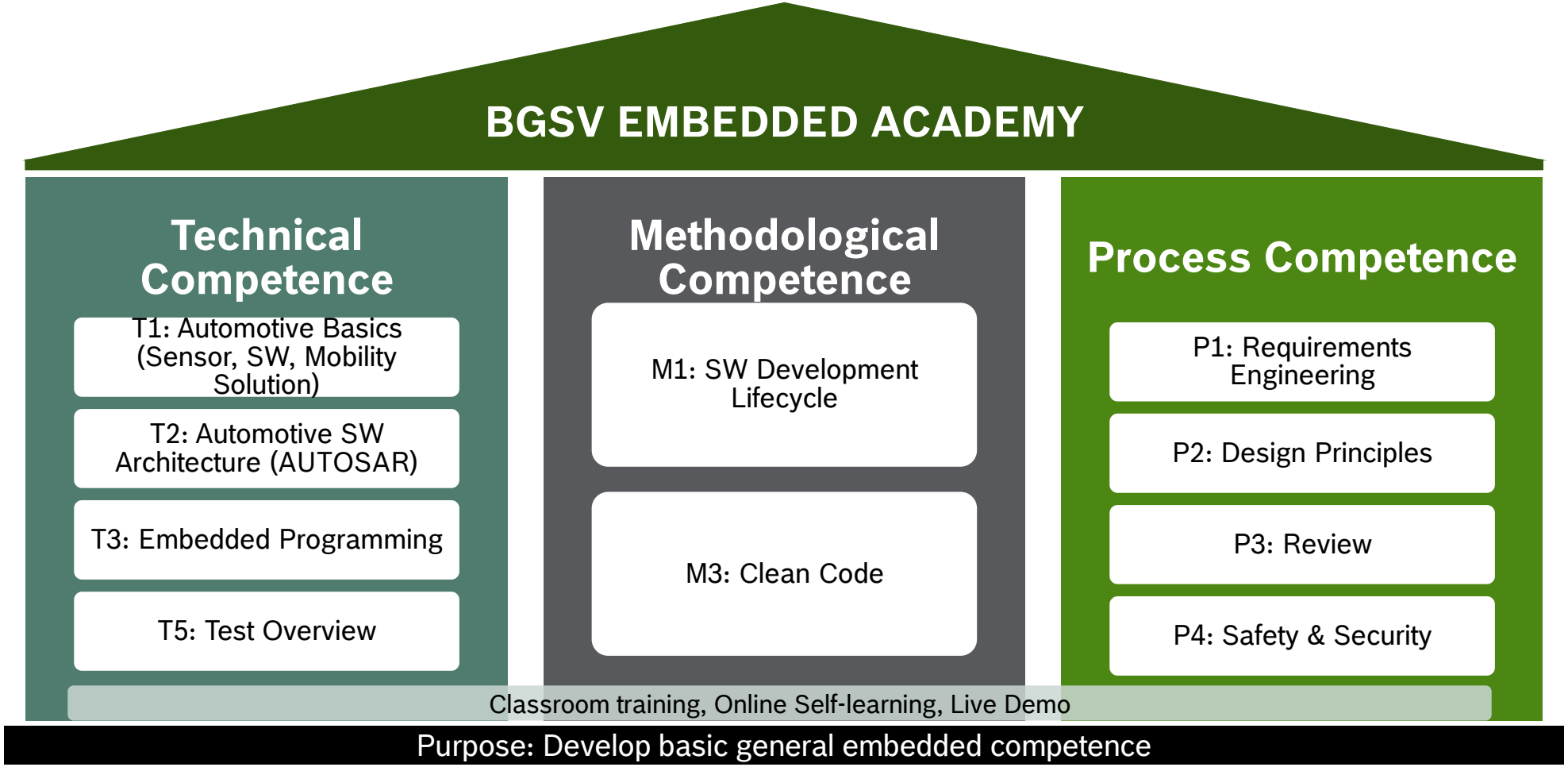
EMBEDDED ACADEMY

PEDAL TO THE MEDAL

BOSCH

BGSV Embedded Academy (BEA)

Focused Program to Develop Embedded Competence

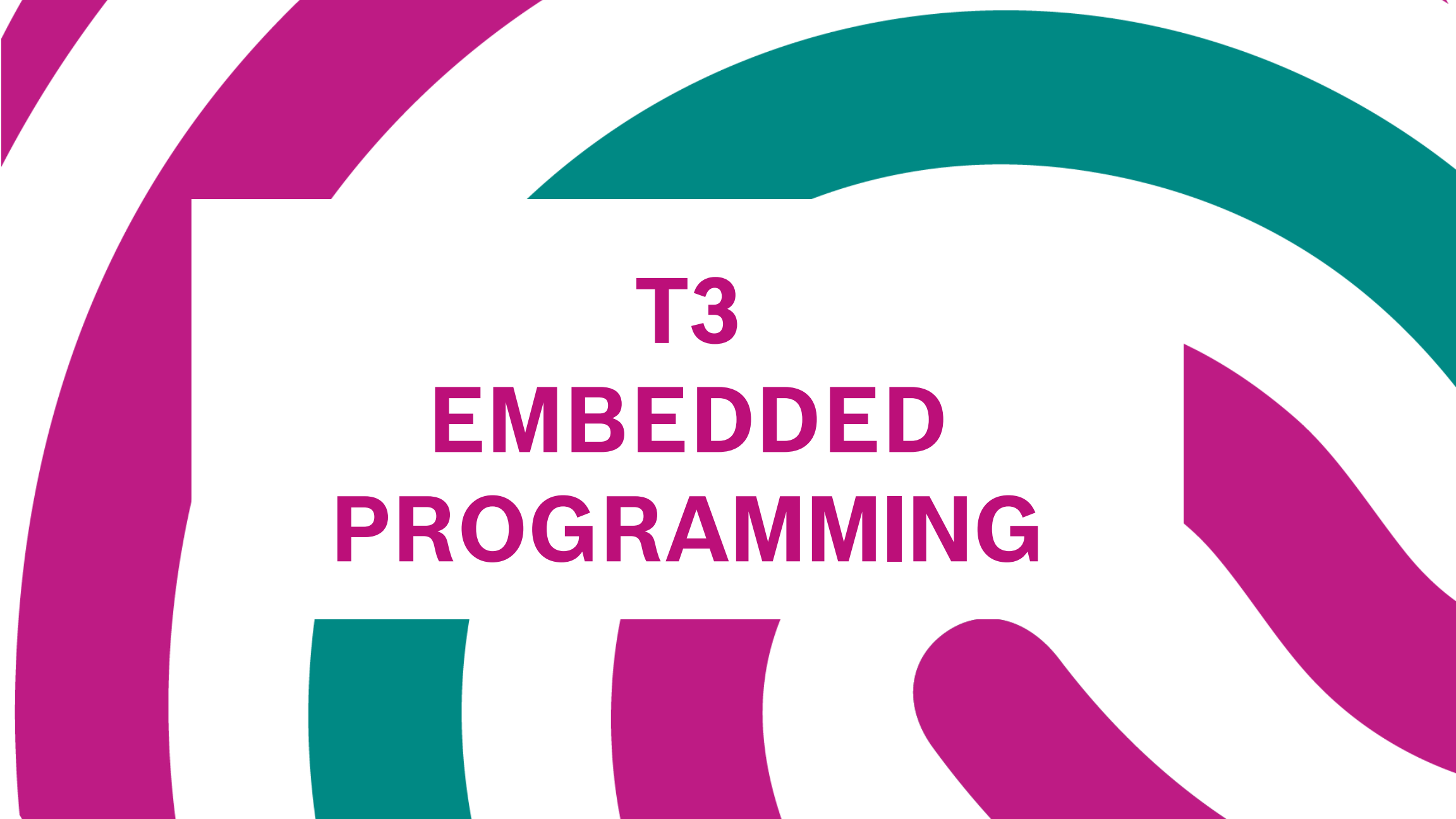


Bosch Global Software Technologies
alt_future

Disclaimer

- This slide is a part of BGSV Embedded Academy (BEA) program and only used for BEA training purposes.
-
- This slide has some copyright images and text, which belong to the respective organizations.

T3
EMBEDDED
PROGRAMMING



Objectives & Assumptions

Remind general programming principles
Remind key embedded system knowledges
How to program for embedded system effectively
Code optimization & Secure coding
Know and practice Object-oriented programming

Experienced with programming at basic level
Not focus on how to programming
C and C++

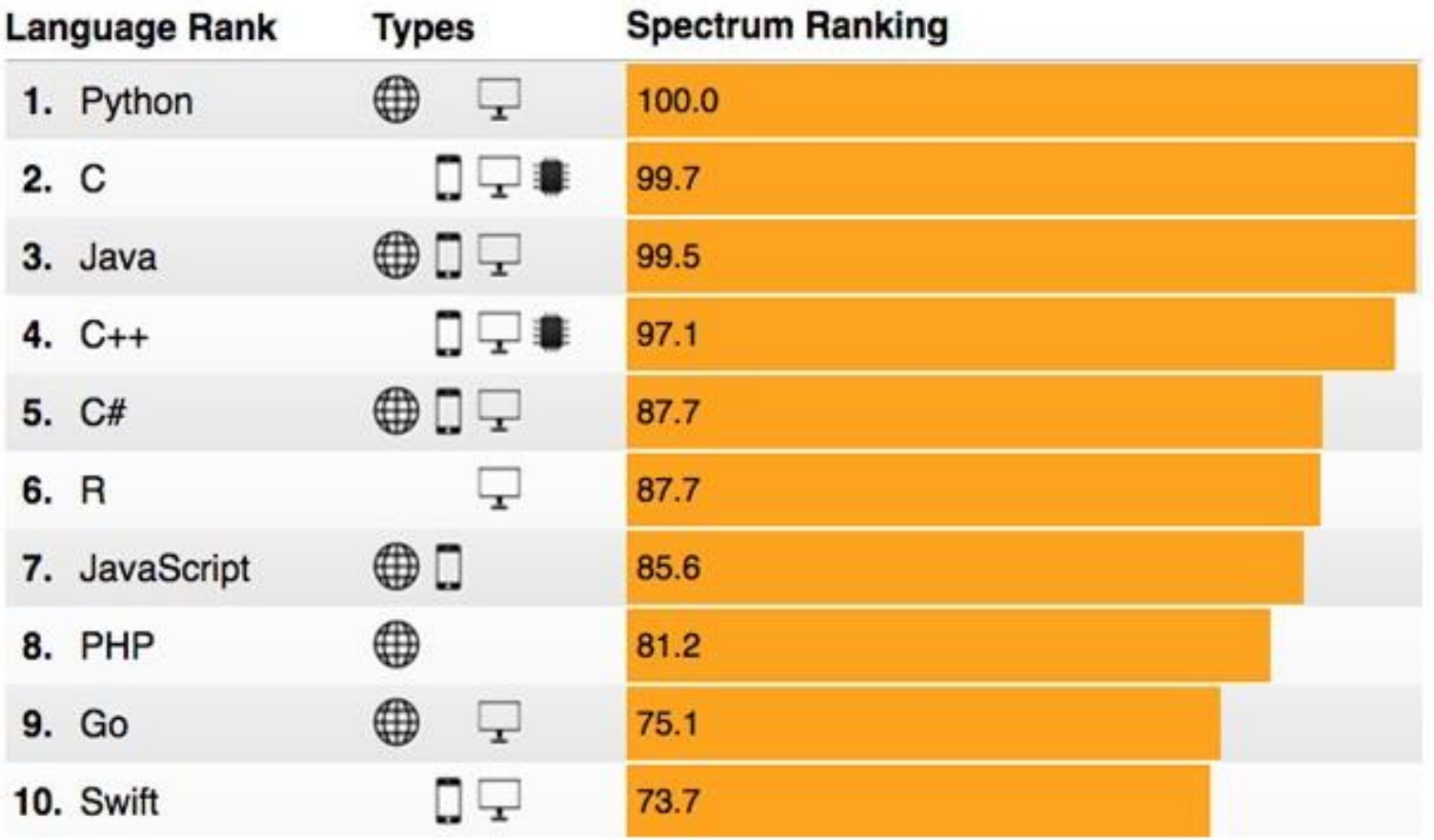
Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process
2. Embedded system programing
    1. What is embedded system?
    2. Which common elements inside Embedded SW?
    3. Constraints affect design choices?
    4. Why C is most common language for Embedded programing?
    5. What skills required for Embedded programmer?
    6. RTOS
3. Some importance topics that related to embedded programming
4. OOP principles basic

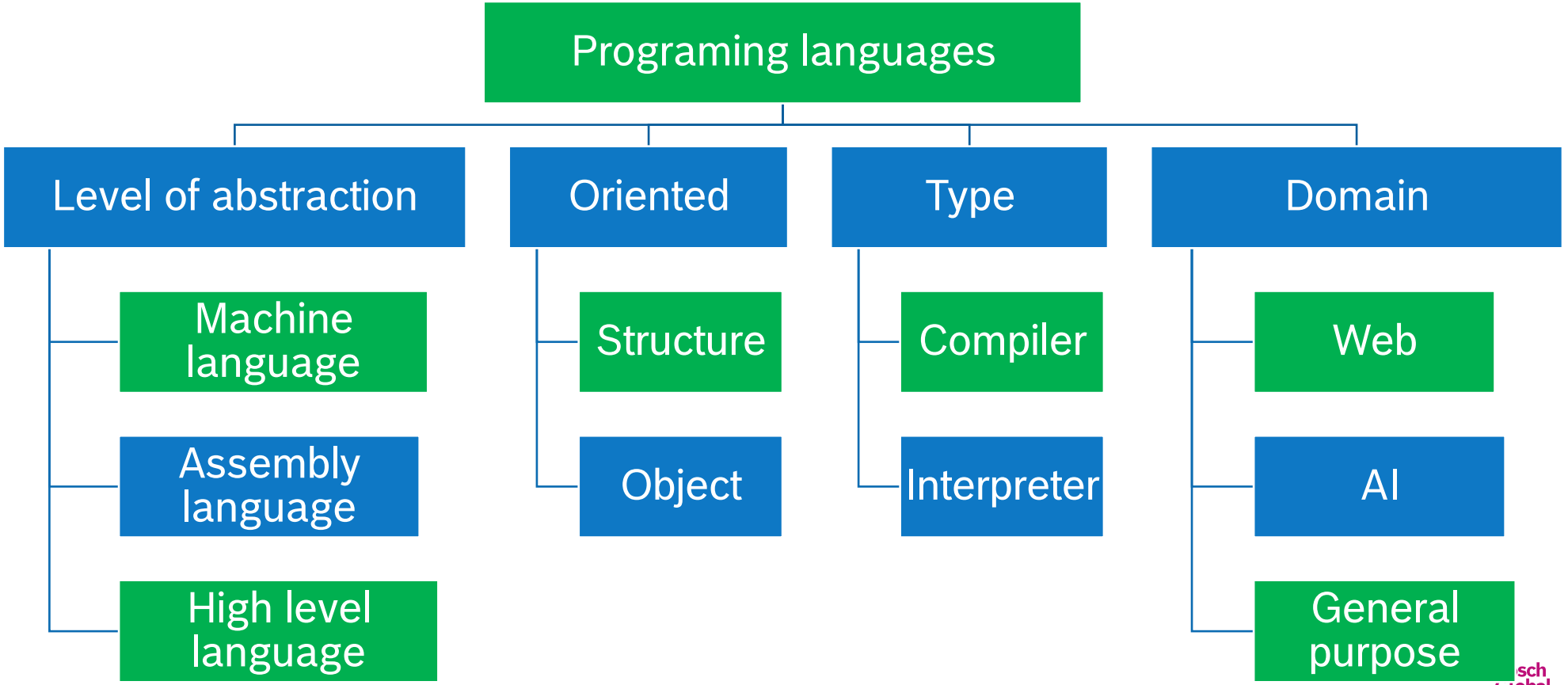
Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process

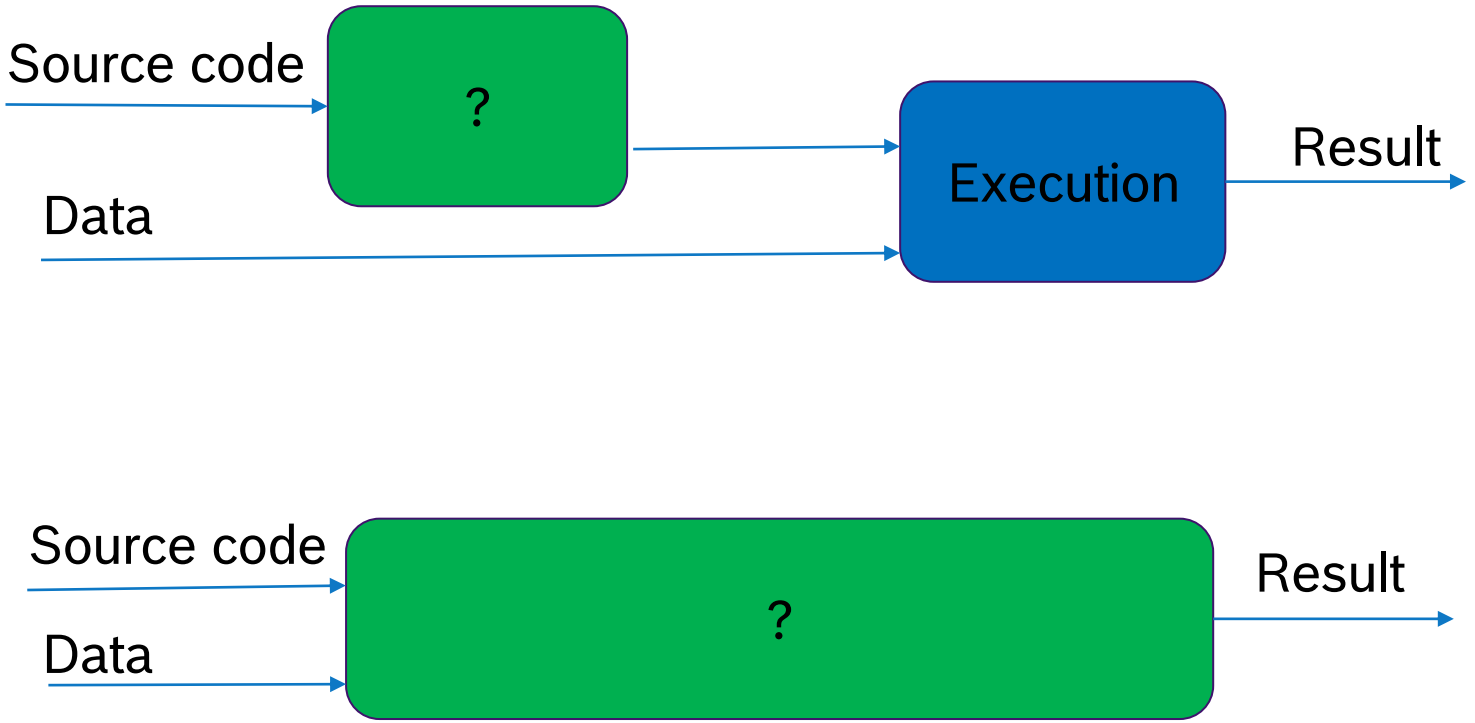
Programing principles
Programing languages



Programing principles
Programing language classification



Programing principles
Compiler vs Interpreter



Programing principles
Programing Domains

- Scientific Applications
  - Fortran, ALGOL60
- Business Applications
  - COBOL
- Artificial Intelligence
  - LIPS, Prolog
- System Programing
  - PL/S, BLISS, Extended ALGOL
- Web Software
  - XHTML, JavaScript, PHP

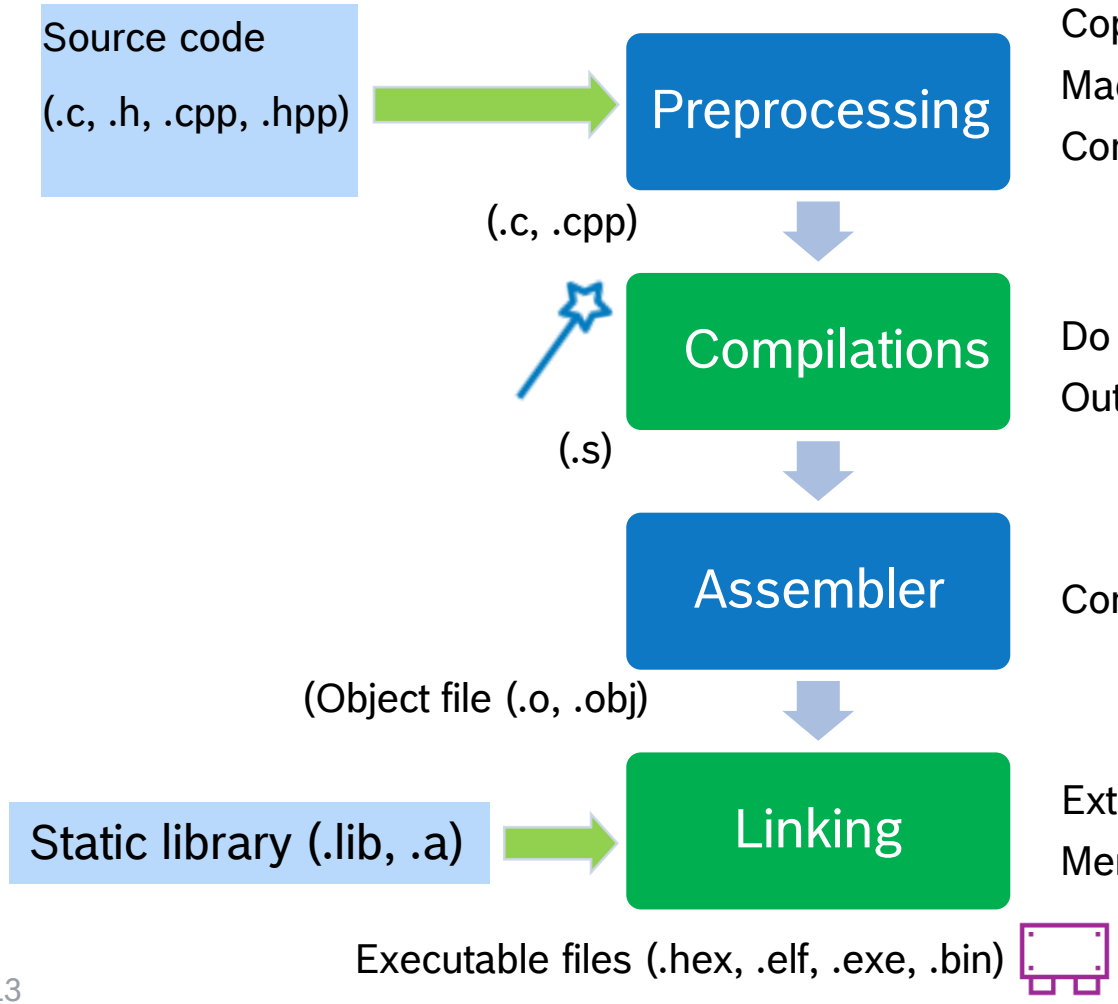
Programing principles
Language Evaluation Criteria and Characteristics

	READABILITY	WRITABILITY	REALIABILITY
Simplicity	*	*	*
Data types	*	*	*
Syntax	*	*	*
Abstraction		*	*
Type checking			*
Exception handling			*

- Cost of learning.
- Cost of writing.
- Cost of compiling, executing, optimization, maintaining, portability…

Programing principles

Compilations steps



Copy all contains of header files (.h, .hpp) into target file (.c, .cpp)

Macros replace (#define)

Conditional compilation (#if, #ifdef, #ifndef, #else, #endif)

Do some magic ☺

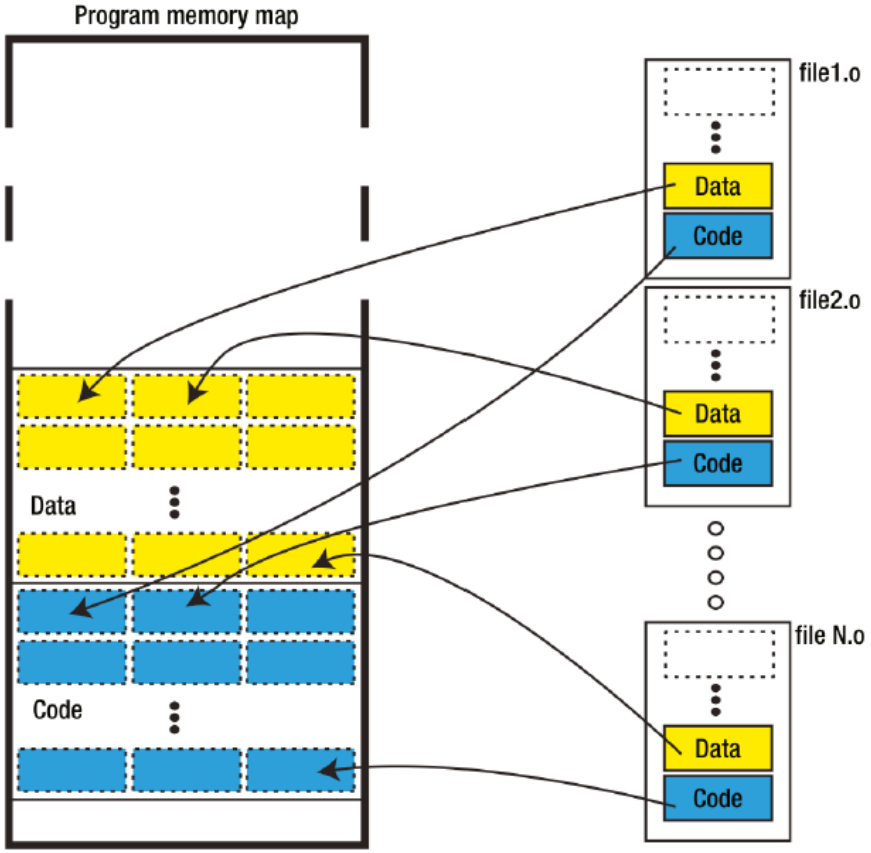
Output as Assembly format (Load r1, X)

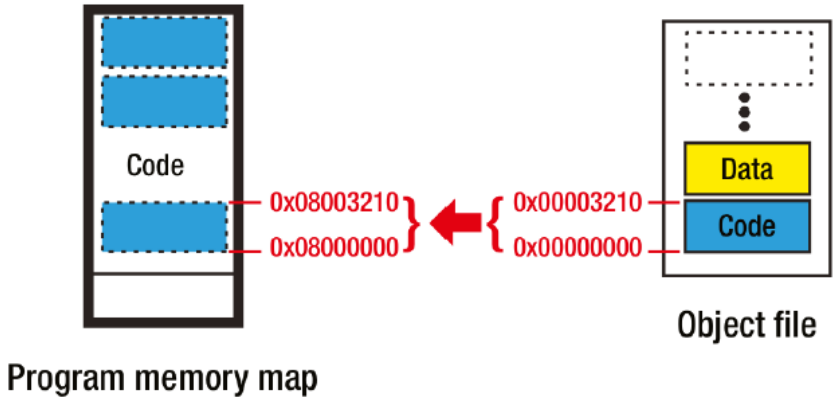
Convert Assembly code to Machine code (0/1)

External symbol linking

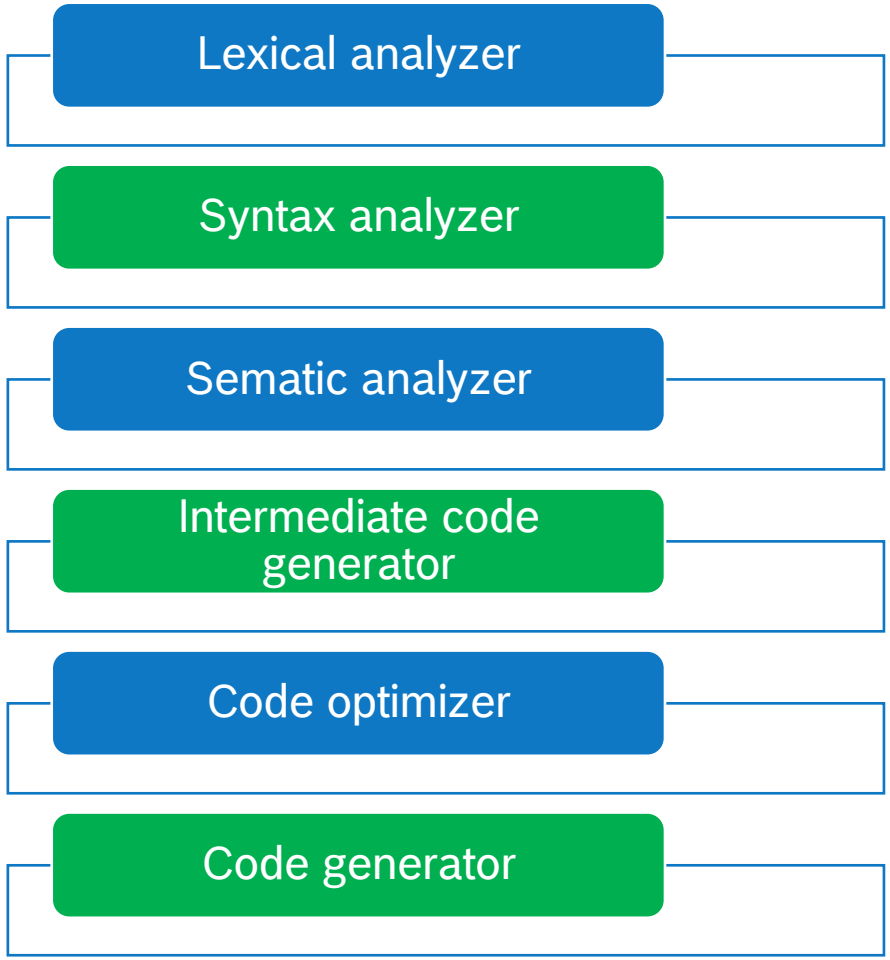
Memory relocation

Programing principles
Linking: memory relocation





Programing principles
Compilation Phases



Programing principles
Q1

```
#define MAX(a,b) (a>b)?a:b
inline int ReturnMaxNumber (int a, int b) {if a > b return a; else return b;}
int ReturnMaxNumber (int a, int b) {if a > b return a; else return b;}
```

Which statement about Macro function and Inline function are Incorrect?

A. Macro function is replaced in Preprocesor phase, while Inline function is replaced in Compiler phase.

B. When running, both Macro function and Inline function will not do context saving and make function call jumping. So they both run faster than normal function.

C. Both may take more memory code size compare to normal function.

D. Both Macro function and Inline function will harder for debugging compare to normal function.

E. An Class member can be access inside both Inline function and Macro function.

Programing principles
Inline function vs Macro

```
inline return_type function_name (parameters)
{
    // inline function code
}
```

```
#define MACRO_NAME Macro_definition
```

Inline function	Macro
Parsed by the compiler	Expanded by the preprocessor
It can be defined inside or outside the class	It is always defined above of the program.
Type safe: debugging is easy for an inline function as error checking is done during compilation	Debugging becomes difficult for macros as error checking does not occur during compilation
No function call, no jumping, replace contains, run faster, more Code size	

Programing principles
Q2

Below statements about Static library and Dynamic library are Correct or Incorrect?

A. (.a, .lib) is static library file; (.so, .dll) is dynamic library file.

B. Static library is created in Compilation phase.

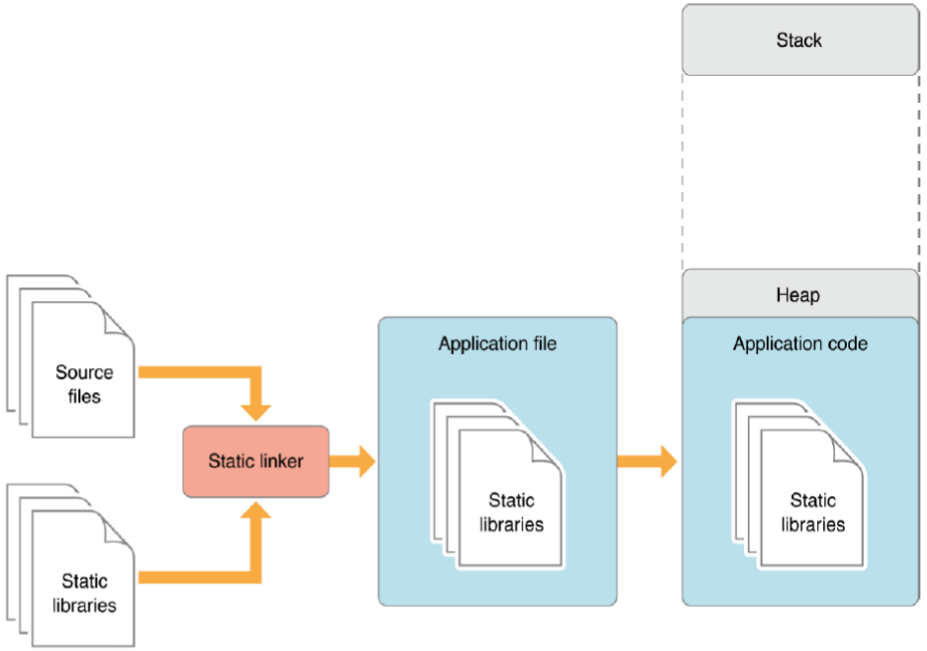
C. Both Static library and dynamic library can be created from multiple source files.

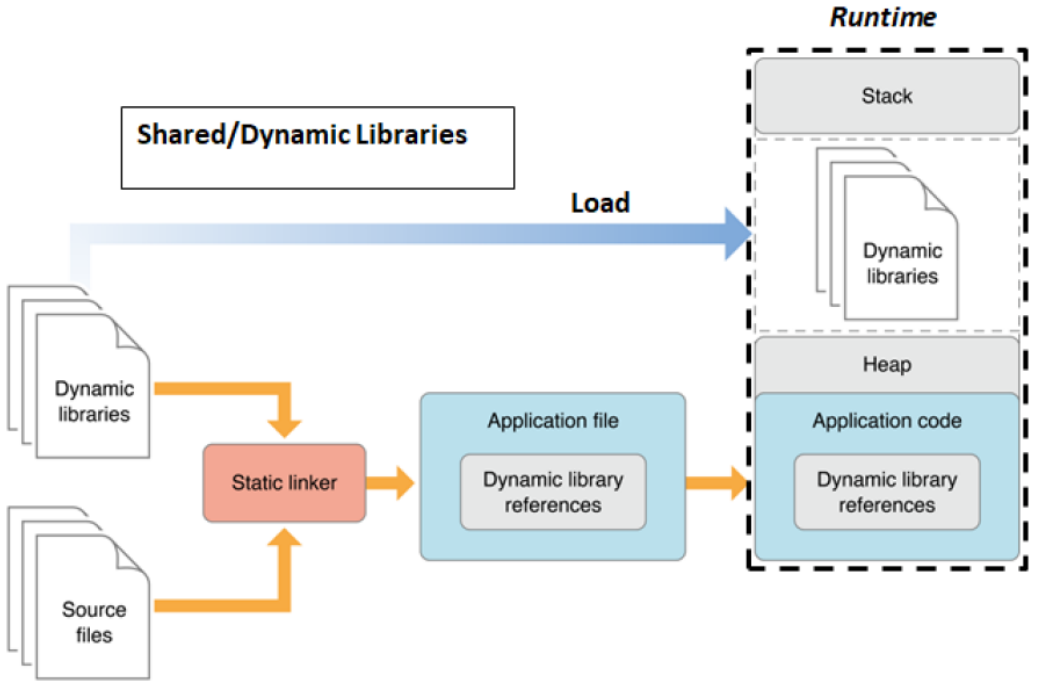
D. Static library is used in Assembler phase.

E. Using Static library can help to reduce SW compilation time.

F. Using Dynamic library can help to reduce SW running time.

Programing principles
Static library and dynamic library





Programing principles

Q3

```
int X = 0;      int Y = 0;
X += X++;
Y = ++Y + Y++;
printf("Value of X: %d\n",X);    printf("Value of Y: %d\n",Y);
```

What are output of above code?

A. X = 1; Y = 3

B. X = 2; Y = 4

C. X = 1; Y = 2

D. Wrong syntax of Y

Programing principles

Q4

```
int X[5] = {0,1,2,3,4};
int Y = 5[X];
printf("Value of Y: %d\n",Y);
```

What are output value of Y?

A. 5

B. 0

C. Wrong syntax. Cannot compiled

D. Unknown value

Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process
2. Embedded system programing
    1. What is embedded system?
    2. Which common elements inside Embedded SW?
    3. Constraints affect design choices?
    4. Why C is most common language for Embedded programing?
    5. What skills required for Embedded programmer?
    6. RTOS & RTOS

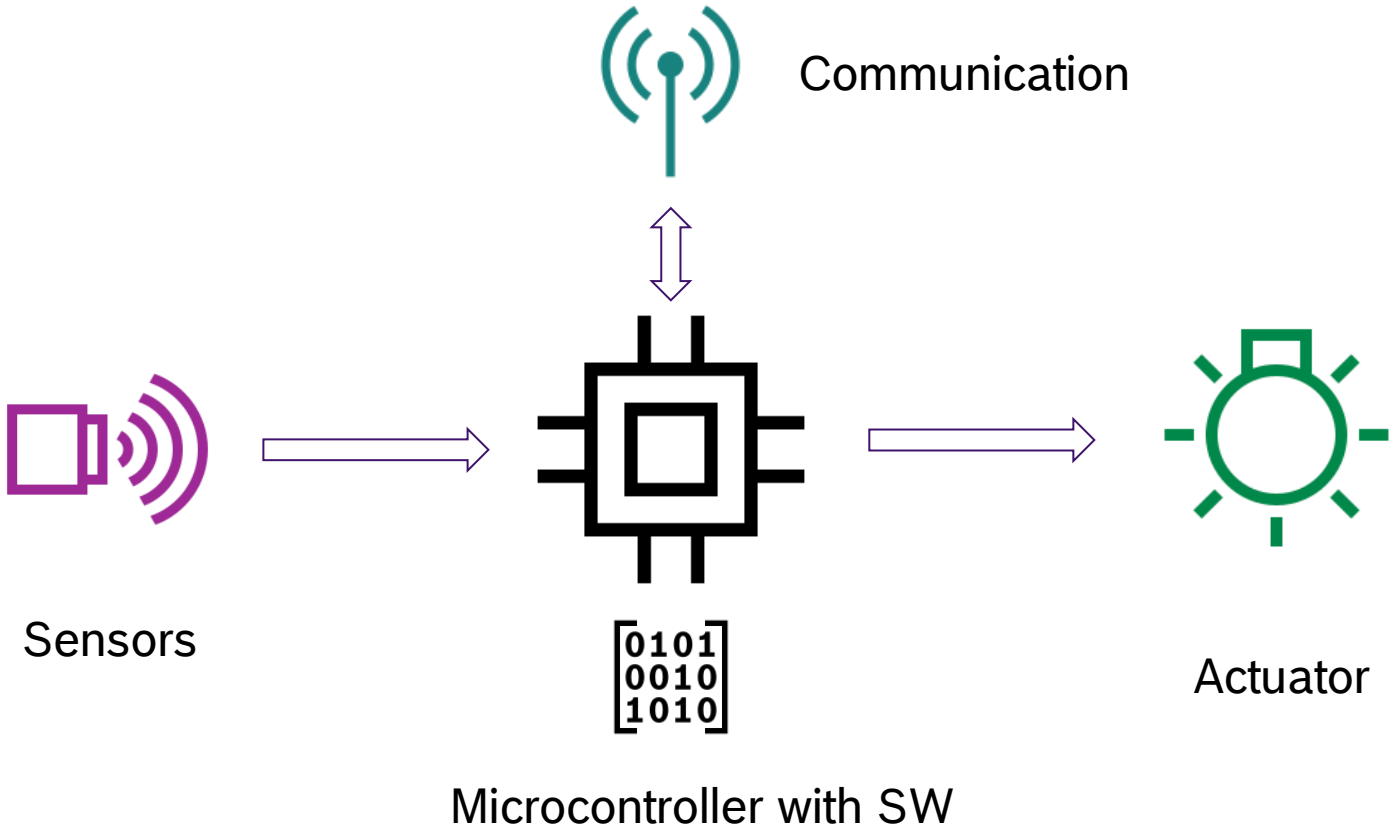
Embedded Programing

Embedded system programing: What is embedded system?

- Example of embedded system?
- Embedded system = Computer Hardware + Software + Additional parts
- Embedded system is combination of Computer HW and SW, and perhaps with additional part (mechanical, electronic) designed to perform a dedicated function.
- Frequently, Embedded system is component within lager systems.
- Automotive embedded systems are connected by Communication networks.

Embedded Programing

Basic elements of traditional Embedded system



Embedded Programing
Embedded system characteristics basic

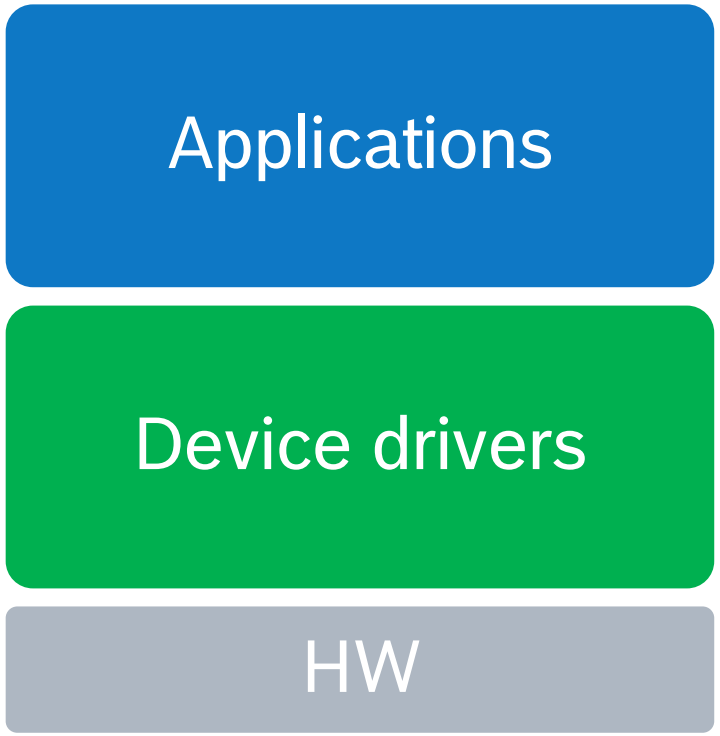
Small Size

Low cost per-unit

Low power consumption

Embedded Programing

Embedded system programing: Which common elements inside Embedded SW?



- Normally, less interact with HW.
- Device drivers developer must have detailed knowledge about using HW of the system

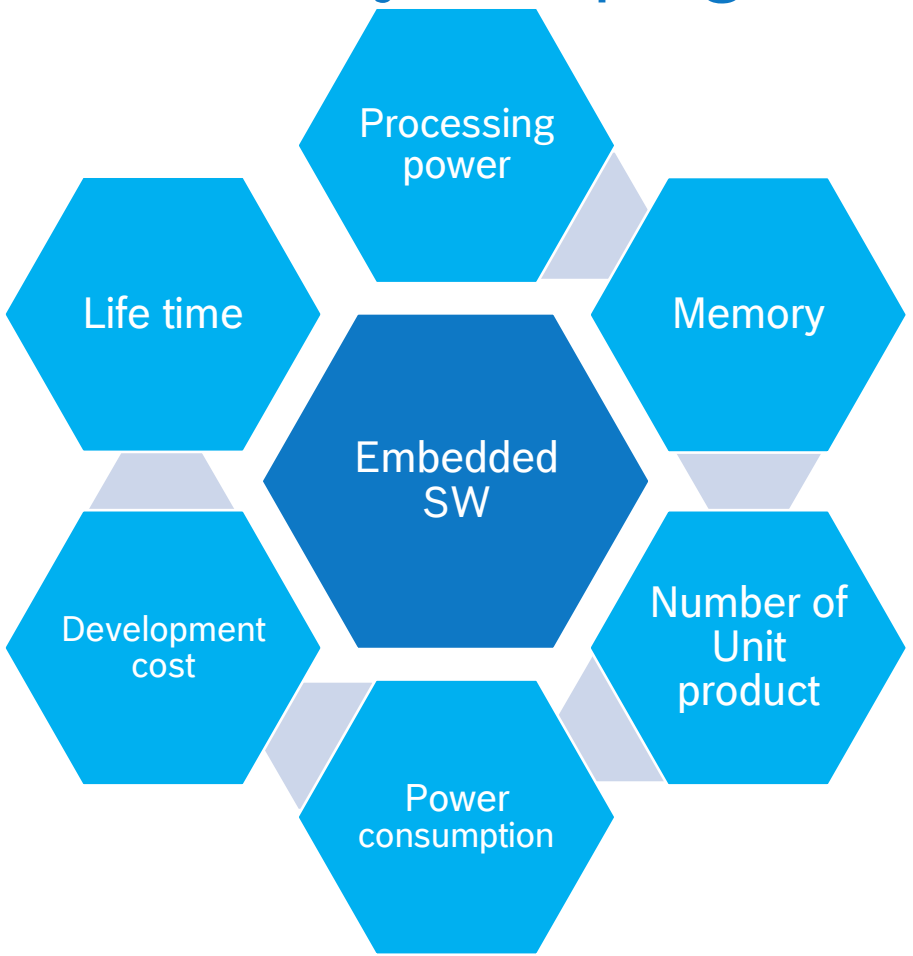
Embedded Programing

What skills required for Embedded programmer?

- HW knowledge: must familiar with microcontroller, circuits, boards, schematic, oscilloscope probe, VOM,...
- Peripheral interfaces knowledge: SPI, I2C, 1-Wire, UART,...
- Efficient code.
- Robust code.
- Minimal resources.
- Reusable code.
- Development tools/Debugging tools using.
- Debug skil

Embedded Programing

Embedded system programing: Constraints affect design choices



- Which constrains are most importance for below example product:
  - Digital Watch
  - Video game player
  - Mars Rover

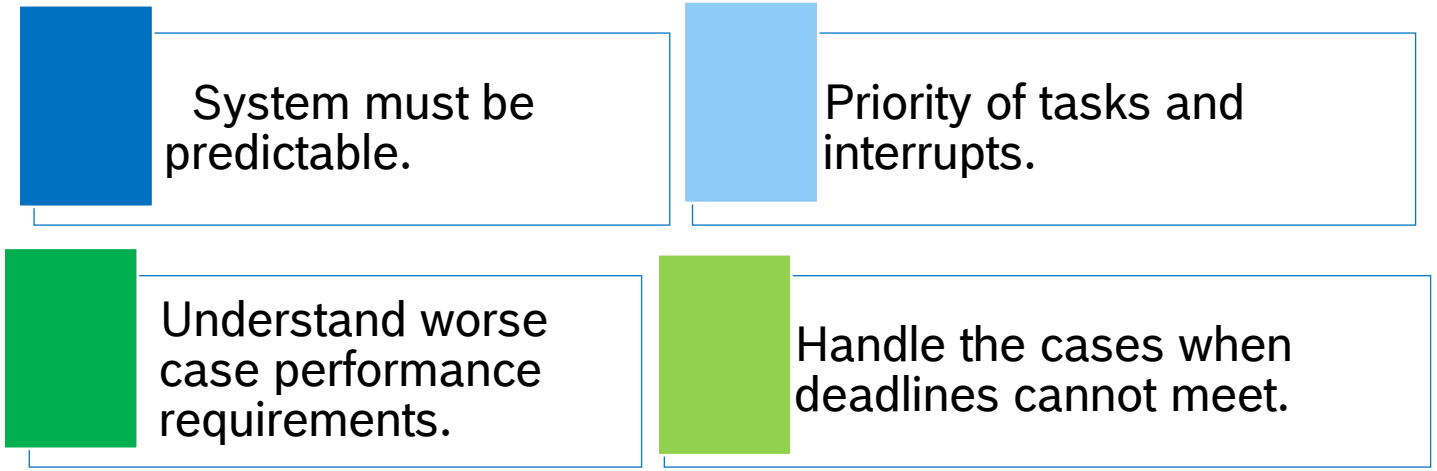
Embedded Programing

Why C is most common language for Embedded programing?

- “Low level” of high level language.
- Close with computer do, interact with HW more easily.
- Many people know and learn.
- Fairly simple to learn.
- Compilers available for most of processors.
- Processer independence.

Embedded Programing
Real Time System (RTS)

- A RTS has timing constraints. The function has deadline for completion.
- The function of a Real time system specified by ability to make calculations/decisions in timely manner.
- A RTS is not simply about the speed. It is about deadline. Guarantee that the deadlines of the system always meet.

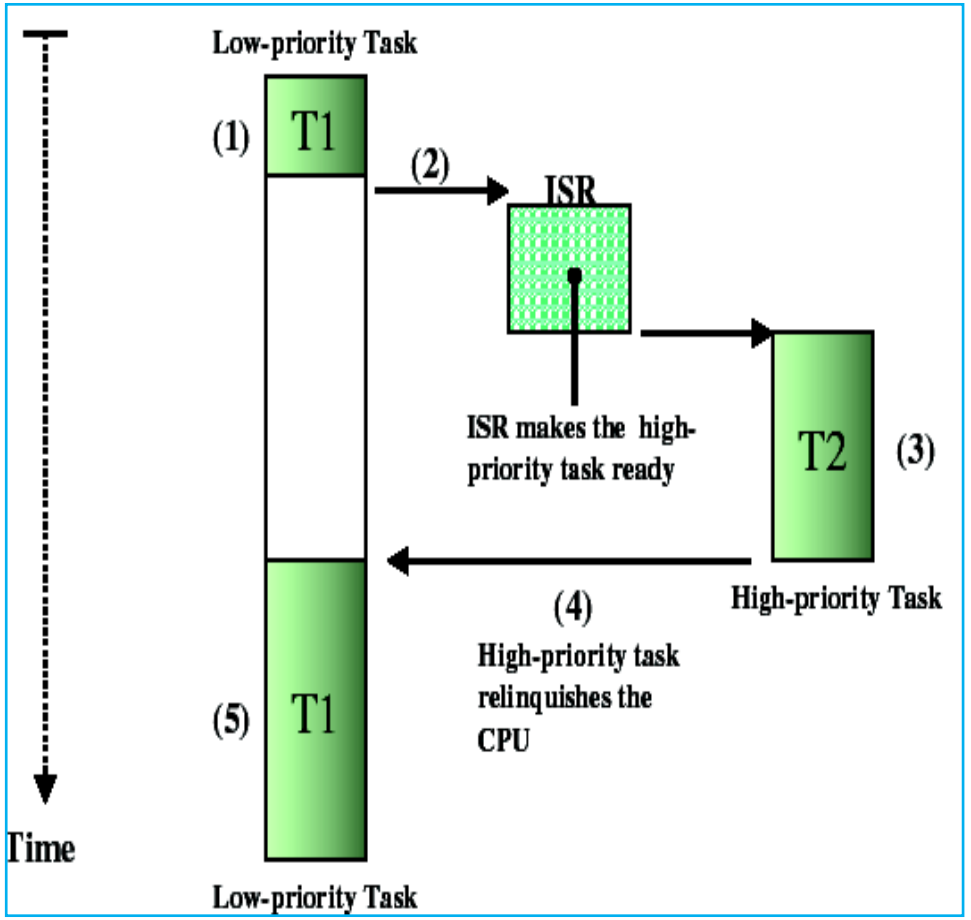


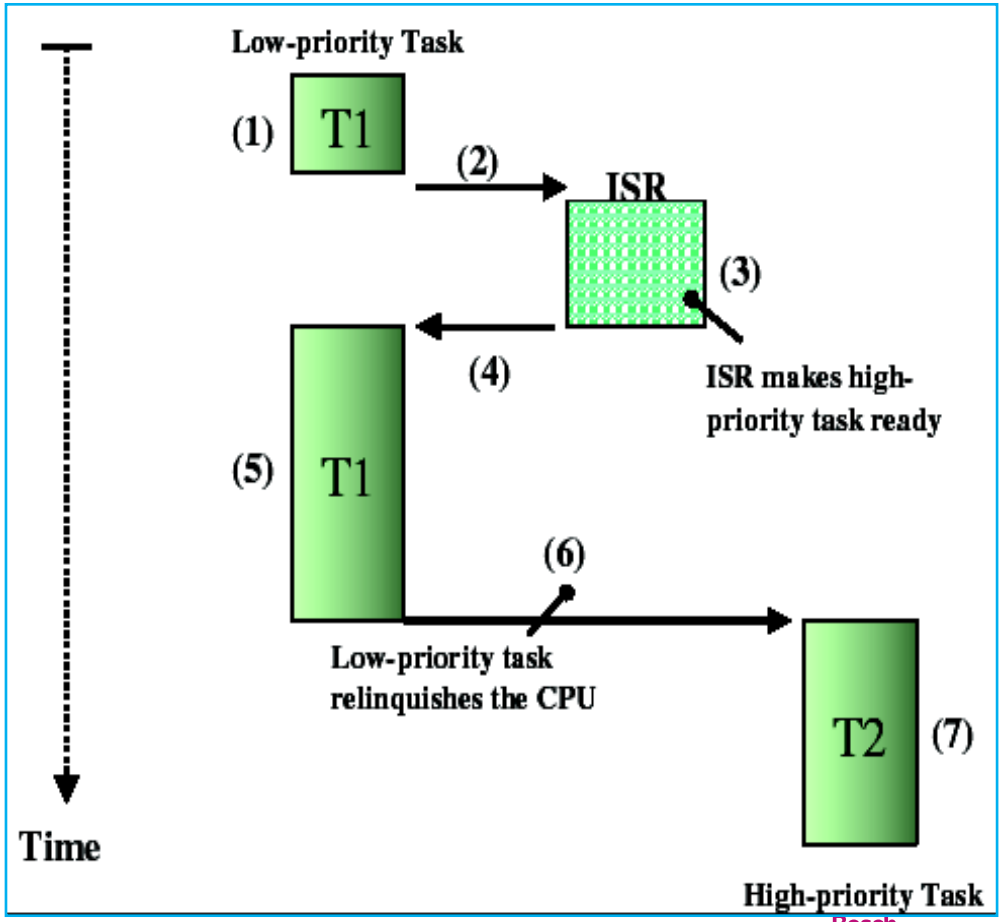
Embedded Programing
Real Time Operation System (RTOS)

- Real time task scheduling
- Resource management

- Task: a group of functions/applications. Common tasks:
  - Initialization task.
  - 1ms task
  - 5ms task
  - 10ms task
  - Background task/Algorithm task
- Scheduling: decide which task should be execute, which task should be suspensed
- Resource management: Mutex, Semaphore

Embedded Programing
Preemptive vs Non-Preemtive





Embedded Programing
Task and Timing

- What is Task Execution Time
- What is Task Deadline
- What is Task Response Time

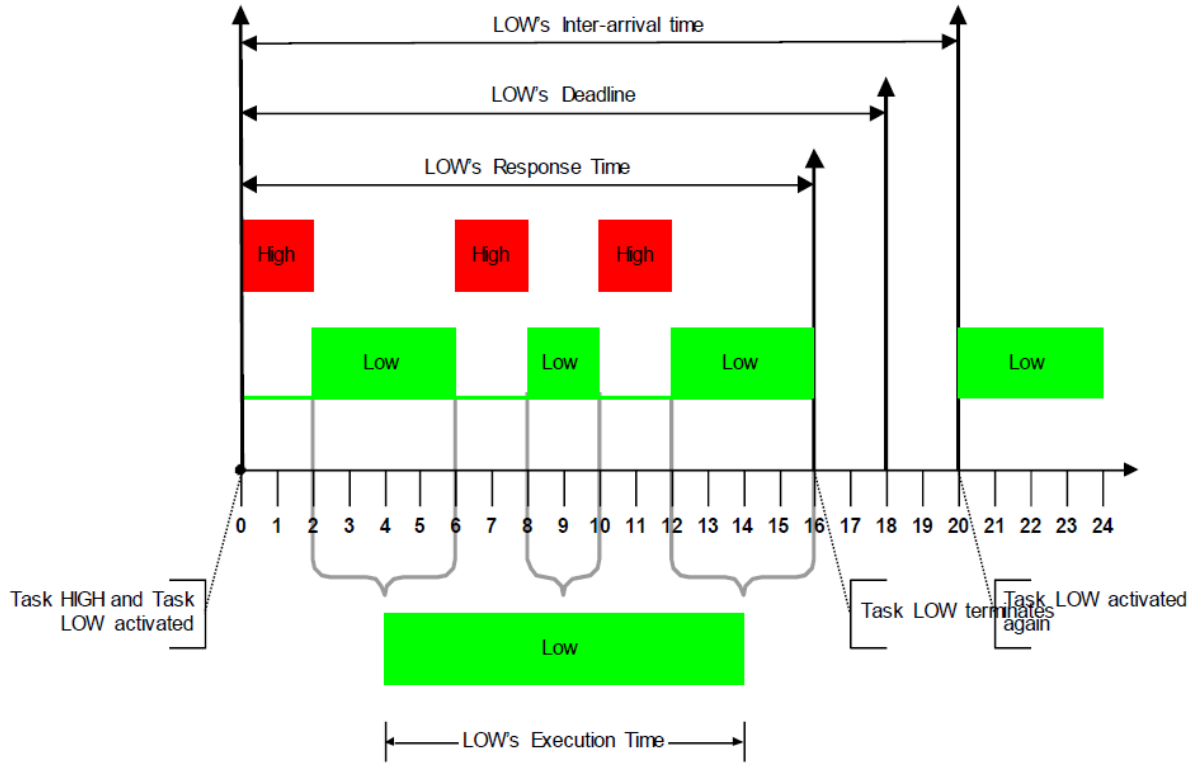
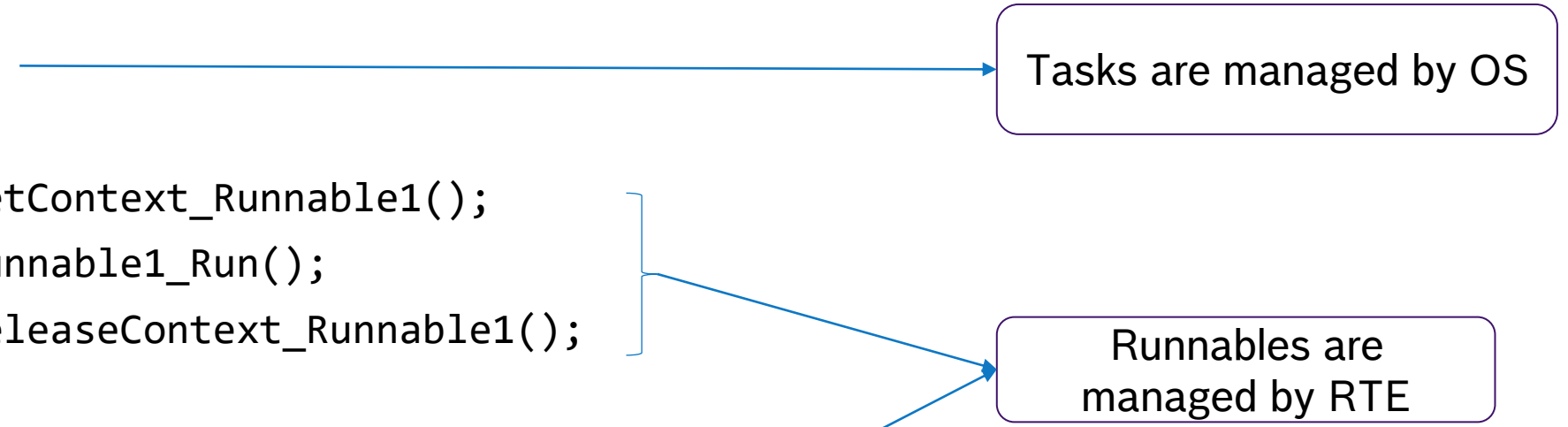


Figure 2.1: Definition of Timing Terminology

Embedded Programing
Task and runnable

```
TASK_10ms
{
        SetContext_Runnable1();
        Runnable1_Run();
        ReleaseContext_Runnable1();

        SetContext_Runnable2();
        Runnable12_Run();
        ReleaseContext_Runnable2();
        …..
}
```



Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process
2. Embedded system programing
    1. What is embedded system?
    2. Which common elements inside Embedded SW?
    3. Constraints affect design choices?
    4. Why C is most common language for Embedded programing?
    5. What skills required for Embedded programmer?
    6. RTOS
3. Some advance programing topic/related topic to embedded programming.

Embedded Programming
Bitwise Operators (1)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Embedded Programming
Bitwise Operators (2)

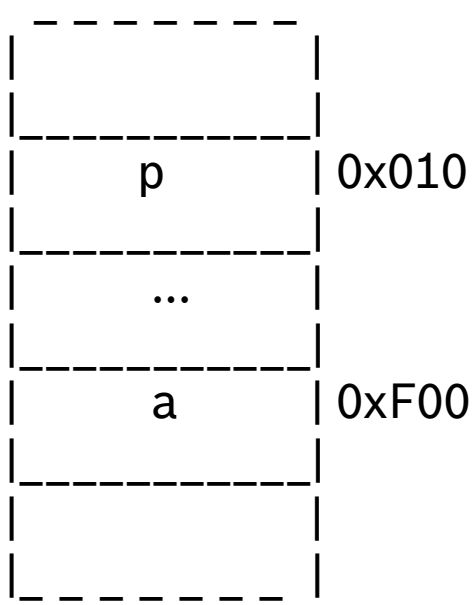
```
unsigned char a = 0xFF;
char b = 0xFF;

printf("%d \r\n", a>>1);
printf("%d \r\n", b>>1);
```

What will be output?

A. 127 ; 127
B. 127 ; -1
C. -1 ; -1
D. 127; 0

Embedded Programing
Pointer remind

```
int a[] = {1, 5, 6, 7};
int* p = a;
```



- Determine type and value of:

1. a = ?
2. &a = ?
3. *a = ?
4. &a + 1 = ?
5. a++ = ?
6. p = ?
7. &p = ?
8. *p = ?
9. p + 1 = ?
10. *p + 1 = ?
11. &p + 1 = ?
12. *(p + 1) = ?
13. p++ = ?

Embedded Programming
const and pointer

```
int* a;                 // Pointer to int
const int* b;           // Pointer to const int
int* const c;           // Const pointer to int
const int* const d; // Const pointer to const int
```

Which statement is showing compiler error ?

```
1. a++;
2. b++;
3. c++;
4. d++;
```

```
5. *a = 1;
6. *b = 1;
7. *c = 1;
8. *d = 1;
```

Why we need to use them?

Embedded Programing
Pointer to function

- Do not know the work of client side, make program more portable.
- Make the program easier to extend.

Syntax:

```
<returnType>* <pointerName> ([type1 param1, type2 param2, …])
```

Example:

```
void (*p)(int, int, int, int, int, int) = nullptr;
p = DrawTriangle;
```

```
int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int main() {
    bool s = true;
    cin >> s;

    int (*p)(int, int) = nullptr;

    if (s == true)
        p = add;
    else
        p = sub;

    cout << p(3, 1);          // Call add or sub?

    return 0;
}
```

Embedded Programing
Switch..case vs multiple if..else

```
Switch(a)
{
Case 1: break;
Case 2: break;
Case 3: break;
}
```

```
If(a == 1) ...
Else if (a == 2) ...
Else if (a == 3)...
```

```
Jump (&a + a)
```

Code optimization hints
Hint 1: Inline function/Macro function

```
sint16 g_mtl_Abs_si16(Sint16 x)(if (x) > 0 return (x); else return (-x);)
```

```
INLINE Sint16 g_mtl_Abs_si16(Sint16 x)(if (x) > 0 return (x); else return (-x);)
#define g_mtl_Abs_mac(x)          (((x) >= (0)) ? (x) : (-(x)))
```

- Use when: function is small but called many places.
- Optimize: Run faster but more code size.

Code optimization hints

Hint 2: Use switch instead of multiple if-else

```
if(x == 1){A();} else if (x == 2) {B();} else if (x == 3) {C();} else {D();}
```

```
switch (x)
{   case 1: A(); break;
    case 2: B();break;
    case 3: C(); break;
    default: D(); break;}
```

- Use when: more than 3 specific integer comparision.
- Optimize: Run faster.

Code optimization hints

Hint 3: Use integer type for loop index/array member access

```
for(unsigned byte i = 0; i >= 100; i++){ArrayBuffer[i] = 0;}
```

```
for(int i = 0; i >= 100; i++){ArrayBuffer[i] = 0;}
```

- Use when: always.
- Optimize: Run faster.

Code optimization hints

Hint 4: Use bit shift instead of division/multiplex

```
unsigned integer a, b;
a = a/2;  b = b*4;
```

```
unsigned integer a, b;
a = (unsign integer) (a>>1);  b = (unsign integer) (b<< 2);
```

- Use when: always
- Optimize: Run faster.

Code optimization hints

Hint 5: Use integer type instead of float/double number

```
float a = 1.9;
if (a > 1.5f) { /* do something */}
```

```
float a = 1.9;
int b =(int)(a*10);
if (b > 15) { /* do something */}
```

- Use when: always
- Optimize: Run faster.

Code optimization hints

Hint 6: Avoid to use multiple/division operator

```
int a = 2; int b = 2;
a = a*2;
```

```
int a = 2;
a = a + a;
```

- Use when: always
- Optimize: Run faster.

Code optimization hints

Hint 7: Use local variable instead of global variable

```
extern int a; A();
void A(void){ if(a > 1) {/* Do something */}}
```

```
extern int a; A(a);
void A(int b){ if(b > 1) {/* Do something */}}
```

- Use when:
- Optimize: Run faster. Take more RAM.

Code optimization hints

Hint 8: Use if branch for higher probability

```
if (a == NULL_PTR){
        /* Null pointer. Do nothing */
} else { A();}
```

```
if (a != NULL_PTR){
        A();
}
```

- Use when: always for most of Microcontroller architechture.
- Optimize: Run faster. Take more RAM.

Code optimization hints

Hint 9: Function is called only as often as needed

```
Com_ReceiveSignal(1, &l_SignalData_ui8);
if (l_SignalData_ui8 == 1) { A(); }
```

```
if(NewMsgReceived_b == TRUE){
    Com_ReceiveSignal(1, &l_SignalData_ui8);
    if (l_SignalData_ui8 == 1) { A(); }}
```

- Use when: always.
- Optimize: Run faster.

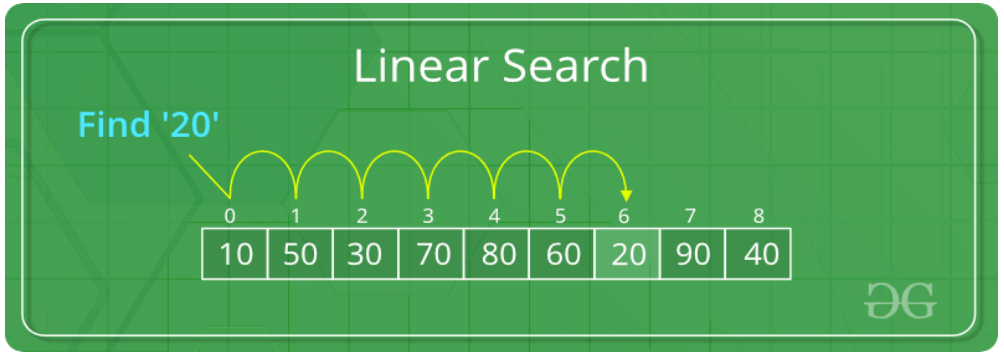
Code optimization hints
Hint 10: Reduce number of loop

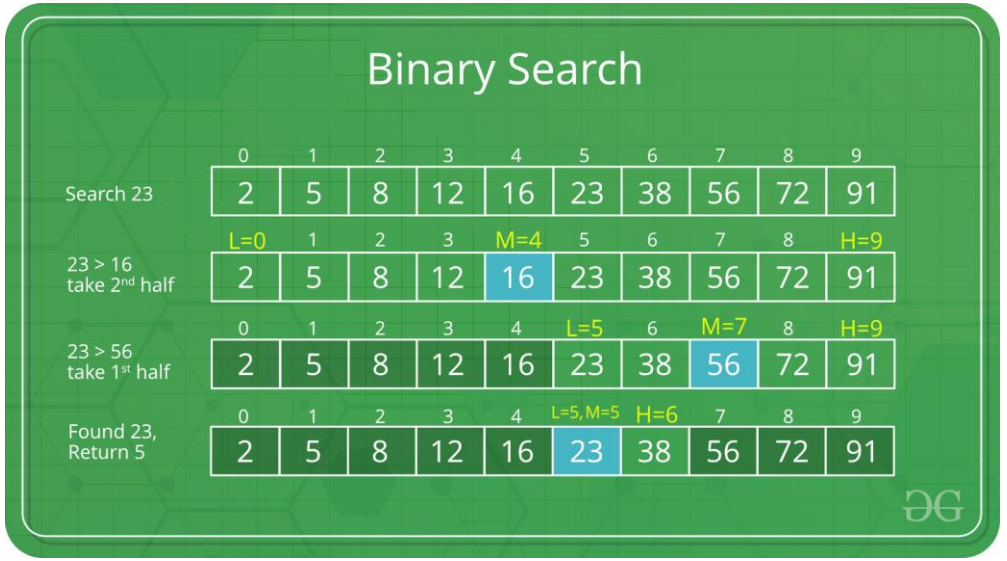
```
for (int i = 0; i < 1000; ++i){
    if(ArrayA[i] > 0) {Flag = TRUE;}
}
```

```
for (int i = 0; i < 1000; ++i){
    if(ArrayA[i] > 0) {Flag = TRUE; break;}
}
```

- Use when: always. Depend on Coding rule.
- Optimize: Run faster.

Code optimization hints
Hint 11: Use better/smarter algorithm! (1)





- Use when: always.
- Optimize: Run faster.

Secure programming

Overflow

Compiler optimization

Undefined behaviour

Use of uninitialized variable

Unspecific behaviour

Divide by zero

Type casting

Use object after destroyed

Race condition

Null pointer

Infinite loop

Secure programming
Undefined behaviour (1)

Example 1

```
int * varA = NULL;          //line 1
*varA = 0;                  //line 2

int varB;                   //line 3
varA = &varB;               //line 4

printf("%i\n", *varA) ; //line5
printf("%i\n", varB) ;  //line 6
```

Example 2

```
#include <limits.h>
int Example(void) {
   int b = INT_MAX-1;
byte c;
a = (byte)b;
   if(b + 100 < b)
   { return 1; }
   return 2;
}
```

Example 3

```
int Add (int a, int b)
{
        return (a + b);
}
```

Bosch Global Software Technologies
alt_future

Secure programming
Undefined behaviour (2)

```c
#define MUL(a, b) a*b
int main()
{
  // The macro is expended as 2 + 3 * 3 + 5, not as 5*8
  printf("%d", MUL(2+3, 3+5));
 return 0;
}
// Output: 16`
```

```c
int main(void)
{
  int a = 0;
  int b = 0;
  return &a < &b; /* undefined behavior */
}
```

```c
int g = 0;
int main (void){
Int a = funcA() ;
Int b = funcB();
 if(a > b)
   /*.....*/
}
int funcA (void){ g++;
    return (g);}
int funcB (void) ){ g--;
    return (g);}
```

Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process
2. Embedded system programing
    1. What is embedded system?
    2. Which common elements inside Embedded SW?
    3. Constraints affect design choices?
    4. Why C is most common language for Embedded programing?
    5. What skills required for Embedded programmer?
    6. RTOS
3. Some advance programing topic
4. OOP principles basic

OOP principles basic
Agenda

1. Introduction
2. Class and Object
3. Inheritance
4. Function signature, overloading, overriding
5. Constructor, Destructor
6. Copy constructor
7. Operator Overloading
8. Virtual Function, Template Function

1. Introduction

Additional Object-Oriented features to C

REMEMBER

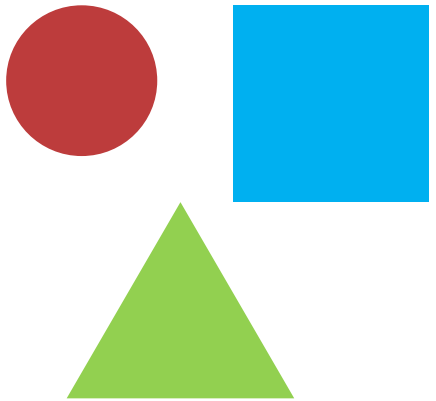
C++ is C with classes adding object-oriented features, such as classes, and other enhancements to the C programming language

4 main concepts of C++:

Encapsulation
Inheritance
Polymorphism
Abstract

2. Class and Object

Circle, Square and Triangle: they are all shapes. We can put them in the same class: Shape

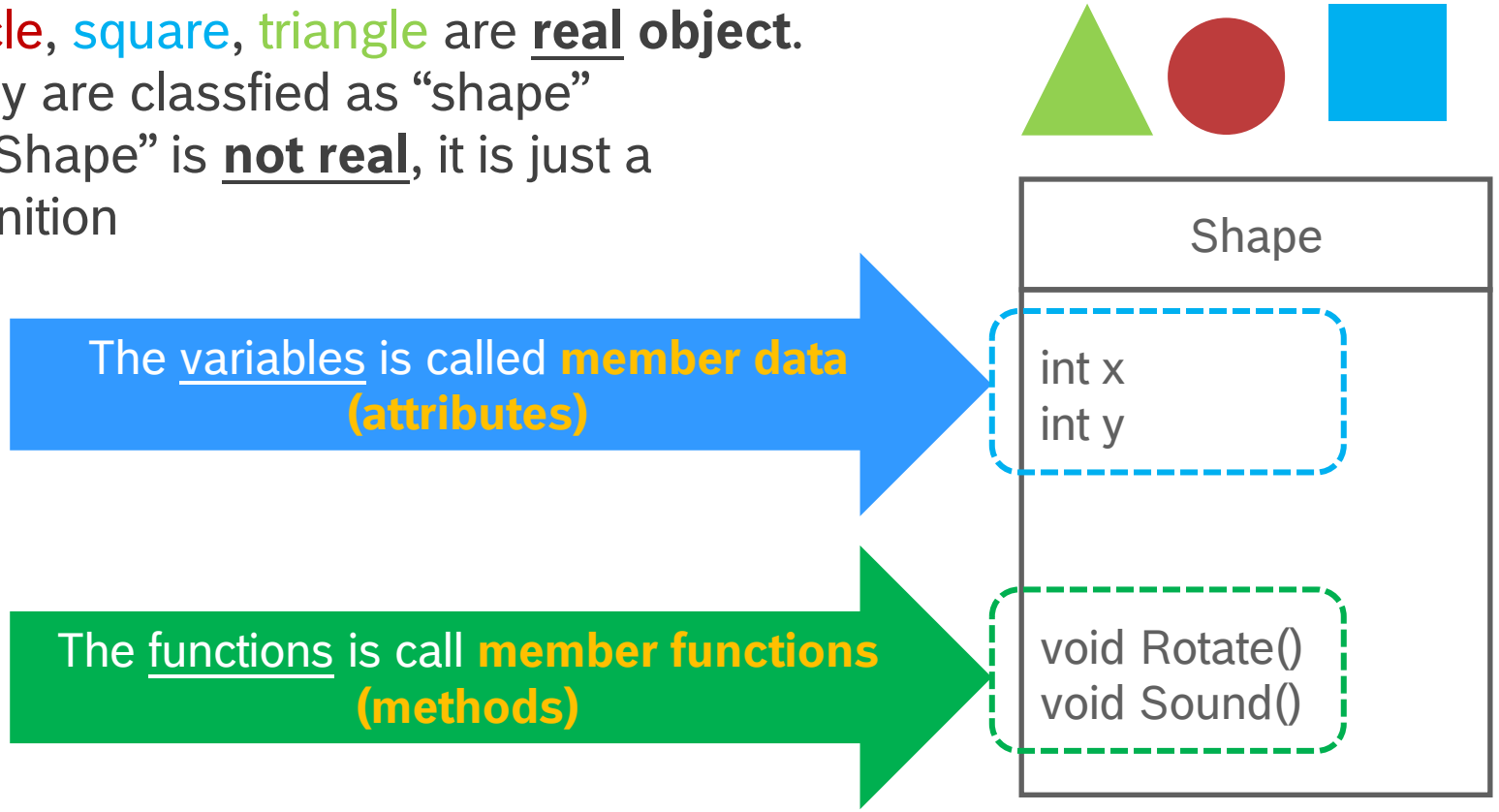


REMEMBER

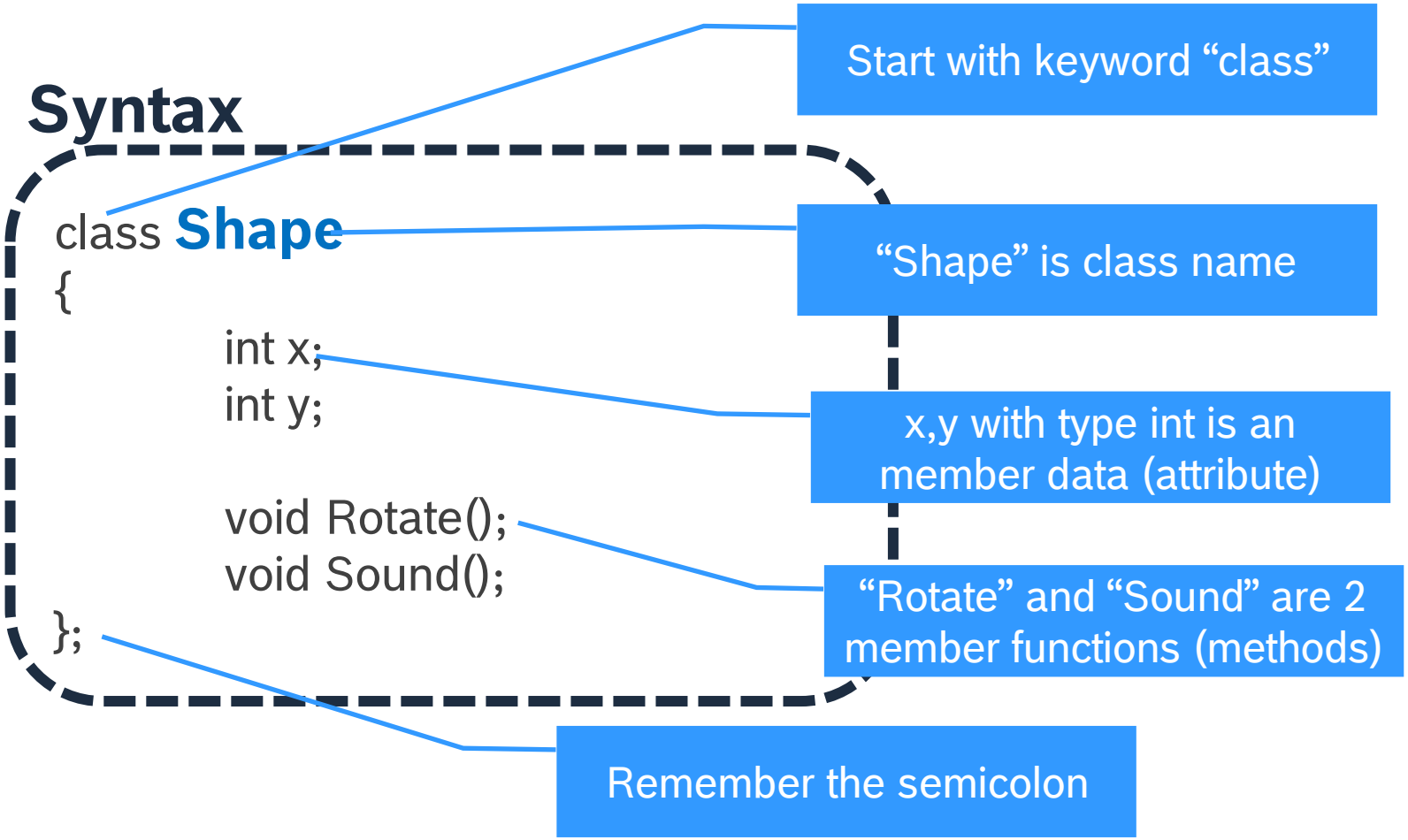
A class is just a collection of variables— often of different types— combined with a set of related functions

2. Class and Object

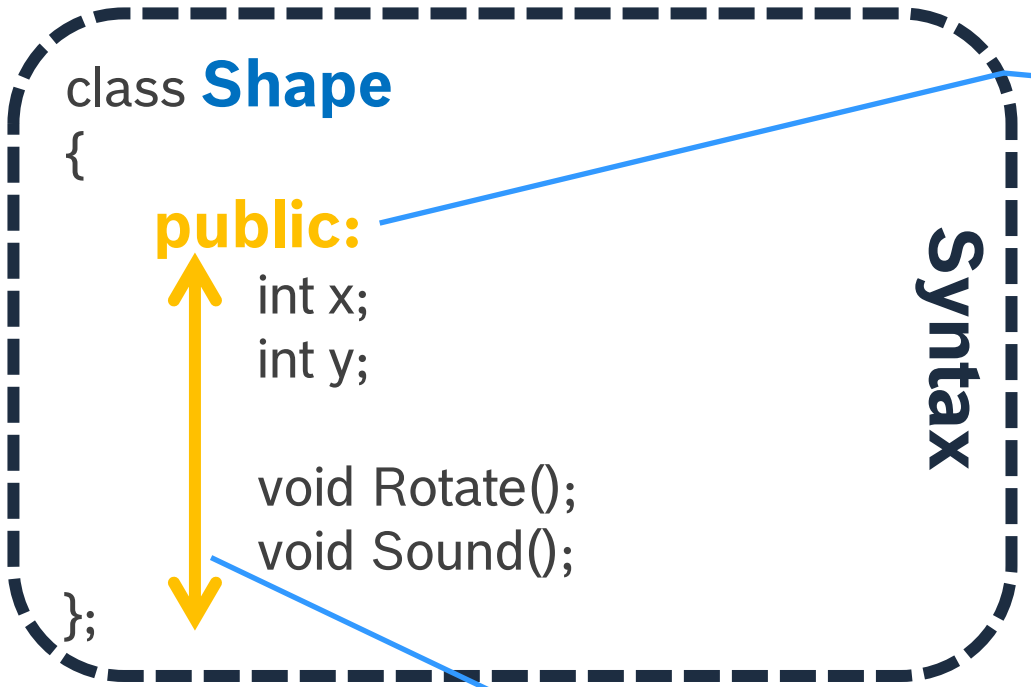
Circle, square, triangle are **real object**.
They are classfied as “shape”
-> “Shape” is **not real**, it is just a definition



2. Class and Object



2. Class and Object

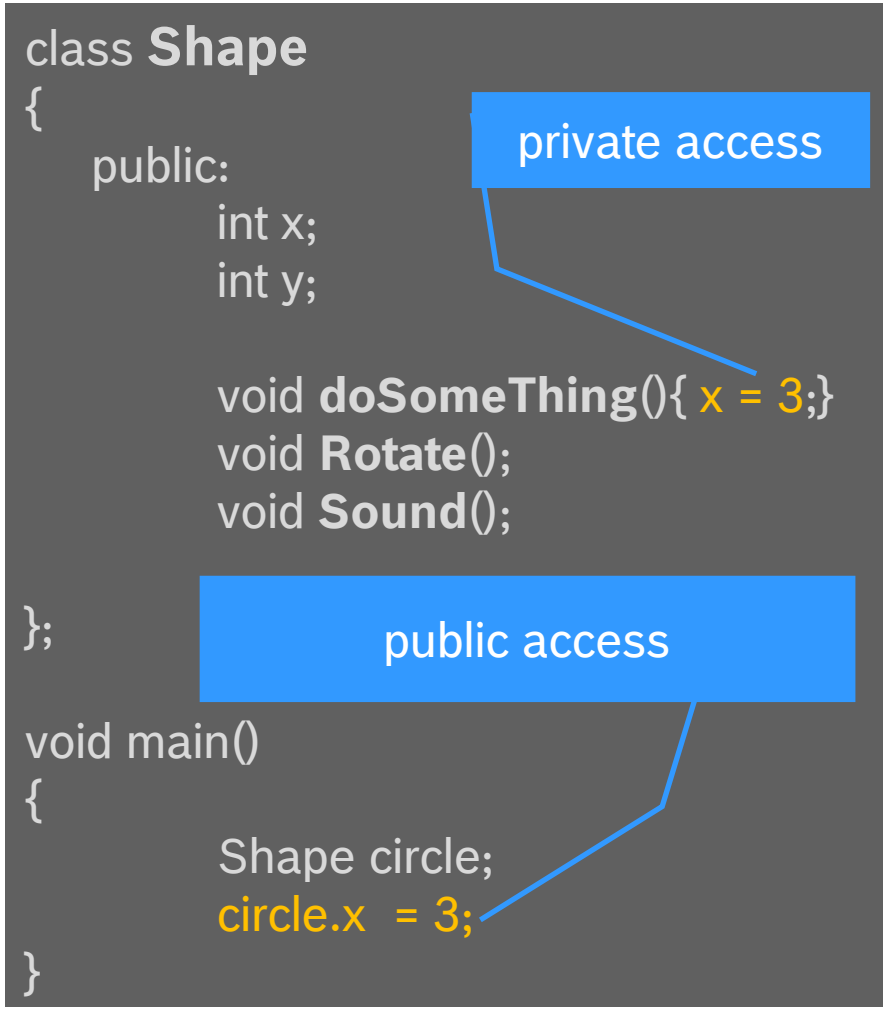


"public" keyword is an access specifier. There are 3 type of access specifiers:

public
protected
private

Access specifier affects the attribute and methods below it until next access specifier.

2. Class and Object



REMEMBER

Public - The members declared as Public are accessible from outside the Class through an object of the class.
Protected - The members declared as Protected are accessible from outside the class BUT only in a class derived* from it.
Private - These members are only accessible from within the class. No outside Access is allowed.

2. Class and Object

01_classandobject.cpp

```
class myClass{
    private: int        privateMember;
    public: int         publicMember;

    public: void setPrivateMember(int x){privateMember = x;};
    public: int getPrivateMember(){return privateMember;};
};

void main()
{
     myClass  A;
     A.publicMember = 5; // Perfectly legal
     A.privateMember = 7; // Syntax error, this will not compile
     A.setPrivateMember(7); // Legal
     cout << A.getPrivateMember(); // Legal
     return 0;   }
```

privateMember have private access specifier: not allow public access

2. Class and Object

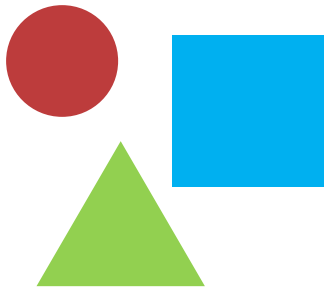
REMEMBER

Encapsulation is used to **hide** the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them

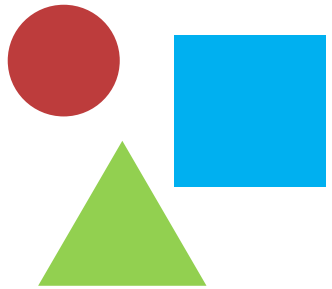
Keyword: **public**, **protected** and **private** adds encapsulation feature in C++

3. Inheritance

Coming back to the story, C++ provides better program design:



3. Inheritance



REMEMBER

Circle inherits Shape. Shape is called **base class**. Circle is called **derived class**

Circle, Square, Triange inherits attributes and methods form Shape

3. Inheritance

Syntax

```
class Shape
{
    public:
        int x;
        int y;
        void Rotate();
        void Sound();
};

class Circle : public Shape
{
};
```

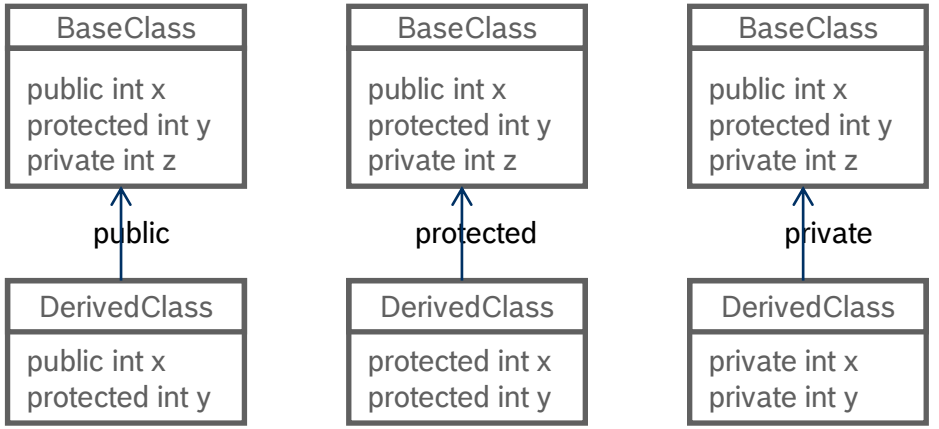
REMEMBER

There are 2 access specifier: one is for data member; one is for inheritance.

- Name of derived class
- Name of base class
- Type/mode of inheritance: public, private, protected

3. Inheritance

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)



Practice

REMEMBER

The access scope for data members/function of derived classed is specify by mode of inheritance.

3. Inheritance

REMEMBER

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

C++ supports multiple inheritance:

```
class Rectangle: public Shape, public PaintCost
```

4. Function signature, overloading, overriding

02_inheritance_hiding_baseclass.cpp

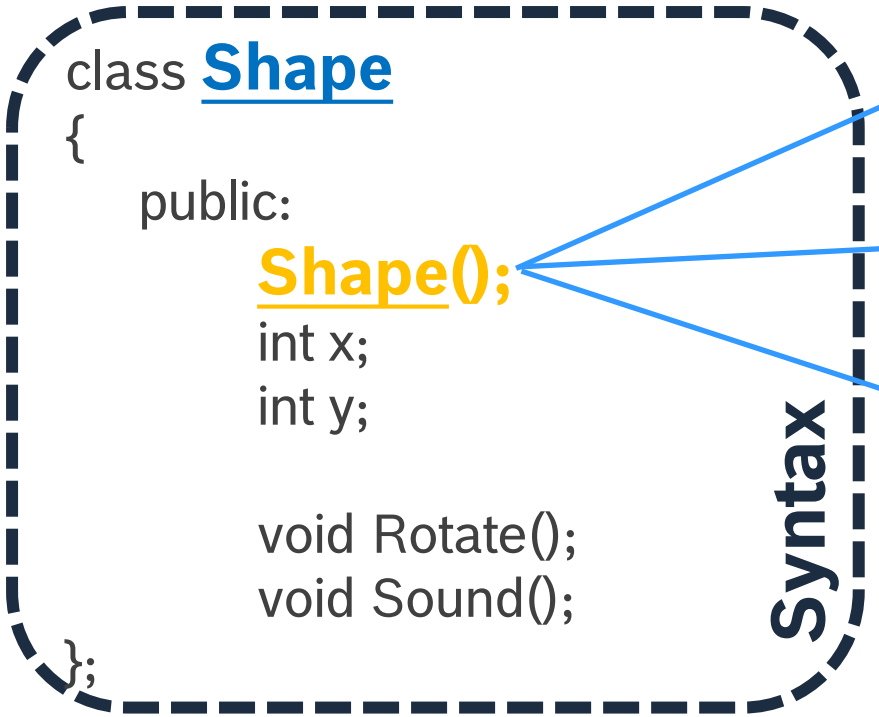
REMEMBER

1. The **signature** of a function consists the following information:
The **name of the function**
The class or namespace scope of that name
The **const** qualification of the function
The types of the function parameters

2. A function **overloads** other function when they are:
- In the same class
- Same function name
- **Different** in signature

3. A function **overrides** other function when they are:
- One in the base class and the other in derrived class
- Same in function name
- **Same** in signature

5. Constructor



Constructor have the same name with class name

Shape() is a constructor

Constructor should be in public access specifier

REMEMBER

Constructor is just a **member function**. Its difference from other member function is that it is called automatically when the object is created.

? If we do not have constructor, what will happened?

? What is the purpose of constructor?

5. Constructor

REMEMBER

The sequence when an object of derived class is created:

1. Memory for "**Derived**" is allocated.
2. The "**Derived**" constructor is called
3. The compiler looks to see if we've asked for a particular **Base** class constructor. If yes, call the base class constructor
4. The **base** class constructor initialization list
5. The **base** class constructor body executes
6. The **base** class constructor returns
7. The **derived** class constructor initialization list
8. The **derived** class constructor body executes
9. The **derived** class constructor returns

5. Destructor

02_inheritance_override.cpp

02_inheritance.cpp

```
class Shape
{
    public:
        ~Shape();
        int x;
        int y;

        void Rotate();
        void Sound();
};
```

Syntax

Destructor have “~” and the same name with class name

~Shape() is a destructor, should be public.

REMEMBER

Destructor is just a **member function**. Its difference from other member function is that it is is called automatically when the object is destroyed (out of scope/delete).

? If we do not have destructor, what will happened?

? What is the purpose of destructor?

6. Copy constructor

```
class Shape
{
    public:
        Shape(const Shape & obj);
        int x;
        int y;

        void Rotate();
        void Sound();
};
```

Syntax

Copy constructor syntax: method has same name of class. Parameter is same type of class with constant reference

REMEMBER

Copy constructor is called when copying an object content to another object content (during object initialization), like in this example:

```
Shape myShape(2);
Shape yourShape(myShape);
Shape herShape= myShape;
func1(myShape);
```

? If we do not have copy constructor, what will happened?

? What is the purpose of copy constructor?

7. Overloading operator

```
class CDate
{
   public:
            int m_nDay;
            int m_nMonth;
            int m_nYear;
            ...
}
```

Read the code

?

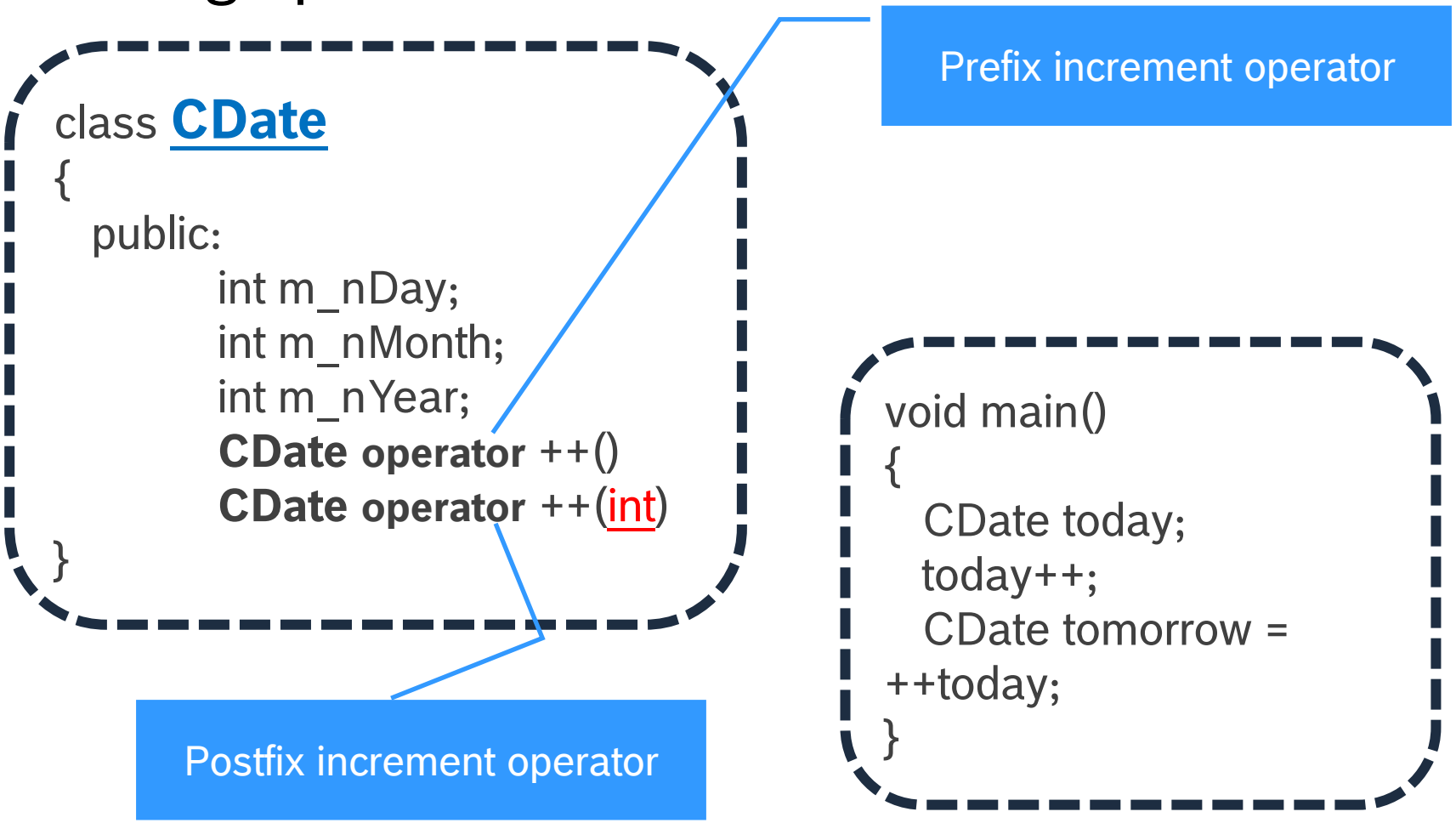
Does the object of CDate class support:

CDate today;
CDate tomorrow = today++;

REMEMBER

To use the operator on object , we need to overload its operator.

7. Overloading operator



7. Overloading operator

04_Operator1.cpp

```
class CDate
{
   public:
            int m_nDay;
            int m_nMonth;
            int m_nYear;
            CDate operator ++()
            CDate operator ++(int)
}
```

Prefix increment operator

Postfix increment operator

```
void main()
{
   CDate today;
   today++;
   CDate tomorrow =
++today;
}
```

7. Overloading operator

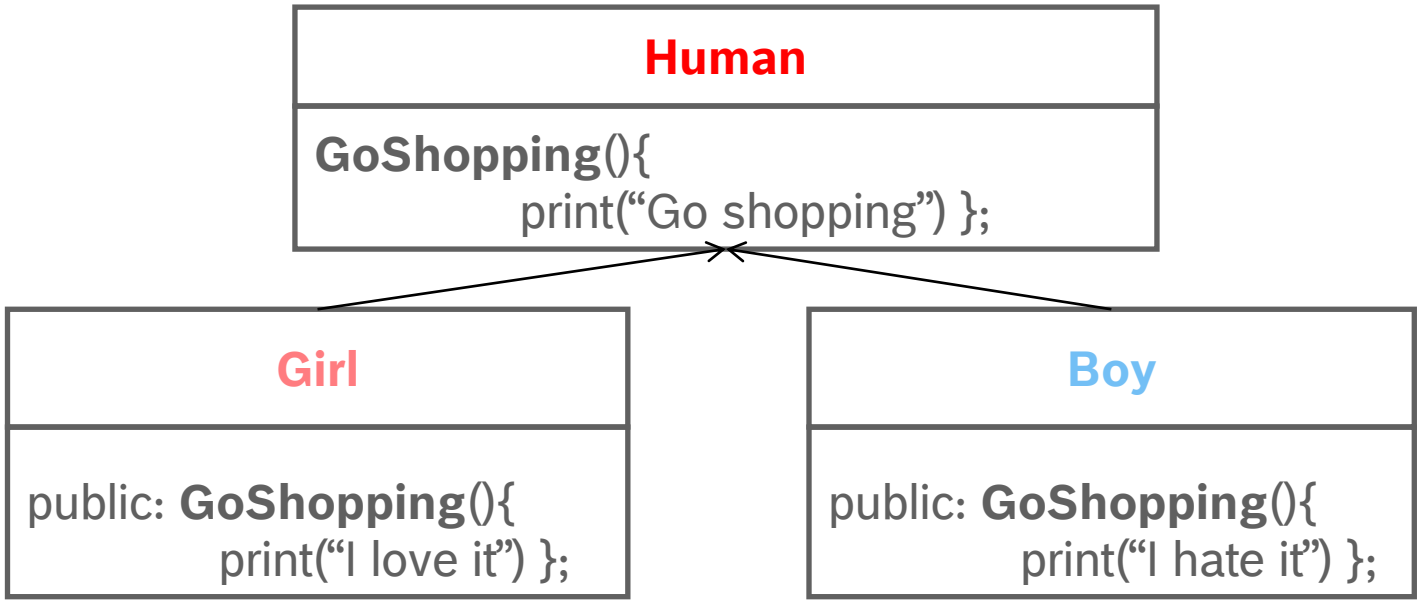
REMEMBER

Operator can be overloaded	Name
++	Increment
--	Decrement
*	Pointer dereference
->	Member selection
!	Logical NOT
&	Address-of
~	One's complement
+ or -	Unary plus/negation
Conversion, binary, comparison, subscript, function operators	Conversion, binary, comparison, subscript, function operators

REMEMBER

Operator cannot be overloaded	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional ternary operator
sizeof	Gets the size of an object/class type

8. Virtual function, polymorphism

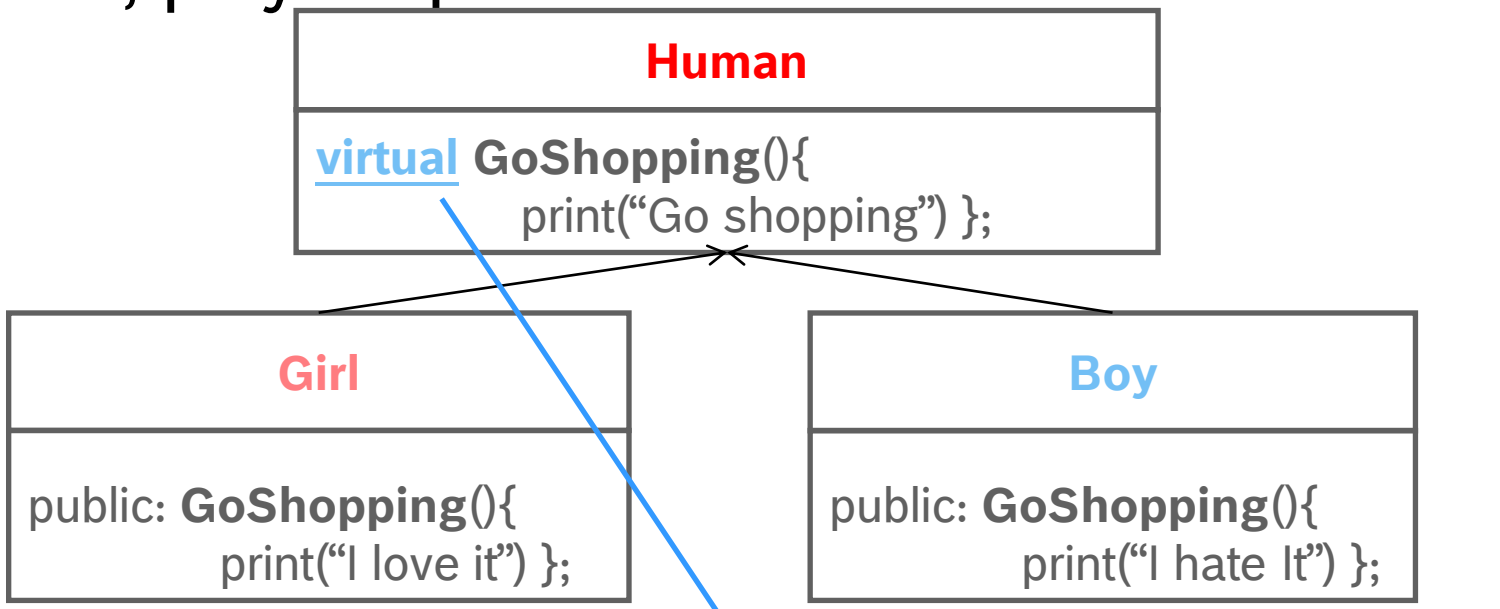



```
main(){
    Human h;
    Girl alice;
    Boy tom;
}
```

?

```
//What in the output
h.GoShopping();
alice.GoShopping();
tom.GoShopping();
```

8. Virtual function, polymorphism

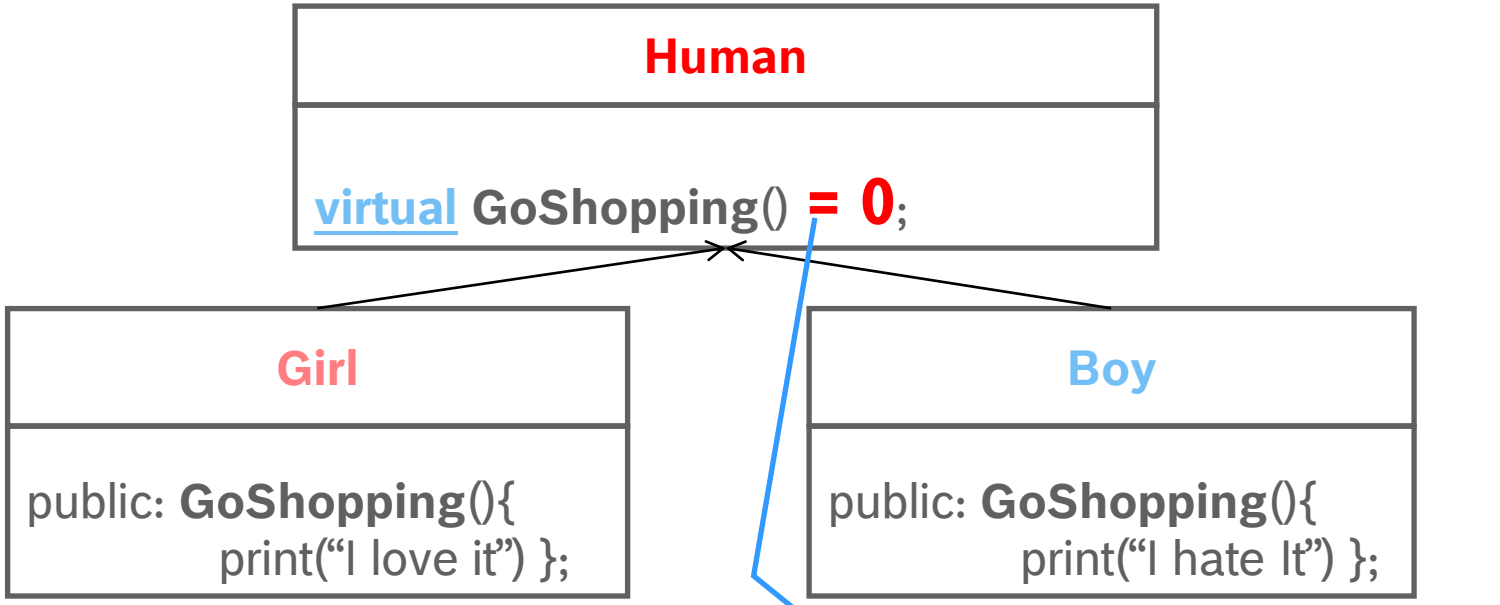


REMEMBER

With polymorphism, you do not need to know the **alice** and **tom** are **Boy** or **Girl**. You just need to know that they are **Human**.

Virtual keyword adds polymorphism

8. Virtual function, polymorphism

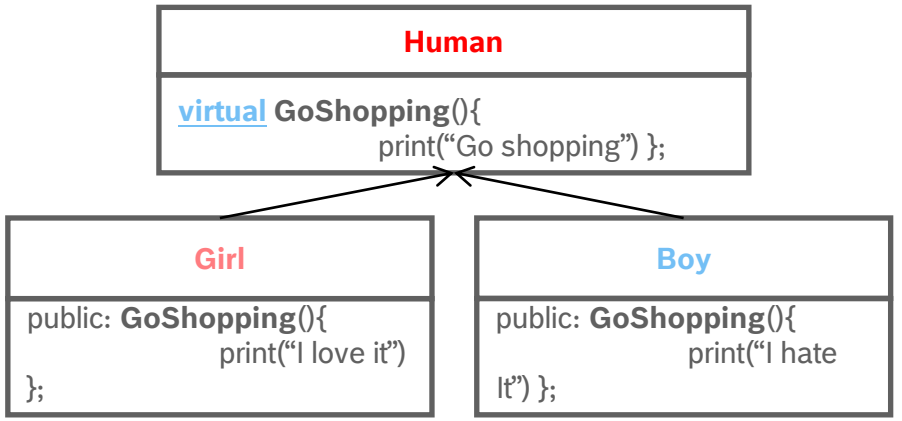


REMEMBER

GoShopping is a virtual method . It becomes **pure virtual method.** GoShopping method does not need implementation. **Human** now becomes an abstract class.

8. Virtual function, polymorphism

03_polymorphism_boy_girl.cpp



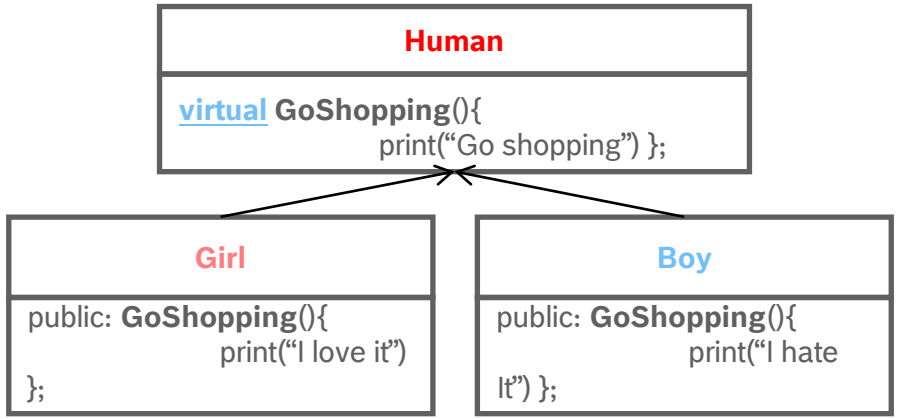

```
Girl alice;
Boy tom;
Human& h1 = alice;
Human& h2 = tom;
h1.GoShopping();
//I love it
h2.GoShopping();
//I hate it
```

REMEMBER

Method 1 to apply polymorphism:
A reference (base class type) refers to an object of derived class

8. Virtual function, polymorphism

03_01_Virtual.cpp




```
Girl alice;
Boy tom;
Human* h1 = &alice;
Human* h2 = &tom;
h1->GoShopping();
//I love it
h2->GoShopping();
//I hate it
```

REMEMBER

Method 2 to apply polymorphism:
A pointer of base class type points to an address of an derived class object.

8. Template Function

Syntax

```
template <typename objectType>
objectType & GetMax (const objectType & value1, const objectType & value2)
{
        if (value1 > value2)
                return value1;
        else
                return value2;
};
```

```
int nInteger1 = 25;
int nInteger2 = 40;
int nMaxValue = GetMax <int> (nInteger1, nInteger2);
double dDouble1 = 1.1;
double dDouble2 = 1.001;
double dMaxValue = GetMax <double> (nDouble1, nDouble2);
```

8. Template Class

Syntax

```
template <typename T>
class CMyFirstTemplateClass
{
public:
      void SetVariable (T& newValue) { m_Value = newValue; };
      T& GetValue () {return m_Value;};
private:
       T m_Value;
};
```

```
CMyFirstTemplate <int> mHoldInteger; // Template instantiation
mHoldInteger.SetValue (5);
std::cout << “The value stored is: “ << mHoldInteger.GetValue ();

CMyFirstTemplate <char*> mHoldString;
mHoldInteger.SetValue (“Sample string”);
std::cout << “The value stored is: “ << mHoldInteger.GetValue ();
```

Recommended learning resources / references

- The C++ Tutorial | Learn C++ (learncpp.com)
- C++ Programming Language - GeeksforGeeks
- C++ Programming Tutorials Playlist - YouTube
- Example references:
  - Amazon.com: C++ in One Hour a Day, Sams Teach Yourself: 9780789757746: Rao, Siddhartha: Books

Thank you!

Bosch
Global
Software
Technologies
alt_future