Separate Compilation

Slides

Comparison

Single Source File Programs

- Programs where all the code is contained within 1 file.
- Large programs results in problems.

Disadvantages

- Very long compile time
- Errors requires recompilation of entire program
- Difficult to edit

Multi-File Programs

- Code for a program is stored in several source files.

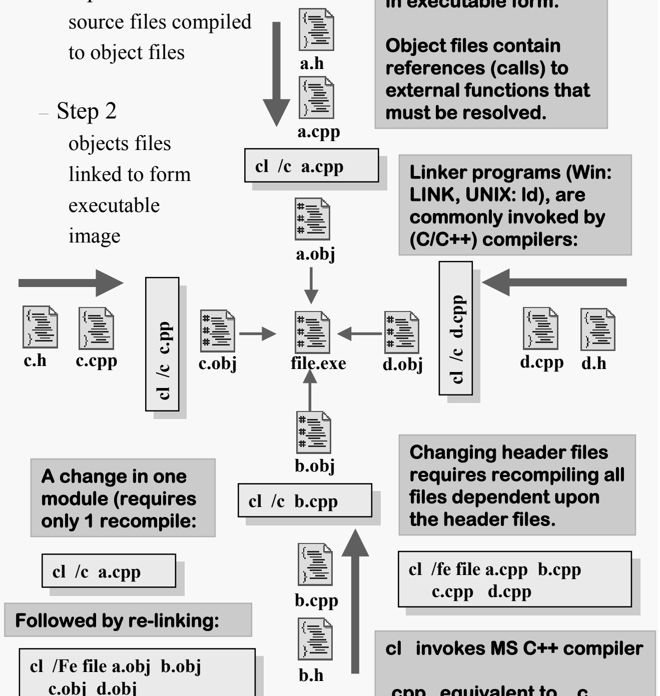
Advantages

- Decreases re-compile time for errors.
  - † Modification of code in one file does NOT require compilation of other files, (exception: if function interfaces have changed).
- Programs can be broken into smaller, simpler subsystems.
- Separate compilation helps support structured methods and modular decomposition for developing large systems.
- Allows languages to be used in conjunction with other programming languages.
- Eases testing in large systems.
- Allows access to system functions, code libraries and packages.
- ♻ Facilitates code reusability.

Compilation Files

Separate Compilation Steps

- Step 1
  source files compiled to object files
- Step 2
  objects files linked to form executable image



Object files contain machine code but not in executable form.

Object files contain references (calls) to external functions that must be resolved.

Linker programs (Win: LINK, UNIX: ld), are commonly invoked by (C/C++) compilers:

```
cl /fe file a.cpp b.cpp
   c.cpp  d.cpp
```

Changing header files requires recompiling all files dependent upon the header files.

A change in one module (requires only 1 recompile:

```
cl /c a.cpp
```

Followed by re-linking:

```
cl /Fe file a.obj b.obj
   c.obj d.obj
```

cl invokes MS C++ compiler

.cpp equivalent to .c
.obj equivalent to .o

Module Structure

C++ Module Components

- Two Files:

a.h

† **Interface File**: (header file .h) contains *public* declarations of all articles that are accessible (visible) and usable by external modules that include the interface file.

‡ Articles consist of constants, typedefs, enum types, class/struct declarations and function prototypes (only parameter types need to be specified, parameter names included are ignored by the compiler).

a.cpp

† **Implementation File**: (code file .c or .cpp) contains *private* declarations and definitions of all articles that are inaccessible (invisible) and **NOT** usable by external modules that include the interface file.

‡ The full declarations of the function protypes given in the header file are specified.

‡ Articles declared in the implementation file that are NOT declared in the interface file are considered local/internal to the module and can only be accessed by the module's code **NOT** by external code.

Security Issues

Traditional C Function Declarations

- Parameter type lists cannot be included in Fn declarations

```
int fn( );
```

- ANSI C compilers will accept traditional C Fn declarations.
- Compiler does NOT know the types of the parameters.
- Compiler cannot perform type checking on the arguments.
- Compiler cannot perform coercion/conversion/promoting between parameters & arguments.

```
longint = fn(longint);
```

† above call will yield incorrect results

Mixing Traditional C & C++ Function Declarations

- Prototypes are required in C++
- In defining functions and their prototypes, using void in the parameter types lists is optional:

```
void    fn();   «=»    void    fn( void );
```

- `void` is NOT a keyword in traditional C

```
int fn();
```

- The above declaration specifies that fn accepts an unknown number of arguments.

Security Issues (continued)

Traditional C Function Invocation Declarations

- Function calls prior to fn definitions results in default declarations.
- Example:

```
fn(x);     /* encountered before fn declaration */
```

- Assumed default declaration:

```
int fn();
```

- Parameter list attributes are unknown.
- No type checking or coercion/conversion can be performed.

Traditional C Function Definitions & Invocation

- Traditional C fn declaration/definition prior to invocation:

```
int     fn(x);
long    x;
{ - - - }
```

- Traditional “C” compilers still treat the parameter list attributes as unknown.
- Programmer has the responsibility for ensuring that the correct number and type of arguments are passed.

C++ Function Prototypes

- Compiler performs type checking of parameters & arguments.

Global Concerns

Declarations & Definitions

- Declarations give only the attributes of identifiers (type).
- Definitions give the attributes of identifiers and reserves storage.
- Identifiers can only be defined once in a program, but may be declared multiple times.
- Definitions should **NOT** be placed in header files.

External Definitions

- Definitions that occur outside all functions in a file.
- Scope extends to the end of the file.
- Cannot be accessed outside of file, unless declared as an **extern** identifier in separately compiled files.

```
extern  int   x;
extern  void  fn(long);
```

Array sizes must be given in definitions but are optional in declarations.

- Extern declarations are NOT definitions (no storage is reserved, no initialization can be performed).

Global (separate compilation) Variables

- extern declarations allow for common storage across compilation units.
- extern declarations are to ***avoided at all costs*** due to the same problems inherent in their use in single file programs.

Header Inclusion Problem

Problem Statement

- Separate files may use the same header file.
  - † Assume a programmer has stored system wide constants in: **const.h**
  - † Assume **const.h** is included in the modulea header file: modulea.h
  - † Assume moduleb includes modulea.h and **const.h**.
- **const.h** header declarations would be duplicated in moduleb.cpp after preprocessing

Solution: Conditional Compilation

- Preprocessor directives:

```
#if       #ifdef  #ifndef #elif   #else
#endif    #undef  #define
```

- Usage: const.h

```
#ifndef   CONST_H
#define   CONST_H

          type definitions,
          constants, etc...

#endif
```

CONST_H is a preprocessor identifier not a C/C++ identifier

- Inclusion of const.h instructs the preprocessor to check if CONST_H has been previously defined during preprocessing. If it has not then it is defined and the const.h declarations are copied into the source, otherwise no inclusion occurs.

Portability

`defined` Preprocessor Directive

```
defined   < identifier >
```

- Evaluates to true if the identifer has been previously defined in the preprocessing.

Platform Specific Compilation

```
#define WIN9x    or    #define WINNT2K
       .          .          .
#if defined(WIN9x )
  // WIN9x specific code
#elif defined(WINNT2K)
  // WINNT2K specific code
#else
                .       .       .
#endif
```

Preprocessor directives other than those covered here are available.

Debugging

- Can be utilized to compile or skip diagnostic output statements
- Two Methods:

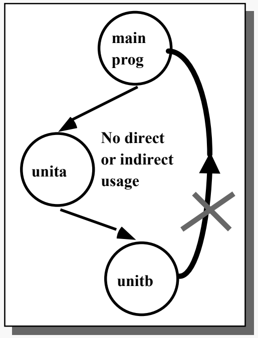
```
#define    DEBUG     1
            .         .         .
#if  DEBUG
   cout << "Debug:" << string1 << endl;
#endif
```

```
#define    DEBUG
            .         .         .
#ifdef  DEBUG
   cout << "Debug:" << string1 << endl;
#endif
```

Module Scope

Lifetime

- Variables declared above all functions in a source file (file-scoped) and inside of main() exist throughout the execution (life) of the program — since main() drives all other routines.
- All other variables declared inside of functions exist only for the lifetime (execution) of the function (automatic) — excluding static (and extern).



- Restricted Access
- Given the diagram at the right:
- The main program only needs to call routines in unit A, and does not require any access to routines, consts, or other elements in unit B.
- Unit A needs only to call the routines in unit B.

```
//main.cpp
#include "main.h"
#include "const.h"
#include "unita.h"
void main()
{

}
```

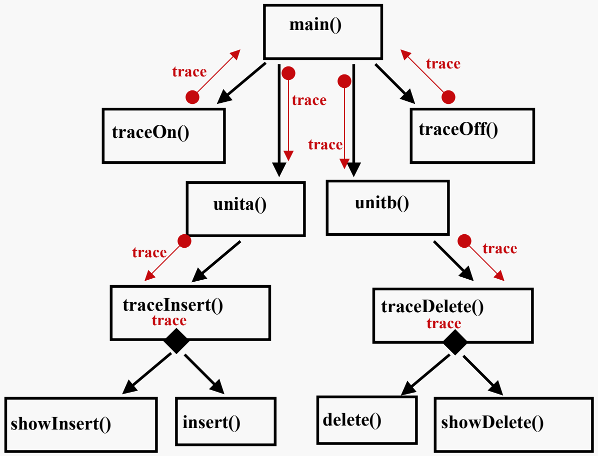
```
// unita.cpp
#include   "const.h"
#include   "unita.h"
#include   "unitb.h"
```

```
// unitb.cpp
#include   "const.h"
#include   "unitb.h"
```

Design Problem

Restricted Access

- Problem: Special graphical trace/debug routines need to be executed at particular times during system execution.
  - † Trace flag is not needed for unitA or unitB.
  - † Complicates interfaces and exposes trace to possible or accidental misuse.



- Solution 1: Make Trace global. Simplifies interfaces but does not solve scope problem.
- **Solution 2**: Hide trace in the private declarations of a separately compiled module, limit access to module functions.

Code Redesign

tracer.h

```
#ifndef   TRACER_H
#define   TRACER_H
  void initTrace(void) ;
  void traceOn(void) ;
  void traceOff(void) ;
  bool traceActive(void);
#endif
```

tracer.cpp

```
#include "tracer.h"
bool trace;
void initTrace(void)
          { trace = false; }

void traceOn(void)
          { trace = true; }

void traceOff(void)
          { trace = false; }

bool traceActive()
          { return trace; }
```

Localized 'global' variable (file scoped)

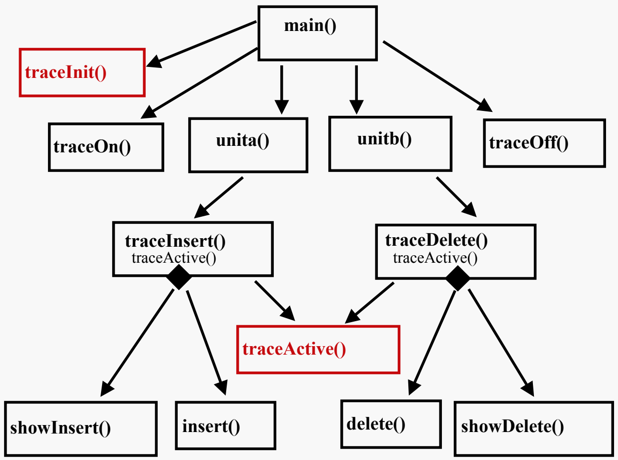
trace can be accessed and changed in any function in tracer.cpp, but NOT in any code external to tracer.cpp

Achieves information hiding/encapsulation that a class implementation would provide.

Redesign Solution

Tradeoff

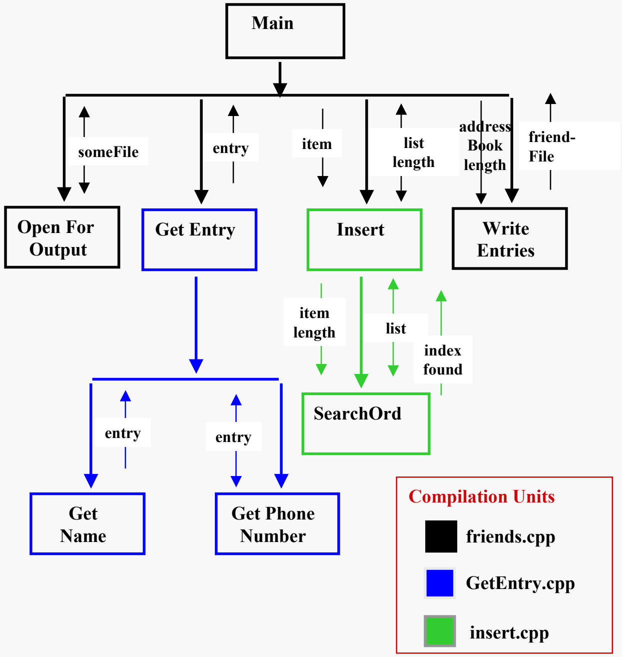
- Interface simplification achieved at increase of number of system modules.



Example: Addresses

Reference

- adapted from: *"Programming and Problem Solving with C++"*, N. Dale, C. Weems & M. Headington, D.C. Heath, ©1996, 794-808.



Example: C++ Code #1

globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

#include <iostream>
#include <iomanip>    // For setw()
#include <fstream>    // For file I/O
#include <cctype>     // For toupper()
using namespace std;

const int MAX_FRIENDS = 150;    // Max number of friends

typedef char String8[9];        //  8 characters plus '\0'
typedef char String15[16];      // 15 characters plus '\0'

struct EntryType
{
    String15 firstName;
    String15 lastName;
    int      areaCode;      // Range 100..999
    String8  phoneNumber;
    int      month;         // Range 1..12
    int      day;           // Range 1..31
    int      year;          // Range 1900..2100
};

void OpenForOutput( ofstream& );
void WriteEntries( const EntryType[], int, ofstream& );

#endif
```

globals.h does NOT correctly model the structure chart program design.

Example: C++ Code #2

friends.cpp

```
//****************************************************
// Friends program
// This program creates an address book by reading
// first names, last names, phone numbers, and birth
// dates from standard input and writing an alphabetical
// listing to an output file
//****************************************************

#include "globals.h"     // global constants & types
#include "GetEntry.h"    // address book building
#include "insert.h"      // insert sort of address book


int main()
{
    EntryType addressBook[MAX_FRIENDS]; // friends' recs

    int       length = 0; // Number entries addressBook
    EntryType entry;      // Current rec being read
    char      response;   // Response char
    ofstream  friendFile; // Output file entries

    OpenForOutput(friendFile);
    if ( !friendFile )
        return 1;



//   . . .

}
```

Example: C++ Code #3

GetEntry.h

```
//*******************************************************
// Friends program
// GetEntry.h header file
//*******************************************************

#ifndef GetEntry_H
#define GetEntry_H

#include "globals.h"    // For global constants & types

void GetEntry( EntryType& );
void GetName( EntryType& );
void GetPhoneNumber( EntryType& );

#endif
```

Why does GetEntry.h incorrectly model the design of the program structure chart?

GetEntry.cpp

```
#include "GetEntry.h"    // For header file

// *******************************************************

void GetEntry( /* out */ EntryType& entry )
{
    GetName(entry);
    GetPhoneNumber(entry);
    cout << "Enter birth date as 3 integers, separated by"
         << " spaces: MM DD YYYY" << endl;
    cin >> entry.month >> entry.day >> entry.year;

//  . . .
}
```

Example: C++ Code #4

insert.h

```
// *****************************************************
// Friends program
// insert.h header file
// *****************************************************

#ifndef INSERT_H
#define INSERT_H

#include <string.h>   // For strcmp()
#include "globals.h"  // For global constants & types

void Insert( EntryType[], int&, EntryType );
void SearchOrd( EntryType[], EntryType, int,
                int&, bool& );

#endif
```

Why does insert.h incorrectly model the design of the program structure chart?

Should `#include`s be placed in the “header.h” files or the “source.cpp” files?

insert.cpp

```
// *****************************************************
// Friends program
// insert.cpp file
// *****************************************************

#include "insert.h"   // For header file

//*****************************************************

void Insert( /*inout*/ EntryType list[],  // Changed List
             /*inout*/ int&      length,  // List Length
             /*in*/    EntryType item   ) // Insert Item
{
// Inserts item into its proper place in the sorted list
// . . .
}
```

Static Array Class Header.h

```
// Array.h:  a static array of Item variables
// This class provides an encapsulation of a fixed size array.
// The Items are stored in ascending order; it is assumed that
// the relational operators are provided for type Item.

#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>
#include <iomanip>
using namespace std;

#include "Item.h"

const int SIZE    = 1000;
const int MISSING = -1;

class Array {
private:
   int     Capacity;    // max # of elements list can hold
   int     Usage;       // # of elements currently in list
   Item    List[SIZE];  // the list

   void ShiftTailUp(int Start);    // implementation exercise
   void ShiftTailDown(int Start);  // implementation exercise

public:
   Array(Item Init = Item()); // Each cell stores copy of Init
   int  getCapacity() const;  // retrieve Capacity
   int  getUsage() const;     //          Usage
   bool isFull() const;       // ask if List is full
   bool isEmpty() const;      //              or empty
   // insert newValue if not a duplicate
   bool Insert(const Item& newValue);
   bool DeleteAtIndex(int Idx); // delete element at index Idx
   // find index of first occurrence of given value
   int  Locate(const Item& toFind) const;
   Item Get(int Idx);          // get element at index Idx
                               // set element at index Idx
   void Set(int Idx, const Item& newValue);

   ~Array();                    // destroy Array object
};

#endif
```

User must supply Item.h file which contains the declaration for the Item type.

Static Array Class Source.cpp

```
// Array Class Implementation
//
#include "Array.h"

#include <cstdlib>        // for NULL
#include <cassert>
using namespace std;

//////////////////////////////////////////////////// constructor

//////////////////////////////////////////////////// Array()
// Initializes list entries to the specified value.
//
// Parameters:
//    Init       value to be stored in each cell
//
// Returns:   none
//
// Pre:       Init has been initialized.
// Post:      Array object has been constructed with List[]
//               of dimension SIZE, usage zero, and each
//               cell holding a copy of Init.
//
// Calls:     none
// Called by: none
//
Array::Array(Item Init) {

   Capacity = SIZE;
   Usage    =    0;

   int Idx;
   for (Idx = 0; Idx < Capacity; Idx++)
      List[Idx] = Init;
}
```

Static Array Class Source.cpp

```cpp
////////////////////////////////////////////////// insert functions

////////////////////////////////////////////////// Insert
// If newValue does not occur in List[], inserts it in
// correct (ordered) position.
//
// Parameters
//     newValue   Item variable to be inserted.
//
// Returns:   true if insertion succeeded; false otherwise
//
// Pre:       newValue has been initialized.
// Post:      If List[] is full no changes are made.
//            Otherwise, if newValue does not occur in
//               List[] it is inserted at the proper index
//               and the Usage is adjusted accordingly.
//
// Calls:     ShiftTailUp()
// Called by: none
//
bool Array::Insert(const Item& newValue) {

   if (Usage == Capacity) return false; // array is full

   int Idx = 0;
   while ( (Idx < Usage) && (newValue > List[Idx]) ) {
      Idx++;
   }

   if (newValue == List[Idx]) return false;  // already present

   ShiftTailUp(Idx);
   List[Idx] = newValue;
   Usage++;
   return true;
}
```

Assumes operator overloading.

Static Array Class Source.cpp

```cpp
////////////////////////////////////////////////// delete functions

////////////////////////////////////////////////// DeleteAtIndex
// Deletes element at specified index, if possible.
//
// Parameters
//     Idx    index of item to be deleted
//
// Returns:   true if deletion is carried out, false otherwise
//
// Pre:       0 <= Idx < Usage
// Post:      If Idx is not valid, no changes.  Otherwise,
//               the Item variable at Idx has been removed.
//
// Calls:     ShiftTailDown()
// Called by: none
//
bool Array::DeleteAtIndex(int Idx) {

   if ( Idx < 0 || Idx >= Usage) return false;

   ShiftTailDown(Idx);
   Usage--;
   return true;
}
```

Static Array Class Source.cpp

```
////////////////////////////////////////////// retrieval functions

////////////////////////////////////////////// Locate
// Returns index of first occurrence of specified value
// in List.
//
// Parameters
//     Value  value to be located in List[]
//
// Returns:   If Value occurs in List[], index of
//               first occurrence; -1 otherwise.
//
// Pre:       Value has been initialized.
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
int Array::Locate(const Item& Value) const {

   if (Usage == 0) return MISSING;

   int Idx = 0;
   while (Idx < Usage) {
      if (List[Idx] == Value)
         return Idx;
      Idx++;
   }
   return MISSING;
}
```

Assumes operator overloading.

Static Array Class Source.cpp

```
////////////////////////////////////////// Get
// Returns Item at List[Idx].
//
// Parameters
//     Idx    index of desired Item variable
//
// Returns:   If Idx is valid, reference to List[Idx];
//               -1 otherwise.
// Pre:       0 <= Idx < SIZE
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
Item Array::Get(int Idx) {

   assert (Idx >= 0 && Idx < SIZE);

   return List[Idx];
}

////////////////////////////////////////// Set
// Sets element at List[Idx].
//
// Parameters
//     Idx    index of desired Item variable
//  newValue  Item to be inserted in array
//
// Returns:   none
// Pre:       0 <= Idx < SIZE
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
void Set(int Idx, const Item& newValue);

   assert (Idx >= 0 && Idx < SIZE);

   List[Idx] = newValue;
}
```

Static Array Class Source.cpp

```
//////////////////////////////////////// array property accessors

//////////////////////////////////////// getCapacity
// Returns Capacity (dimension) of List
//
// Returns:   number of elements List[] can hold
//
// Pre:       none
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
int Array::getCapacity() const {

   return Capacity;
}

/////////////////////////////////////////////////////// getUsage
// Returns Usage (# cells used) of List
//
// Returns:   number of List cells in use
//
// Pre:       none
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
int Array::getUsage() const {

   return Usage;
}
```

Static Array Class Source.cpp

```
/////////////////////////////////////////////////////// isFull
// Indicates whether List is full.
//
// Returns:   true if List is full; false otherwise
//
// Pre:       none
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
bool Array::isFull() const {

   return (Usage >= Capacity);
}

/////////////////////////////////////////////////////// isEmpty
// Indicates whether List is empty.
//
// Returns:   true if List is empty; false otherwise
//
// Pre:       none
// Post:      no changes.
//
// Calls:     none
// Called by: none
//
bool Array::isEmpty() const {

   return (Usage == 0);
}

/////////////////////////////////////////////////////// destructor

//////////////////////////////////////////////////
// Dummy destructor included for modifiability.
//
Array::~Array() {
}
```