CSE1121
Structured & OOP Language

Sumaya Kazary
Assistant Professor
Department of Computer Science and Engineering
Dhaka University of Engineering & Technology, Gazipur

Acknowledgement

Thanks to the authors of all the books and online tutorials used in this slide.

Object Oriented Programming

Ref. Book:

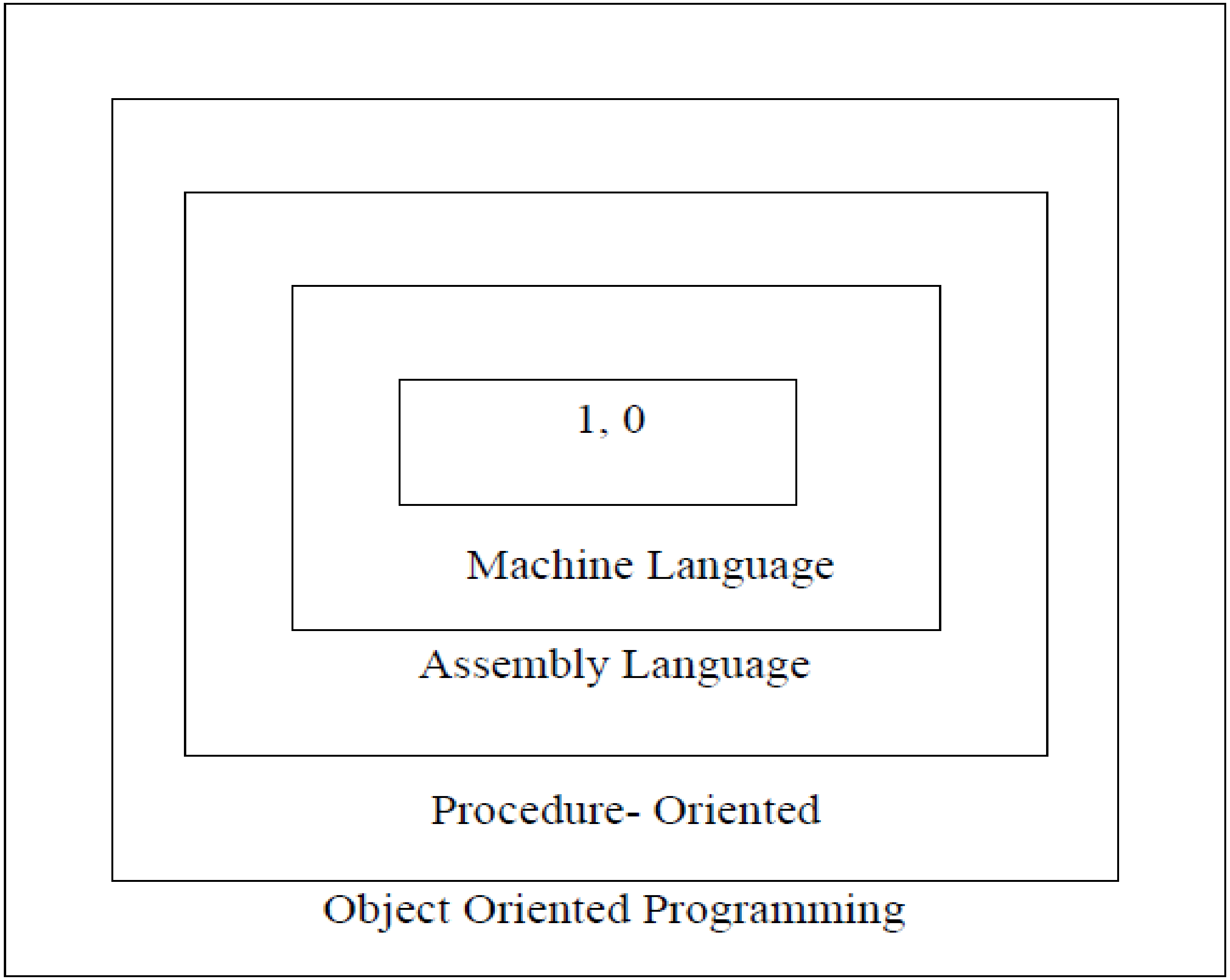
- Object oriented Programming by C++
  by Robert Lafore
- Teach yourself C++
  by H.schildt

An Overview of C++

Objectives

- Software Evaluation
- Two versions of C++
- Some differences between C and C++
- C++ console I/O
- C++ comments
- Introducing function overloading
- C++ keywords
- Function overloading
- Default Arguments
- Inline Functions

Software Evaluation



Structured Programming

- Using function
- Function & program is divided into modules
- Every module has its own data and function which can be called by other modules.

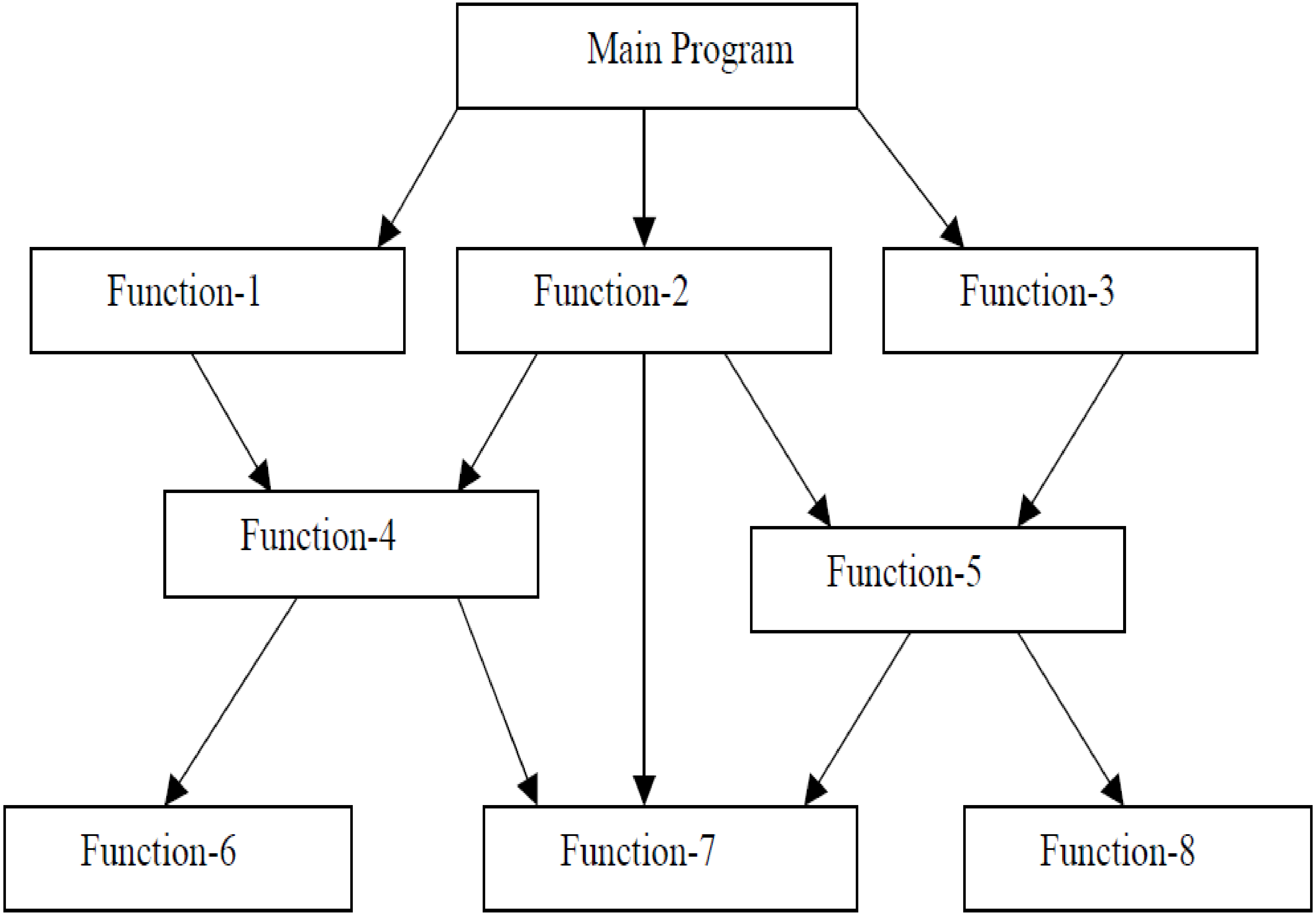
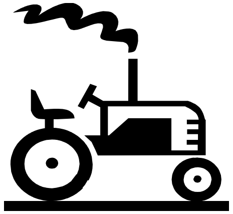


Fig. 1.2 Typical structure of procedural oriented programs



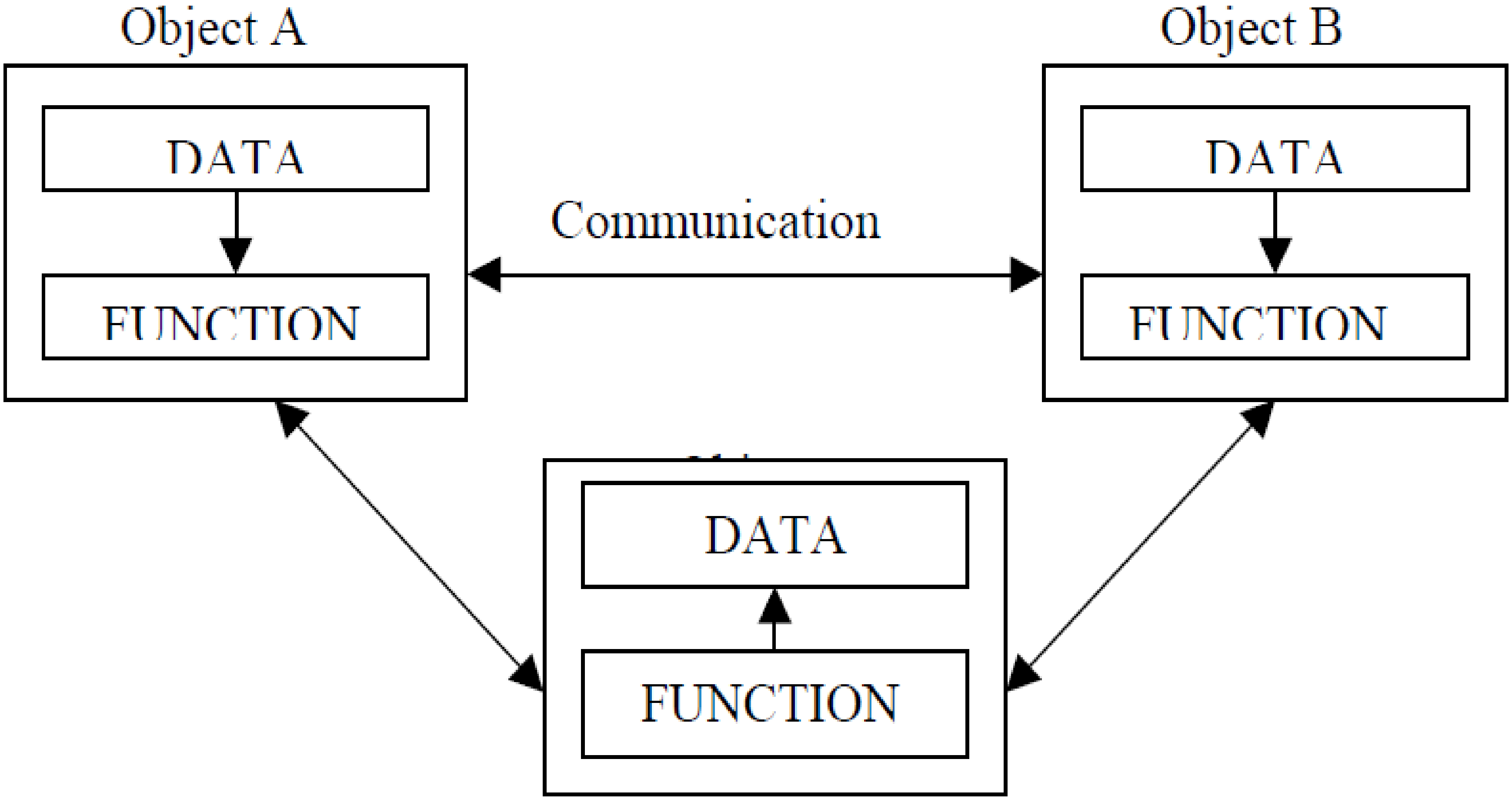
OBJECT ORIENTED PROGRAMMING

- Objects have both data and methods
- Objects of the same class have the same data elements and methods
- Objects send and receive *messages* to invoke actions

Key idea in object-oriented:

The real world can be accurately described as a collection of objects that interact.

Organization of data and function in OOP



Structure of Object-Oriented Programming

Classes

Blueprints of objects, attributes and methods

Objects

Are instances of a class

Methods

Describe the behavior of an object

Attributes

Represent the state of an Object

Background of C++

- C++ was developed by Bjarne Stroustrup at Bell Laboratories
    - Originally called “C with Classes”
    - The name C++ is based on C’s increment operator (++)
        - Indicating that C++ is an enhanced version of C

- Widely used in many applications and fields
- Well-suited to “Programming in the Large”

INTRODUCTION

- C++ is the C programmer’s answer to Object-Oriented Programming (OOP).
- C++ is an *enhanced version of the C language*.
- C++ adds support for OOP without sacrificing any of C’s power, elegance, or flexibility.
- Both object-oriented and non-object-oriented programs can be developed using C++.

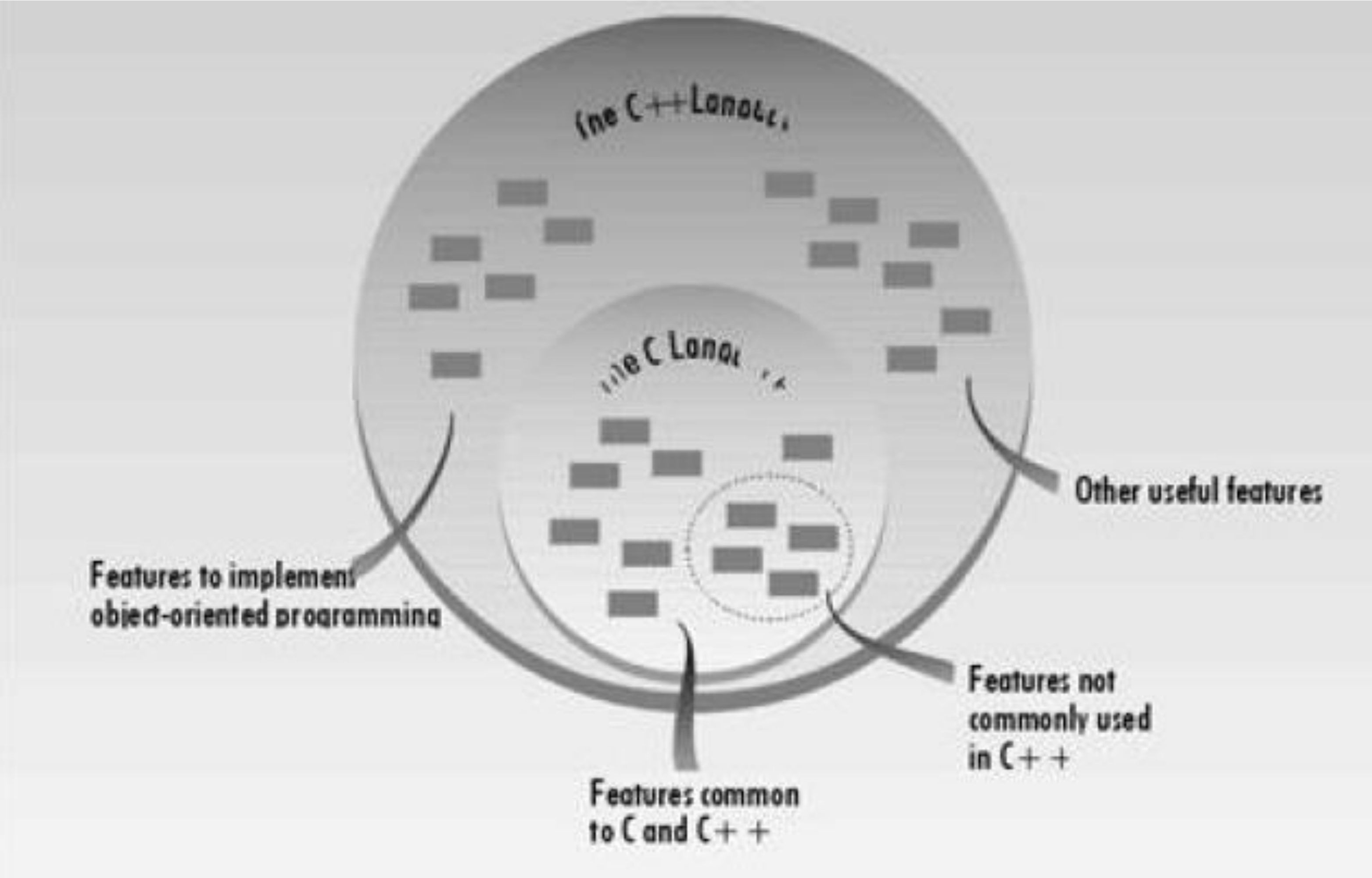

The C++ Language
The C Language
Other useful features
Features to implement object-oriented programming
Features not commonly used in C++
Features common to C and C++

Fig 1.4: The relationship between C and C++.

TWO VERSIONS OF C++

- A traditional-style C++ program -

```
#include <iostream.h>

int main()
{
        /* program code */
        return 0;
}
```

TWO VERSIONS OF C++ (CONT.)

- A modern-style C++ program that uses the new-style headers and a namespace -

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

int main()
{
        /* program code */
        return 0;
}
```

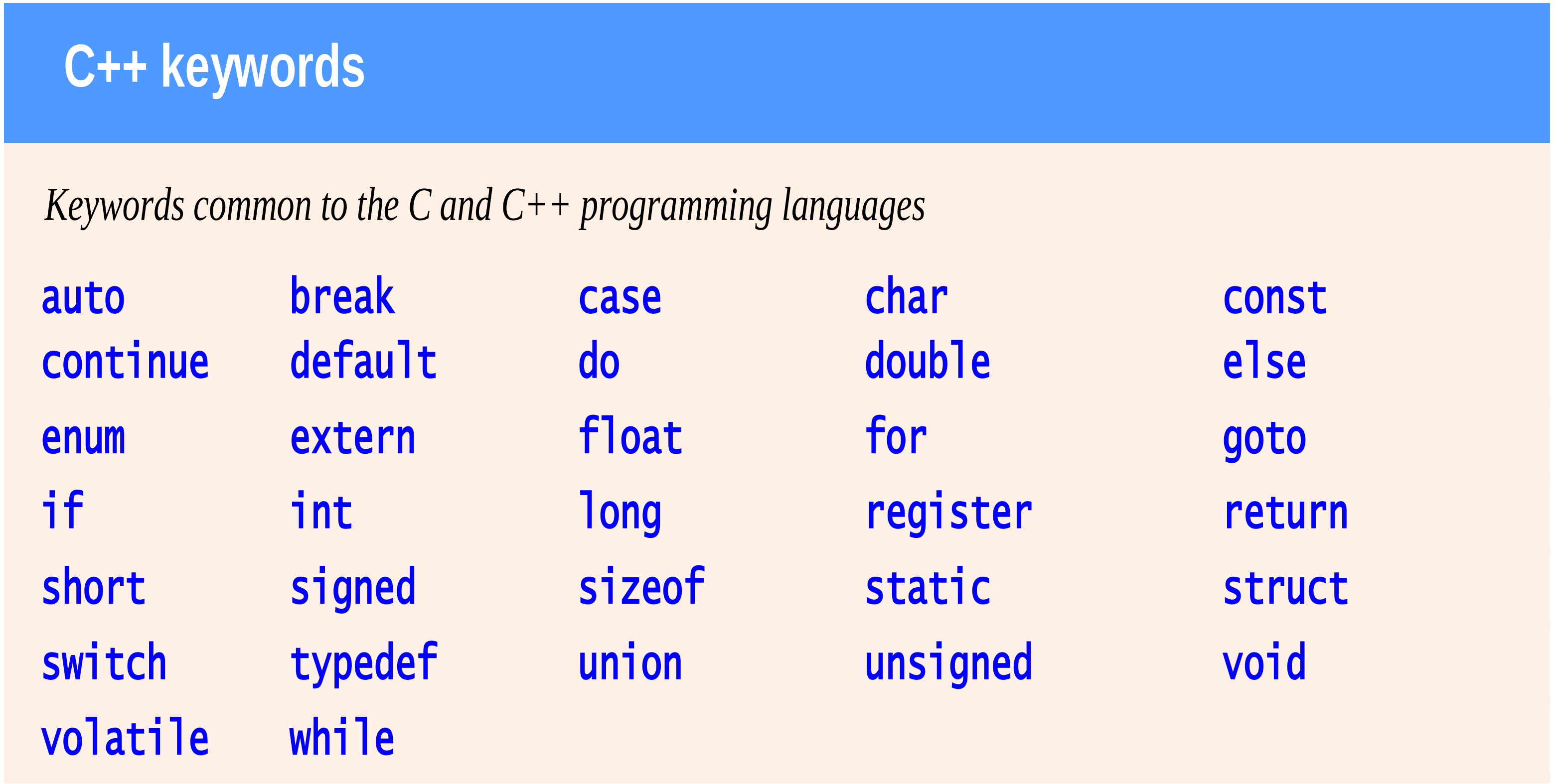
Some Differences Between C and C++

- All functions **must be** prototyped.
- If a function is declared as returning a value, it ***must* return a value.**
- Return type of all functions must be declared explicitly.
- ***Local variables*** can be declared ***anywhere***.
- C++ defines the **bool** datatype, and keywords **true** (any nonzero value) and **false** (zero).

IMPORTANT DIFFERENCES BETWEEN C & C++

Key	C	C++
Introduction	C was developed by Dennis Ritchie in around 1969 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979.
Language Type	As mentioned before C is procedural programming.	On the other hand, C++ supports both procedural and object-oriented programming paradigms.
OOPs feature Support	As C does not support the OOPs concept so it has no support for polymorphism, encapsulation, and inheritance.	C++ has support for polymorphism, encapsulation, and inheritance as it is being an object-oriented programming language
Data Security	As C does not support encapsulation so data behave as a free entity and can be manipulated by outside code.	On another hand in the case of C++ encapsulation hides the data to ensure that data structures and operators are used as intended.
Driven type	C in general known as function-driven language.	On the other hand, C++ is known as object driven language.
Feature supported	C does not support function and operator overloading also do not have namespace feature and reference variable functionality.	On the other hand, C++ supports both function and operator overloading also have namespace feature and reference variable functionality.

Keywords Shared with *C*

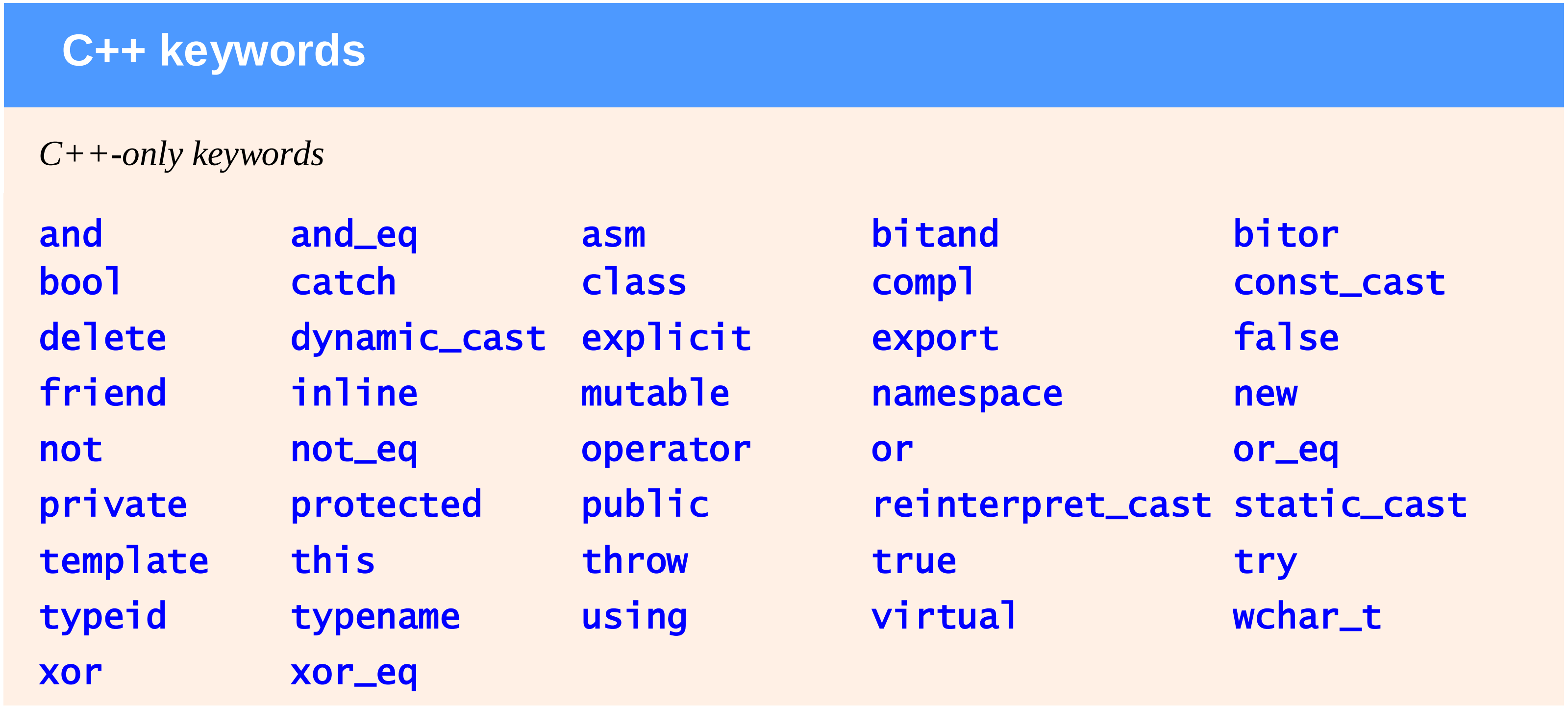


C++ keywords

Keywords common to the C and C++ programming languages

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

New Keywords in C++



C++ keywords

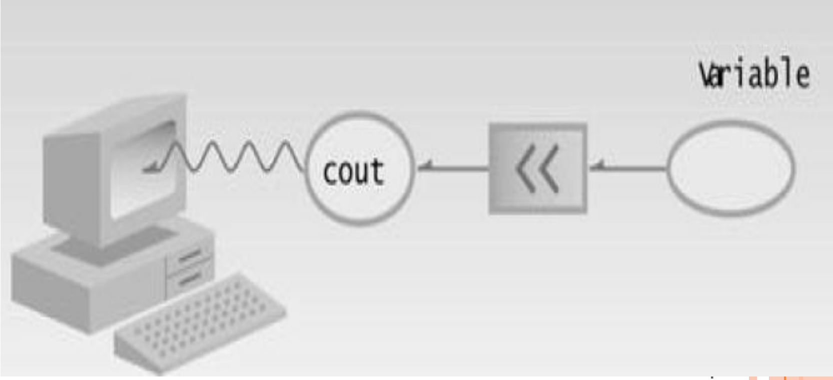
C++-only keywords

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

The New C++ Headers

- The new-style headers do not specify filenames.
- They simply specify standard identifiers that might be mapped to files by the compiler, but they need not be.
  - <iostream>
  - <vector>
  - <string>, not related with <string.h>
  - <cmath>, C++ version of <math.h>
  - <cstring>, C++ version of <string.h>
- Programmer defined header files should end in ".h".

C++ Console I/O (Output)

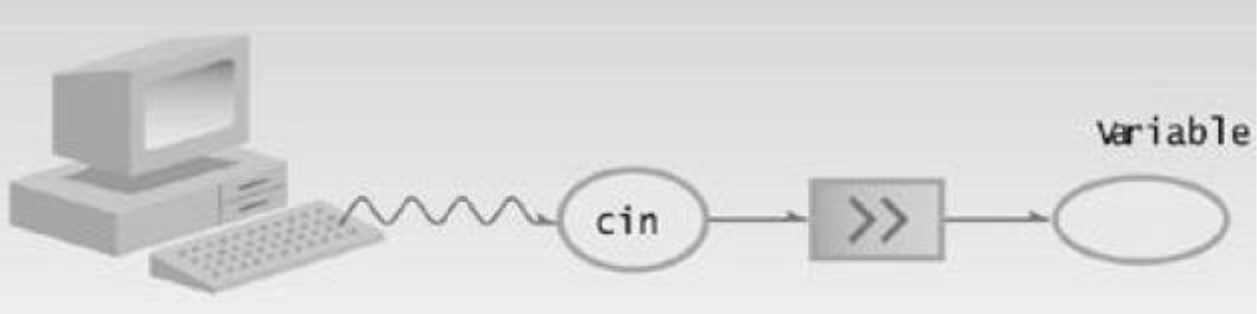


- cout << “Hello World!”;
  - printf(“Hello World!”);
- cout << iCount;   /* int iCount */
  - printf(“%d”, iCount);
- cout << 100.99;
  - printf(“%f”, 100.99);
- cout << “\n”, ***or*** cout << ‘\n’, ***or*** cout << endl
  - printf(“\n”)

- In general, **cout << *expression***;

C++ Console I/O (Input)

- cin >> strName; /* char strName[16] */
  - scanf("%s", strName);
- cin >> iCount; /* int iCount */
  - scanf("%d", &iCount);
- cin >> fValue; /* float fValue */
  - scanf("%f", &fValue);
- In general, **cin >> *variable*;**



C++ Console I/O (I/O chaining)

- cout << “Hello” << ‘ ‘ << “World” << ‘!’;
- cout << “Value of iCount is: ” << iCount;
- cout << “Enter day, month, year: ”;
- cin >> day >> month >> year;
  - cin >> day;
  - cin >> month;
  - cin >> year

C++ Console I/O (example)

```
include <iostream>
int main()
{
   char str[16];
   std::cout << “Enter a string: ”;
   std::cin >> str;
   std::cout << “You entered: ” << str;
    return 0;
}
```

```
include <iostream>
using namespace std;
int main()
{
   char str[16];
   cout << “Enter a string: ”;
   cin >> str;
   cout << “You entered: ” << str;
   return 0;
}
```

C++ Comments

- Multi-line comments
  - /* one or more lines of comments */
- Single line comments
  - // …

Introducing Function Overloading

- Provides the mechanism by which C++ achieves one type of polymorphism (called **compile-time polymorphism**).
- Two or more functions can share the same name as long as either
  - The type of their arguments differs, OR
  - The number of their arguments differs, OR
  - Both of the above

Introducing Function Overloading (cont.)

- The compiler will automatically select the correct version.
- The return type alone is not a sufficient difference to allow function overloading.
- **Example:** p-34.cpp, p-36.cpp, p-37.cpp.

Q. Can we confuse the compiler with function overloading?
A. Sure. In several ways. Keep exploring C++.

Example

// abs is overloaded three ways

```
int abs(int n);
long abs(long n);
double abs(double);
void main(){
    cout<<“Absolute value of -10:”<<abs(-10)<<endl;
    cout<<“Absolute value of -10L:”<<abs(-10L)<<endl;
    cout<<“Absolute value of -10.01:”<<abs(-10.01)<<endl;
 }
int abs(int n){
    cout<<“In integer abs()<<endl;
     return n<0 ? –n : n;
}
```

```
long abs(long n){
    cout<<“In long abs()<<endl;
     return n<0 ? –n : n;
}
```

```
double abs(double d)
{
cout << "Using double abs()\n";
return d<0.0 ? -d : d;
}
```

USING DEFAULT ARGUMENTS

- It is related to function overloading.
  - Essentially a shorthand form of function overloading
- It allows to give a parameter a default value when no corresponding argument is specified when the function is called.
  - void f1(int a = 0, int b = 0) { … }
  - It can now be called in three different ways.
    - f1();   // inside f1() 'a' is '0' and b is '0'
    - f1(10);   // inside f1() 'a' is '10' and b is '0'
    - f1(10, 99); // inside f1() 'a' is '10' and b is '99'
  - We can see that we cannot give 'b' a new (non-default) value without specifying a new value for 'a'.
  - So while specifying non-default values, **we have to start from the leftmost parameter and move to the right one by one.**

USING DEFAULT ARGUMENTS

- Default arguments must be specified only once: either in the function's prototype **OR** in its definition.
- All default parameters must be to the right of any parameters that don't have defaults.
  - void f2(int a, int b = 0); // no problem
  - void f3(int a, int b = 0, int c = 5); // no problem
  - void f4(int a = 1, int b); // compiler error
- So, once you begin to define default parameters, you cannot specify any parameters that have no defaults.
- Default arguments must be constants or global variables. They cannot be local variables or other parameters.

USING DEFAULT ARGUMENTS

- Relation between default arguments and function overloading.
  - void f1(int a = 0, int b = 0) { … }
  - It acts as the same way as the following overloaded functions –
    - void f2( ) { int a = 0, b = 0; … }
    - void f2(int a) { int b = 0; … }
    - void f2(int a, int b) { … }
- Constructor functions can also have default arguments.
- It is possible to create copy constructors that take additional arguments, as long as the additional arguments have default values.
  - MyClass(const MyClass &obj, int x = 0) { … }
- This flexibility allows us to create copy constructors that have other uses.
- <u>See the examples from the book to learn more about the uses of default arguments.</u>

Overloading and Ambiguity

- Due to automatic type conversion rules.
  - Example 1:
    - void f1(float f) { … }
    - void f1(double d) { … }
    - float x = 10.09;
    - double y = 10.09;
    - f1(x); // unambiguous – use f1(float)
    - f1(y); // unambiguous – use f1(double)
    - f1(10); // ambiguous, compiler error
      - Because integer ‘10’ can be promoted to both “float” and “double”.

OVERLOADING AND AMBIGUITY (contd.)

- Due to the use of reference parameters.
  - Example 2:
    - void f2(int a, int b) { … }
    - void f2(int a, int &b) { … }
    - int x = 1, y = 2;
    - f2(x, y);
      - // ambiguous, compiler error

OVERLOADING AND AMBIGUITY (contd.)

- Due to the use of default arguments.
- Example 3:
  - void f3(int a) { … }
  - void f3(int a, int b = 0) { … }
  - f3(10, 20);
    - unambiguous – calls f3(int, int)
  - f3(10);
    - // ambiguous, compiler error

In-line Functions

- Inline function is a **C++ enhancement** designed to speed up programs

- When normal function is called, processor will usually save all the register information, memory information, then jump to the location of the function

- When the normal function finish execution, it will then restore all the registers and memory information, then jump back to the point in the program after the function execution

- With C++ inline function, C++ compiler compiles the function "in line" with the other code in the program

In-line Functions

- Functions that are not actually called but, rather, are expanded in line, at the point of each call.

- The compiler replaces the function call with the corresponding function code in the machine language level, so no jump of function call is necessary

- **Advantage**
  - Have no overhead associated with the function call and return mechanism.
  - Can be executed much faster than normal functions.
  - Safer than parameterized macros. ***Why ?***
- **Disadvantage**
  - If they are too large and called too often, the program grows larger.

Syntax Note: Inline Functions

- Preface the function definition with the keyword **inline**

- Place the function definition above all the functions that call it

- Note that you have to place the entire definition (meaning the function header and all the function code), not just the prototype, above the other functions

In-line Functions

- The **inline** specifier is a ***request*****,** not a command, to the compiler.
- An inline function **must be defined before** it is first called.
- Some compilers will **not in-line a function** if it contains
  - A **static** variable
  - A **loop**, **switch** or **goto**
  - If the function is **recursive**

In-line Functions

```
inline int even(int x)
 {
        return !(x%2);
 }

int main( )
{
    if(even(10)) cout << “10 is even\n”;
                                        // becomes if(!(10%2))

    if(even(11)) cout << “11 is even\n”;
                                        // becomes if(!(11%2))

    return 0;
}
```

```
10 is even
```

inline.cpp -- use an inline function

```
#include <iostream.h>
// an inline function must be defined before first use
inline double square(double x) { return x * x; }
int main(void)
{
    double a, b;
    double c = 13.0;
    a = square(5.0);
    b = square(4.5 + 7.5);   // can pass expressions
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c squared = " << square(c++) << "\n";
    cout << "Now c = " << c << "\n";
    return 0;
}
```

Automatic In-lining

- Defining a member function inside the class declaration causes the function to automatically become an in-line function.

- In this case, the **inline** keyword is no longer necessary.
  - However, it is not an error to use it in this situation.

- Restrictions
  - Same as normal in-line functions.

Automatic In-lining

```
// Manual in-lining
class myclass
{
   int a;
public:
   myclass(int n);
   void set_a(int n);
   int get_a();
};
inline void myclass::set_a(int n)
{
   a = n;
}
```

```
// Automatic in-lining
class myclass
{
   int a;
public:
   myclass(int n) { a = n; }
   void set_a(int n) { a = n; }      int
   get_a() { return a; }
};
```

Lecture Contents

- Teach Yourself C++
  - Chapter 1 (1.1-1.4,1.6-1.7)
  - Chapter 2 (2.6-2.7)
  - Chapter 5 (5.4-5.5)