Chapter 1 – Types and Declarations:

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name and (actually, to the entity referred to by the name) and how such operations are interpreted.

Ex: float x; → x is a floating point variable
int y = 7 → y is an integer var. with the intial value 7
float f(int); → f is a function taking an argument of type int and returning a floating-point number

"In this lecture, we'll demonstrate language features. These simple elements are used to construct C++ programs. You must know these elements, plus the terminology and simple syntax that goes with them, in order to complete a real project in C++, especially to read code written by others."

- Boolean, character and integer types → integral types.
- Integral and floating point types are collectively called arithmetic types
- Enumerations and user defined types are called user-defined types
other types are → built-in types

"The integral and floating point types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision and the range available for computations.
Assumption is that a computer provides:
bytes for holding characters,
words for holding integers
some entity (holding) suitable for floating point computations,
addresses referring to these entities.
The C++ fundamental types, together with pointers and arrays present the machine-level notions to the programmer in a reasonably implementation independent manner.

Booleans: {true, false}.

```
- void f(int a, int b)
  {
    bool b1 = a == b;
    // ...
  }
```

- bool is_open(file*);
- bool greater(int a, int b) { return a>b; }

- bool $\xrightarrow{\text{to}}$ integer : true → 1
  false → 0

- int $\xrightarrow{\text{to}}$ bool : non-zero int → true
  0 → false

Ex: bool b = 7; ⇒ b → true
int i = true; ⇒ i → 1

* In arithmetic and logical expressions, booleans are converted to integers.

Ex: void g()
{
bool a = true;
bool b = true;

bool x = a+b; → a+b = 2 so x = true
bool y = a/b; → a/b = 1 so y = true
}

* A pointer can be implicitly converted to a bool.

- Nonzero pointer → true
- zero-valued → false

Character Types:

- Almost universally, a "char" has 8 bits, so that it can hold 256 different values

- We should avoid making assumptions about the representation of objects. This general rule applies even to characters.

- There are different character sets defined by ISO-646.
  Ex: ASCII char. set provides the characters that appear on your keyboard

- Each character constant has an integer value.

- A type wchar_t is provided to hold characters of a larger character set such as Unicode. The size of wchar_t is implementation-dependent and large enough to hold the largest character set supported by the implementation's locale.
  (name is leftover from C. In C, wchar_t is a typedef rather than a built-in type. _t is added to distinguish built-in typedefs)

Character literals: A character literal, often called a character constant, is a character enclosed in single quotes, ex: 'a' and '0'.
- Type of a character literal is char.
- Such character literals are really symbolic constants for the integer value of the characters in the character set of the machine on which the C++ program is to run.

Integer Types: In 3 forms: int (plain int)
signed int
unsigned int

In 3 sizes: short int (short)
int
long int (long)

- plain ints are always signed.

Integer Literals:
- Decimal literals: 0 1234 95 12768
- 0x) literal starting with zero followed by X ⇒ hexadecimal base (base 16)
- a literal starting with zero followed by a digit is an octal (base 8) number

Ex = decimal: 0 2 63 83
octal = 00 02 077 0123
hexadecimal = 0x0 0x2 0x3f 0x53

a, b, c, d, e, f → 10, 11, 12, ..., 15 respectively.

→ Using these notations to represent genuine numbers can lead to surprises.
Ex: If we represent 65535 as 0xffff on a machine on which an int is represented as a two's complement 16-bit integer: Then 0xffff is the negative decimal number = -1. if it is 32 bit (for example, it would be 65535)

2's complement ⇒ Take complement, add 1 and the number is negative

1 1 1 1 1 1 1 1 ⇒ 255 (unsigned)
-1 ⇒ signed (2's compl.)

Void: This type is syntactically a fundamental type.

- There are no objects of type void. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type.

Ex: `void x;` → error: there are no void objects
`void f();`
`void *pv` → pointer to object of unknown type.

Enumerations: is a type that hold a set of values specified by the user. Once defined, enumeration is used very much like an integer type.

```
enum { ASM, AUTO, BREAK };
       ↓    ↓     ↓
       0    1     2      → Enumerators
```

by default start from 0

- Each enumeration is a distinct type. The type of an enumerator is its enumeration.

Ex: ~~AUTO~~ enum keyword { ASM, AUTO, BREAK };

AUTO is of type keyword.

Declarations :

→ Every name (Identifier) in a C++ program must be declared before its use. That is the type must be specified, to inform the compiler to what kind of entity the name refers.

Ex :
```
char ch;
string s;
int count = 1;
const double pi = 3.1415;
extern int error_number;

char * name = "Hacer";
char * season[] = {"spring", "summer", "fall", "winter"}

struct Date { int d, m, y; };
int day (Date *p) { return p->d; }
double sqrt(double);

typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

- most of these declarations are also definitions.

Ex: char ch ; → memory will be allocated (it's both decl. and defn.)

day → is a specified function

only
```
double sqrt(double);
extern int error_number;
struct User;
```
} not definitions just declarations

- The entity they refer to must be defined somewhere else.

* There must always be exactly one definition for each name in a C++ program. However, there can be many declarations. (should be consistent!)

Ex:
```
int count;
int count; // already defined.
```
- extern int error_code;
- extern short error_code // error inconsistent type mismatch. → decl.

- U : unsigned literals ⇒ 3U (explicit)
- L : long literals

Floating Point Types:

- 3 sizes : float → single precision
  double → double-precision
  long double → (extended precision)
  } exact defn.'s are implementation dependent

Floating Point Literals

- by default floating point literal is of type double.

Ex : 1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15

(1.2e10 and 1.23e-15: no space)

- if you want a floating point literal of type float, use suffix f or F :

Ex : 3.14f 2.0F

Sizes:

- Size of C++ objects are represented as multiples of the size of a char. (so size of char is 1)

- sizeof operator

sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)

sizeof (N) ≡ sizeof (signed N) ≡ sizeof (unsigned N)

* if a value v can be represented exactly in a variable of type T, a conversion of v to T is value-preserving and no problem.
The conversions that are not value preserving are best avoided.

- Any declaration that specifies a value is a definition.
- Type cannot be left out of a declaration.

  Ex: const c=7; → error: no type
  gt(int a, int b) { return (a>b)? a:b;} → error: no return type

- It is possible to declare several names in a single declaration (a list of comma-separated declarators)

  Ex: int a, b; → ≡ int a; int b;

- operators apply to individual names only:

  Ex: int* p, y; ⇒ p: int* y=int (not int*)

  int x, *q;
  int v[10], *pv;
  } less readable ⇒ avoid this usage.

Scope: A declaration introduces a name into a scope; that is a name can be used only in a specific part of the program text.

⇒ { } → defines a block ⇒ hence a scope

→ global name ⇒ defined outside of any function, class or namespace
↳ The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs.

- A declaration in a block can hide a declaration in an enclosing block or a global name. After exiting from a block, the name reaches its previous meaning.

```
int x;                  → global x
void f()
{
    int x;              → local x, hides global x
    x = 1;
    {
        int x;          → hides first local x
        x = 2;          → assign to second local x
    }
    x = 3               → assign to first local x
}
int *p = &x;            → take addr. of global x.
```

* Name hiding should be minimized. (Error prone). Using names such as i and x for global variables for example is asking for trouble.

- A hidden name can be referred to using scope resolution operator ::.

```
int x;
void f2()
{
    int x=1;
    ::x=2      → assign to global x
    x=2;       → assign to local x
    :
}
```

- There's no way to use a hidden local name

Look at this example:

```
int x=11;
void f4()
{
    int y=x;      → global x , y=11
    int x=22;     →
    y=x;          → local x  => y=22
}
```

- function argument names are considered declared in the outermost block of a function:

```
void f5(int x)
{
    int x;        //error : already defined.
}
```

Initialization:

- if a global, namespace or a local static object (collectively called static objects) is initialized to 0, if no explicit initialization is done.

ex: int a; → a=0;
double d; → d=0.0;

- Local variables and dynamic objects (objects created runtime) are not initialized by default.

Ex: void f()
{
int x; ⇒ does not have a well defined value.
:
}

array initializer: int a[] = {1, 2};

Point z(1, 2); ⇒ initialization by constructor

int f(); → function declaration

Objects and Lvalues

- An object is a contiguous region of storage; an lvalue is an expression that refers to an object.

- An lvalue that has not been declared const is often called a modifiable lvalue.

- An object declared in a function is created when its definition is encountered and destroyed when its name goes out of scope. Such objects are called automatic objects

- Objects declared in global or namespace scope and statics declared in functions or classes are created on initialized once (only) and "live" until the program terminates. Such objects are called static objects.

- Array elements and nonstatic structure or class members have their lifetimes determined by the objects of which they are part.

→ using new and delete, you can create and control lifetimes of objects.

Typedef: declares a new name for the type (rather than a new variable declaration)

Ex: typedef char* Pchar;

Pchar p1, p2; ⇒ char* p1, *p2;

Advices:

- Keep scopes small
- Don't use the same name in both a scope and an enclosing scope
- Declare one name (only) per declaration
- Keep common and local names short, keep uncommon and nonlocal names longer
- avoid similar looking names
- maintain a consistent name style
- choose names carefully to reflect meaning rather than implementation
- Use a typedef to define a meaningful name for a built-in type, in case in which the built-in type used to represent a value might change
- Every declaration must specify a type
- Avoid unnecessary assumptions about the numeric value of characters
- Avoid unnec. assumpt. about the size of integers
- Avoid making unnecess. ass. about the sizes of objects
- Be careful about signed to unsigned conversions and vice-versa
- " " " floating point to integer conversions
- " " " conversions from a big to a small type.