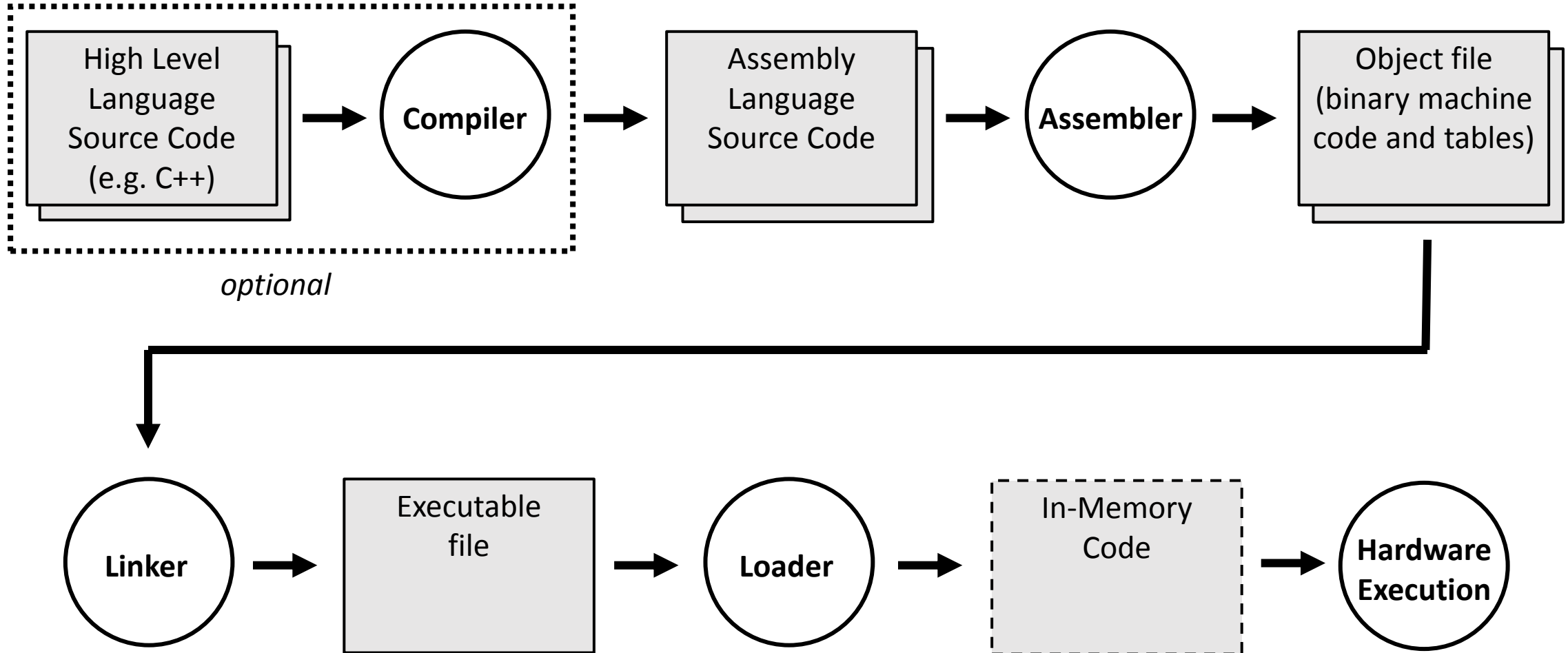


CMPE 220

Week 6 – Linkers, Loaders, and IDEs

Building Software



System Dependent Components

(Compilers not yet discussed)

Assembler	Linker	Loader
<ul style="list-style-type: none">• Parses assembly language source code• Assigns addresses• Assembles instructions• Generates <i>object (*.o) files</i>• Generates listings• Reports errors	<ul style="list-style-type: none">• Links <i>object files</i>• Relocates modules and adjusts addresses• Generates <i>executable file</i>• Reports errors	<ul style="list-style-type: none">• Relocates executable file• Loads executable file• Inserts startup / terminate code• Launches <i>executable file</i>• Reports errors
Operating System		
<ul style="list-style-type: none">• Memory Management• Process Management		
Hardware		
<ul style="list-style-type: none">• Instruction set• Addressing modes• Memory address space		

The Link Step

- Linkers combine multiple object (*.o) files into a single executable file
- Linkers connect symbols defined in one code module (EXTDEFS) with symbol references in another module (EXTREFS)
- The linker may automatically search system libraries for external routines (*automatic linking*)
- Most systems require a *link* step, even for a single object file
- The linker creates a properly formatted executable file
 - Object file (*.o) format differs from executable file format

Historical Aside

- As programs grew larger, it became desirable to break the development into several parts... and even to share parts of programs
- One of the earliest linkers was developed by Grace Hopper in 1952
- She called it a “compiler” because it compiled several parts into one program



Linker Relocation: (review)

Symbol Table

Symbol	Type	Address
LOOP	ADDR	1017
INCR	ADDR	1027
CONT	ADDR	1055
COUNTER	ADDR	1095
MAXINDEX	EQU	300

Symbol Reference Table

Symbol	Type	Reference
EXPO	EXTREF	1021
INCR	REF	1026
LOOP	REF	1030
LOOP	REF	1048
COUNTER	REF	1070
MAXINDEX	REF	1076

- Note: some symbolic information that was important to the assembler is not needed for module relocation by the linker or the loader.
 - May still be important for symbolic debuggers

Module Relocation: Linker

External Symbol Table

Symbol	Type	Address

External Symbol Reference Table

Symbol	Type	Reference
EXPO	EXTREF	1021

Modification Table

Reference
1021
1026
1030
1048
1070
1076

- Modification Table holds the addresses of every instruction that contains an address (and therefore needs to be relocated).
 - $\text{new address} = \text{old address} + \text{new module base address}$

Module Relocation: Linker

External Symbol Table

Symbol	Type	Address

External Symbol Reference Table

Symbol	Type	Reference
EXPO	EXTREF	1021

Modification Table

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	1		0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Modification Table holds a bit flag corresponding to every instruction that contains an address (and therefore needs to be relocated).
 - $\text{new address} = \text{old address} + \text{new module base address}$

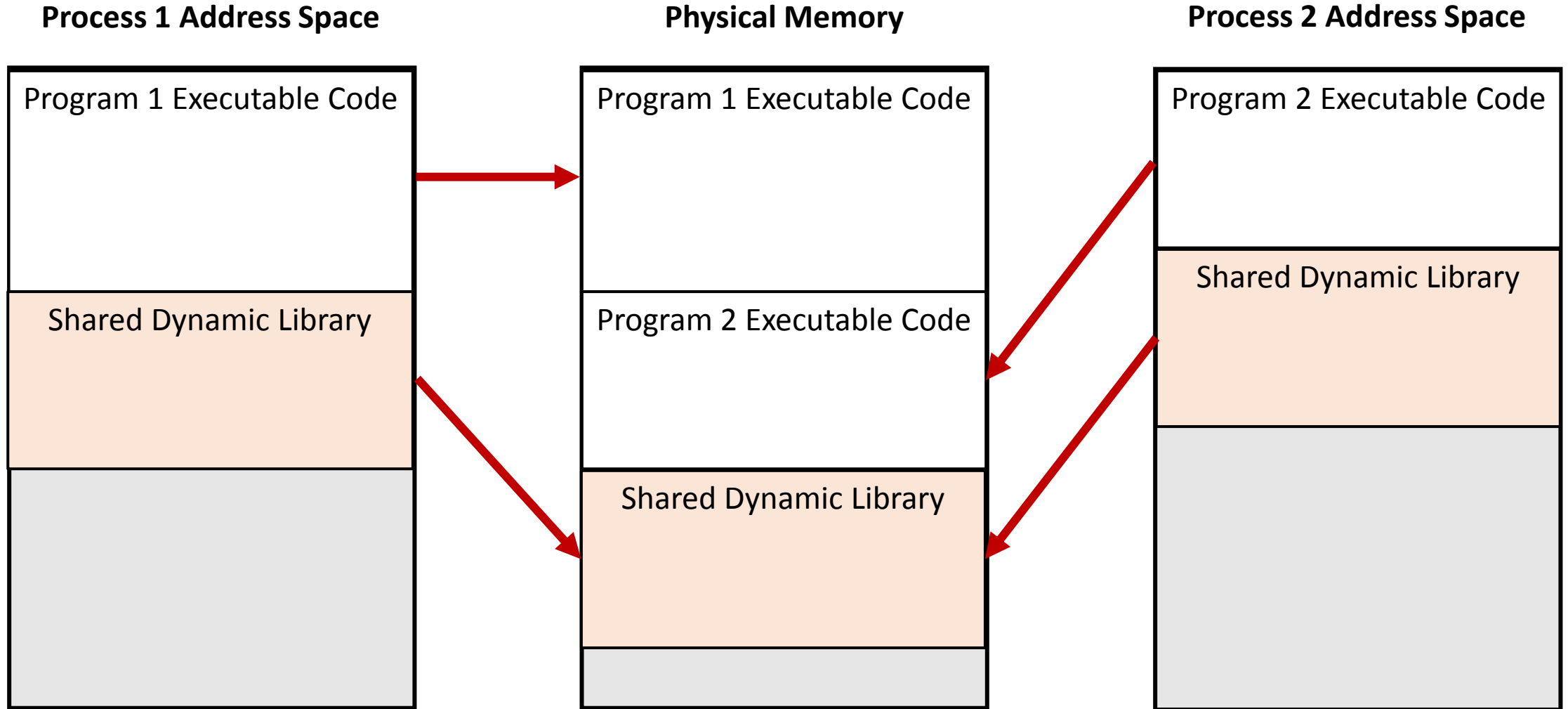
Dynamic Linking

- On some systems, shared libraries are *not* linked into the executable file.
- Instead, the linker inserts a call to a special system function for *dynamic* linking and loading (also called linking on demand).
- Supported, with variations, on most modern platforms:
 - Windows: DLL (Dynamic Link Library) files
 - POSIX & Solaris: ELF (Executable and Linkable Format) files
 - MacOS: .dylib (Dynamic Library) files

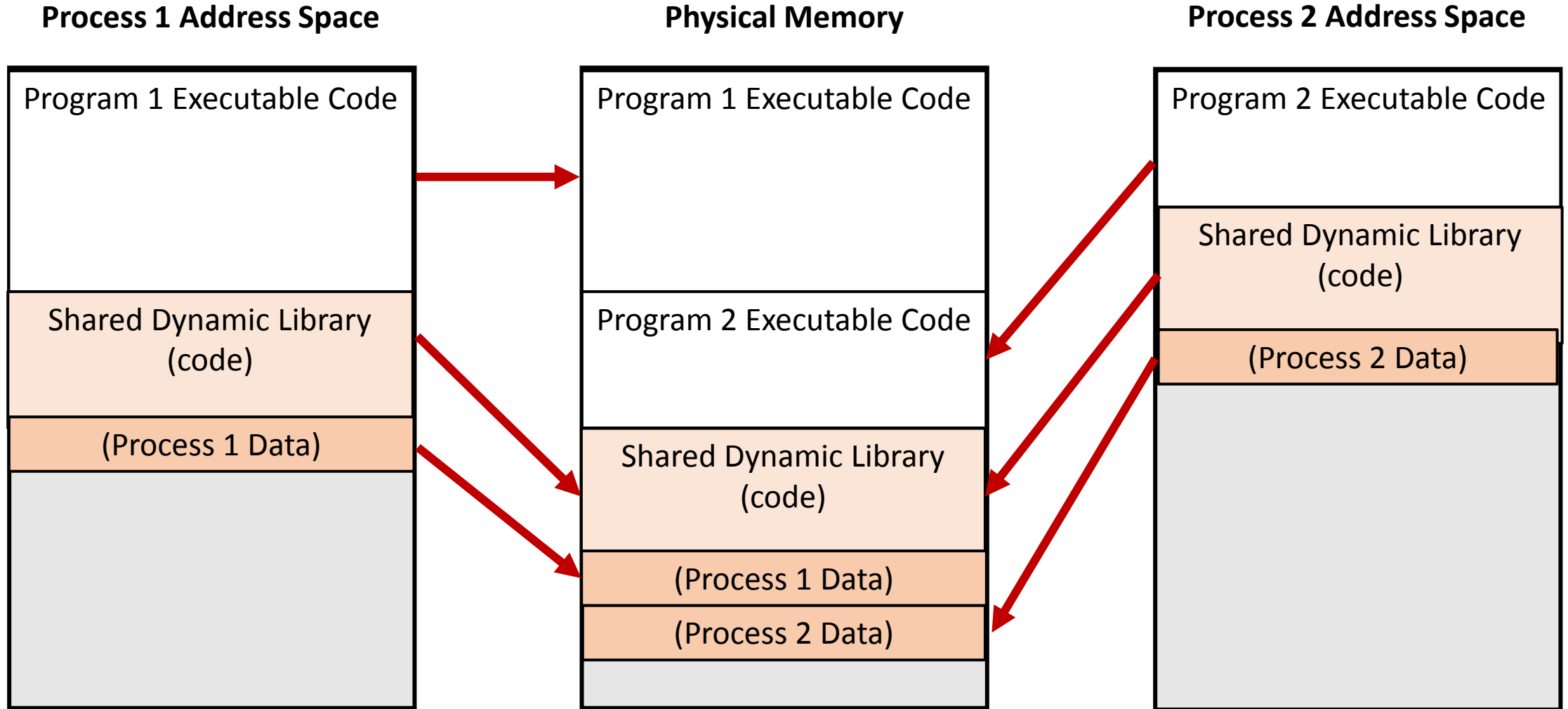
Dynamic Linking Advantages

- Reduces the memory footprint of the executable
 - Loads only routines that are actually used
- Allows run-time binding, so different versions of shared library routines may be used
 - E.g. a version with debugging or tracing code
- A single copy of a dynamic library may be *shared* among processes via MMU mapping.
 - An MMU, or Memory Mapping Unit, is a hardware device that maps addresses in a process address space to actual locations in physical memory

Shared Dynamic Libraries



Shared Dynamic Libraries (Data Separation)

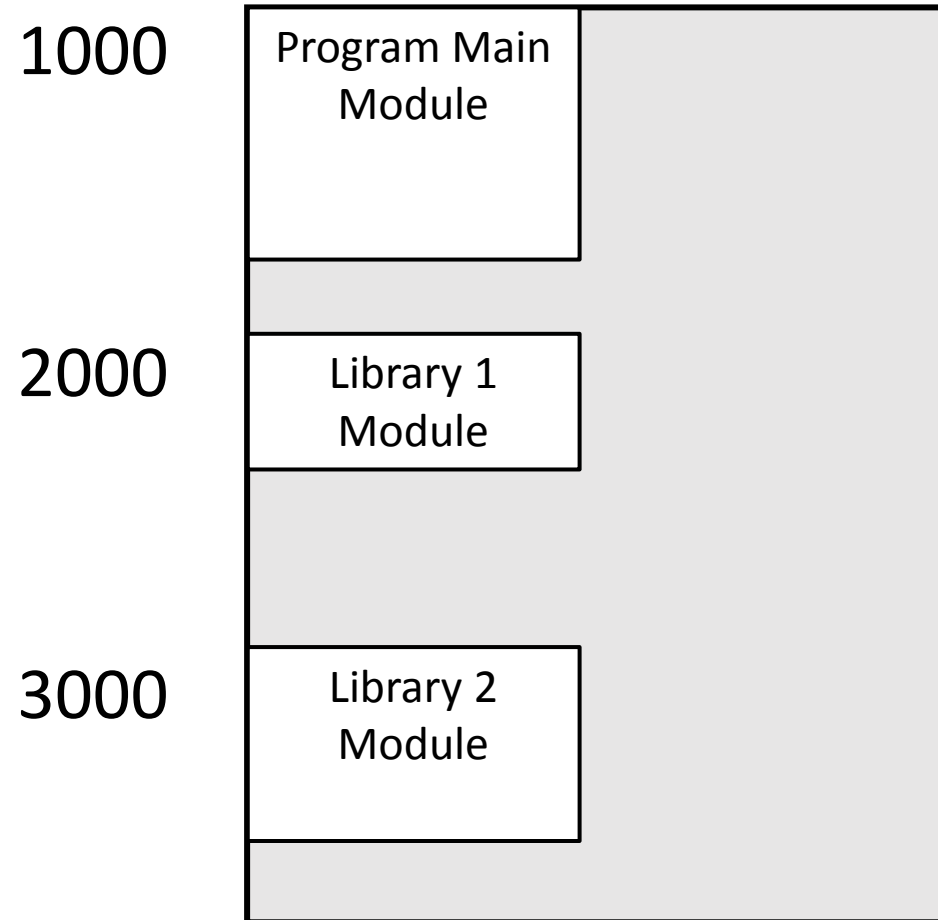


The Load Step

- Loads an executable file into memory
- *May* relocate the file
- Performs any required system initialization
- Launches the program (starts execution)

Absolute Addressing

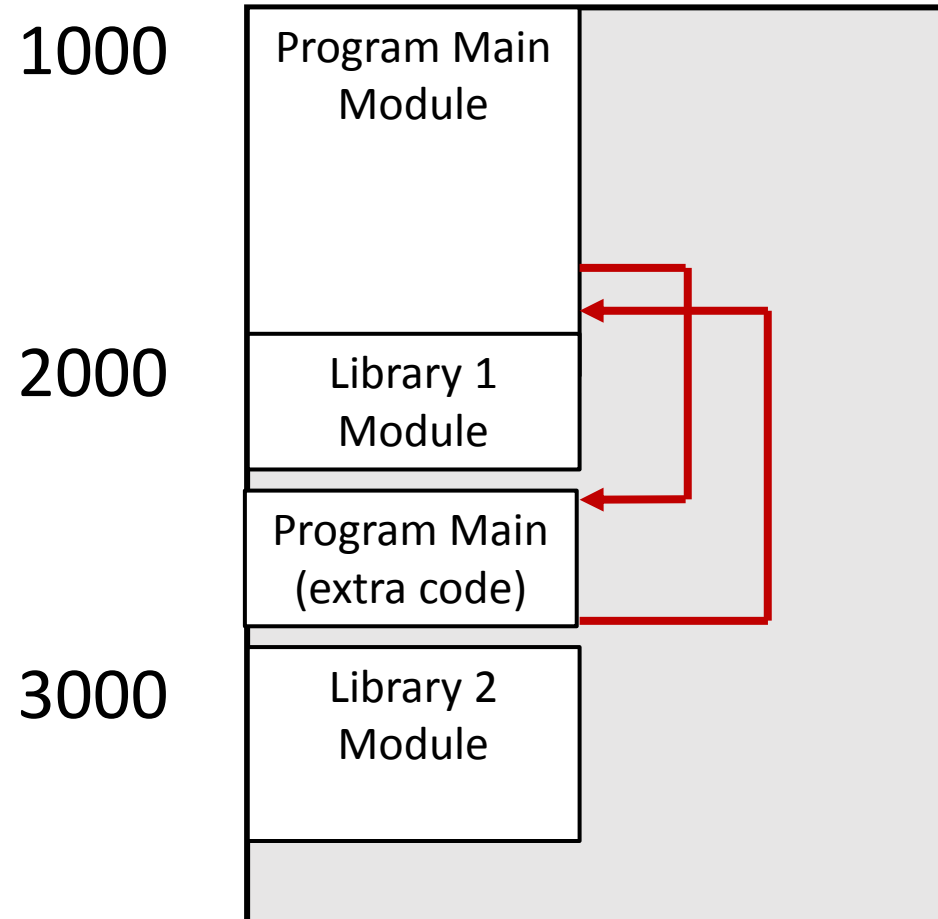
Predefined Addresses



- Very early systems
- Embedded systems
- Routines can be located at pre-designated addresses
- *Separate assembly files*
- No linker required
- *Absolute Loader*

Absolute Addressing Problem: code growth

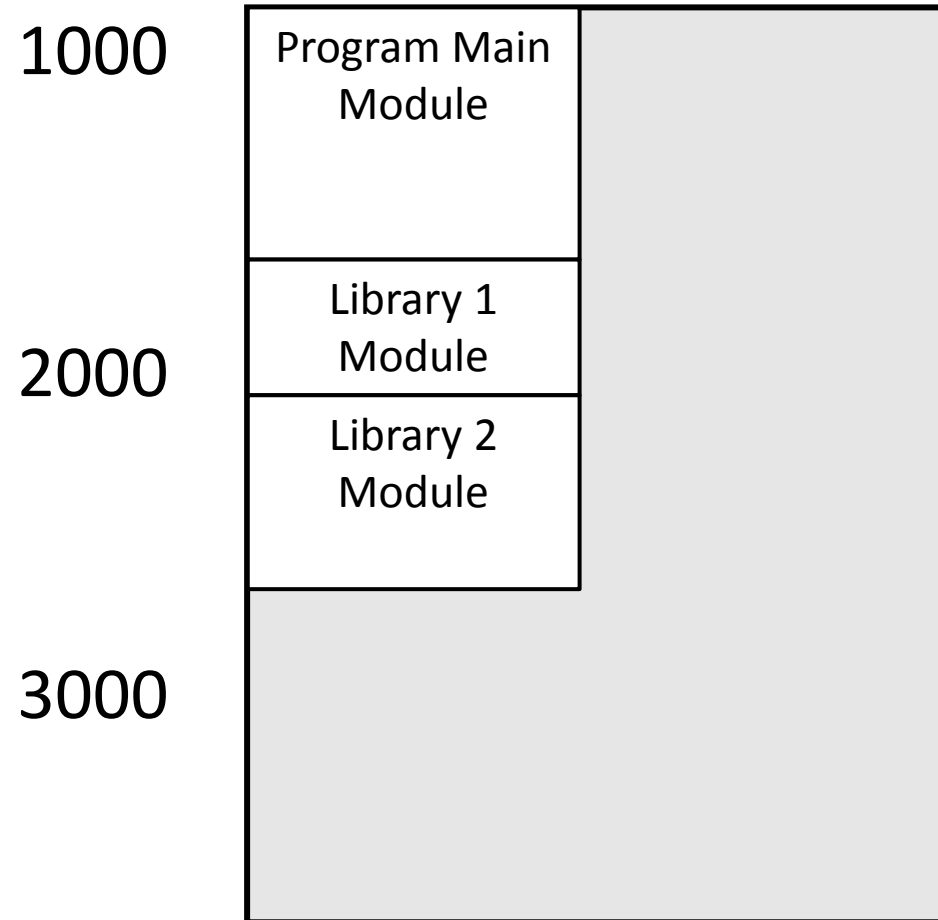
Predefined Addresses



- Very early systems
- Embedded systems
- Routines can be located at pre-designated addresses
- Separate assembly files
- *Absolute Loader*
- **Memory management is the responsibility of the programmer**

Absolute Addressing – contiguous code

Predefined *Starting* Address – Contiguous Code



- Very early systems
- Embedded systems
- Routines can be located at pre-designated addresses
- Ways to generate:
 - Single assembly file (no linker)
 - Multiple files (linker required)
- *Absolute Loader*

Absolute Loader

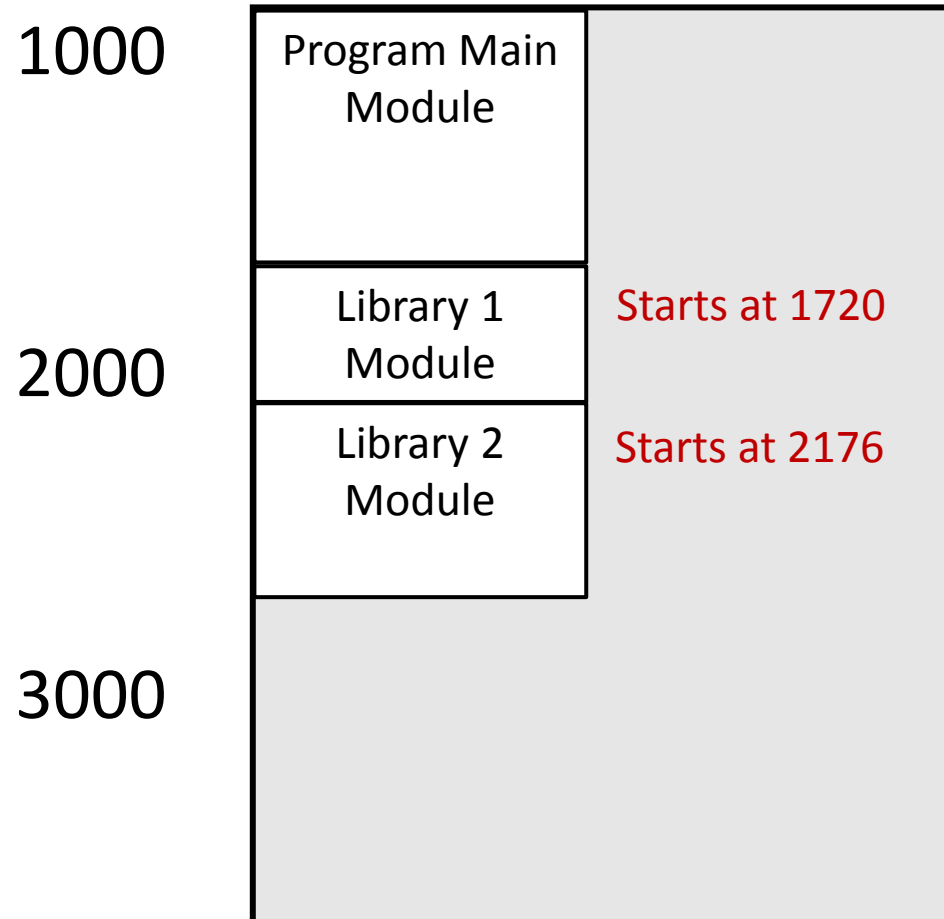
- Loads an executable file, with a *defined start address* and size, into memory
 - “Code” format in the executable file may be binary, or character-encoded (e.g. hex character codes)
- No relocation is needed at load time
- Executable file may be generated from a single assembly source file, or may be linked.
 - *Linker* relocates modules in order to combine them into a single address space

Absolute Loader - Use Cases

- “Primitive” machine – fixed address layout
 - Many embedded system
- Bootstrap loader
 - Load the initial software (typically the OS) when a computer is started
- A system that uses a hardware MMU – software doesn’t need to be relocated.

Relocating Loader

Linking / Loading with Relocation

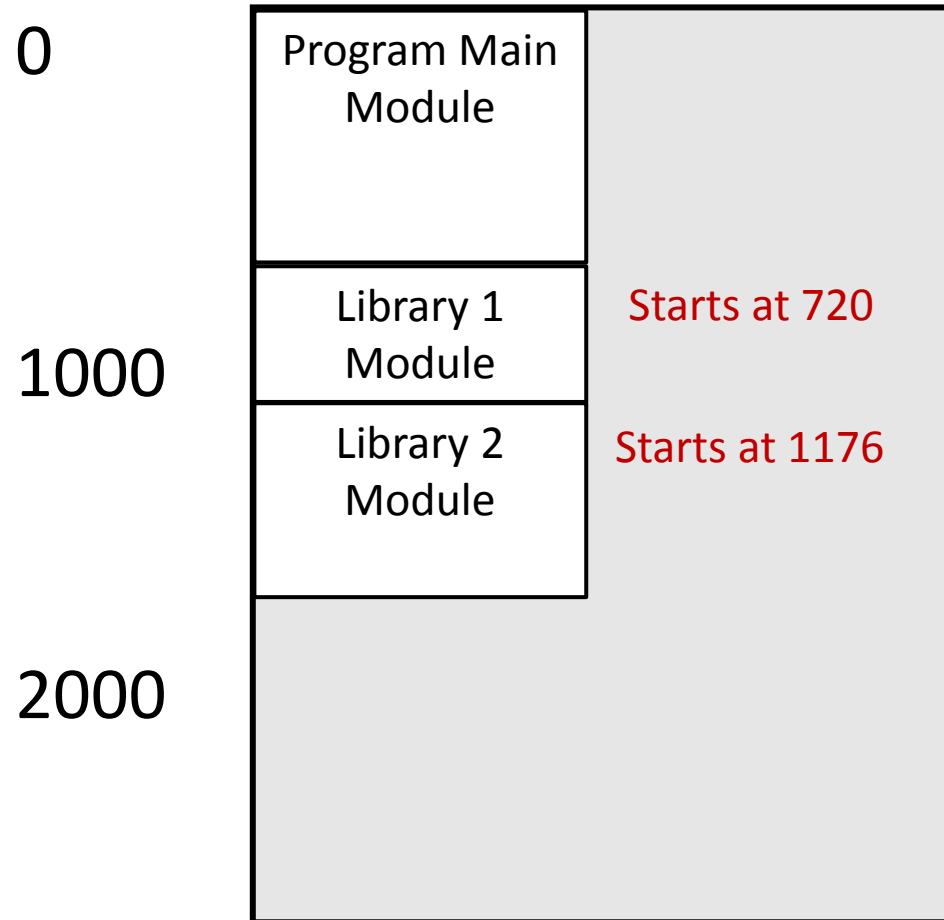


With START Instruction

- **Assembler:** set addresses based on START
- **Linker or Loader:** update addresses
 - $\text{new address} = \text{old address} - \text{START} + \text{new module base address}$

Relocating Loader

Linking / Loading with Relocation



Without START Instruction

- **Assembler:** ignore START; set addresses based on 0
- **Linker or Loader:** update addresses
 - $\text{new address} = \text{old address} + \text{new module base address}$
- START instruction in assembly language is superfluous

Program Relocation: Loader

Modification Table

Reference
1021
1026
1030
1048
1070
1076

Modification Table

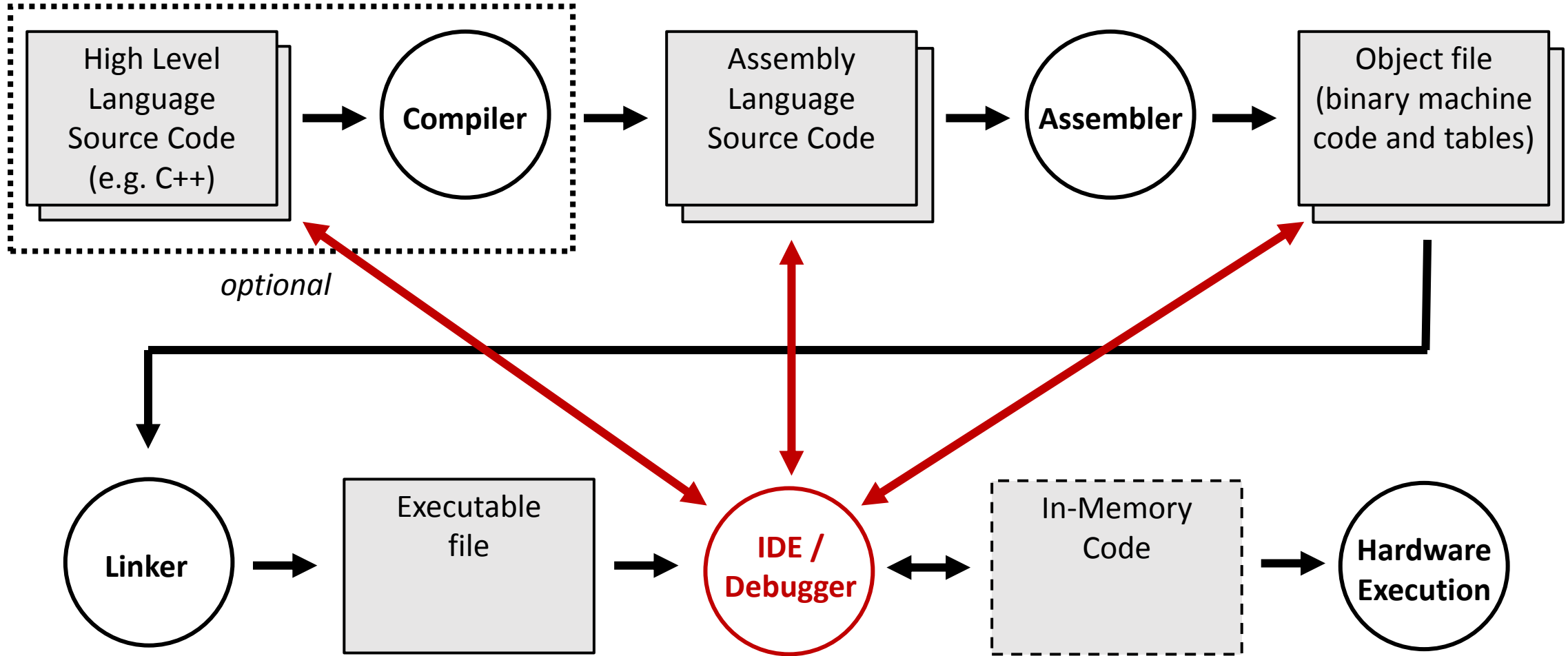
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- A relocating loader, just like a linker, may use a modification table to determine locations that must be adjusted.

Terminology

- **Linker:** Links multiple object (*.o) files into a single executable file
- **Linkage Editor:** an old-fashioned term for *linker*.
- **Dynamic Linker:** allows links to be made to shared library routines at run time (on demand)
- **Linker/Loader, or Linking Loader:** links multiple object (*.o) files into a single executable file, which it then loads and runs
- **Loader:** loads and runs an executable file
- **Absolute Loader:** loads an executable file at its specified address in memory
- **Relocating Loader:** relocates an executable file, allowing it be loaded at any address

Interactive Development Environments (IDEs)



IDEs: a Development Framework and Toolset

- In the first lecture, we talked about:
 - **Shells:** used for a wide range of ad hoc and automated tasks (chiefly on POSIX systems)
 - **Make:** automates the program build process
-

- IDEs are used during the development phase
- IDEs:
 - contain system dependent components
 - impact the file formats of object files and executable files
 - add complexity to assemblers, linkers, and loaders

Interactive Development Environments

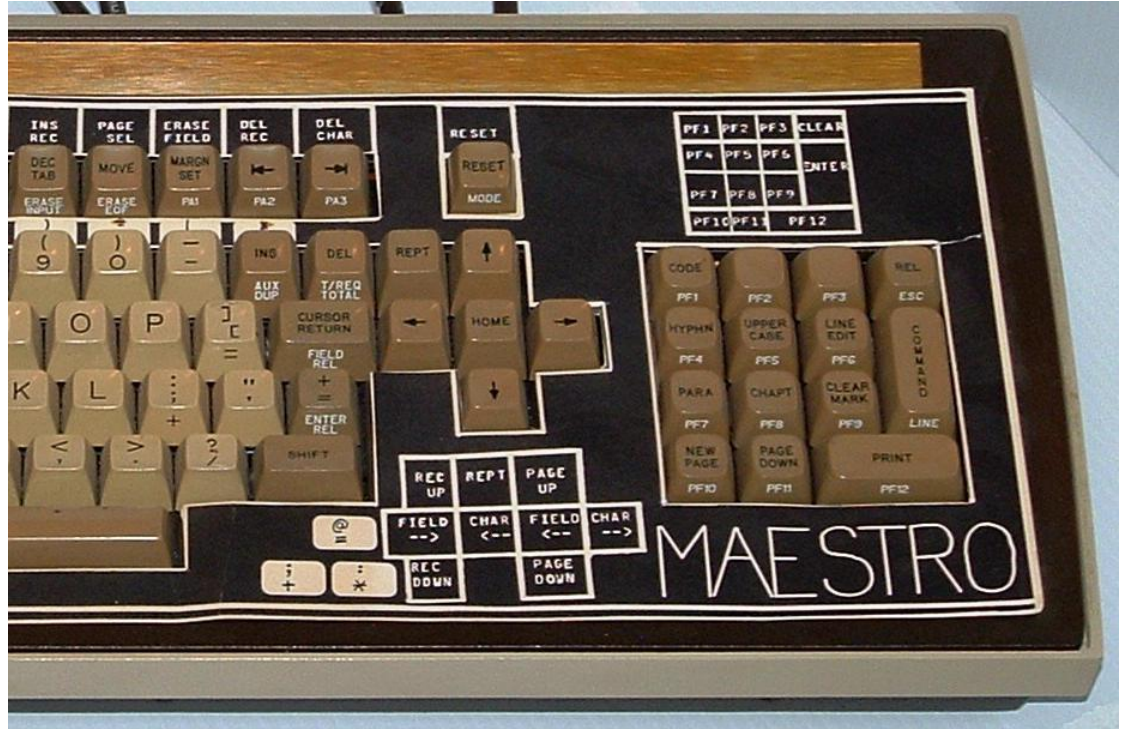
- Interactive (typically GUI based)
- Complete Development Environment:
 - “Smart” source code editor
 - Integration with program build systems (e.g. make)
 - Program debugger
- Additional Tools:
 - Integration with version control systems
 - Visual layout editor
 - Code refactoring
 - Performance profiling

Why IDEs?

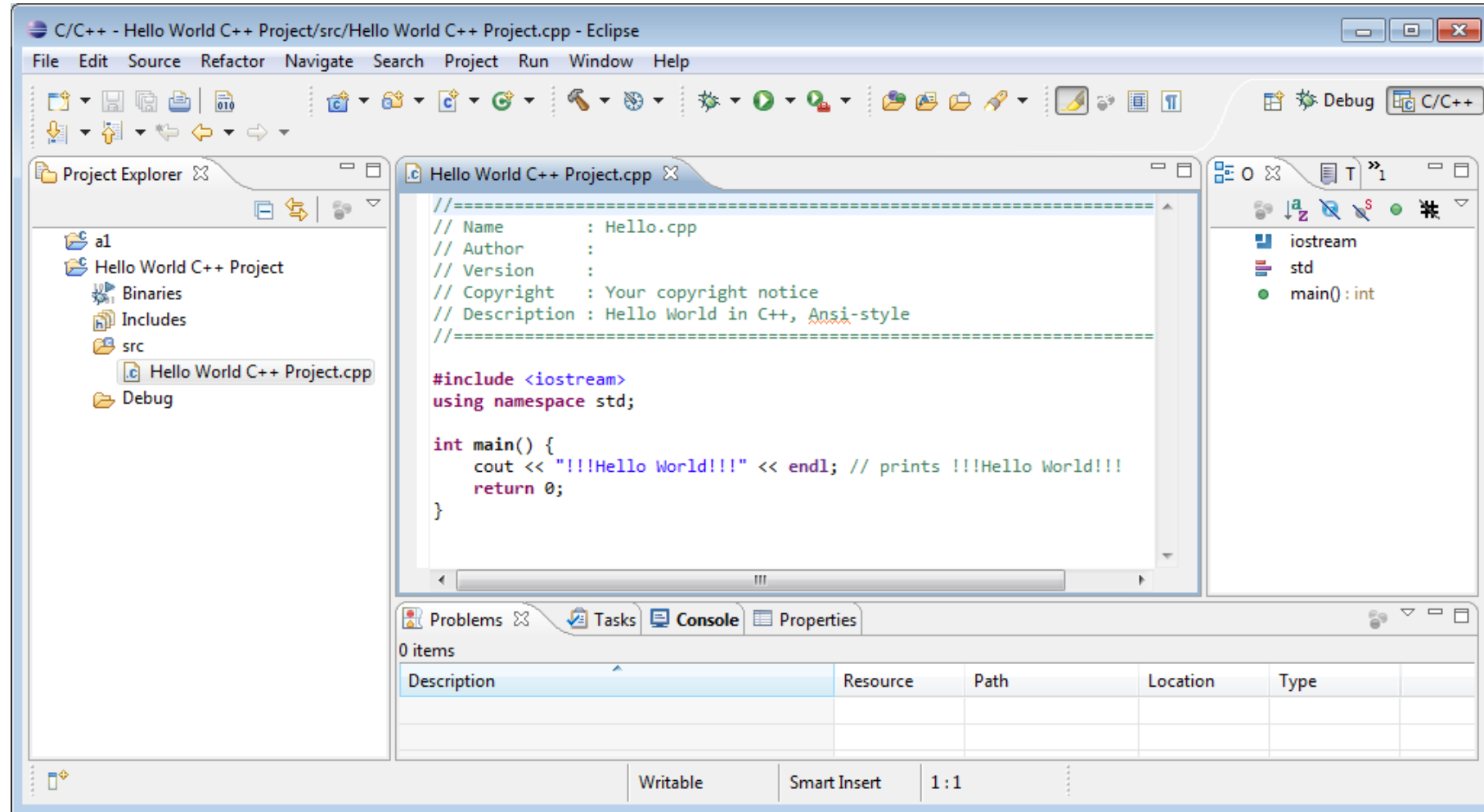
- Unlike *structured programming*, IDEs do not represent a paradigm shift... they simply allow programmers to do common tasks more quickly and efficiently.
- **Smart Editor:** because programmers spend more time editing than any other activity.
- **Debugger:** because ongoing support is the greatest cost in the software lifecycle.

History

- The first modern, GUI-based IDE was a commercial product, **Maestro I**, released by Softlab Munich in 1975
 - It was a leased hardware/software system, which included a custom keyboard and display
- Turbo PASCAL – 1983
- Visual Basic – 1991
- Delphi - 1995



Eclipse – a Popular Open Source IDE



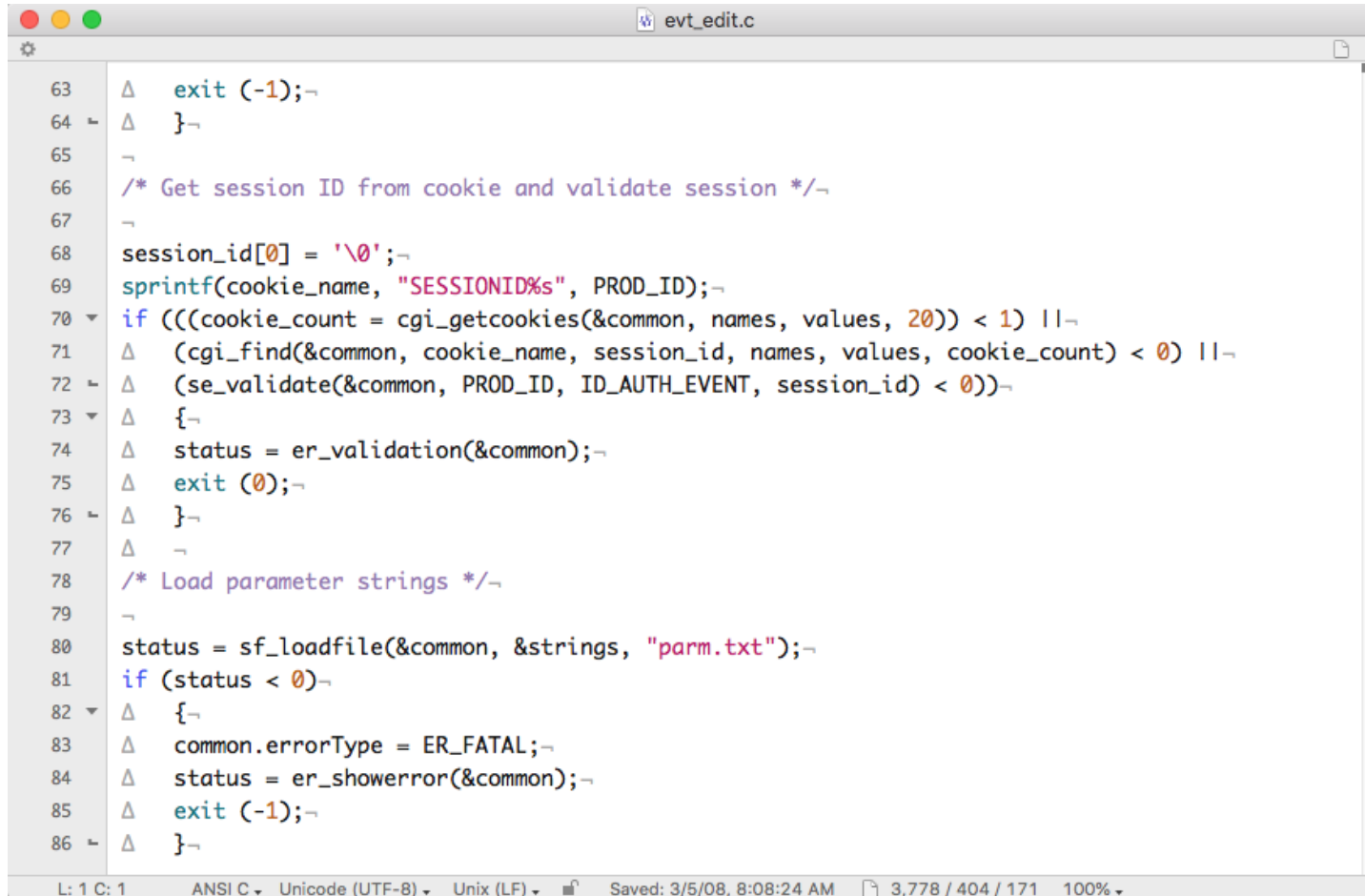
Most Popular IDEs

IDE	Developer
Visual Studio	Microsoft
Eclipse	Open Source
Xcode	Apple
Android Studio	Google (based on IntelliJ)
NetBeans	Oracle / Open Source
IntelliJ IDEA	Proprietary - JetBrains
PyCharm	Proprietary - JetBrains
Komodo	Proprietary - Komodo
Xojo (formerly RealBasic)	Proprietary - Xojo
<i>Bash Shell</i>	<i>Open Source / POSIX</i>

IDE Portability Versus Hardware Dependency

Feature	Machine Dependent?
Smart Editor / Code Editor	NO
Program Build	NO
Version Control	NO
File Browser	NO
High Level Language Support	NO
Code Refactoring	NO
Assembly Language Support	YES
Visual Layout Editor	YES
Performance Profiling	YES
Debugging	YES

Smart / Code Editing



```
63  Δ  exit (-1);-
64  Δ  }-
65  -
66  /* Get session ID from cookie and validate session */-
67  -
68  session_id[0] = '\0';-
69  sprintf(cookie_name, "SESSIONID%s", PROD_ID);-
70  Δ  if (((cookie_count = cgi_getcookies(&common, names, values, 20)) < 1) ||-
71  Δ  (cgi_find(&common, cookie_name, session_id, names, values, cookie_count) < 0) ||-
72  Δ  (se_validate(&common, PROD_ID, ID_AUTH_EVENT, session_id) < 0))-
73  Δ  {-
74  Δ  status = er_validation(&common);-
75  Δ  exit (0);-
76  Δ  }-
77  Δ  -
78  /* Load parameter strings */-
79  -
80  status = sf_loadfile(&common, &strings, "parm.txt");-
81  if (status < 0)-
82  Δ  {-
83  Δ  common.errorType = ER_FATAL;-
84  Δ  status = er_showerror(&common);-
85  Δ  exit (-1);-
86  Δ  }-
```

L: 1 C: 1 ANSI C ▾ Unicode (UTF-8) ▾ Unix (LF) ▾ Saved: 3/5/08, 8:08:24 AM 3,778 / 404 / 171 100% ▾

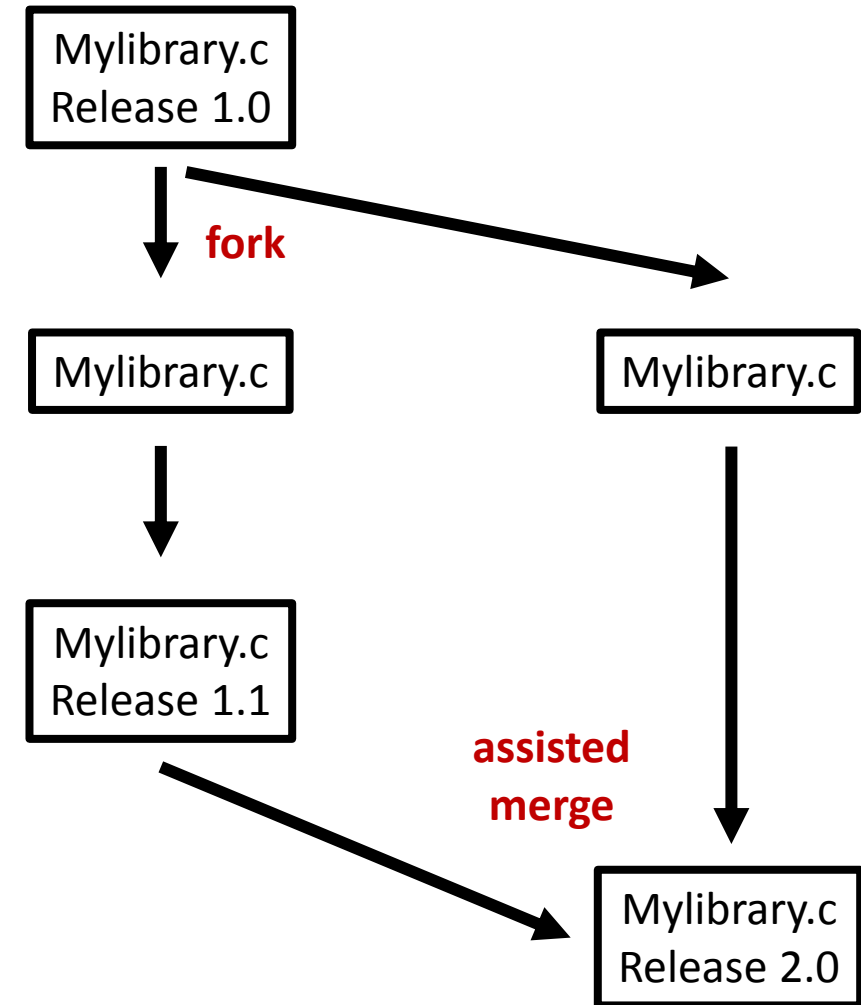
- Multiple Languages
- Syntax checking
- Auto-complete
- Indentation and code cleanup
- Color coding (configurable)
- Block checking / bracket matching / paren matching

Source Code Control / Version Control

- Allows multiple people to collaborate on a project
- Files are “checked in” to the version control system, and “checked out” for editing
- Checking in files typically requires a list of changes (comments), allowing a kind of audit trail
- The version control system stores all versions of a file, allowing users to *revert* to a previous version
- Some version control systems allow the project team to manage “forks” and to intelligently merge forked versions.
 - New version versus bug fixes

Source Code Control: Checkpoints and Forks

- When a project is “released” all files are *checkpointed* – given a Release number
- Developers may later *fork* files to work on different versions independently
- The Source Code Control System provides assistance for merging forked versions



Popular Version Control Systems

Version Control System	Model	Developer
Revision Control System (RCS)	Local	Open Source
Source Code Control System (SCCS)	Local	Open Source
Git	Distributed	Linus Torvalds / Open Source
Concurrent Versions System (CVS)	Client / Server	Open Source
Subversion (SVN)	Client / Server	Open Source
AWS CodeCommit (based on Git)	Distributed	Amazon
Team Foundation Server	Client / Server	Microsoft
Rational ClearCase	Distributed	IBM
Mercurial	Distributed	Open Source


- Many IDEs allow integration with multiple Version Control Systems

Visual Layout

The screenshot shows a 'Clients' application window with a 'Client Detail' tab selected. The form is divided into two main columns. The left column contains fields for 'Company', 'DBA', 'Client Since', 'Ship Addr', 'Bill Name', 'ATTN', 'Bill Addr', 'Tax Zone', 'Agreement', and 'Credit'. The right column contains fields for 'Short Name', 'Contact', 'E-Mail', 'Phone', 'Alt Phone', 'FAX', 'URL', 'FTP', 'Bus. Type', and 'FEIN'. A 'Copy' button is located between the 'Bill Addr' and 'Tax Zone' fields. The 'Bus. Type' dropdown menu is open, showing a list of options with '(E)ntertainment' selected. At the bottom of the window, there is a summary bar with 'CLIENT ID', 'Balance: \$ 0.00', 'Statement: 1/1/01', and a 'Save' button.

- Drag-and-Drop components
- Attach code to components

Debugging (Sample GUI)



	LDX	#0	
	STX	matches	matches = 0
	STX	index	Index = 0
LOOP	LDX	index	Load the index
	LDA	arraya, X	Get data from array A
	COMP	arrayb, X	Compare to element in array B
	JEQ	INCR	If they match, go incr count
CONT	LDA	index	Increment index by 3 bytes
	ADD	#3	
	STA	index	
	COMP	end	"end" contains 300
	JLT	LOOP	In index < limit, continue loop

Label	Addr	Byte Value	Word Value
LOOP	1017	74	5330938
CONT	1042	107	7410384
matches	1062	0	12
arraya	1066	2	135017
arrayb	1365	3	204293

Insert Break

Delete Break

Run

Debugging


Label	Addr	Byte Value	Word Value
LOOP	1017	74	5330938
CONT	1042	107	7410384
matches	1062	0	12
arraya	1066	2	135017


Register	Byte Value	Word Value
A	112	7810374
B	44	2349534
X	0	0

- Allows the user to view the address and content of memory locations by label
- May allow different display formats (decimal, binary, hex, character, etc)
- Allows the user to alter the content of memory locations
- Allows users to view and alter register contents

Debugging (Breakpoints)

(ANIMATE)



	LDX	#0	
	STX	matches	matches = 0
	STX	index	Index = 0
LOOP	LDX	index	Load the index
	LDA	arraya, X	Get data from array A
	COMP	arrayb, X	Compare to element in array B
	JEQ	INCR	If they match, go incr count 
CONT	LDA	index	Increment index by 3 bytes
	ADD	#3	
	STA	index	
	COMP	end	"end" contains 300
	JLT	LOOP	In index < limit, continue loop

Label	Addr	Byte Value	Word Value
LOOP	1017	74	5330938
CONT	1042	107	7410384
matches	1062	0	12
arraya	1066	2	135017
arrayb	1365	3	204293



Insert Break

Delete Break

Run

Debugging (Breakpoints)

	LDX	#0	
	STX	matches	matches = 0
	STX	index	Index = 0
LOOP	LDX	index	Load the index
	LDA	arraya, X	Get data from array A
	COMP	arrayb, X	Compare to element in array B
	JEQ	INCR	If they match, go incr count
CONT	LDA	index	Increment index by 3 bytes
	ADD	#3	
	STA	index	
	COMP	end	"end" contains 300
	JLT	LOOP	In index < limit, continue loop



Label	Addr	Byte Value	Word Value
LOOP	1017	74	5330938
CONT	1042	107	7410384
matches	1062	0	12
arraya	1066	2	134219
arrayb	1365	2	134219

Insert Break

Delete Break

Run

Setting a Breakpoint

(ANIMATE)

	LDX	#0	
	STX	matches	matches = 0
	STX	index	Index = 0
LOOP	LDX	index	Load the index
	LDA	arraya, X	Get data from array A
	COMP	arrayb, X	Compare to element in array B
	JSUB	BREAK	GIVE CONTROL TO IDE
CONT	LDA	index	Increment index by 3 bytes
	ADD	#3	
	STA	index	
	COMP	end	"end" contains 300
	JLT	LOOP	In index < limit, continue loop

Instruction Save Table

1054	JEQ	INCR
------	-----	------

Hitting a Breakpoint

Pseudocode

BREAK:

- Save L Register (JSUB return address)

- Save all Registers

- Save CC (condition code)

- Update Program Display (show position in code)

- Update Memory Display (show values of memory locations)

Continuing After a Breakpoint

When the User Clicks RUN

- Restore saved registers

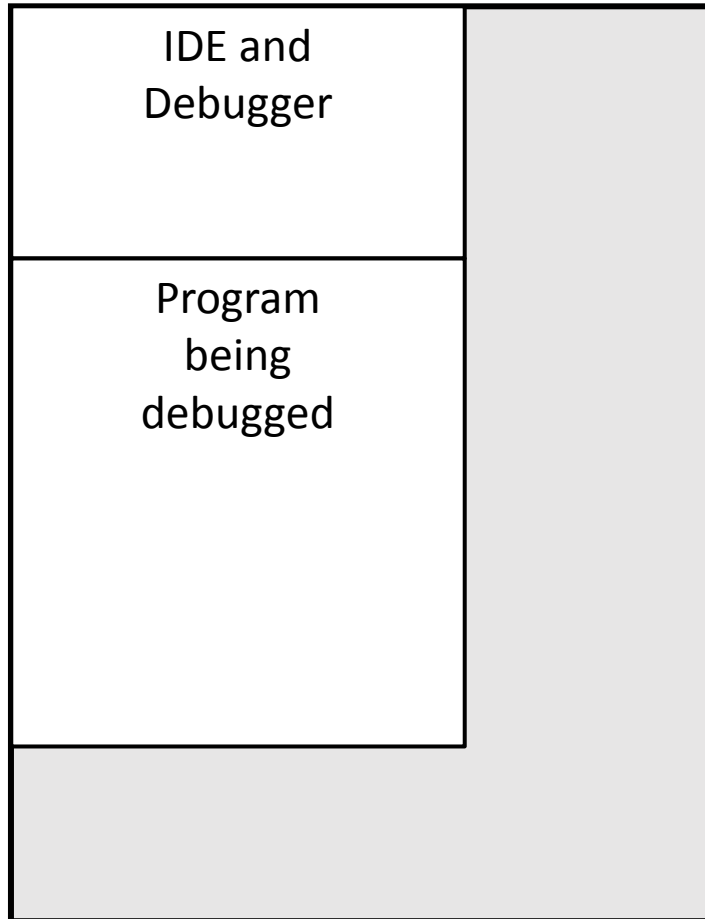
- Restore CC (condition code)

- Execute Saved Instruction

- Jump to return address (L register)

Impact on Memory and Addressing

Memory



- Debugger and program are a single *process*
- Debugger and program reside in the same address space
- Debugger needs to *relocate* program when loading it
- Debugger is able to read and write program memory

Breakpoints Using Interrupts

- An interrupt is a hardware mechanism for interrupting the normal flow of program execution.
 - Causes an 'immediate' branch to an *interrupt handler* routine
- Interrupts may be triggered by
 - A timer
 - An I/O operation
 - An instruction that triggers an interrupt

Setting a Breakpoint

(ANIMATE)

	LDX	#0	
	STX	matches	matches = 0
	STX	index	Index = 0
LOOP	LDX	index	Load the index
	LDA	arraya, X	Get data from array A
	COMP	arrayb, X	Compare to element in array B
	INT	01	GIVE CONTROL TO IDE
CONT	LDA	index	Increment index by 3 bytes
	ADD	#3	
	STA	index	
	COMP	end	"end" contains 300
	JLT	LOOP	In index < limit, continue loop

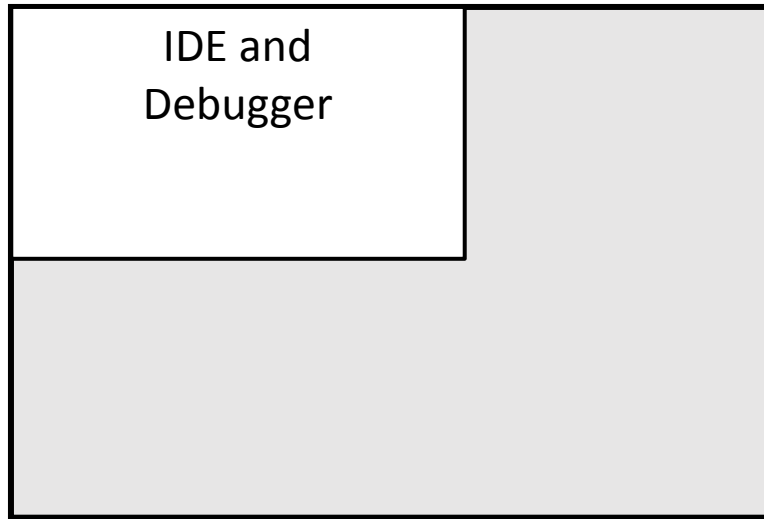
Instruction Save Table

1054	JEQ	INCR
------	-----	------

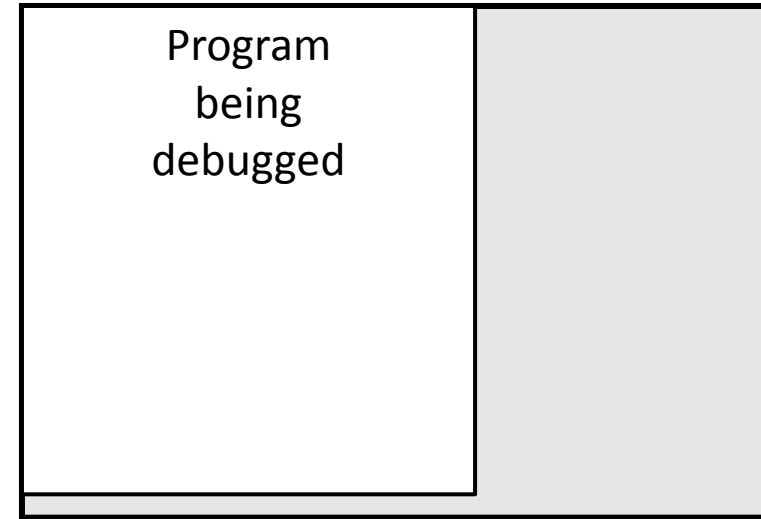
- For example only... the SIC instruction set does not have an INT (interrupt) instruction.*

Impact on Memory and Addressing

Process 1 Address Space



Process 2 Address Space



- Debugger and program are separate *processes*
- Debugger and program each reside in their own address space
- Debugger does not need to *relocate* program when loading it
- Debugger must be able to read and write across address spaces

Requirements for Assembler, Linker & Loader

- Write additional information to object (*.o) files and executable files

Code Table

1017		LDX	#0	
1020		STX	matches	matches = 0
1023		STX	index	Index = 0
1026	LOOP	LDX	index	Load the index
1029		LDA	arraya, X	Get data from array A
1032		COMP	arrayb, X	Compare to element in array B
1035		JEQ	INCR	If they match, go incr count
1038	CONT	LDA	index	Increment index by 3 bytes
1041		ADD	#3	
1044		STA	index	
1047		COMP	end	"end" contains 300
1050		JLT	LOOP	In index < limit, continue loop

Symbol Table

Symbol	Type	Address / Value
LOOP	REF	1017
CONT	REF	1042
matches	REF	1062
arraya	REF	1066
arrayb	REF	1365
index	REF	1368

Debugging (High Level Languages)

➡

```
status = cgi_get(&common, field_buffer, 1000);  
  
field_count = cgi_vars(&common, field_buffer, names,  
code[0] = '\0';  
  
if ((cgi_find(&common, "code", &code, names, values,  
    {  
    strcpy(common.message, "You must select a  
  
    common.errorType = ER_USER;  
  
    status = er_showerror(&common);  
  
    exit (0);
```

Label	Addr	Value
status	1017	1
field_count	1042	17
common	1062	structure
code	1066	array
field_buffer	1365	structure

Insert Break

Delete Break

Run

Debugging (High Level Languages)

- The debugger must now be able to connect high-level source code statements to machine code addresses, to allow breakpoints
- The debugger must understand and be able to display variable arrays, different data types, structures, etc.

Additional File Information

- The information required by debuggers adds a great deal of data to the object and executable files
- Typically an *option* when compiling, assembling, and linking:
gcc -g
 - gcc -g generates debug information to be used by GDB debugger

For Next Thursday

- Log in to Canvas and complete Assignment 3
- Read Chapter 4 – Macro Processors