CS-202
Course Introduction
C++ Primer – GNU Compilation starters

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Your Instructor

Christos Papachristos

Research Assistant Professor

Department of Computer Science & Engineering

- Areas of Activity:

  Robotics Research @ Autonomous Robots Lab (ARL) - UNR

  Coding / SW Engineering for Field Robotics

- Education

  MS in Electrical & Computer Engineering

  PhD in Autonomous (Aerial) Robotics

- Find me at:

  ARF004 / Autonomous Robots ARENA (a.k.a. "the HighBay")

  cpapachristos@unr.edu , papachric@gmail.com

Course Requirements

- **CS-135 (@least C- grade)**

  Basic program design
  Control structures for selection , iteration
  Use functions , employ/implement prototypes, pass parameters
  Multi-dimensional arrays (declaration/usage/passing as parameters)
  Basic stream I/O (with external file usage)

ALL are *Strict* Prerequisites !

Course Objectives

Introductory-level experience to:

- Object-oriented programming (**OOP**)
- Object-oriented design (**OOD**)

Basic software engineering techniques , Proper program design principles

Pointers , Dynamic Memory

C++ libraries , Classes – operators , the STL

(Coding) Tools:

- C++ programming language
- GCC (GNU Compiler)
- The GNU Make tool
- Linux Operating System

Course Rules

Grading Policy (tentative):

Component	Percentage
Projects & Assignments	50% (Breakdown: 40% – 10% each)
Midterm Exam	20%
Final Exam	30%

You cannot earn a passing grade in the course without a passing grade on both:

- The average of the Midterm and Final exams.
- The average of the Projects.

Plus/Minus grading will be assigned as indicated in the Syllabus.
Grade re-scaling may be assigned based on an outstanding or inferior Final exam.

Class presence will be required, tracked, and factored in, for the course's Lab Sections.
For general university policy regarding class absence, see UAM 3,020.

Course Rules

Tasks & Responsibilities:

Weekly Projects – Bi-weekly MAX
Turn in via WebCampus! LATE submission (24hrs max) incurs 20% penalty.
After 24hrs NO SUBMISSION WILL BE ACCEPTED.
Lab Assignments – In-Lab! NO SUBMISSION WILL BE ACCEPTED AFTER.

Academic Dishonesty:

Cheating, plagiarism or otherwise obtaining grades under false pretenses constitute academic dishonesty according to the code of this university. Academic dishonesty will not be tolerated and penalties can include filing a final grade of "F"; reducing the student's final course grade one or two full grade points; awarding a failing mark on the coursework in question; or requiring the student to retake or resubmit the coursework. For more details, see the University of Nevada, Reno General Catalog.
(Also, refer to Academic Standards in course syllabus and online)

Course Rules

Tasks & Responsibilities:

Weekly Projects – Bi-weekly MAX
Turn in via WebCampus! LATE submission (24hrs max) incurs 20% penalty.
After 24hrs NO SUBMISSION WILL BE ACCEPTED.
Lab Assignments – In-Lab! NO SUBMISSION WILL BE ACCEPTED AFTER.

Academic Dishonesty:

Note:
There exist widely accessible and reliable tools to cross-compare you code:
vs your classmates' code !
vs students' code from previous semesters !
Semantic analysis and comparison means IT WILL CATCH similarly structured code, even if you rename your variables / change indentation / etc. !

Course Help

Course & Projects:

Peer-Assisted Study Session (PASS) Leader:

Kurtis Rodrigue kurtisr@nevada.unr.edu

3 hrs in-class, 4 hrs PASS sessions

Teaching Assistants:

XinYing Wang, Hemanta Sapkota, Shuvo Kumar Paul, Yuchuan Liu

xinyingw@, hsapkota@, shuvo.k.paul@, ycliu@ [nevada.unr.edu]

50 mins per Lab section , Tutoring & quizzes

Disability Services:

Any student with a disability needing academic adjustments or accommodations is requested to speak with the Disability Resource Center as soon as possible to arrange for appropriate accommodations.

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (8 Sections)	
	CLASS		CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Projects will start **this** Thursday 1/24.

Labs will start **this** Thursday 1/24.

Challenges

Getting started:

The Linux environment (some transfer students could have a hard time).

Start with your Projects early on !

Familiarize yourselves with the **NoMachine™** client and the **ECC** systems.

Don't play down the significance of the Lab sessions.

Don't wait to seek help. Benefit from all sources.

Your TAs.

Your PASS Leader.

WebCampus material (lectures, samples, etc.) & discussions.

The *web*… http://en.cppreference.com/w/cpp

http://www.cplusplus.com/reference/

Today's Concepts

Differences between programming languages

- Compiled *vs* Interpreted programs
- Programming "style" restrictions

C++ concepts (Learning C++ is only part of the deal)

- Classes
- Object-Oriented Programing
  - Encapsulation
  - Inheritance
  - Polymorphism
- Object-Oriented Design

(But Remember: C++ is the "*Latin*" of programming languages)

Today's Practices

Using the Linux Operating System.

Developing under the GCC (GNU Compiler Collection) suite.

- Linux basics, Shell usage introduction.
- Text editor for code writing – gedit / emacs / nano / vim.

  (IDEs such as Eclipse, XCode, Codeblocks etc. are not part of the supported toolchains, your programs will have to compile & run on installations like the ones on the machines of SEM321).
- Compilation toolchain invocation.
- Basics for large (sort of) software development project setup.

Code – Compile – Run

C++ Basic Info

Created in 1979 by Bjarne Stroustrup – *the originator*. At Bell Labs (home of UNIX and C).

The story: "Invent a computer programming language so arcane and complex that no one except him would be able to use it."

The background: "Languages like Simula and Ada."

The anecdote: "Colleagues laughed a his first cut at an inscrutably hard language. Then, he went back to his lair and didn't emerge until he had added references and templates."

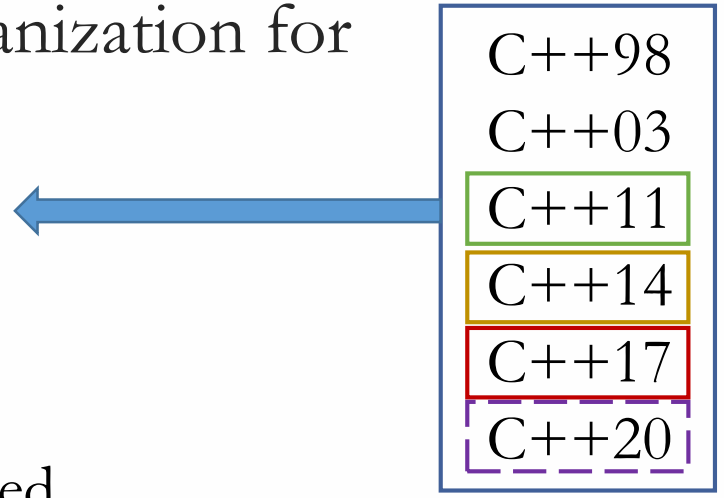
C++ Basic Info

Added object-oriented features to C.

Renamed to C++ in honor of auto-increment operator.

Later standardized with several International Organization for Standards (ISO) specifications.

- Constantly expanding/updating standards:



Greatly influenced Java development (1991).

- Popular modern OO language
- Wide industry usage
- Used in many types of applications

- Object-Oriented
- Portable (not as much as Java, but fairly so)
- Efficient
- Retains much of its C origins

Procedural *vs* Object-Oriented

Procedural

Modular units: functions

Program structure: hierarchical

Data and operations are not bound to each other

Examples:

C, Pascal, Basic, Python

Object-Oriented (OO)

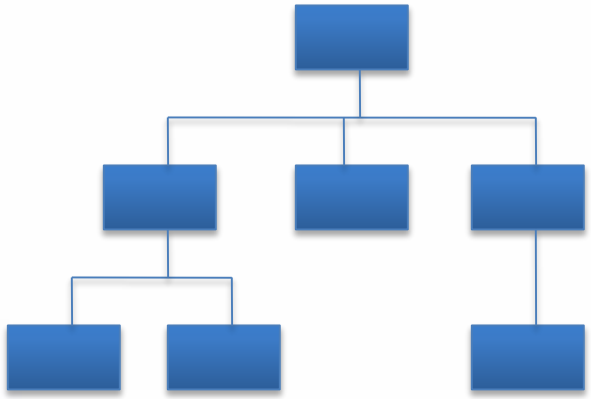
Modular units: objects

Program structure: a graph

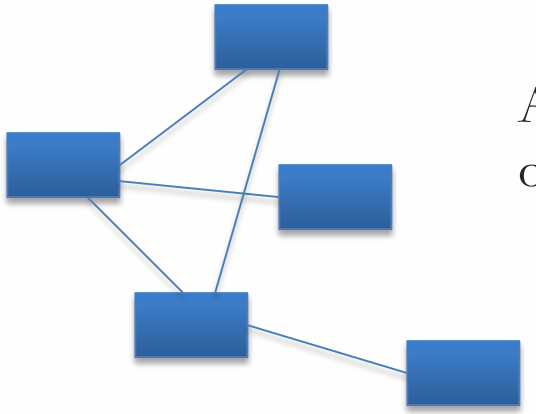
Data and operations are bound to each other

Examples:

C++, Java, Python



A hierarchy of functions



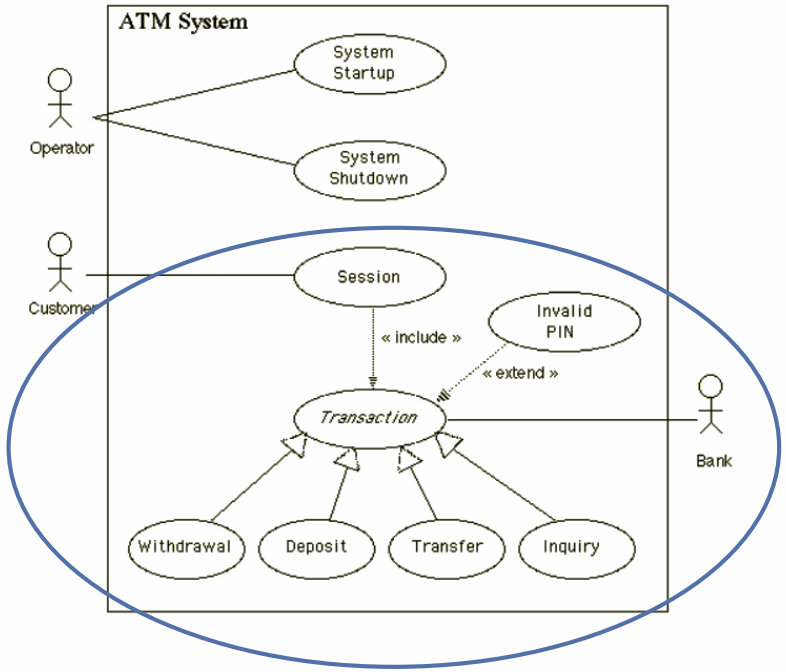
A collection of Objects

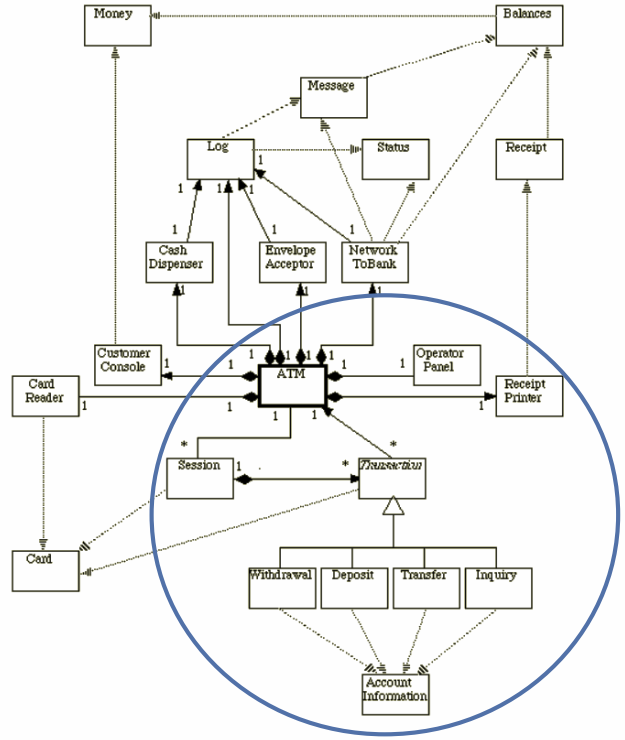
Procedural *vs* Object-Oriented

The ATM Machine paradigm

Fundamental Software Development problems:

- Maintainability
- Adaptability (modularity)
- Expressiveness





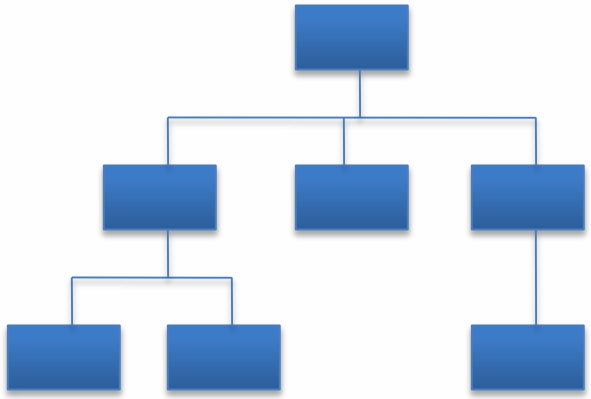
Procedural *vs* Object-Oriented

Procedural

Focused on the question: "**What** should the program **do next**?" Structure program by:

- Splitting into sets of tasks and subtasks.
- Make ***Functions*** for tasks.
- Perform **tasks in sequence** (computer).

Large amount of data and/or tasks makes projects/programs unmaintainable.



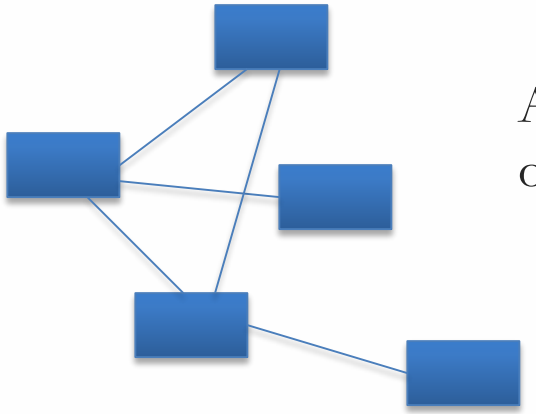
A hierarchy of functions

Object-Oriented (OO)

Package-up **self-contained** & **modular** pieces of code.
The world is made up of **interacting *Objects***.
Pack away details into boxes (***Objects***), allowing us to consider them in their abstract form.
Focus on their (numerous) interactions.

Key concepts:

- Encapsulation
- Inheritance
- Polymorphism

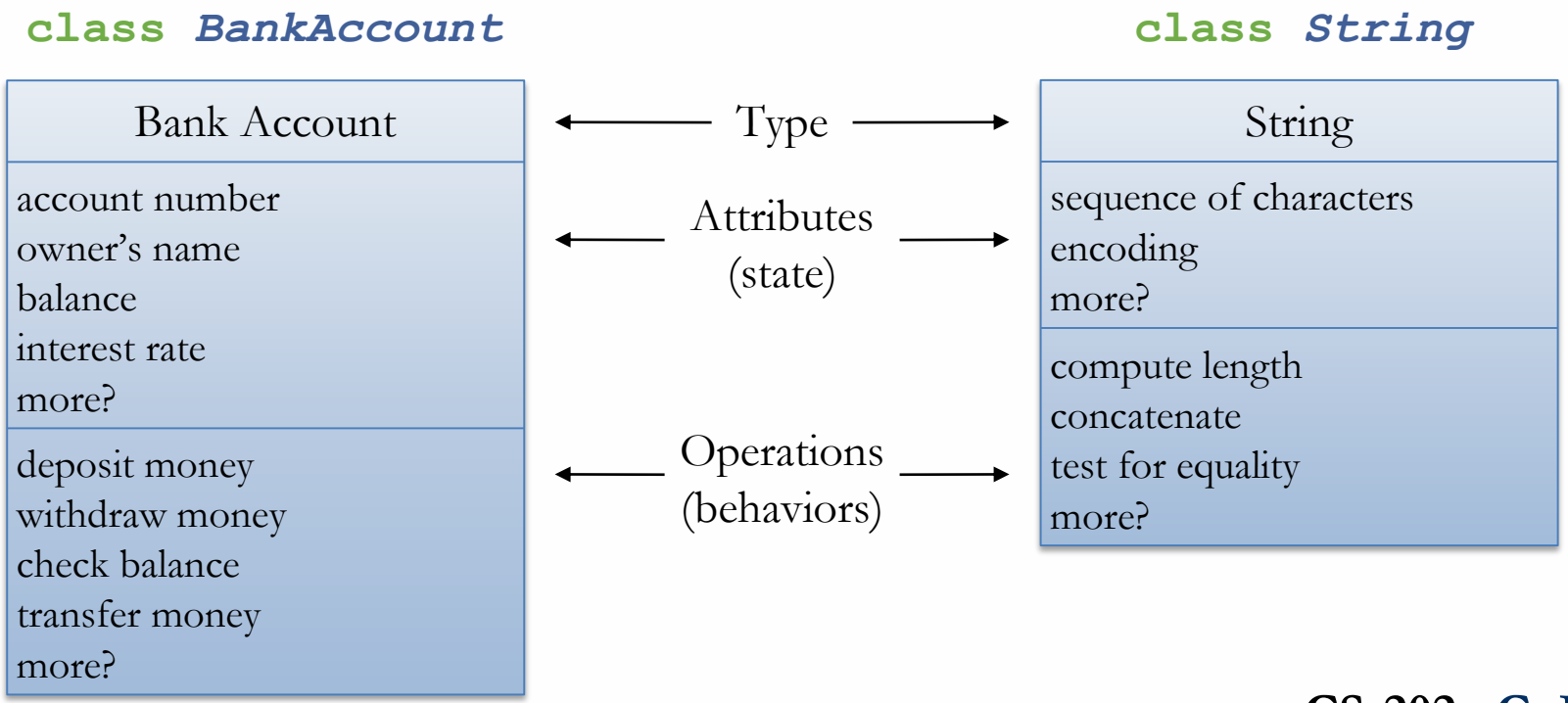


A collection of Objects

Classes

What is a … **Class**

C++ Classes are very similar to C Structs, in that they both include user-defined sets of data items, which collectively describe some entity such as a Student, a Book, an Airplane, or a data construct such as a String, a ComplexNumber, etc…

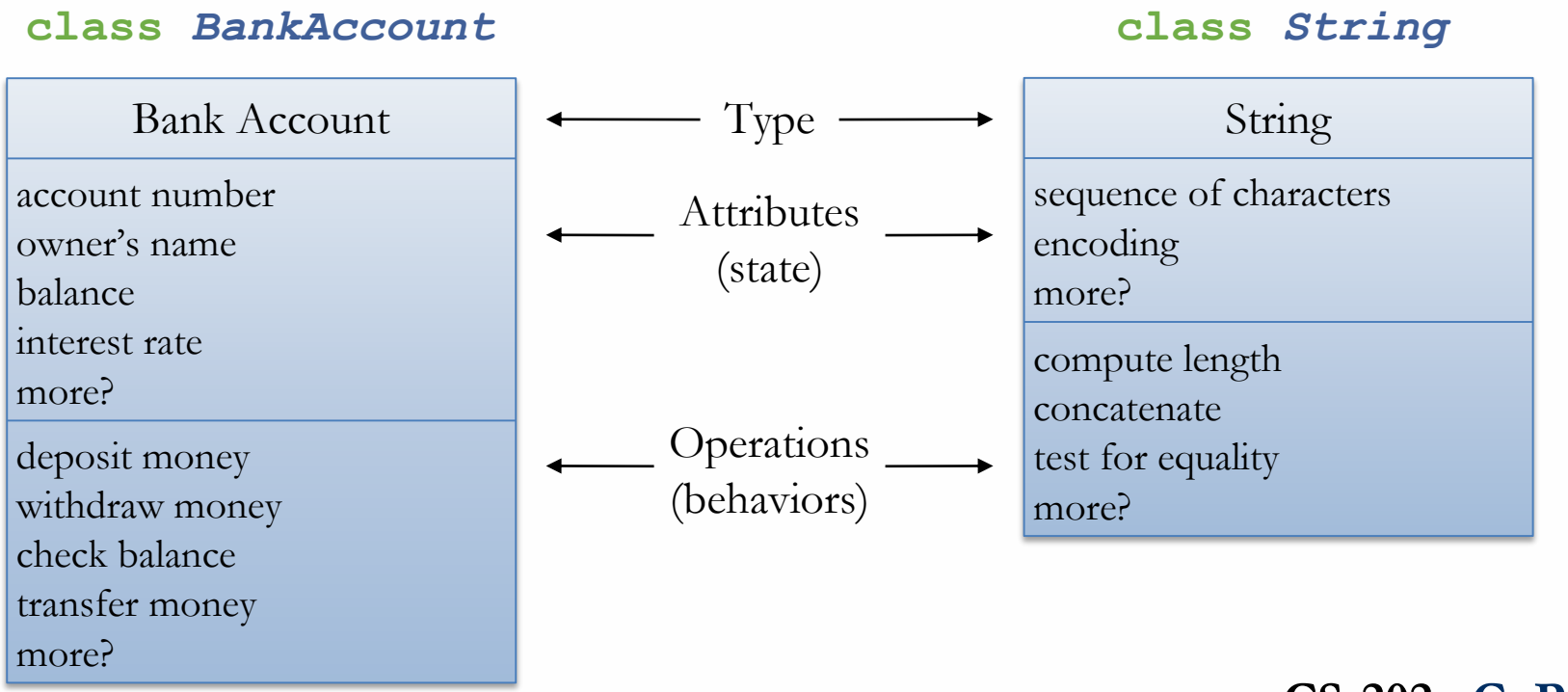


Classes

What is a … **Class**

A *non-primitive* & *user-defined* data type containing:

- Attributes – make up the object's state
- Operations – define the object's behaviors



Objects

What is an … **Object**

A particular instance of a class.

```
class BankAccount
```

Bank Account
account number owner's name balance interest rate
deposit money withdraw money check balance transfer money

Scrooge's Account

12-345-6
Ebenezer Scrooge
$ 9,999,999,999 . 99
0.0125 %

Wilde's Account

65-432-1
Oscar Wilde
$ 8 . 45
3.5 %

Smith's Account

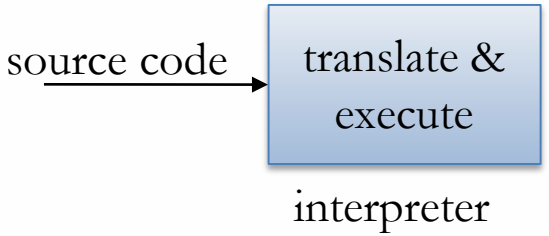
43-261-5
Adam Smith
$ 1,766 . 00
2.5 %

For any of these accounts, one can…

- Deposit money
- Withdraw money
- Check the balance
- Transfer money

Interpreters, Compilers, Hybrids

Interpreted Languages (e.g. JavaScript, Perl, Ruby)

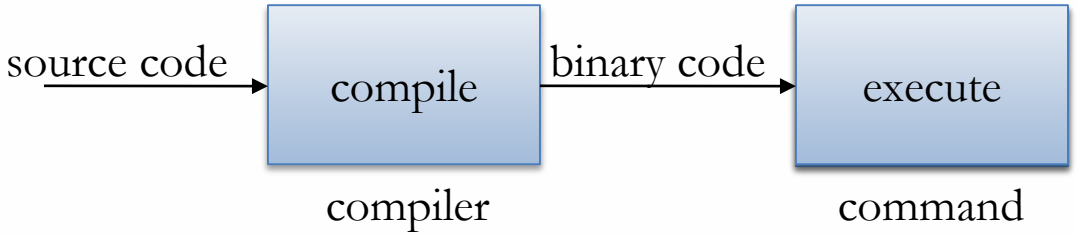


Interpreter translates source into binary operations and executes it.

Small, easy to write

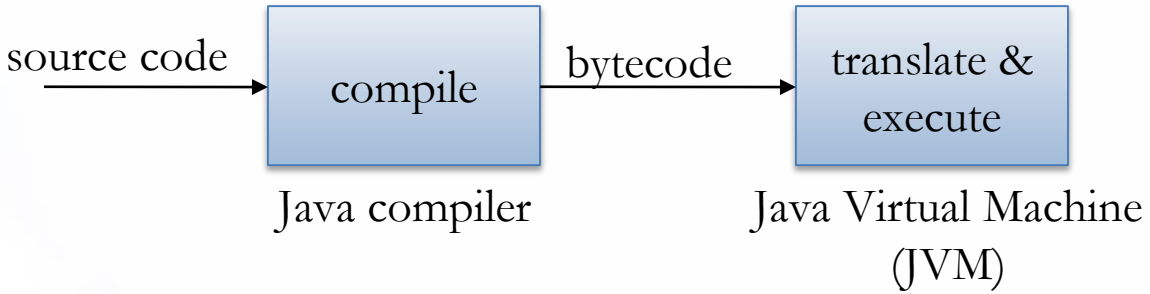
Interpreter is a *program* unique to each platform (i.e. operating system).

Compiled Languages (e.g. C, C++)



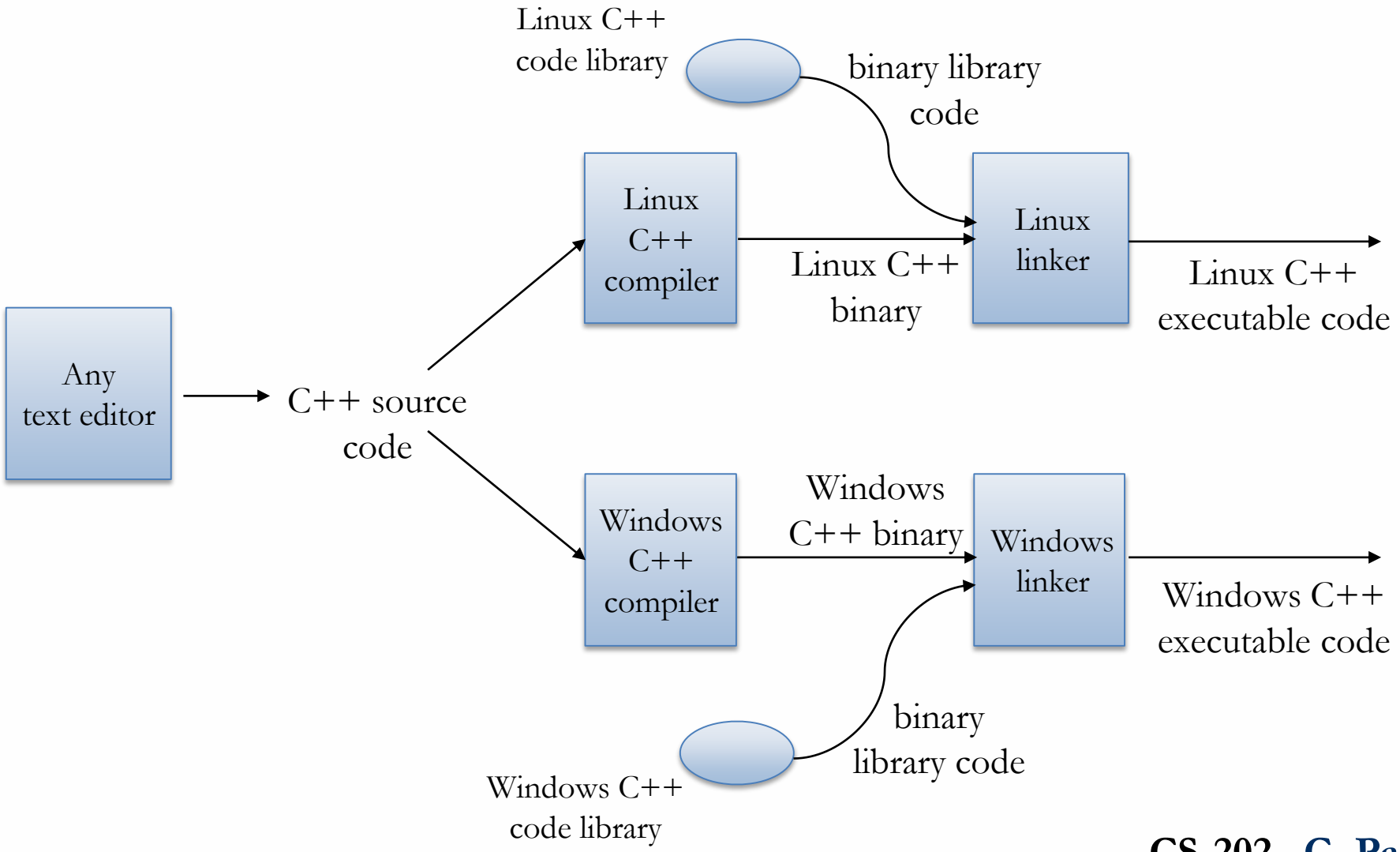
Compiler is platform dependent.
Code once compiled is expressed in the *instructions* of the target machine (e.g. target architecture).

Many other models: (e.g. Java, Python etc.)

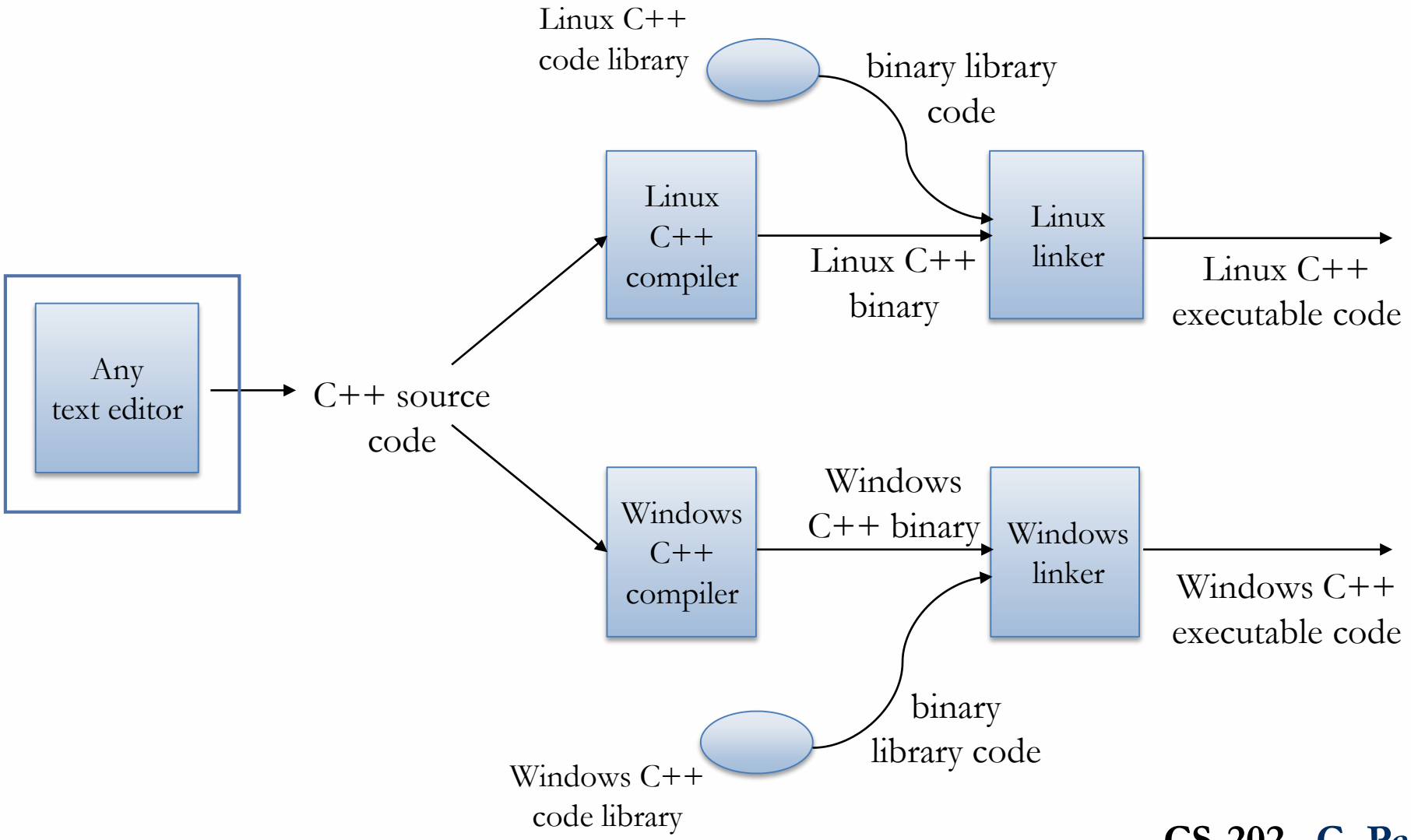


Bytecode is platform independent
Java *Virtual Machine* (the target platform) is an interpreter that is platform dependent.

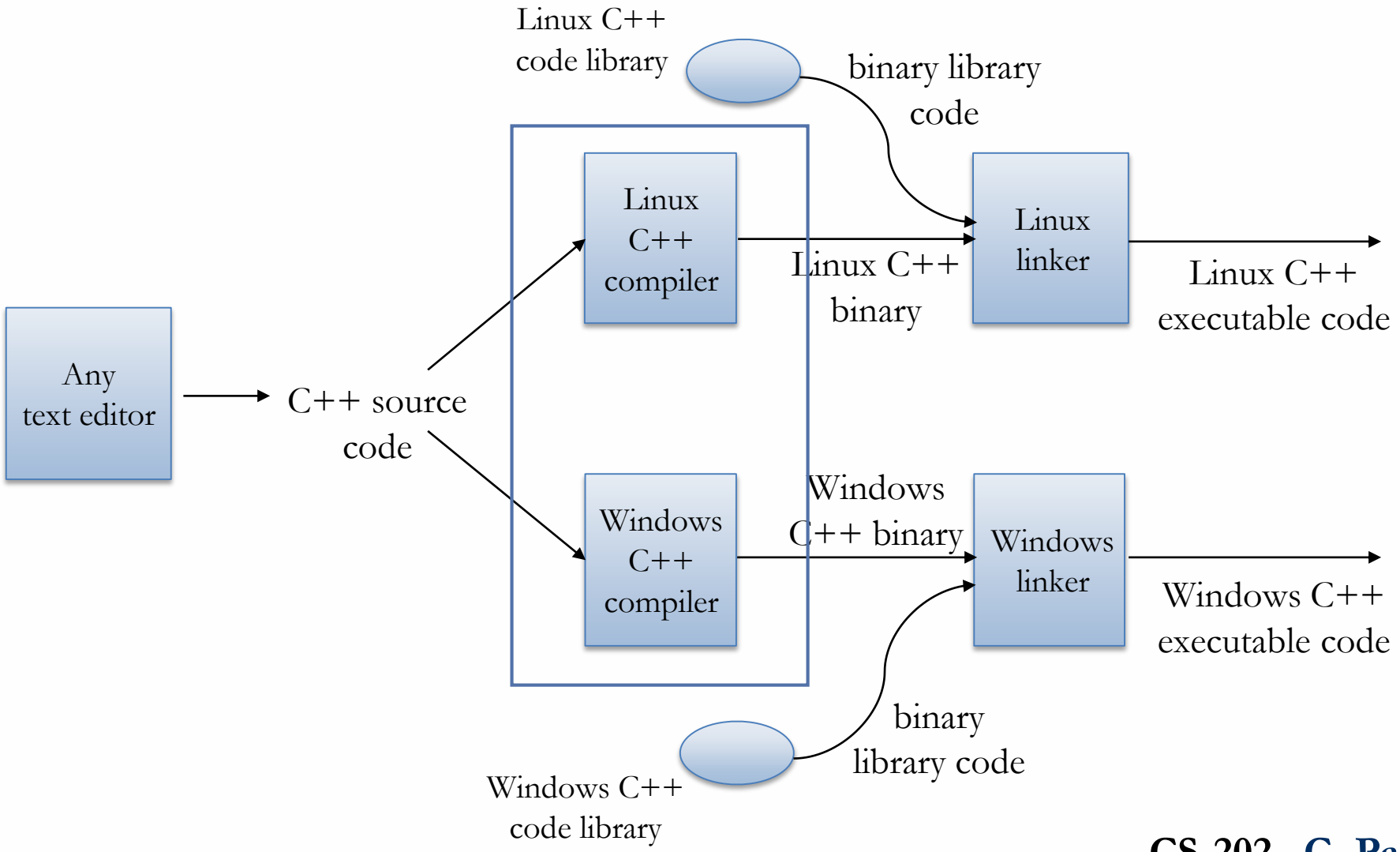
C++ Compilation & Linkage



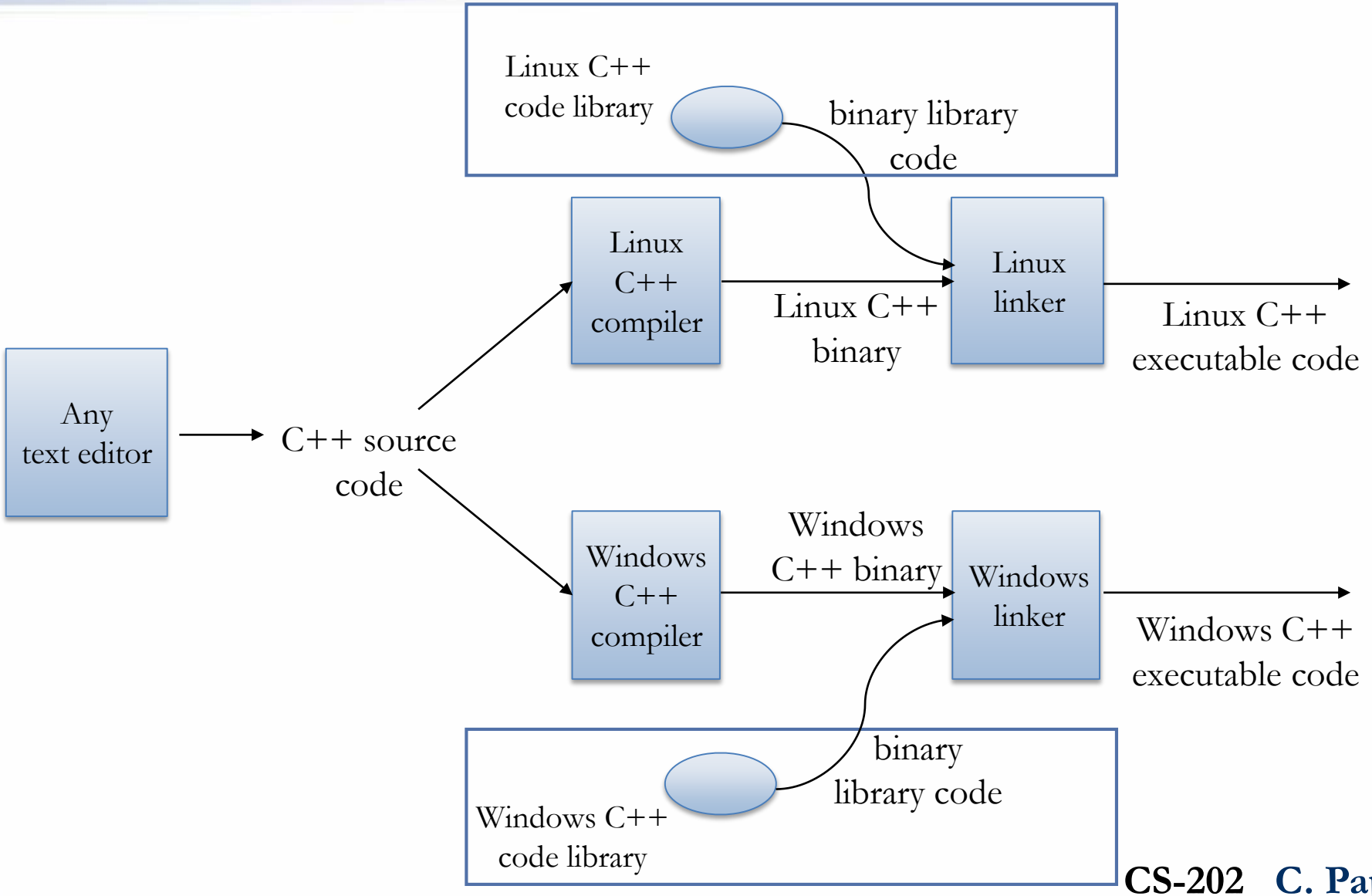
C++ Compilation & Linkage



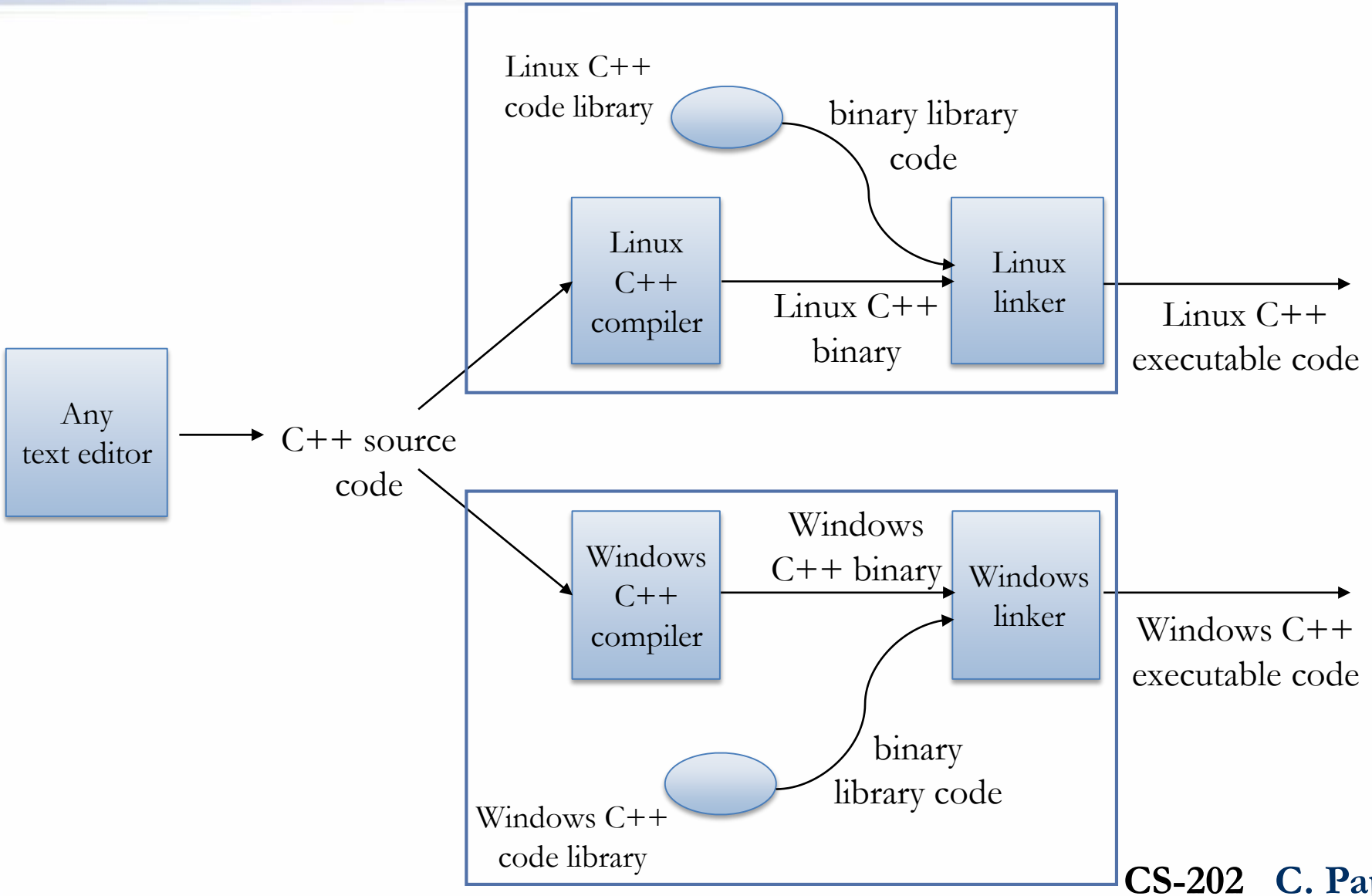
C++ Compilation & Linkage



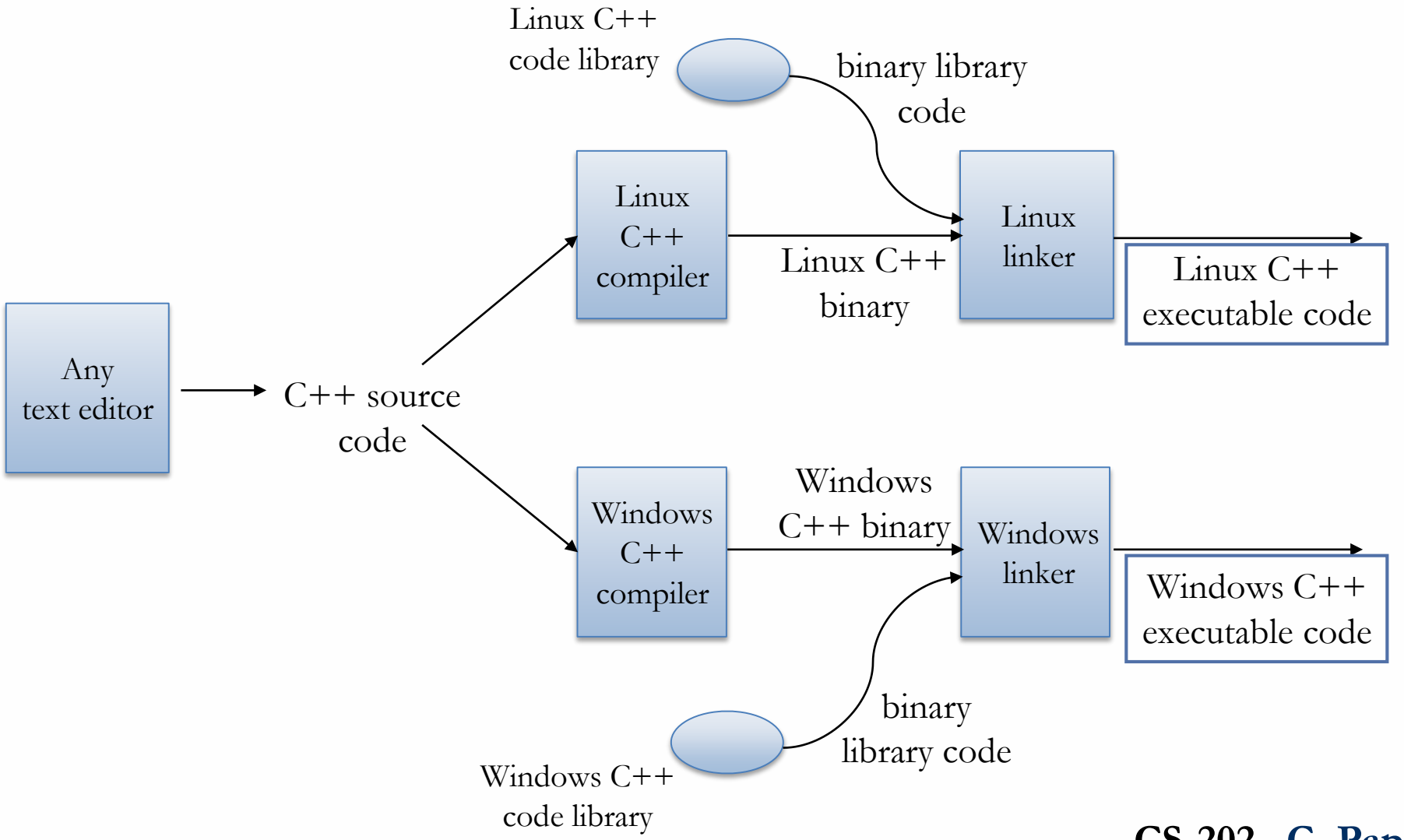
C++ Compilation & Linkage



C++ Compilation & Linkage



C++ Compilation & Linkage



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

The C++ compilation process

Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named `prog1.cpp` and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using `#define` with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the `-E` option:

   ```
   g++ -Wall -std=c++11 -E prog1.cpp
   ```

   The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

2. To stop the process after the compile step, you can use the `-S` option:

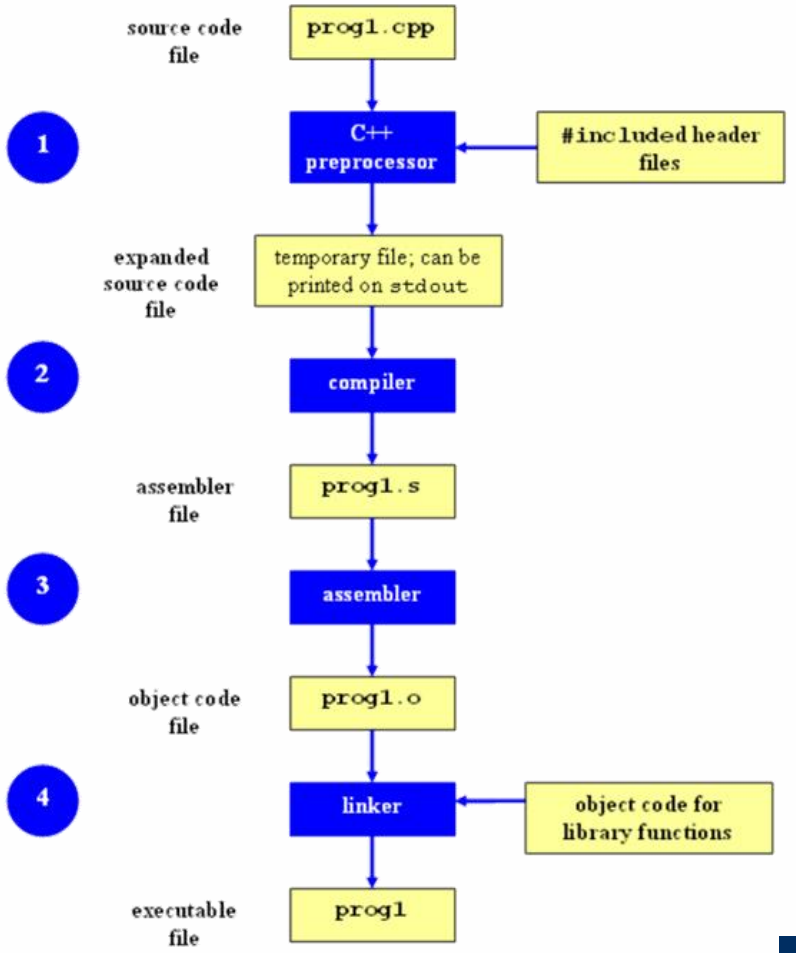
   ```
   g++ -Wall -std=c++11 -S prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.

3. To stop the process after the assembly step, you can use the `-c` option:

   ```
   g++ -Wall -std=c++11 -c prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*.



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

The C++ compilation process

Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named `prog1.cpp` and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using `#define` with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the `-E` option:

   ```
   g++ -Wall -std=c++11 -E prog1.cpp
   ```

   The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

2. To stop the process after the compile step, you can use the `-S` option:

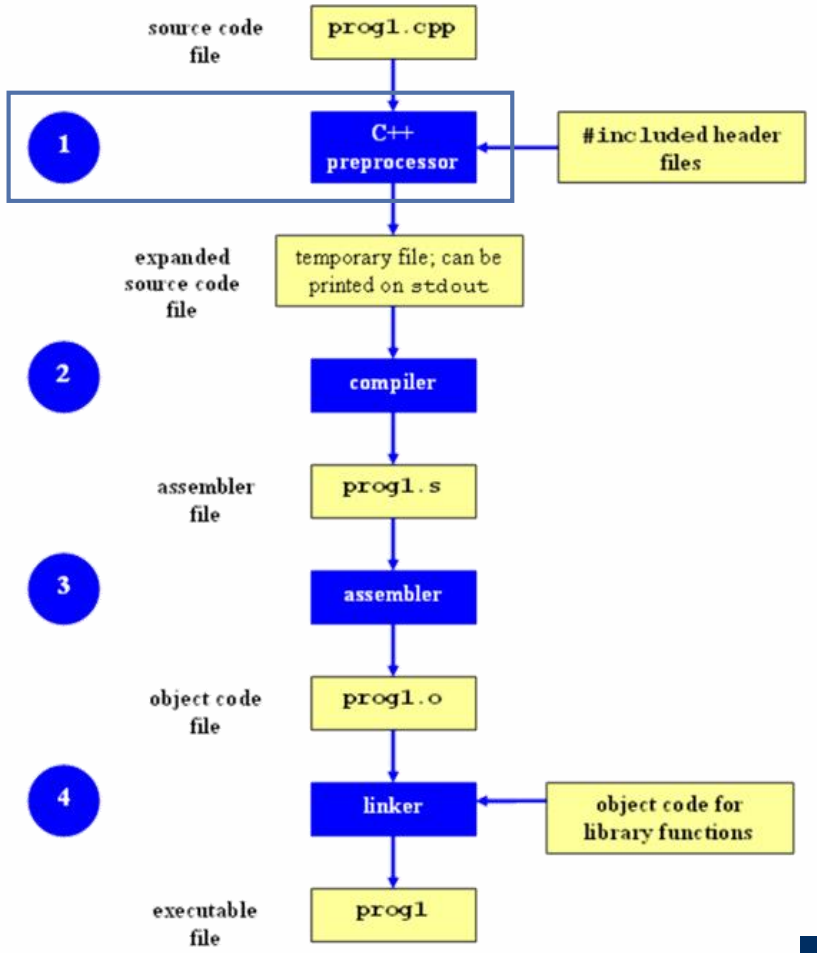
   ```
   g++ -Wall -std=c++11 -S prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.

3. To stop the process after the assembly step, you can use the `-c` option:

   ```
   g++ -Wall -std=c++11 -c prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*.



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

The C++ compilation process

Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named `prog1.cpp` and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using `#define` with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the `-E` option:

   ```
   g++ -Wall -std=c++11 -E prog1.cpp
   ```

   The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

2. To stop the process after the compile step, you can use the `-S` option:

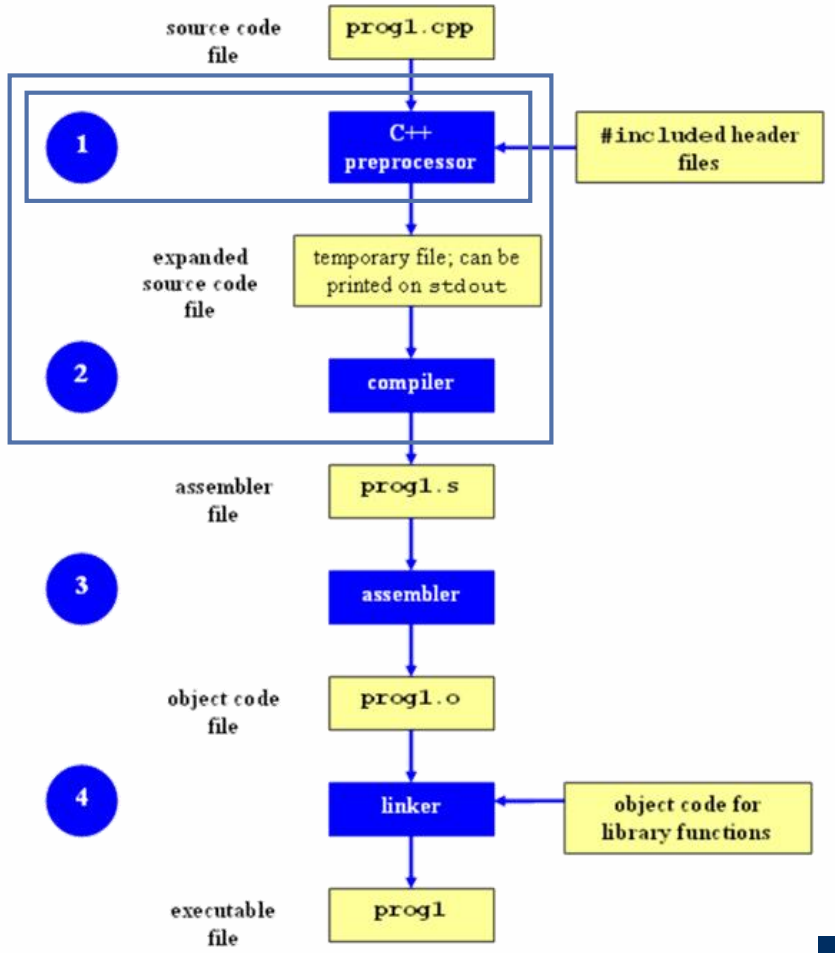
   ```
   g++ -Wall -std=c++11 -S prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.

3. To stop the process after the assembly step, you can use the `-c` option:

   ```
   g++ -Wall -std=c++11 -c prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*.



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

The C++ compilation process

Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named `prog1.cpp` and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using `#define` with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the `-E` option:

   ```
   g++ -Wall -std=c++11 -E prog1.cpp
   ```

   The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

2. To stop the process after the compile step, you can use the `-S` option:

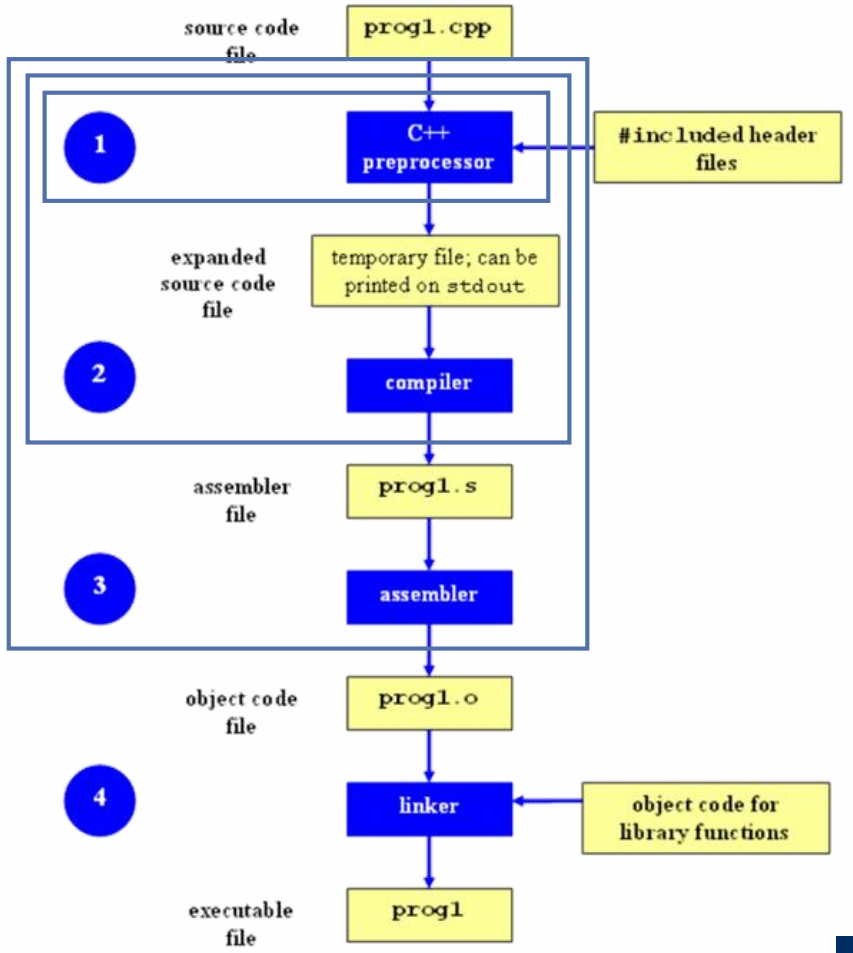
   ```
   g++ -Wall -std=c++11 -S prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.

3. To stop the process after the assembly step, you can use the `-c` option:

   ```
   g++ -Wall -std=c++11 -c prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*.



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

The C++ compilation process

Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named `prog1.cpp` and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using `#define` with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the `-E` option:

   ```
   g++ -Wall -std=c++11 -E prog1.cpp
   ```

   The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

2. To stop the process after the compile step, you can use the `-S` option:

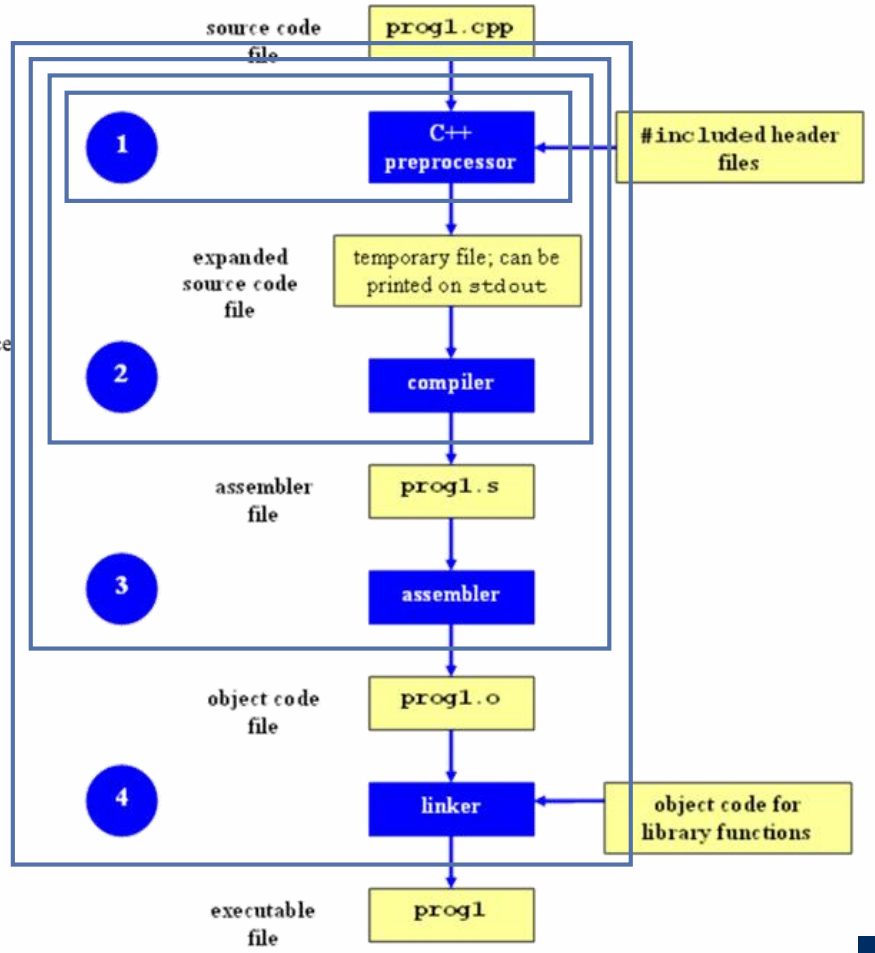
   ```
   g++ -Wall -std=c++11 -S prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.

3. To stop the process after the assembly step, you can use the `-c` option:

   ```
   g++ -Wall -std=c++11 -c prog1.cpp
   ```

   By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*.



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

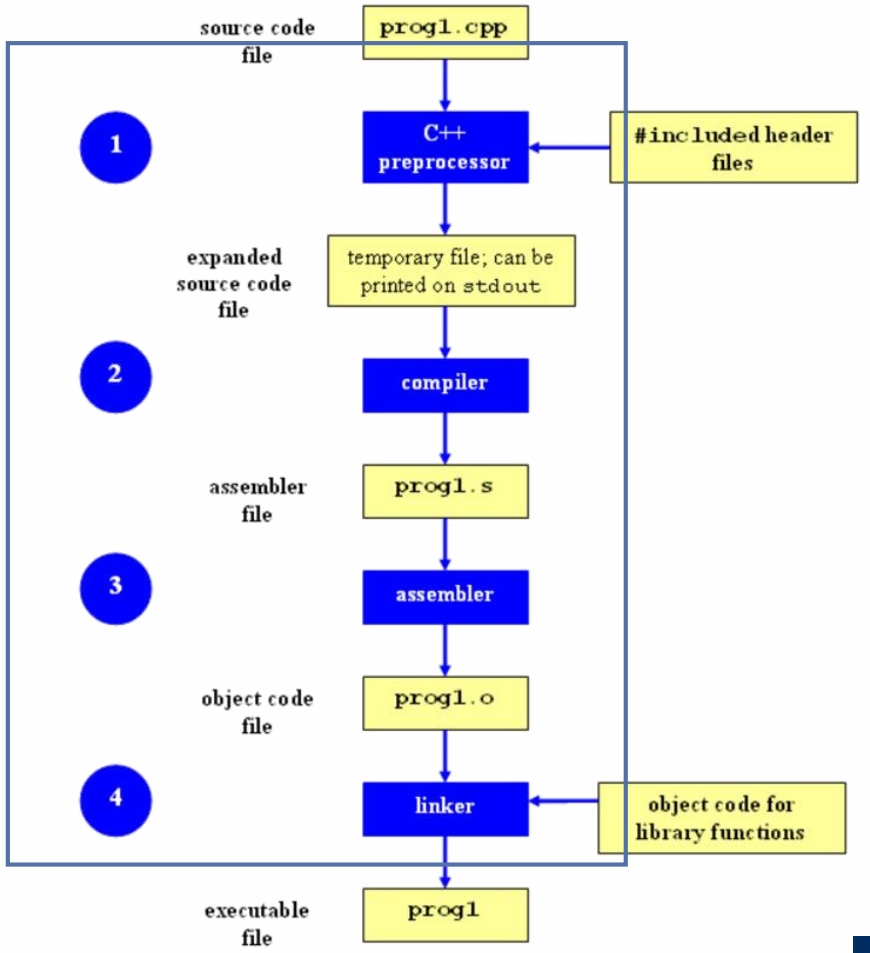
```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile (Optional)

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

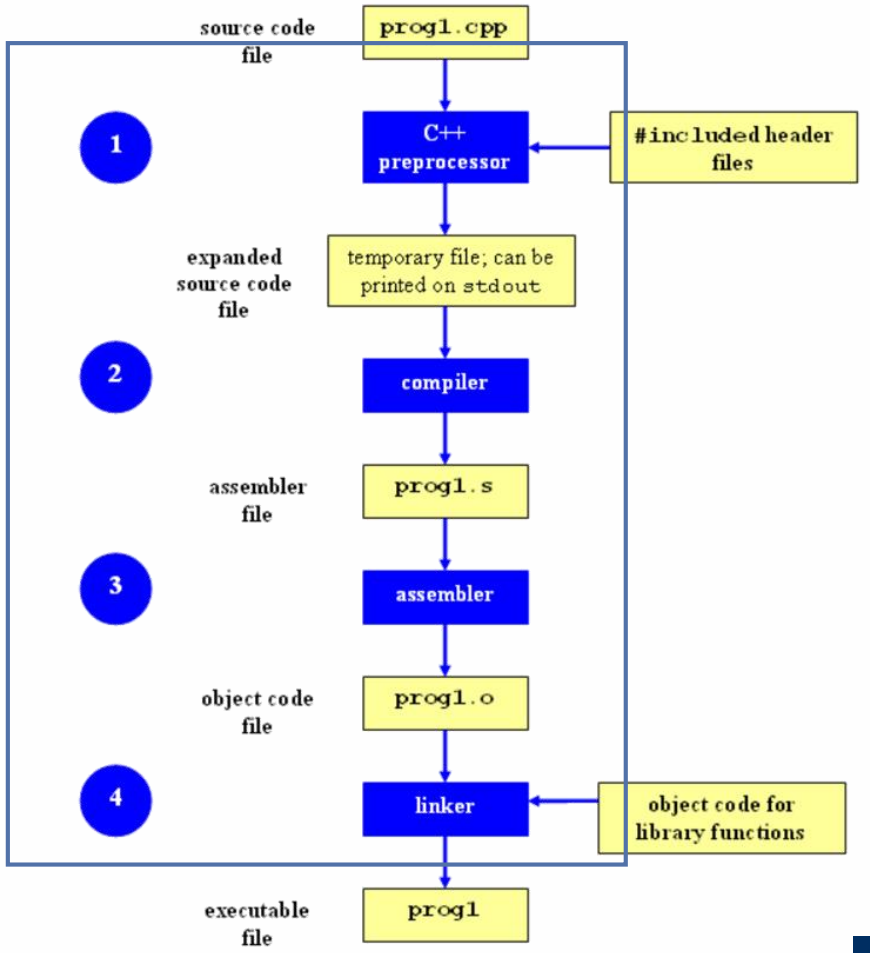
```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

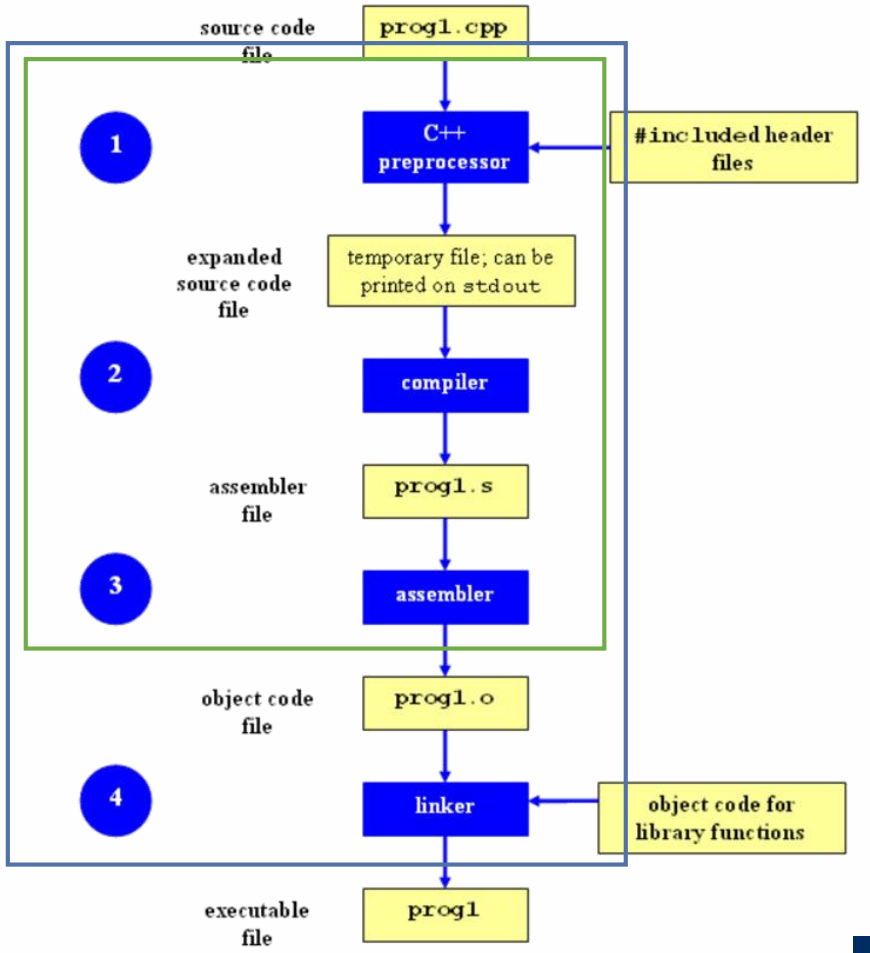
```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link — Just Compile them

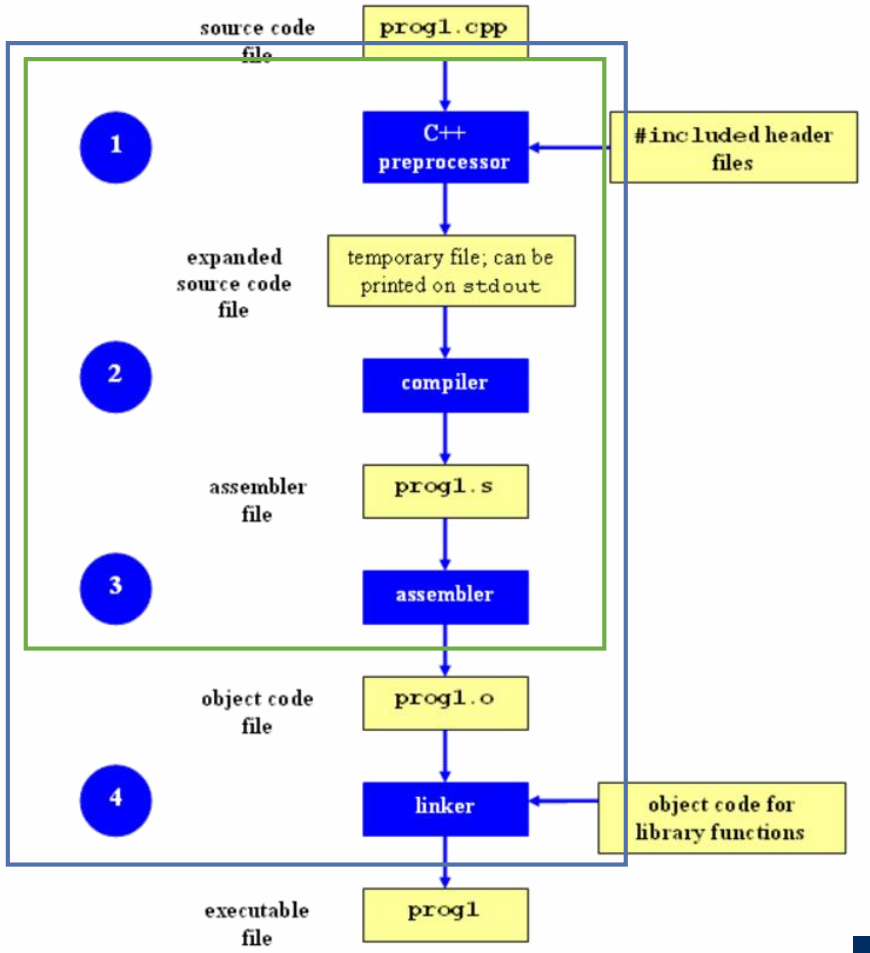
```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp
```

And then Link them

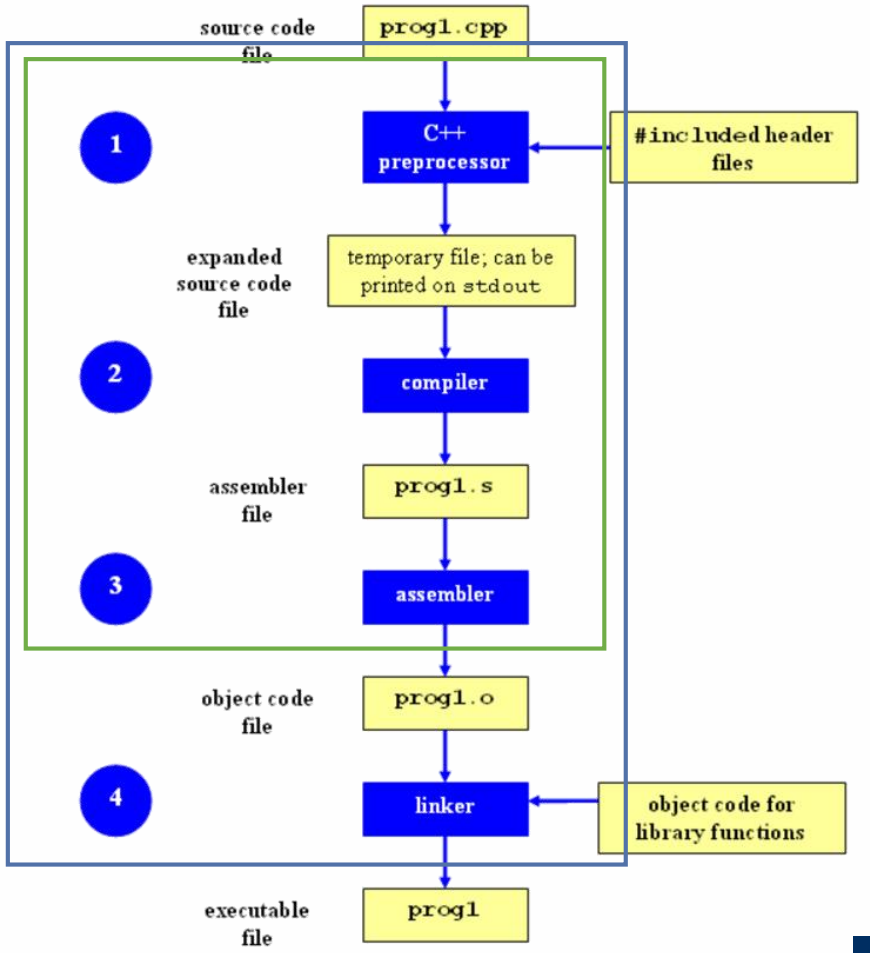
```
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

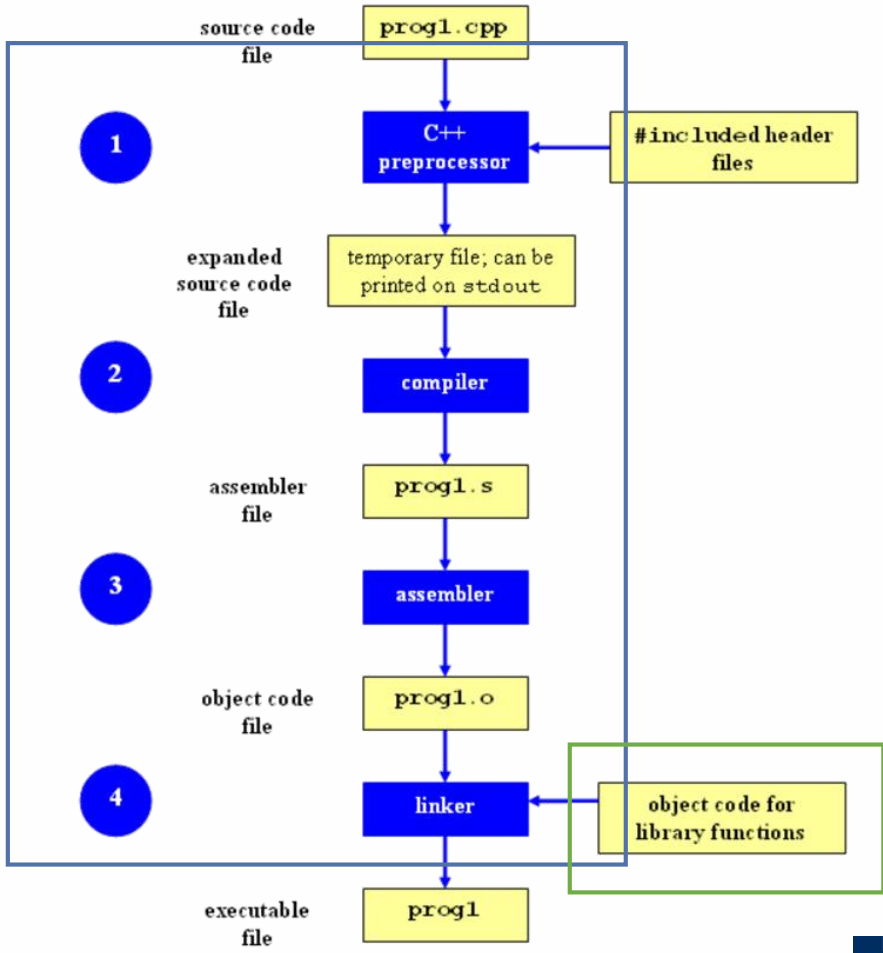
Linking — Also Link to an external Library **libm.so**

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Note: Actually **libstdc++** requires **libm**, so with **g++** it is automatically linked.

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

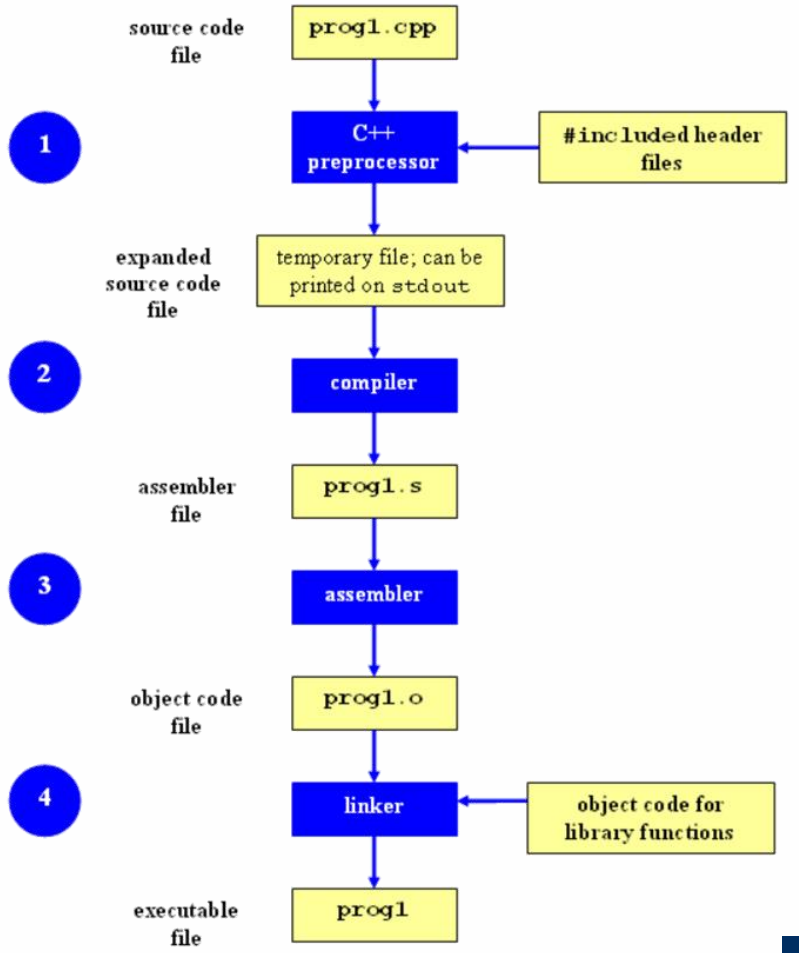
```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



C++ Compilation & Linkage

http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Standalone Compile

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp
```

Compile Objects & Link

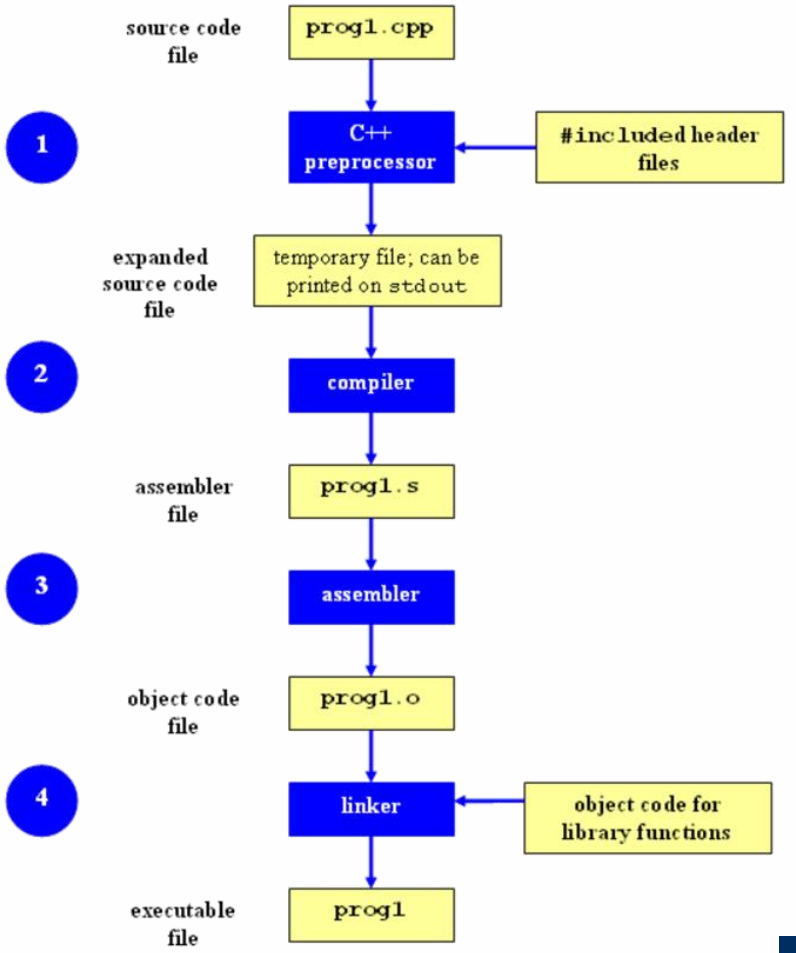
```
g++ -Wall -O3 -std=c++11 -c -o prog1.o prog1.cpp
g++ -Wall -O3 -std=c++11 -c -o prog2.o prog2.cpp

g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o
```

Linking

```
g++ -Wall -O3 -std=c++11 -o prog1 prog1.cpp -lm
g++ -Wall -O3 -std=c++11 -o prog12 prog1.o prog2.o -lm
```

Running under (Linux)

```
./prog1
```



Hands-on C++ Compilation

Install the **NoMachine™** Client:

https://www.nomachine.com/

Connect to the **CSE** Ubuntu Virtual Machine Environment:

https://unr.canvaslms.com/files/2324648

1) Right click on the desktop screen, choose "Create Document" → "Empty File", name it **`proj_1.cpp`**
2) Double-click to open it (with the default text editor – gedit), write your code and save it.
3) Back on the Desktop screen, click the Blue sign on the top-left, and then click on "Terminal Emulator"
4) In the terminal give the command "**`cd Desktop`**" to go to the Desktop folder
5) In the same terminal give the command "**`g++ -o proj_1 proj_1.cpp`**" to compile your code
6) If it compiles correctly you will have a **`proj_1`** file created which is your executable
7) You can run to test the executable by giving the command "**`./proj_1`**" in the same terminal screen. If it compiles and runs, you can take the source code (**`proj_1.cpp`**) and put it in an archive file (zip, tar.gz) together with the documentation file. Then upload this compressed archive file on WebCampus.
8) You will find the program that creates compressed files by clicking on the Blue sign on the top-left, and then go to "Accessories" → "Archive Manager". You can add the files to compress via drag-and-drop.

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

C++ Primer

```
1  #include <iostream>
2  using namespace std;

3  int main( )
4  {
5      int numberOfLanguages;

6      cout << "Hello reader.\n"
7           << "Welcome to C++.\n";

8      cout << "How many programming languages have you used? ";
9      cin >> numberOfLanguages;

10     if (numberOfLanguages < 1)
11         cout << "Read the preface. You may prefer\n"
12              << "a more elementary book by the same author.\n";
13     else
14         cout << "Enjoy the book.\n";

15     return 0;
16 }
```

Preprocessor directives (e.g. Includes), Function Prototypes

- also, **namespace**s

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

The main – Note: **returns** an **int**

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

Variable Definition

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

Console Output

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

Console Input

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

Selection Structures / Flow Control

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 0
Read the preface. You may prefer
a more elementary book by the same author.
```

User types in 0 on the keyboard.

SAMPLE DIALOGUE 2

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 1
Enjoy the book
```

User types in 1 on the keyboard.

C++ Primer

```
1   #include <iostream>
2   using namespace std;

3   int main( )
4   {
5       int numberOfLanguages;

6       cout << "Hello reader.\n"
7            << "Welcome to C++.\n";

8       cout << "How many programming languages have you used? ";
9       cin >> numberOfLanguages;

10      if (numberOfLanguages < 1)
11          cout << "Read the preface. You may prefer\n"
12               << "a more elementary book by the same author.\n";
13      else
14          cout << "Enjoy the book.\n";

15      return 0;
16  }
```

Output

C++ Identifiers & Variables

C++ Identifiers

Can't use keywords/reserved words.

Case-sensitivity and validity of identifiers.

Meaningful names!

Used for variables, class names, and more.

Variables

Must declare all data before use in program.

- A memory location to store data for a program.
- Referenced by the variable name.

Variable Declaration

C++ Variables

When we declare a variable, we tell the compiler:
When and where to set aside memory space for the variable.
How much memory to set aside.
How to interpret the contents of that memory;

- I.e., the specified data type.

```
int a ;
double b ;
```

What name we will be referring to that location in memory;

- I.e. by its identifier, or name.

```
int a ;
double b ;
```

Variable Declaration

C++ Variables

Syntax: `<type> <legal identifier> ;`

Examples:

```
int sum ;
float average ;
double grade = 98 ;
```

Must be declared before being used

Must be declared to be of a specific & known type (e.g. `int`, `float`, `char`, etc.)

Don't forget the semicolon at the end!

Variable Declaration

C++ Variables

Naming conventions are rules for names of variables to improve readability

- Different standards exist, suggested:
  - Start with a lowercase letter
  - Indicate "word" boundaries with an uppercase letter
  - Restrict the remaining characters to digits and lowercase letters

```
topSpeed    bankRate1    timeOfArrival
```

  - Indicate "word" boundaries with an underscore

```
top_speed    bank_rate_1    time_of_arrival
```

Note: variable names are still case sensitive!

Widely-used standard:

- Google standard https://google.github.io/styleguide/cppguide.html

Data Types

Primitive Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits
`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Data Types

Primitive Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits
`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Mostly seen/used primitive types.

Data Assignment

You can (and often *should*) initialize data in declaration statement

Note: Otherwise results *can* be "Undefined" if you don't initialize!

```
int myValue;      →  1; 0; -1; 234251; -32141521;   ???
int myValue = 0;  →  0;                            OK
```

Assigning data during execution

Value Categories (C++ Heritage from C) :

Lvalues (left-side) & *Rvalues* (right-side)

Note: Actually C++ has *GLvalues*, *Xvalues*, *PRvalues*)

Lvalues must be variables

Rvalues can be any expression

Note: More complicated than where they can appear, but keeping it simple for now …

Example: `distance = rate * time;`

Lvalue: "`distance`"

Rvalue: "`rate * time`" (Note: The *entire* expression)

Data Assignment

You can (and often *should*) initialize data in declaration statement

Note: Otherwise results *can* be "Undefined" if you don't initialize!

```
int myValue;   →  1; 0; -1; 234251; -32141521;   ???
int myValue = 0;  →  0;   OK
```

Assigning data during execution

Value Categories (C++ Heritage from C) :

Lvalues (left-side) & *Rvalues* (right-side)

Lvalues must be variables

Rvalues can be any expression

Note: "Reading" from an Uninitialized variable is **Undefined Behavior**

4.1 Lvalue-to-rvalue conversion

1 - A glvalue of a non-function, non-array type T can be converted to a prvalue. … If the object to which the glvalue refers is not an object of type T … or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.

Example: `distance = rate * time;`

Lvalue: "`distance`"

Rvalue: "`rate * time`" (Note: The *entire* expression)

Data Assignment

Compatibility of Data Assignments

- Type mismatches
  - Cannot place value of one type into variable of another type !
  - But sometimes, a *conversion* is possible …

`int intVar = 2.99;` ➡ `2` is assigned to **`intVar`**!

Only the integer part "fits", so that's all that goes …

Called "Implicit Type Conversion"

Literals

`2`, `5.75`, `'Z'`, `"Hello World\n"`

Also known as "constants": can't change in program

Literals / Literal Data

Examples:

```
2                       // Literal constant int
5.75                    // Literal constant double
'Z'                     // Literal constant char
"Hello World\n"         // Literal constant string
```

Cannot change their values during execution

Called "literals" because you "literally typed" them in your program!

Constants / Constant Data

You should not use literal constants directly in your code –

It might seem obvious to you, but not so:

```
limit = 52 ;
```

Is this weeks per year... or cards in a deck?

- Instead, use named constants.
- Give the constants names.

Also allows you to change multiple instances in a central place.

Constants / Constant Data

Two ways to go about this:

One way: **Preprocessor Define**

```
#define WEEKS_PER_YEAR 52
```

(Note: there is no "=")

This means that the Preprocessor will just replace the `WEEKS_PER_YEAR` text in the code before compilation.

- `WEEKS_PER_YEAR` itself is *not a variable* !

Alternative way: **Constant Variable**

Append "`const`" keyword to declaration

```
const float PI = 3.14159;
```

`PI` is *a fully-fledged variable* !

- This means we know its type and can dependably predict its behavior across code !

CS-202

Time for Questions !