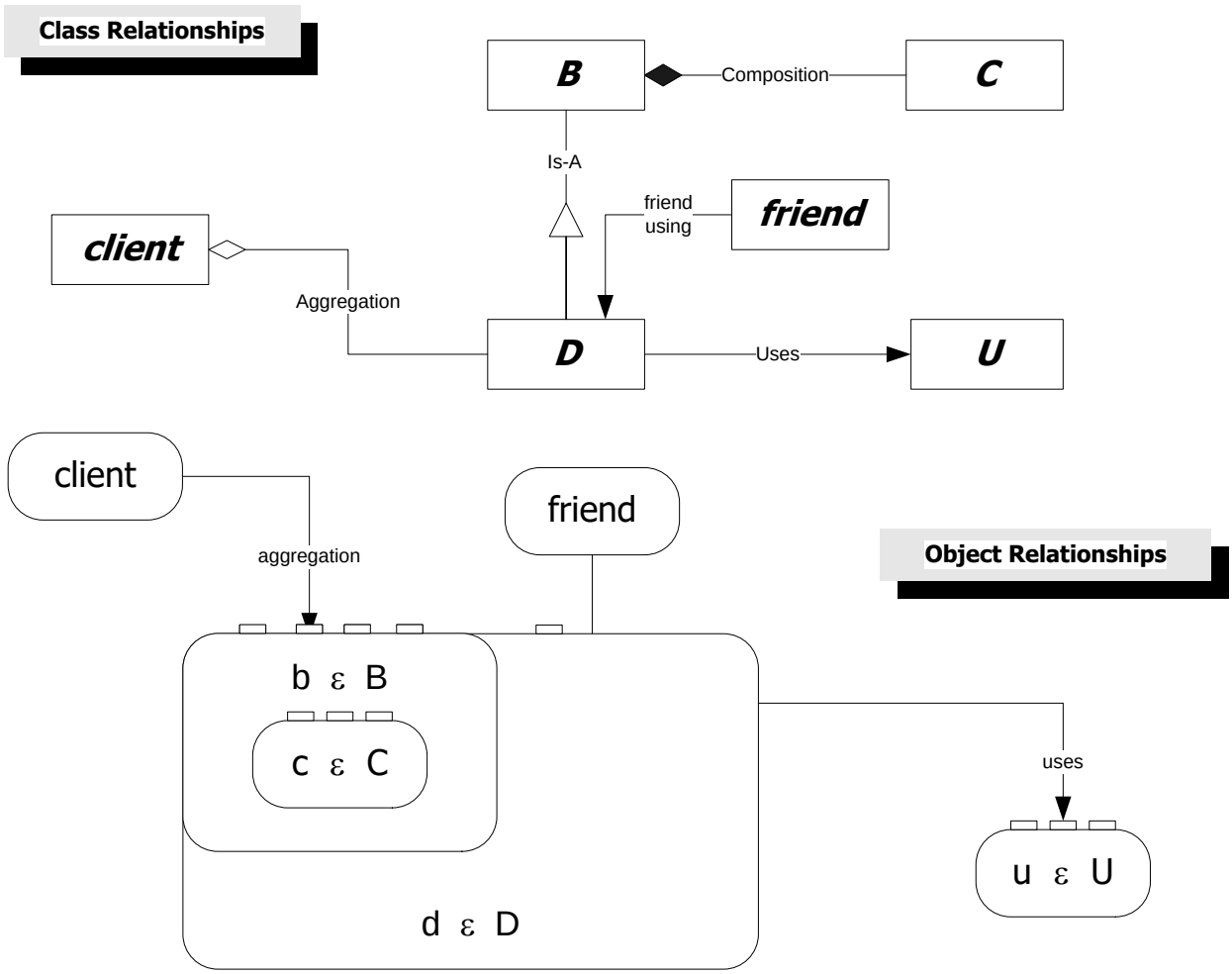


C/C++ Computational Model Contents



Object Oriented Design Context

1. Class is an encapsulated service
2. Class relationships
    a. Strong composition (holds data member) – object contains composed object
    b. Weak composition or Aggregation – object holds pointer or reference to another object
    c. Inheritance – class derives from one or more base classes
        i. Public inheritance supports substitution of derived types. It creates a specialization of the base type.
        ii. Public, protected, and private inheritance support sharing code and resources defined and held by base classes
    d. Using – member functions take an instance of a used type on which to operate
3. Types
    a. Primitives: char, int, double, …
    b. class, struct, enum
    c. Qualifiers: const, static, volatile, *(pointers), &(C++ references)
    d. Arrays of primitives or instances of classes or instances of structs
    e. Classes and structs can hold instances or references to any of the above.
4. Templates: like text substitution at late compile time
    a. Actually much deeper than that as we will see when Template Metaprogramming is discussed.
    b. Intended to have only one source for code that only depends on a specified type. That type is not specified in the template design, but is specified when a client uses the template.
    c. Also intended to replace macros, which are a source of problems in many contexts.
5. Typedefs – type aliases
    a. Used to provide universal names for template types that are unknown at template compile time, e.g., early binding.
    b. Used to provide short names for verbose template syntax


Class Relationships
B
Composition
C
Is-A
friend
using
friend
client
Aggregation
D
Uses
U
client
friend
aggregation
Object Relationships
b ε B
c ε C
uses
u ε U
d ε D