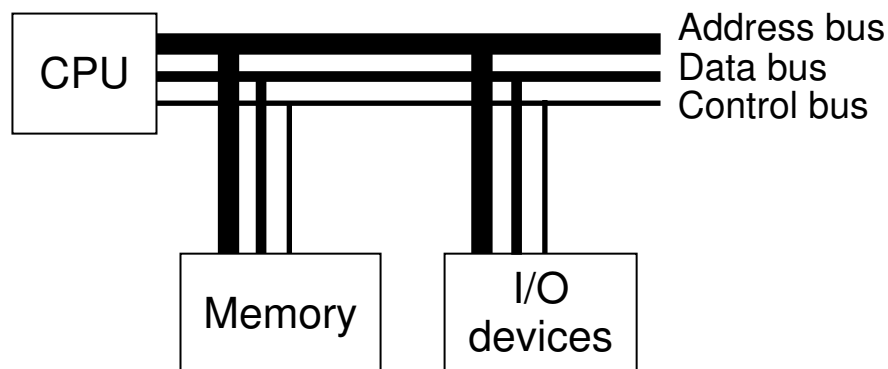


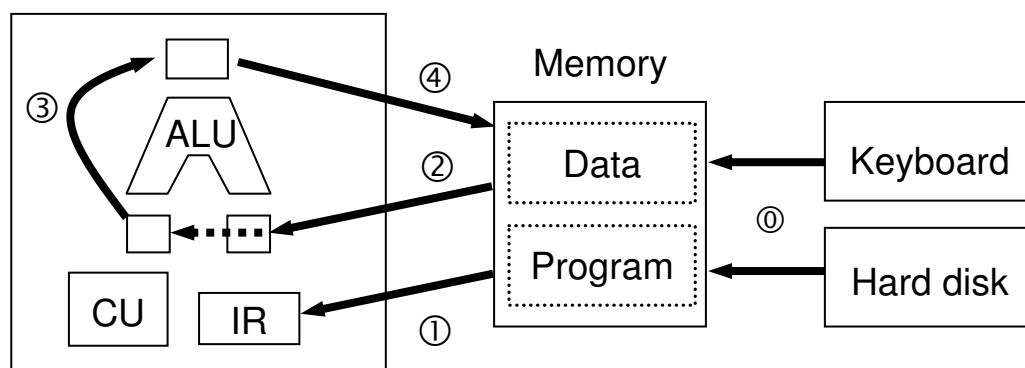
Lecture – Computer languages

Computer hardware

- Components of a computer



- Program execution



ALU Arithmetic and Logic Unit
CU Control Unit
IR Instruction Register

- ① Programs and data are loaded into memory before execution.

Four steps of execution

- ① Fetch
- ② Decode
- ③ Execute
- ④ Store

Computer languages

- Machine languages

Operation	Opcode	Memory address	
+	00	000	100
-	01	001	101
*	10	010	110
/	11	011	111

Machine instruction

00 001 110 011

Add the numbers stored in memory locations 001 and 110, and store the result in memory location 011

Machine languages are for machine, not for human.

- Assembly languages

- 1 Symbolize opcodes by CPU designers, e.g. Intel, AMD
- 2 Symbolize memory addresses by programmers

Operation	Opcode	Memory address	
+	ADD		
-	SUB	x	2
*	MUL		3
/	DIV	z	

Variables, e.g. x, y, and z, denote memory locations.

Assembly instruction

ADD x, y, z

Properties of machine and assembly languages

- 1 Machine dependent
- 2 One instruction per operation

Subtract x/y from $x*y$ and store the result in z

```
MUL x,y,R1
DIV x,y,R2
SUB R1,R2,z
```

where **R1** and **R2** are names of two registers.

- High-level languages

- 1 Machine independent
- 2 Easy to express complex computation

$z = x*y - x/y$

↙
assignment operator, rather than equality operator

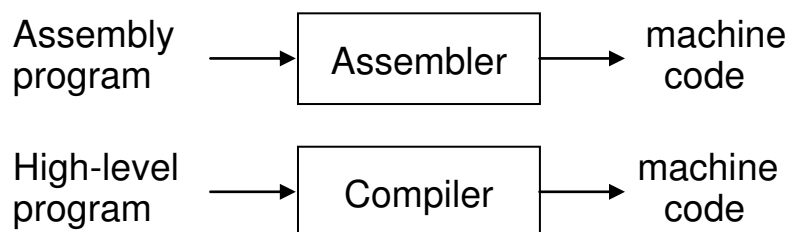
C-based languages use $=$, e.g. C, C++, C#, Java, Perl

Algol-like languages use $:=$, e.g. Algol60, Pascal, Ada

- Classification of high-level languages

- 1 Imperative languages, e.g. C, Algol60
- 2 Object-oriented languages, e.g. C++, Java
- 3 Functional languages, e.g. Scheme, SML, Haskell
- 4 Logic languages, e.g. Prolog

- Translators



Two main jobs of translators

- 1 Syntax analysis
- 2 Generate machine code, in case of no syntax errors

Problem solving

- Problem solving process
 - 1 Understand the problem, especially the I/O specification (which is of particular importance in online judge)
 - 2 Design an algorithm
 - 3 Coding
 - 4 Compile, test, and debug

- Algorithms

An algorithm is a step-by-step procedure for solving a problem.

Example – The summation problem

Input An integer $n \geq 0$

Output $1 + 2 + \dots + n$

Algorithm A – $O(1)^\dagger$, i.e. $O(n^0)$

Step 1 input n

Step 2 output $n*(n+1) / 2$

Algorithm B – $O(n)^\dagger$ accumulation algorithm

sum = 0

sum = sum + 1

sum = sum + 2

\vdots

sum = sum + n

} sum = sum + i , $i = 1, 2, \dots, n$

Step 1 input n

Step 2 sum = 0

Step 3 for $i = 1$ to n do sum = sum + i

Step 4 output sum

n

3

sum

0	1	3	6
---	---	---	---

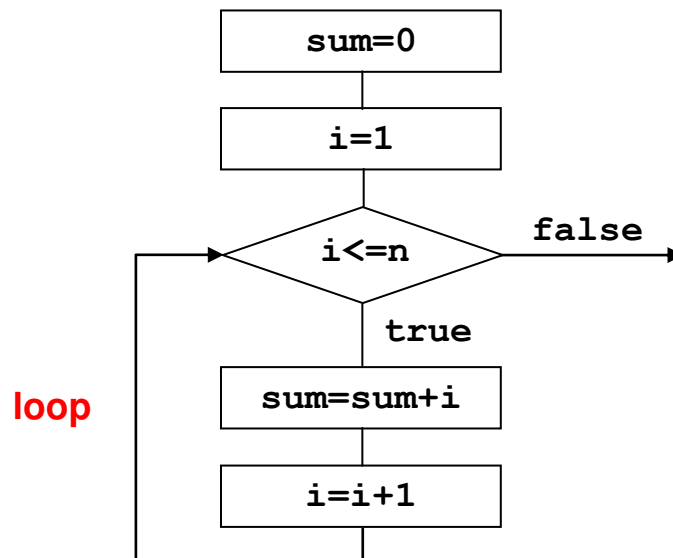
i

1	2	3	4
---	---	---	---

$^\dagger O(1)$ means constant running time.

$^\dagger O(n)$ means the running time is proportional to n .

- Flowchart



- Coding in C/C++

for statement — termination condition

`for (exp1; exp2; exp3) body;`

↑
initialization

↑
incrementation of the control variable

```
sum=0;
```

```
for (i=1; i<=n; i=i+1) sum=sum+i;
```

while statement

```
while (exp) body;
```

```
sum=0;
```

```
i=1;
```

```
while (i<=n) {
```

```
    sum=sum+i;
```

```
    i=i+1;
```

```
}
```

N.B. C/C++ are free format – a statement may occupy one or multiple lines as long as it is terminated by a ; or a }.

N.B. The loop body contains a single statement.

```
while (i<=n) } an infinite loop
    sum=sum+i;
    i=i+1;
```

Elements of a programming language

- Constant `0, 1`
Keyword `for, while`
Identifier `n, i, sum`
Operator `+, =, <=`

Expression – An expression yields a value.

Arithmetic expression	<code>sum+i, i+1</code>
Relational expression	<code>i<=n</code>
Assignment expression	<code>i=1, sum=0</code>

Statement – A statement does an action.

Iteration statement	<code>for, while</code>
Compound statement	<code>{ ... }</code>
Expression statement	<code>exp;</code>

Data type

Built-in type	<code>int, float, etc.</code>
User-defined type	

Function

Built-in (library) function	<code>sqrt, etc</code>
User-defined function	

- On the syntax of identifiers

An identifier begins with an `_` or a letter, followed by a possibly empty sequence of `_`'s, letters, and digits.

Identifiers are case-sensitive in C/C++, i.e.

`sum, Sum, sUm, suM, SUM, SuM, sUM, SUM`

are all distinct identifiers.

- On keywords and reserved words

In C/C++, keywords are reserved, i.e. they cannot be used as identifiers, e.g.

`for = 0;` `// illegal`

- On expression statements

Syntax *exp*;

Semantics Evaluate the expression *exp* and then discard its value

Meaningless cases

The expressions have no side effects, i.e. they only yield values.

sum+i;

i<=n; All are redundant.

5;

Meaningful cases

The expressions have side effects, i.e. they not only yield values but also do some actions.

i=i+1	Action	increment the value of i by 1
	Value	the updated value of i

i=i+1;	Action	increment the value of i by 1
---------------	--------	--------------------------------------

Note

Only two kinds of expressions have side effects – assignment expressions and I/O expressions.

For these expressions, more often than not, only side effects are desired, e.g. **sum=sum+i; i=i+1;**

Occasionally, their values are also wanted, e.g. **x=y=0;**

- On assignment operators

Simple assignment	=	sum=sum+i
-------------------	---	------------------

Compound assignment	+=, *=, etc	sum+=i
---------------------	-------------	---------------

Increment and decrement	++, --	
-------------------------	--------	--

- On increment and decrement operators

Prefix **++, --**

++i	Action	increment the value of i by 1
	Value	the updated value of i

++i;	Action	increment the value of i by 1
-------------	--------	--------------------------------------

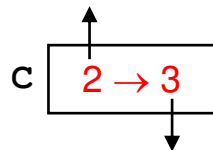
- On increment and decrement operators (Cont'd)

Postfix ++, --

<code>i++</code>	Action	increment the value of <code>i</code> by 1
	Value	the original value of <code>i</code>
<code>i++;</code>	Action	increment the value of <code>i</code> by 1

Example

value of `C++`



In every case, `C` becomes 3.

value of `++C`, `C=C+1`, and `C+=1`

Example

```
for (i=1; i<=n; i=i+1) sum=sum+i;
```

```
i=1;
while (i<=n) {
    sum=sum+i; i=i+1;
}
```

May be replaced by `i++`, `++i`, or `i+=1`

The following code makes use of the value of `i++`.

```
for (i=1; i<=n;) sum=sum+i++;
```

or, equivalently,

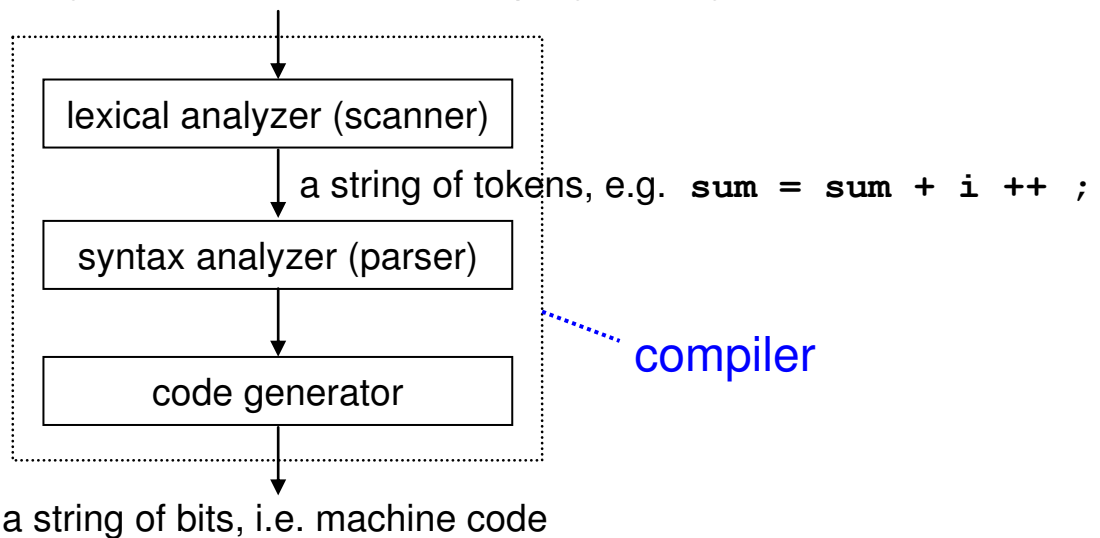
```
i=1;
while (i<=n) sum=sum+i++;
```

In the preceding code, `i++` can't be replaced by `++i`, `i=i+1`, or `i+=1`, as in

```
sum=sum+ ++i;    // sum=sum+++i; is different.
sum=sum+ (i+=1); // sum=sum+i+=1; is illegal.
```


- Comment on lexical analyzer and syntax analyzer

a string of characters, i.e. source program, e.g. `sum=sum+i++;`



Example

The following two statements are scanned differently.

`sum=sum+ ++i;`

`sum=sum+++i;`

The following two statements are parsed differently.

`sum=sum+ (i+=1) ;`

`sum=sum+i+=1;`

Rule – Tokens extend as far to the right as possible.

Q: What is the meaning of `sum+++i`, i.e. how is it scanned?

- 1) `sum + + + i`
- 2) `sum + ++ i`
- 3) `sum ++ + i`

Principle

Insert spaces whenever necessary to guide the scanner.

- Data types

Data have types, such as integer, real, boolean, character, string, etc.

Binary encoding

Data are encoded in binary form in one way or another.

Type checking

Check if the operands of an operation are of correct type.

For example, given

`sum + i`

do the variables `sum` and `i` hold values of numeric type?

Clearly, the machine cannot tell the type of the binary pattern stored in a memory location.

<code>sum</code>	<code>001 ... 110</code>
<code>i</code>	<code>000 ... 010</code>

One approach[†] requires the types of variables be declared by the programmer and charges the compiler with the task of type-checking.

<code>int sum,i,n;</code>	}	<code>int sum=0,i=1,n;</code>
<code>sum=0;</code>		
<code>i=1;</code>		
<code>while (i<=n) {</code>		
<code>sum=sum+i;</code>		
<code>i=i+1;</code>		
<code>}</code>		

Variables must be declared before they are used.

[†] This approach does type-checking at compile time. Another approach does type-checking at run time

- Functions

Mathematical functions

$$s : \mathbb{N} \rightarrow \mathbb{N}$$

$$s(n) = n(n + 1)/2$$

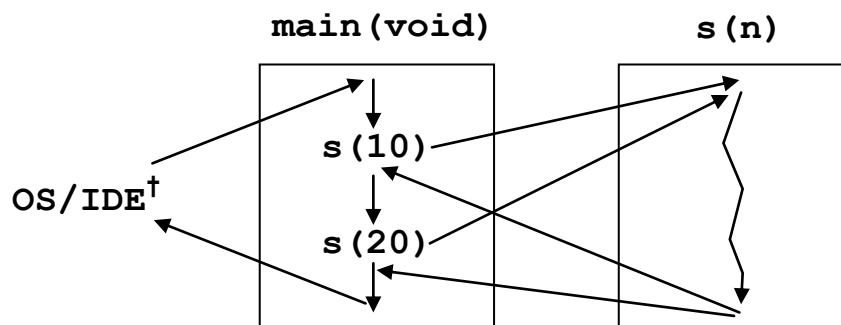
$$s(10) = 10(10 + 1)/2 = 55 \quad \text{substitute 10 for } n$$

Programming language functions

```
int s(int n)
{
    return n*(n+1)/2;
}
```

Unlike mathematical functions, programming language functions are executed.

1 Function call and return



[†] An IDE (Integrated Development Environment) integrates several programs, e.g. editor, compiler, debugger.

2 Parameter passing

actual parameter/argument formal parameter/parameter

```

s(10)                      int s(int n)      10 n
{
    return n*(n+1)/2;
}
```

Instead of substituting 10 for `n`, read the value 10 from the memory location for `n`.

- Functions (Cont'd)

- 3 Function body

A math function is usually defined by an expression; but, a programming function may contain assignments, loops, etc

- The function **main**

A C/C++ program shall contain a function called **main**.

Differences among C89 (or C90), C99, and C++ (1997, 2003)

N.B. ANSI – American National Standards Institute

ISO – International Organization for Standardization

- 1 It shall return a value of type **int** and may be passed command line arguments.

However, C99 allows an implementation to define its own extension of **main**, e.g. the return type needn't be **int**.

- 2 In C++, if control reaches the end of **main** without encountering a **return** statement, the effect is that of executing **return 0;**

In C89, the value returned is undefined.

C99 behaves like C++, if **main** is declared to return an **int**; otherwise, it behaves like C89.

- 3 It shall not be called from within a C++ program. But, it can be called from within a C program. N.B. $C \not\subset C++$

Throughout this course, we shall usually write

```
int main(void) { body }
```

and express no concern about the value returned to OS.

- On the return type of a function

In C89, if the return type of a function isn't specified, it defaults to **int**. But, in C99 and C++, the return type must always be specified. Thus,

```
s(int n){ return n*(n+1)/2; }
```

is legal in C89, but illegal in C99 and C++.

- **Type checking (revised)**

Check if the arguments, if any, of a function call are of correct type.

Functions must be declared before they are used.

```
int s(int n) { return n*(n+1)/2; }
int main(void)
{
    10+s(10);      // type check the calls to s and +
}
```

OR

```
int s(int);        // signature or prototype
int main(void)
{
    10+s(10);
}
int s(int n) { return n*(n+1)/2; }
```

Although the names of parameters are irrelevant with type checking, they may appear in a prototype:

```
int s(int n);      // Any parameter name will do.
```

Comments on C89

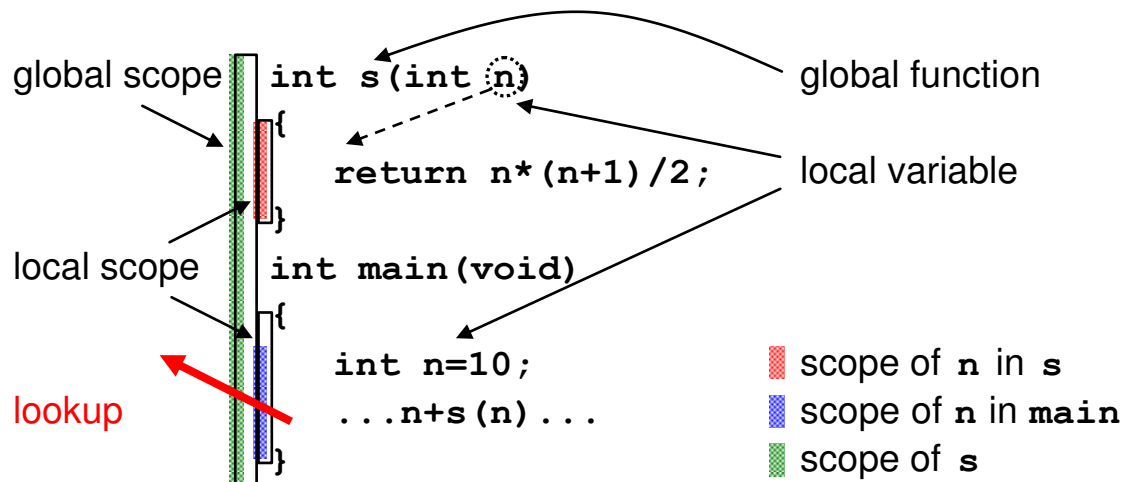
In fact, C89 is weak in type checking in that function prototypes may be omitted. In that case, function calls aren't type checked and are error-prone.

```
int main(void)
{
    s(7,8);        //
    s();           // All are erroneous, but aren't detected.
    s(3.14);       // Also, the behaviors are undefined.
}
int s(int n) { return n*(n+1)/2; }
```

Lesson – Don't omit function prototypes in C89.

- Block-structured languages

A block is a construct that delimits the scopes of names declared in it.



```

int s(int);
int main(void)
{
    int n=10;
    ...n+s(n)...
}
s is visible here
int s(int n)
{
    return n*(n+1)/2;
}

```

```

int main(void)
{
    int n=10;
    int s(int);
    ...n+s(n)...
}
s is invisible here
int s(int n)
{
    return n*(n+1)/2;
}

```

} int n=10, s(int);

N.B. Analogous to math functions

$$f(x, y) = x^2 + y^2$$

$$g(x) = y^2 - f(x - 1, y + 2) + 5, \text{ where } y = (x - 1)^2 / 2$$

Clearly, the two pairs of x and y don't interfere with each other. Also, the function g uses the function f .

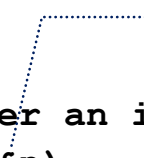
● Built-in (library) functions

Math `sqrt, abs, sin, ...`

I/O `printf, scanf, ...`

etc

```
#include <stdio.h>
int s(int n) { return n*(n+1)/2; }
int main(void)
{
    int n;
    printf("Enter an integer >=0: ");
    scanf("%d", &n)
    printf("1+2+...+%d=%d\n", n, s(n)); ①
}
```



Sample run `// the new line character`

Enter an integer >=0: 10↵

1+2+...+10=55

- ① The format string may contain both ordinary characters and conversion specifications.

Header files

Header files contain, among other things, signatures of built-in functions.

Math `math.h`

I/O `stdio.h`

Header files for built-in functions are usually included in the global scope, because they often contain certain declarations that have to be declared in the global scope.

- **Typical layout of C/C++ programs**

preprocessor commands

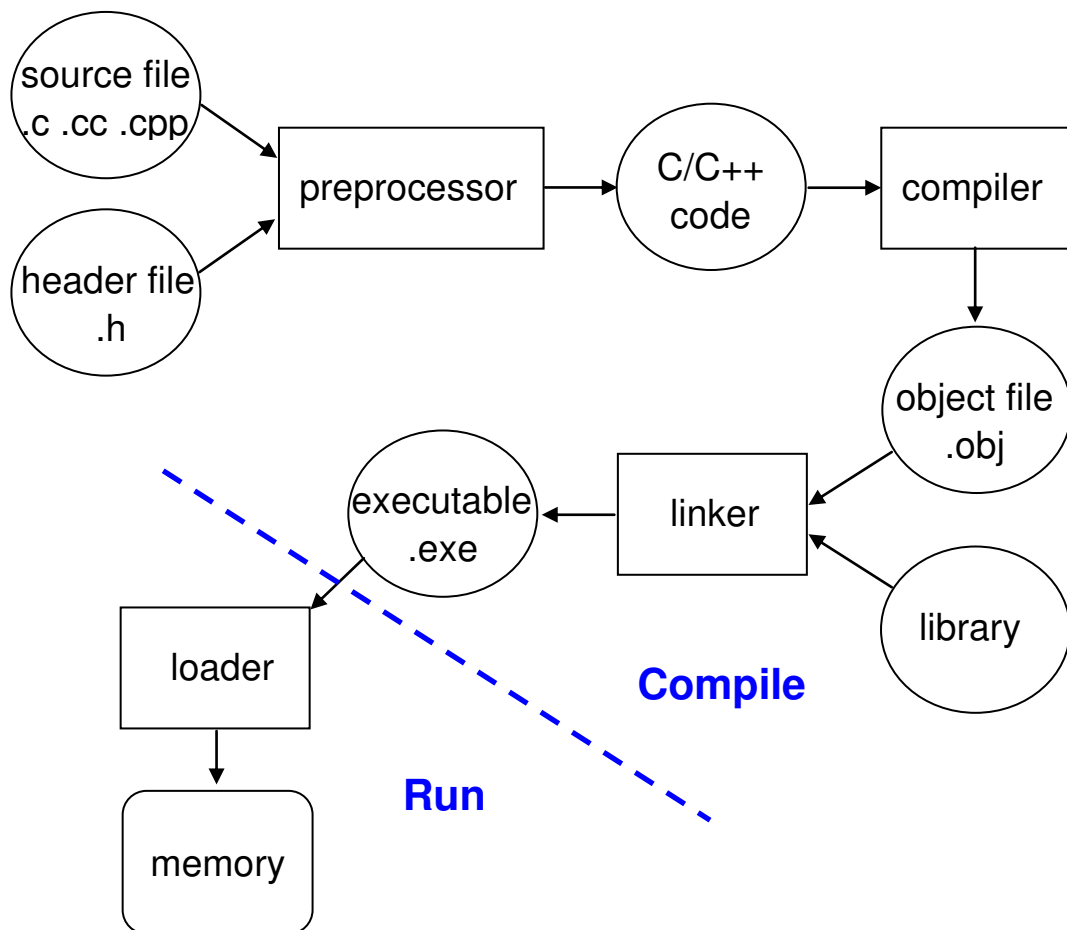
signatures of functions

```
int main(void) { ... }
```

definitions of other functions

N.B. C/C++ allows no local functions – all functions are global.

- **Preprocessor, compiler, linker, and loader**



Most compilers treat .c files as C programs, and .cc or .cpp files as C++ programs.