Chapter 2: Introducing C++

Department of Computer Science and Engineering
Kathmandu University

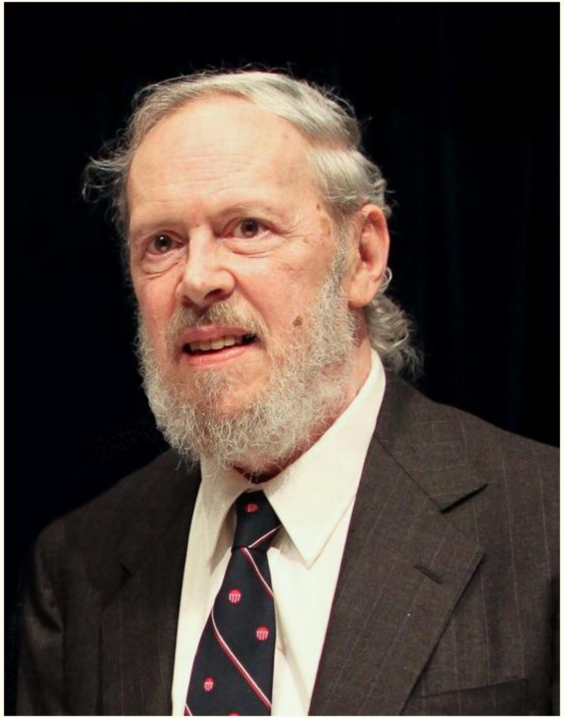
Rajani Chulyadyo, PhD

Contents

History of C++

1972

- Dennis Ritchie at Bell Labs designed C, a general-purpose, high level programming language.
- The Unix kernel, originally implemented in assembly language, was re-implemented in C.



Source: https://en.wikipedia.org/wiki/Dennis_Ritchie

History of C++

Late 70's	**OOP becomes popular**
1979	**Bjarne Stroustrup began developing "C with Classes", which was later renamed to C++**
1985	**The first commercial implementation of C++ was released**
1989	**C++ 2.0 was released**
1998	**C++98 was released, standardizing the language. That was a major release.** After C++98, C++ evolved relatively slowly.

Source: https://en.wikipedia.org/wiki/Bjarne_Stroustrup

History of C++

2003 — **A minor update (C++03) was released**

2009 — **C++0x got released**

2011 — **C++11 was released**

That was a major release that added numerous new features.

2014 — **C++ 14 was released**

Since 2011, C++ updates every 3 years regularly.

2017 — **C++17 was released**

2020 — **C++20 is planned to be released**

2023 — **C++23 is planned to be released**

Source: https://en.wikipedia.org/wiki/Bjarne_Stroustrup

- C++ is a **compiled, strongly-type, open ISO-standardized** language
  - compiles directly to a machine's native code
  - expects the programmer to know what he or she is doing
  - is standardized by a committee of the ISO (The purpose of standardization is to ensure that programs written to work with one compiler/interpreter will work with another)
- Many **C++ compilers** are available
  - e.g.,Clang, GCC, C++Builder, (Microsoft) Visual C++, Oracle C++ compiler etc.
- C++ offers many paradigm choices: **procedural, generic, OOP paradigms**
- C++ is **portable**

Myths vs Reality

Myth: C++ is only or even mostly used in legacy systems.

Reality: C++ is everywhere.

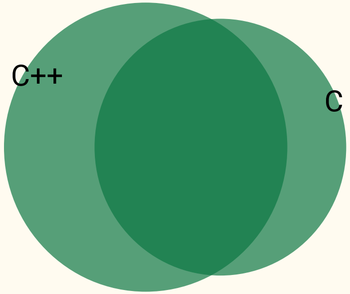
- System programming: Operating systems, device drivers etc.
- Databases, browsers, bank applications
- Graphics, game engines
- Embedded systems: Appliances, robotics, automobiles etc.
- High-level libraries: Machine Learning libraries
- Interpreters, compilers
- etc.

Myths vs Reality

Myth: C is a subset of C++

Reality: C and C++ are two different languages. They are closely related but have many significant differences.[1]

C++ began as a fork of an early, pre-standardized C, but they evolve separately.



[1] http://www.stroustrup.com/bs_faq.html#C-is-subset

Compilers

Computers understand only machine language, which consists of sets of instructions made of ones and zeros.

Imagine programming a computer directly in machine language using only ones and zeros!!

To make programming simpler and more understandable, high level languages have been developed.

Compilers (also interpreters or assemblers) translates programs written in high-level languages into machine language.

Compilers

Compiler	Author	Operating System			License type	IDE	Standard conformance		
		Windows	Unix-like	Other			C++11	C++14	C++17
C++Builder (modern, bcc*c)	Embarcadero (LLVM)	Yes (bcc32c,bcc64)	⟨iOS⟩ (bccios*), ⟨Android⟩ (bcca*)	No	Proprietary	Yes	Yes	Yes	Yes
Clang (clang++)	LLVM Project	Yes	Yes	Yes	UoI/NCSA	Xcode, QtCreator (optional)	Yes	Yes	Yes
GCC (g++)	GNU Project	MinGW, MSYS2, Cygwin, Windows Subsystem	Yes	Yes	GPLv3	QtCreator, Kdevelop, Eclipse, NetBeans, Code::Blocks, Geany	Yes	Yes	Yes
Oracle C++ Compiler (CC)	Oracle	No	Linux, Solaris	No	Proprietary (Freeware)	Oracle Developer Studio, NetBeans	Yes	Yes	No
Turbo C++ (tcc)	Borland (Code Gear)	No	No	DOS	Proprietary (Freeware)	Yes	No	No	No
Visual C++ (cl)	Microsoft	Yes	Linux, macOS; ⟨Android⟩, ⟨iOS⟩	No	Proprietary	Visual Studio	Yes	Yes	Partial

Source: https://en.wikipedia.org/wiki/List_of_compilers#C.2B.2B_compilers

For more info on compiler support: https://en.cppreference.com/w/cpp/compiler_support

How C++ works

Source code files → Preprocessing → Compilation and Assembly → Linking → Executable program

Preprocessing: Modifies the original source code files according to the preprocessor directives (such as #include, #define etc.)

Compilation and Assembly: Translates the source code into object files containing machine language code.

Linking: Links the object code with the libraries and produces a single executable file.

A simple C++ program

```
#include <iostream>


int main()
{
    std::cout << "Hello World!" << std::endl;
    std::cin.get();

    return 0;
}
```

A simple C++ program

```cpp
#include <iostream> // Tells the compiler to include "iostream" header
file
// Required for std::cout, std::cin, and std::endl

int main() // The program starts by executing the main() function
{
    std::cout << "Hello World!" << std::endl; // Prints "Hello World!"
    std::cin.get(); // Get a character from the user

    return 0; // The program's return value to "the system"
}
```

A simple C++ program

To get an executable:

```
g++ HelloWorld.cpp
```

Or

```
g++ -o HelloWorld.out HelloWorld.cpp
```

Syntax for g++ is

```
g++ [options] <inputs>
```

A simple C++ program

To enable C++11:

```
g++ -std=c++11 HelloWorld.cpp
```

To enable C++17

```
g++ -std=c++17 HelloWorld.cpp
```

A simple C++ program

To only run the preprocessor:

```
g++ -E HelloWorld.cpp
```

To only run preprocess, compile, and assemble steps:

```
g++ -c HelloWorld.cpp
```

Basic input/output

C++ uses a convenient **abstraction** called **streams** to perform input and output operations in sequential media such as the screen, the keyboard or a file.

A stream is an entity where a program can either insert or extract characters to/from.

Stream	Description
`cin`	standard input stream
`cout`	standard output stream
`cerr`	standard error (output) stream
`clog`	standard logging (output) stream

Basic input/output: cout

`cout` writes characters to the standard output (screen/console).

For formatted output operations, `cout` is used together with the insertion operator (<<).

```
std::cout << "Output sentence"; // prints Output sentence on screen
std::cout << 120;               // prints number 120 on screen
std::cout << x;                 // prints the value of x on screen

// Multiple insertion operators may be chained
std::cout << "This " << " is a " << "single C++ statement";
std::cout << "I am " << age << " years old and my zipcode is "  <<
zipcode;
```

Basic input/output: cin

`cin` is used to access the standard input device (keyboard by default).

For formatted input operations, `cin` is used together with the extraction operator (<<).

```
int a;
int b;

std::cout << "Enter two numbers: " ;
std::cin >> a >> b;     // Equivalent to std::cin >> a; std::cin >> b;
```

Lab 1 exercise

Question 1: Write a program that takes two integers from the user and prints the following:

a. Sum of the two numbers
b. Product of the two numbers

(20 mins)

Basic input/output: cin

Suppose you enter “This is a sentence” to the following program. What is the expected output?

```
#include <iostream>

int main() {
    std::string str;
    std::cout << "Enter a sentence: ";
    std::cin >> str;
    std::cout << "The entered sentence is : " << str << std::endl;
}
```

Basic input/output: cin

`cin` extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted.

Thus, only the first word will be printed in the previous program.

To get an entire line from `cin`, use `getline` function.

```
#include <iostream>
int main() {
   std::string str;
   std::cout << "Enter a sentence: ";
    getline(std::cin, str);
   std::cout << "The entered sentence is : "<< str << std::endl;
}
```

Basic concepts

- Keywords
- Identifiers
- Data types
- Variables
- Constants
- Operators
- Expressions
- Statements
- Reference variables
- Inline functions
- Function overloading

Keywords

Reserved words that are not available for re-definition or overloading.

Examples:

`alignas`, `alignof`, `and`, `and_eq`, `asm`, `auto`, `bitand`, `bitor`, **`bool`**, **`break`**, **`case`**, `catch`, **`char`**, `char16_t`, `char32_t`, **`class`**, `compl`, **`const`**, `constexpr`, `const_cast`, **`continue`**, `decltype`, **`default`**, **`delete`**, **`do`**, **`double`**, `dynamic_cast`, **`else`**, **`enum`**, `explicit`, `export`, **`extern`**, **`false`**, **`float`**, **`for`**, **`friend`**, **`goto`**, **`if`**, **`inline`**, **`int`**, **`long`**, `mutable`, **`namespace`**, **`new`**, `noexcept`, `not`, `not_eq`, **`nullptr`**, **`operator`**, `or`, `or_eq`, **`private`**, **`protected`**, **`public`**, `register`, `reinterpret_cast`, **`return`**, **`short`**, **`signed`**, **`sizeof`**, **`static`**, `static_assert`, `static_cast`, **`struct`**, **`switch`**, **`template`**, **`this`**, `thread_local`, **`throw`**, **`true`**, **`try`**, **`typedef`**, `typeid`, **`typename`**, `union`, **`unsigned`**, **`using`**, **`virtual`**, **`void`**, **`volatile`**, `wchar_t`, **`while`**, `xor`, `xor_eq`

Specific compilers may also have additional specific reserved keywords.

Identifiers

- An identifier is an arbitrarily long sequence of **digits, underscores**, lowercase and uppercase **Latin letters**, and **most Unicode characters**.
  - e.g. `var`, `var1`, `my_var`, `vär`, `_var`, `MyClass`, `Func` etc.
- A **valid identifier** must begin with a non-digit character (Latin letter, underscore, or Unicode non-digit character).
  - e.g. `var1`, `_var1` are valid identifiers but `1var` is not.
- Identifiers are **case-sensitive** , and every character is significant.
  - e.g. `Var` and `var` are different.

Identifiers

An identifier can be used to name objects, references, functions, enumerators, types, class members, namespaces, templates, template specializations, parameter packs, goto labels, and other entities, with the following exceptions:

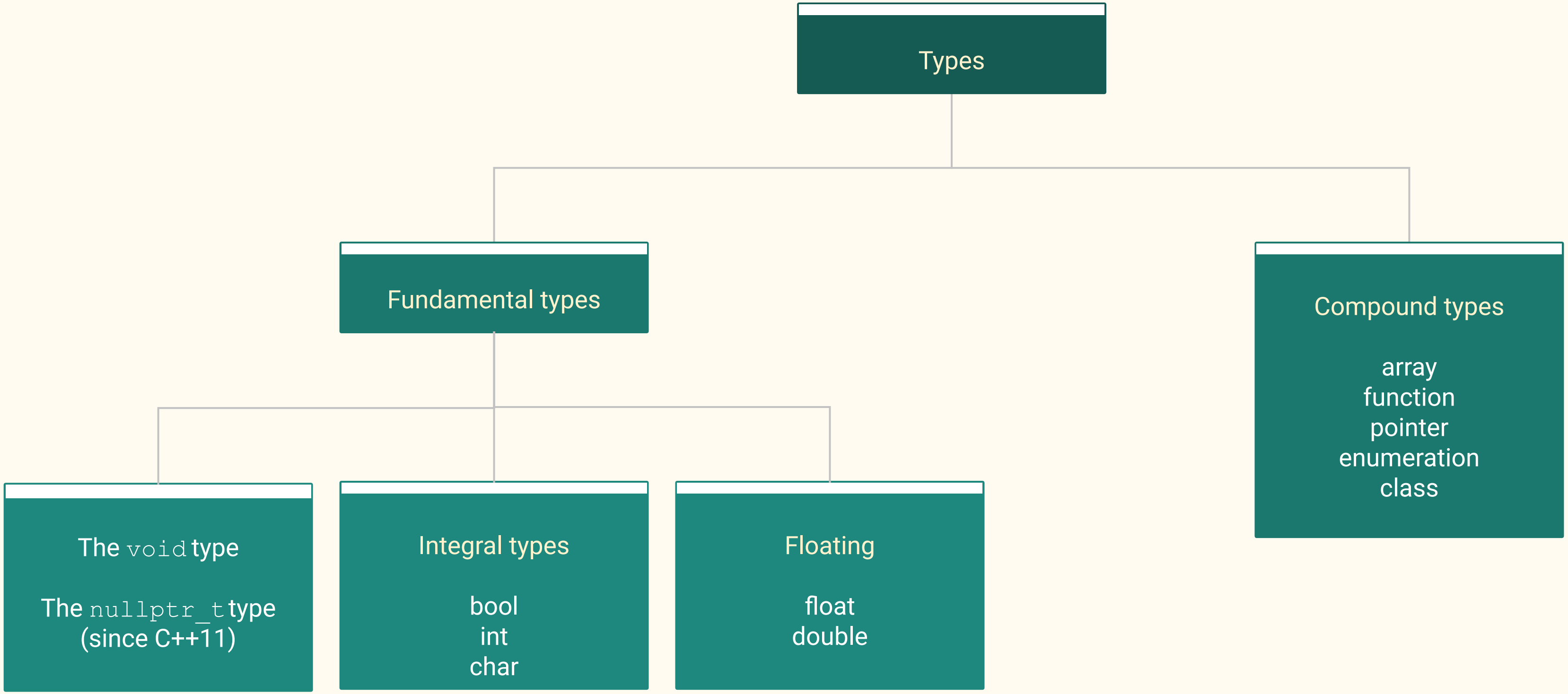
1. The identifiers that are **keywords** (e.g. `double`, `define` etc.) cannot be used for other purposes;
2. The identifiers **with a double underscore anywhere** (e.g. `my__var`, `__my_func`) are **reserved**;
3. The identifiers that **begin with an underscore followed by an uppercase letter** (e.g. `_MyVar`, `_My_func`, `_Object`) are reserved;
4. The identifiers that **begin with an underscore** are **reserved** in the **global namespace**.

Types

Objects, references, functions, and expressions have a property called **type**, which

- **restricts the operations** that are permitted for those entities and
- **provides semantic meaning** to the otherwise generic sequences of bits.

Types


Types
Fundamental types
Compound types
array
function
pointer
enumeration
class
The void type
The nullptr_t type
(since C++11)
Integral types
bool
int
char
Floating
float
double

Fundamental types

The `void` type

- Represents the absence of type

```
void func(int);   // Returns nothing
void* ptr;  // A void pointer
```

The `nullptr` type (since C++11) (will be covered later in this chapter)

Fundamental types: Integral types

- **Boolean type**
  - Can only represent one of two states, `true` or `false`.
- **Integer type**
  - Can store a whole number value, such as 7 or 1024.
  - Exist in a variety of sizes, and can either be `signed` or `unsigned`, depending on whether they support negative values or not.
- **Character type**
  - Can represent a single character, such as 'A' or '$'.
  - The most basic type is char, which is a one-byte character.
  - Other types are also provided for wider characters.

Fundamental types: Integral types

Group	Type	Storage size	Value range
Character types	**`char`**	Exactly one byte in size. At least 8 bits.	0 to 255 (by default) or -128 to 127 (when compiled with `--signed_chars`)
	`char16_t`	Not smaller than char. At least 16 bits.	?
	`char32_t`	Not smaller than char16_t. At least 32 bits.	?
	`wchar_t`	Can represent the largest supported character set.	?
Integer types (signed)	**`signed char`**	8 bits	-128 to 127
	`signed` **`short`** *`int`*	2 bytes	?
	`signed` **`int`**	Not smaller than short. At least 16 bits. 2 or 4 bytes depending on the compiler and the system architecture	?
	`signed` **`long`** *`int`*	Not smaller than int. At least 32 bits.	?
	`signed` **`long long`** *`int`*	Not smaller than long. At least 64 bits.	?
Integer types (unsigned)	**`unsigned char`**	(same size as their signed counterparts)	?
	`unsigned short` *`int`*		
	`unsigned` *`int`*		
	`unsigned long` *`int`*	https://en.cppreference.com/w/cpp/language/types	
	`unsigned long long` *`int`*	http://cplusplus.com/doc/tutorial/variables/	

Fundamental types: Floating types

They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.

- **`float`**
  - Single precision floating point type. Usually IEEE-754 32 bit floating point type
- **`double`**
  - Double precision floating point type. Usually IEEE-754 64 bit floating point type
- **`long double`**
  - Extended precision floating point type.

Compound types

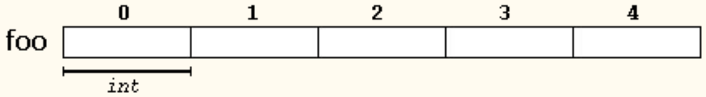
Compound types are composed of more than one types.

- Array
- Reference
- Pointer
- Function
- Class
- Enumeration

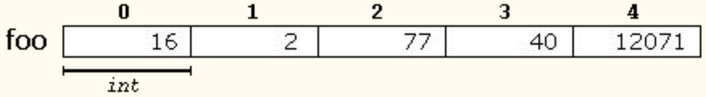
Compound types: Array

- Allows to store multiple pieces of information of the same type.
- The elements are placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

```
int foo[5];   // Declaration
```



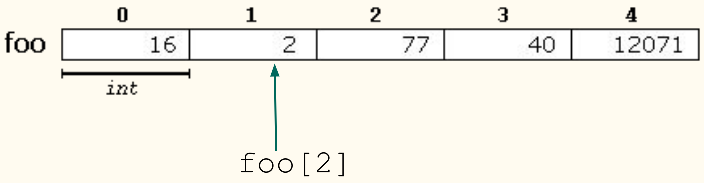

```
int foo [5] = { 16, 2, 77, 40, 12071 }; // Initialization
```



Compound types: Array

Accessing the values in an array

Syntax: `name[index]`



Iterating over an array

```
for (int i=0; i < 5; i++) {
    std::cout << foo[i] << std::endl;
}
```

```
for (int elem : foo) {
    std::cout << elem << std::endl;
}
```

Compound types: References

A **reference variable** provides an **alias** for a **previously defined variable.**

Syntax for creating a reference variable:

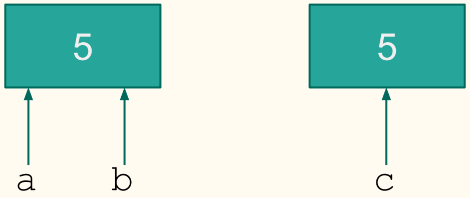
```
data-type & reference-name = variable-name;
```

Example:

```
int a = 5;

int & b = a;

int c = a;
```



Compound types: References

```
int a = 5;
int & b = a;   // b is now a reference variable that refers to a
// int &b;   // This is invalid. A reference variable must be initialized.

a++; // Both a and b will be 6

int c = 10;
b = c;    // A reference variable cannot be change. b will always refer to a.
          // Here, the content of c will be copied to a but b will not refer to c.

c++;    // Now c will be 11 but b and a will remain unchanged.
```

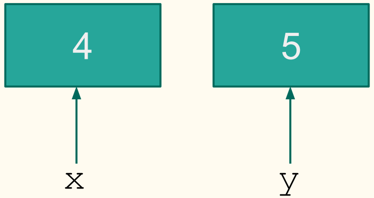
Compound types: References

A major application is in passing arguments to functions.

Pass by value

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



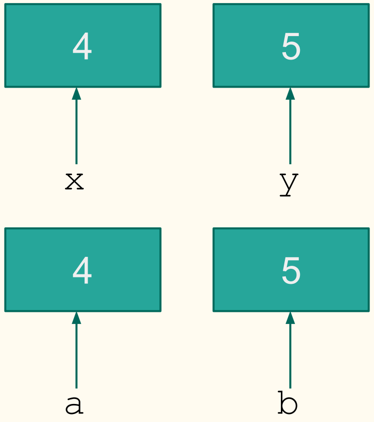
Compound types: References

A major application is in passing arguments to functions.

Pass by value

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



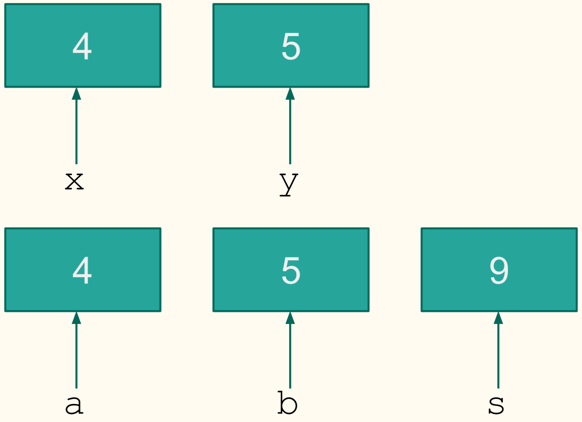
Compound types: References

A major application is in passing arguments to functions.

Pass by value

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



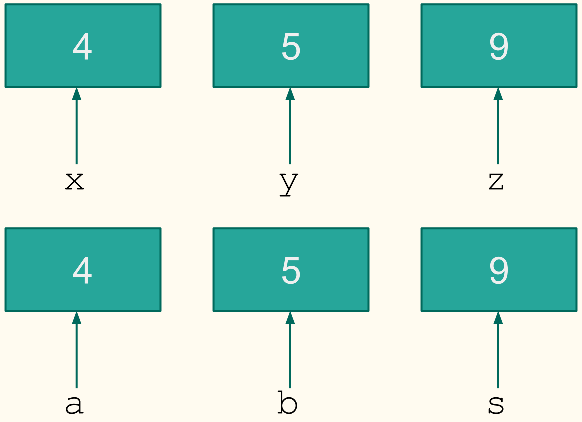
Compound types: References

A major application is in passing arguments to functions.

Pass by value

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



Compound types: References

A major application is in passing arguments to functions.

Pass by value

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```

Pass by reference

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```

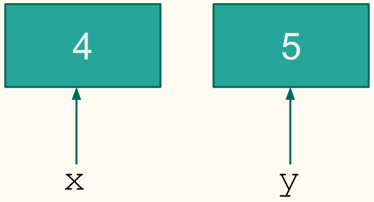
Compound types: References

A major application is in passing arguments to functions.

Pass by reference

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



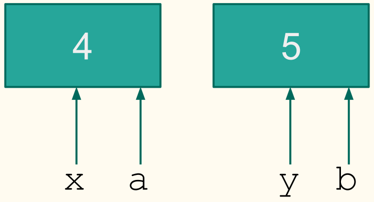
Compound types: References

A major application is in passing arguments to functions.

Pass by reference

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



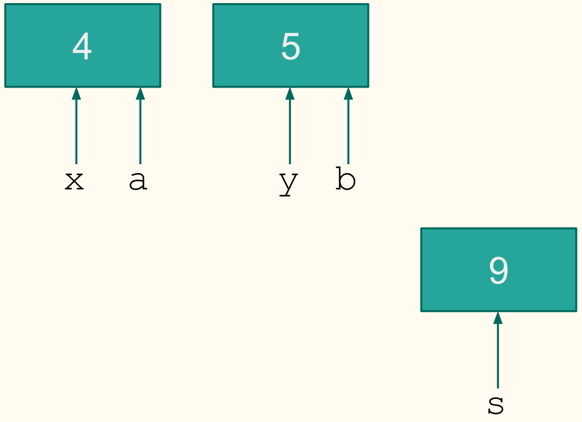
Compound types: References

A major application is in passing arguments to functions.

Pass by reference

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



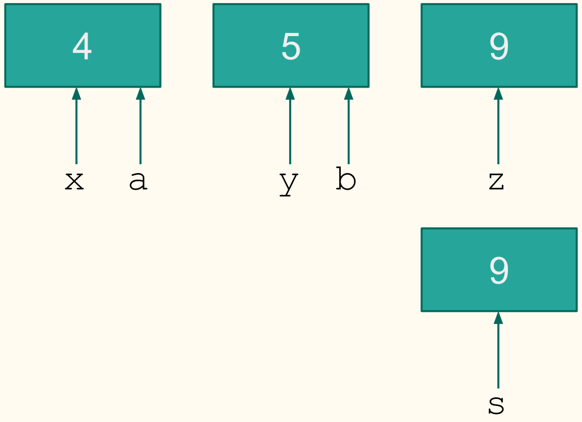
Compound types: References

A major application is in passing arguments to functions.

Pass by reference

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

int main() {
    int x = 4, y = 5;
    int z = add(x, y);
}
```



Pass by reference

```
void add (const int & a, const int & b, int & result) {
   result = a + b;
}

int main() {
   int x = 4, y = 5;
   int z;
   add(x, y, z);
}
```

Why pass by reference?

- It eliminates the copying of object parameters back and forth. (Copying large/complex objects may be time-consuming/complicated)
- It enables functions to return multiple values

  Examples:

```
void func(int input1, int input2, int & output1,
          int & output2) { … }
void func(const int & input1, const int & input2,
          int & output1, int & output2) { … }
```

Lab 1 exercise

Question 2: Write a function to swap two numbers (using references).

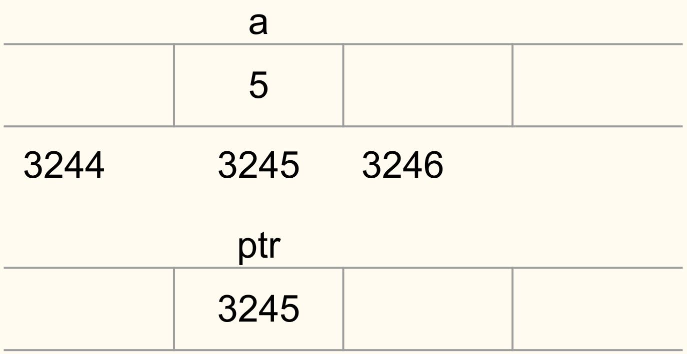
(15 mins)

Compound types: Pointer

- A variable that stores the memory address as its value.
- Is created with the * operator

```
int a = 5;
int* ptr = &a;  // Assign the address of a to the pointer
```

Address-of operator



Compound types: Pointer

- Pointers are said to "point to" the variable whose address they store.
- Pointers can be used to directly access the variable they point to using the dereference operator (*).

```
int a = 5;
int* ptr = &a; // ptr points to a
*ptr = 10;     // a's value will now be 10

int b = *ptr;  // A new variable b will be created.
               // a's value will be copied to b
```

Dereference operator

Lab 1 exercise

Question 3: Try this out and explain the output. (10 mins)

```
#include <iostream>

int main() {
    int a = 10;
    int* ptr = &a;

    std::cout << "ptr = " << ptr << std::endl;
    std::cout << "&ptr = " << &ptr << std::endl;
    std::cout << "&a = " << &a << std::endl;
    std::cout << "a = " << a << std::endl;
    std::cout << "*ptr = " << *ptr << std::endl;

    *ptr = 20;
    std::cout << "a = " << a << std::endl;
}
```

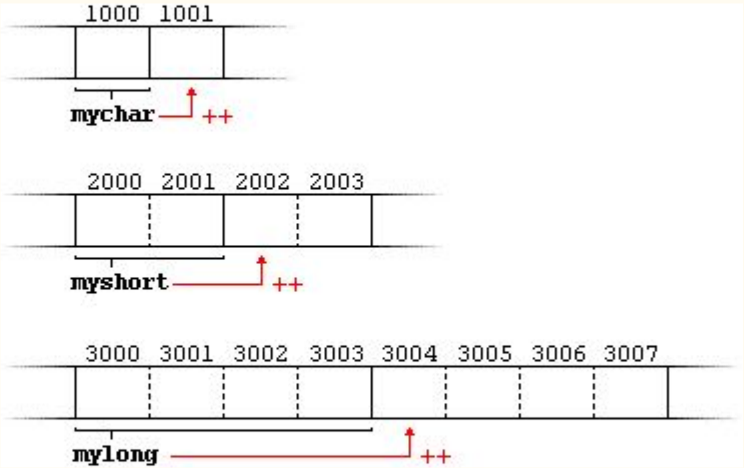
Compound types: Pointer

Pointers arithmetics

Addition and subtraction operations are allowed on pointers.

```
char *mychar;
short *myshort;
long *mylong;

// ++ will move the pointer to the next
// x byte(s), x being the size of the type
++mychar; // Moves to the next byte
++myshort;
++mylong;
```



Compound types: Pointer

Pointers and arrays

- Arrays work very much like pointers to their first elements.
- An array can always be implicitly converted to the pointer of the proper type.

```
int arr[10];
int* ptr = arr;
*ptr++ = 5;
*(ptr+3) = 6;
arr[5] = 9;

std::cout << "arr contains ";
for (int ele  : arr) {
    std::cout << ele << ", " ;
}
std::cout << std::endl;
```

Compound types: Pointer

Pointer initialization

```
int var;
int* ptr = &var;

int a;
int* ptr1;   // It is an uninitialized pointer
ptr1 = &var; // Now it is initialized to point to var

int* ptr2 = nullptr;  // A null pointer points to nowhere

int* ptr3 = 0;           // Null pointer.
```

Compound types: Pointer

Pointers and dynamic memory allocation

Dynamic memory is allocated using operator `new` followed by a data type specifier.

```
int* ptr = new int(2);
```

It creates and initializes objects with dynamic storage duration, i.e., objects whose lifetime is not limited by the scope in which they were created.

It returns a pointer to the beginning of the new block of memory allocated.

Compound types: Pointer

Pointers and dynamic memory allocation

- Once the memory allocated using operator `new` is no longer needed, it can be freed using operator `delete` so that the memory becomes available.

```
delete ptr;
```

Memory leaks

- If the memory allocated using operator `new` is not `delete`d, and the original value of pointer is lost, then the memory cannot be deallocated: a memory leak occurs.

Compound types: Pointer

Memory leak example

```
double square(double num) {
    double* ptr = new double(num);

    return (*ptr) * (*ptr); // Return without deallocating ptr
}


int main() {
    double s = square(100.);
}
```

Compound types: Function

A function type is the type of a variable or parameter to which a function has or can be assigned.

A function type depends on the type of the parameters and the result type of the function.

```cpp
#include <iostream>

int func1(int a) { return a/2; }
int func2(int a) { return a*2; }

int main() {
   int (*F)(int);
```

```cpp
   F = func1;
   std::cout << F(100) << std::endl;

   F = func2;
   std::cout << F(100) << std::endl;
}
```

Compound type: Class

(will be covered in Chapter 3)

Compound type: enumeration

- Enumerated types are types that are defined with a set of custom identifiers, known as enumerators, as possible values.

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

- Objects of these enumerated types can take any of these enumerators as value.

```
colors_t mycolor;
mycolor = blue;
if (mycolor == blue) {
    mycolor = red;
}
```

Compound type: enumeration

- Values of enumerated types declared with enum are implicitly convertible to an integer type, and vice versa.

- If the elements of an enum are not assigned an integer, they are always assigned an integer starting from 0.

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Here, black =0, blue = 1, green = 2 and so on.

```
enum colors_t {black = 4, blue = 20, green};
```

Here, green = 21

Variables : Declaration

- C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use.
- This informs the compiler the size to reserve in memory for the variable and how to interpret its value.

```cpp
int x, y;
```

```cpp
int x;
int y;
```

Variables : Initialization

3 ways of initializing a variable:

```
int x = 1; // Copy initialize an integer

int y(1);  // Direct initialize an integer

int z{1}; // Uniform initialization of an integer (Only since C++11)
```

Constants

Expressions with a fixed value. Can be evaluated at compile time

- Literals
- Typed constant expressions
- Preprocessor definitions

Constants: Literals

They are used to express particular values within the source code of a program.

Integer literals

```
int a = 5; // Here, 5 is a literal constant (of type int)
```

Just like variables, literal constants have a type. **By default, integer literals are of type `int`.** We append **`u`** and/or **`l`** to an integer literal to specify a different integer type.

```
523u // unsigned int
523l // long int
523lu or 523ul // unsigned long int
```

Constants: Literals

Floating Point Numerals

They express real values, with decimals and/or exponents.

```
3.0    // double
3.14   // double
2.1e7  // double
```

By default, floating-point literals are of type `double`

```
3.14f     // float
3.14l     // long double
```

Constants: Literals

Character and string literals are enclosed in quotes.

Examples:

```
'a'           // char
"Hello World" // string
```

Special characters, such as new line, tab, backslash etc., are preceded by a backslash

```
'\n' // New line
'\t' // Tab
'\\' // Backslash
```

Constants: Literals

Other literals:

boolean literals are values of type **`bool`**, that is **`true`** and **`false`**

`nullptr` is the pointer literal which specifies a null pointer value (since C++11)

Constants

Typed constant expressions

```
const int a = 5; // Value of a cannot be modified
```

Preprocessor definitions

```
#define MAX 10   // Defines a constant MAX whose value is 10
```

Operators

- Assignment operators
  - Assign a value to a variable
  - Examples:
    - `x = 1;`
    - `x = y;`
    - `x = (y = 3) + 1;`
    - `x = y = 2;`
- Arithmetic operators
  - + (addition), - (subtraction), * (multiplication), / (division), % (modulo)

Operators

- Compound assignment operators
  - Modify the current value of a variable by performing an operation on it
  - Examples:
    - `x += 5;`
    - `x -= 5;`
    - `x /= 6;`
  - `+=, -=, *=, ≠, %=, >>=, <<=, &=, ^=, |=`
- Increment / decrement operators
  - Equivalent to `+= 1` and `-= 1`
  - `++, --`
  - Pre-increment, post-increment
  - Pre-decrement, post-decrement

Operators

- Relational comparison operators
  - `==`, `!=`, `>`, `<`, `>=`,`<=`
  - `<=>` (3-way comparison available since C++20)
- Logical operators
  - `!`, `!=`, `&&`, `||` (Alternative spellings: **`and`** for `&&`, **`or`** for `||`, **`not`** for !, **`not_eq`** for `!=`)
  - && - If the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated)
  - || - If the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated)
  - What will be the output?

```
z = 10;
if (z < 10 && ++z > 10){
}
std::cout << "z = " << z << std::endl;
```

```
z = 10;
if (++z > 10 && z < 10){
}
std::cout << "z = " << z << std::endl;
```

Operators

- Conditional ternary operator
  - Evaluates an expression, and returns one value if that expression evaluates to true, and a different one if the expression evaluates as false
  - condition ? result1 : result2
  - Example: `(a > b ? a : b)`
- Comma operator
  - Separates two or more expressions that are included where only one expression is expected
  - Example:
    - `a = (b= 3, c= 3, b + c * 2);`

Operators

- Bitwise operators
    - Modify variables considering the bit patterns that represent the values they store

Operator	ASM equivalent	Description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

Operators

a	b	OR	XOR
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

- Bitwise operators
  - Examples:

```
int y{7};
int z{2};

std::cout << "y & z = " << (y & z) << std::endl; // Returns 2. Why?
std::cout << "y | z = " << (y | z) << std::endl; // Returns ?
std::cout << "y ^ z = " << (y ^ z) << std::endl; // Returns ?
std::cout << "y << 2 =" << (y << 2) << std::endl; // Returns 28. 00000111 << 2 = 00011100
std::cout << "y >> 2 =" << (y >> 2) << std::endl; // Returns ?
std::cout << "~y = " << ~y << std::endl; // Returns -8 ?
```

Operators

- Special operators
  - Conversion operators
    - Convert from one type to another using `()`, `static_cast`, `dynamic_cast` etc.
    - `int b = (int) a;`
    - `int b = static_cast<int>(a);`
  - Member access operators
    - `a[b], *a, &a, a -> b, a.b, a->*b, a.*b`
  - `new`
  - `delete`
  - `sizeof`
  - `typeid`
  - `noexcept` (since C++11)
  - Scope resolution operator (`::`)

Precedence of operators

A single expression may have multiple operators.

```
y = 2 + 10 / 5;
```

```
z = 7 + 4 ^ 3;
```

What will the value of y be? 4 or 2?
And how about z? 14 or 8?

Precedence of operators

A single expression may have multiple operators.

```
y = 2 + 10 / 5;

z = 7 + 4 ^ 3;
```

What will the value of y be? 4 or 2?
And how about z? 14 or 8?
Why?

Operators are evaluated according to their precedence.

From greatest to smallest priority, C++ operators are evaluated in the following order:

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(*type*)	C-style type-casting	
4	Pointer-to-member	.* ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or	|	bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction	||	logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= |=	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

Source: http://www.cplusplus.com/doc/tutorial/operators/

Expressions

An expression is a sequence of operators and their operands, that specifies a computation.

Operators may be unary, binary or ternary.

Examples:

```
a + b
```

```
++a
```

```
(a > b) ? a : b
```

Statements

A **simple C++ statement** is each of the individual instructions of a program, like the variable declarations and expressions

They always end with a semicolon (;), and are executed in the same order in which they appear in a program.

Example:

```
int x = 5, y = 6;

int z = x + y;
```

Statements

A **compound statement** is a group of statements, all grouped together in a block, enclosed in curly braces {}.

```
{ statement1; statement2; statement3; }
```

Flow control statements

- Selection statements
  - `if-else, switch`
- Iteration statements
  - `while` loop, `for` loop
  - Range-based for loop (since C++11)
    `for (int element : elements) { statement; }`
- Jump statements
  - `goto, break, continue`

Lab 1 exercise

Question 3: Write a program to input 10 double precision floating point numbers from the user, store them in an array, and then compute mean and standard deviation of the array. Note that the standard deviation σ of a collection of numbers x_j, j = 1,2,..., N is given by

$$\sigma = \sqrt{\frac{\sum_{j=1}^{N} (x_j - \bar{x})^2}{N-1}}$$

where $\bar{x}$ is the mean of the numbers.

(20 mins)

Default arguments

- A default value to be passed to a parameter.
- Used when the function call does not specify an argument for that parameter.
- Must be the rightmost argument in the function's parameter list.

```
int multiply(int a, int b = 1) {
    return a * b;
}
int main() {
    int product;
    product = multiply(8);     // 8
    product = multiply(8, 5); // 40
}
```

Inline functions

- Calling a function generally causes a certain overhead (stacking arguments, jumps, etc.)
- For very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.
- This can be done by preceding the function declaration with the `inline` specifier
- It suggests the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

Inline functions

```
inline int square(int a) {
   return a * a;
}
int main() {
   int a = 4;
   int sq = square(a);
   return 0;
}
```

Note that the `inline` specifier merely indicates the compiler that inline is preferred for this function, although **the compiler is free to not inline it.**

Scope and namespace

Scope of a variable is the area of the program where the variable is valid.

A **global variable** is valid from the point it is declared to the end of the program.

A **local variable**'s scope is limited to the block where it is declared and cannot be accessed (set or read) outside that block.

We will discuss the following two scores here, other scopes will be covered later.

- Global scope
- Block scope
- Namespace score

Scope and namespace

Block scope

The potential scope of a variable introduced by a declaration in a block (compound statement) begins at the point of declaration and ends at the end of the block.

```
int main() {
   std::string str("The scope of this variable is within the main() function.");

   {
       std::string str("The scope of this variable is within this block, which ends at line 6.");
   }
}
```

Namespace scope

A **namespace** is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.

It is used to organize code into logical groups and to prevent name collisions that can occur especially when your codebase includes multiple libraries.

Syntax to define a namespace:

```
namespace namespace_name {
    statements;
}
```

Scope and namespace

Points to remember while defining a namespace

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- No need to give semicolon after the closing brace of the definition of the namespace.
- Namespace definition can be split over several units.

```
namespace n1 {
   namespace n2 {
       int a;
   }
}
namespace n3 {
   int a;
}
```

Scope and namespace

A symbol, by default, exists in a **global namespace**, unless it is defined inside a block starts with keyword namespace, or it is a member of a class, or a local variable of a function.

Scope and namespace

There are three ways to use a namespace in the program,

1. Scope Resolution Operator (::)
   - Example:
     ```
     int b = n1::a;
     int c = ::a; // defined in global namespace
     ```
2. The using directive
   - Example:
     ```
     using namespace n1;
     int b = a;
     ```
3. The using declaration
   - Example:
     ```
     using namespace n1::a;
     b = a;
     ```

```
namespace n1 {
    namespace n2 {
        int a;
    }
}
int a;
```

Recall the HelloWorld program

```cpp
#include <iostream> // Required for std::cout, std::cin, and std::endl
// cout, cin and endl are defined in namespace  std

int main()
{
    std::cout << "Hello World!" << std::endl;
    std::cin.get();

    return 0;
}
```

Recall the HelloWorld program

```
#include <iostream> // Required for std::cout, std::cin, and std::endl
// cout, cin and endl are defined in namespace  std
using namespace std;
int main()
{
    cout << "Hello World!" << endl;
    cin.get();

    return 0;
}
```

Scope and namespace

```cpp
#include <iostream>

std::string str("This is global.");
namespace n1 {
   std::string str("This is inside namespace n.");
   namespace n2 {
       std::string str("This is inside namespace n2 of n1.");
   }
}

int main() {
   std::string str("This is local.");

   std::cout << "str = " << str << std::endl;
   std::cout << "::str = " << ::str << std::endl;
   std::cout << "n1::str = " << n1::str << std::endl;
   std::cout << "n1::n2::str = " << n1::n2::str << std::endl;
   {
       std::string str("This is inside the nested block.");
       std::cout << "str inside the nested block = "<< str << std::endl;
   }
}
```

Static variables

Static variables keep their values and are not destroyed even after they go out of scope.

```
#include <iostream>
int generateID()
{
    static int s_id{ 0 };
    return ++s_id;
}
int main()
{
    std::cout << generateID() << '\n'; // Prints 1
    std::cout << generateID() << '\n'; // Prints 2
    std::cout << generateID() << '\n'; // Prints 3
}
```

Function overloading

Multiple **functions** in the **same scope** may have the **same name**, as long as their **parameter lists** are **different**. This is known as **function overloading**.

Function declarations that differ only in the return type cannot be overloaded.

```
int add(int a, int b) {  return a + b;  }
int add(int a, int b, int c) {   return a + b + c;    }
// long add(int a, int b) {  return a + b;  }  // Not possible


int main() {
   int x, y, z;
   int sum;

   sum = (x = 4, y = 3, add(x, y));  // sum = 7

   sum = (x = 4, y = 3, z = 9, add(x, y, z)); // sum = 16
}
```

String

C++ has a class for strings

```
#include <string>
int main() {
    std::string str1 = "Hello";
    std::string str2 = "World";
    std::string str3 = str1 + " " + str2;

    std::cout << "Length of str3 = " << str3.length() << std::endl;
}
```

Exercise: Explore the following string methods: `append(str)`, `c_str()`, `clear()`, `compare(str)`, `find(str [, index])`, `insert(index, str)`, `push_back(ch)`, `replace(index, len, str)`, `substr(start [, len])`
Also explore: `stoi(str[, idx, base])` and `stringstream`

The stack and the heap

Lab 1 exercise

Question 5:

(15 mins)

Comparison between C and C++

C

- Procedure-oriented
- Very light-weight, compiled
- No support for data encapsulation
- Good for embedded devices, system-level code etc.

C++

- Object-oriented
- Light-weight, compiled
- Provides data encapsulation
- Good for developing games, networking, server-side applications etc.

More comparison at the end of the course

Resources

1. https://en.cppreference.com/w/cpp/
2. http://cplusplus.com/doc/tutorial/
3. https://www.learncpp.com
4. https://www.edureka.co/blog/namespace-in-cpp/

Assignment

Assignment # 1

1. Explain how C++ programs work.
2. A C++ program that compiles in one compiler may not compile in another compiler. Why?
3. A C++ program that compiles in one version of a compiler may not compile in another version of the same compiler. Why?
4. What do you understand by operator precedence and associativity?
5. What are the differences between pointers and references?
6. What are the differences between pass by value and pass by reference?
7. Explain the purpose of namespaces.
8. Compare inline function and normal function on the basis of memory usage, execution time and also explain trade-off between them.
9. Differentiate between pointer variable and reference variable?