Introduction to C++

. cpp file

```
# include<iostream>

int main() {

    std:: cout << "Hello World" << std:: endl;

}
```

1. main() : starting point to execute a C++ program
   return type int, but main() can be left without a return value, which means it returns 0 by default.

2. No semi-colon after # include<iostream>

3. Write std:: to use C++ standard library
   Bad practice: using namespace std;
   cout << "Hello World" ;
   It brings all the identifiers from the std namespace to the global namespace

4. cin , cout
   pay attention to the directions

```
int x , y;
std:: cout << "please enter two integers : ";
std :: cin >> x >> y ;
int sum = x+y;
std:: cout << " x+y is :  " << sum << std:: endl;
```

fundamental data types

Key Word	description	literals / examples	operators
char	8-bit character	'a', 'Q', '+' '\t', '\\', '\n' '\r', '\'', '\"'	+ : This is not Concatenation e.g. 'a' + '+' = 140 ↑ ASCII:97 ↑ ASCII:43
int	integers from -2147483648 to 2147483647	3, 5	+, −, *, / %
long	long integers Linux-32bit 2^{31} Linux-64bit 2^{63}	4294967294	+, −, *, / %
unsigned	non-negative integers	0, 4294967295	
double	double-precision floating numbers point	3.141592 3.1415E5 (3.1415×10^5)	+, −, *, /
bool	boolean values	true, false	!, &&, ||

other data types

float single-precision floating-point
32-bit precision (7 decimal digits of precision)
So when double is not needed, float is faster.

std::string

```
e.g.  std::string  s1 = "Hello";
      std::string  s2 = "World";
      std::string  s3 = s1 + s2;    //string concatenation
```

auto new feature since C++11

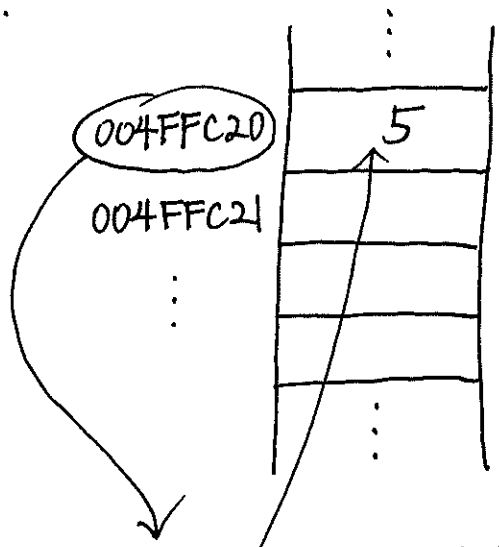
```
e.g.  auto t2 = 256;
        ↑
      t2 is an int variable
      can't assign non-int value to t2

      auto b1 = true;
        ↑
      bool variable
```

* if-else, for loop, for each loop, while loop, !, &&, ||, etc. are the same as Java. You will see more in the future.

* Memory address , pointer

Memory is like a succession of memory cells. Each cell has a unique address. (Just like each house has a unique address on the street)

e.g.




```
int a; // meanwhile, a memory address is assigned to a
a = 5; // we can use variable name to access a memory
          cell and ignore its physical address.
```

* But we can use & operator to find the physical address of a variable.

e.g. `std::cout << &a;`

output: 004FFC20

* A <u>pointer</u> is a variable that stores the memory address.

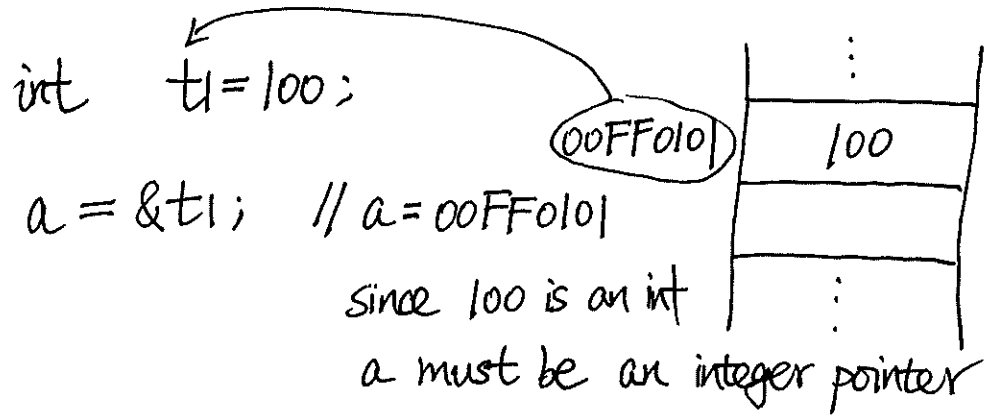
- Syntax to declare a pointer

  Data_Type* variable_name ;

  e.g. int* a ; // a is a pointer (integer pointer)
  a stores an address
  The value in the address is an int

  char* b;

  (I do NOT recommend this: int *a;)



- Dereference Operator

  * operator returns the value of a memory address

  e.g. std::cout << *a ;

  output: 100

  Note: int* a ;
  *a ;
  } two * are different

  int pointer int*

  dereference operator

- More examples

```
std::string  food = "Pizza" ;
std::string* ptr = &food ;
```

food ? ptr ?

&food ? *ptr ?

Good practice:

```
using std::string ;
string food = "Pizza" ;
```

- Let us define two functions

```
void swap (int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
void swap2(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main() {
    int num1 = 5;
    int num2 = 3;
    swap(num1, num2);
```

A copy of num1, num2 are passed into swap(). Swap two copies won't change the origins.

```
    std::cout << num1 << "  " << num2 << std::endl;

    swap2(&num1, &num2);
```

changes are made based on physical addresses.

☆ reference & const

* When a variable is declared as reference, it becomes an alternative name for an existing variable.

* Syntax to declare a reference:

```
Data_Type&  ref_name = var_name;
```

Since a reference is just an alternative name, roughly saying, a reference is coded for the compiler and do <u>Not</u> require memory space.

e.g.
```
int x = 10;
int& ref = x;
ref = 20;
std::cout << x;
x = 30;
std::cout << ref;
```

20

30

Notice: `int& ref = x;` and `&ref` are different.
- `int& ref = x;` ← declare reference
- `&ref` ← find the address of ref (operator)

Why we need references?

1. save memory space

2.
```
void swap3(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
}
```
← references changed then origins also changed.

Any dangers when using references ?

- If we accidentally changed a reference, the original variable's value will also be changed.

- Solution ? (Java : final)

key word : const

```
int c2=10; //need it later
```

e.g. `const int c1=5;` // You must initialize a const var

(also written as `int const c1=5;`) // `c1=6;` is illegal

① `const int* cp1=&c1;` // cp1 is a pointer.

(also written as `int const* cp1=&c1;`) // cp1 stores the memory address of c1.

// `*cp1=6;` is illegal.

But `cp1=&c2;` // This is legal. Address can be reassigned.

②

```
int* const cp2=&c1; // cp2=&c2; is illegal
```

But `*cp2=6;` // This is legal.

③ `const int* const cp3=&c1;`

The ultimate const

try these examples yourself.

C++ Array

Definition: A collection of fixed number of elements of the same type.

e.g. `double f[3];` // an array of 3 double numbers `f[0]`, `f[1]`, `f[2]`

`double* p[3];` // an array of 3 pointers `p[0]`, `p[1]`, `p[2]`

```
f[2] = 21.4;
p[0] = &f[2];   // What is the & here?
std::cout << *p[0];   // What is the *? What is the output?
int w[] = {10, 11, 12, 13, 14};
char c[] = {'c', 'a', 't'};
int k=3;
```

! → `double points[k];` is illegal

★★★ Array & Pointer: The name of an array can be used as a pointer to the array's first element, and vice versa.

★★★ This means when you declare a pointer, it is also an array.

(Why?)

e.g. `char c[] = {'c', 'a', 't'};`

`char* p2 = c;` // c is a pointer to c[0]

What is *c ?

What is p2[2] ?

- You can also declare an array like this :

```
int K = 3;
int* p3 = new int[K]; // Note, int p3[K]; is illegal
p3[0] = 5;
p3[1] = 6;
p3[2] = 12;
```

- Use <u>delete</u> to release memory space

  for array objects, use `delete[]`

  e.g. `delete[] p3;`

  for non-array objects, use `delete`

- If an object is allocated with `new[]`,
  it should be deallocated with `delete[]`

Expressions

1. `int K = 3;`

`K++;` `//int i=K++;` `i=3` `K=4` post-increment

`++K;` `//int i = ++K;` `i=4` `K=4` ← more efficient pre-increment

2. `int i=10;`
`int j=5;` `// int i=10, j=5;`

`K=1;`

`std::string S="Yes";`

`i-=4;` `// i=i-4;` `i=6`

`j*=-2;` `// j=j*(-2);` `j=-10`

`K<<=1;` `// K=K<<1;` shift K to the left by 1.

↑
This is a bitwise operator.

$\underline{10}_{\text{base 2}} = 2$

`S+="or No";` `// S=S+" or No";` `S="Yes or No"`

`K<<2;` `//shift K to the left by 2`

$1000|_2 = 8$

3. converting types

```
int t = 14;
double d1 = (double) t;   // old C-style cast
double d2 = double (t);   // C++ functional cast
                  ↑
              parameter

int t1 = 18, t2 = 16;
double dv1 = t1 / t2 ;        // 1.0
double dv2 = double (t1) / double (t2);   // 1.125
double dv3 = double (t1/t2) ;  // 1.0
```

header files and classes

In Java, each .java file is a class. But for C++, it is not the same. Suppose you have a class person in a single .cpp file named A.cpp. To use a function in the class, the compiler needs to have seen the declaration of the function.

However, you want to reuse the class person in another .cpp file name B.cpp. Then you need to copy-paste class person in B.cpp. Then, there is duplication. A and B can be compiled separately. But if you compile A and B together, there will be a huge problem.

So to prevent code duplication and facilitate code reuse, we want to write the class person in a header file (.h).

Then A.cpp and B.cpp can both include the header file.

Syntax: `#include "person.h"`

Normally, there is a pair: person.h and person.cpp.