Higher Level Languages

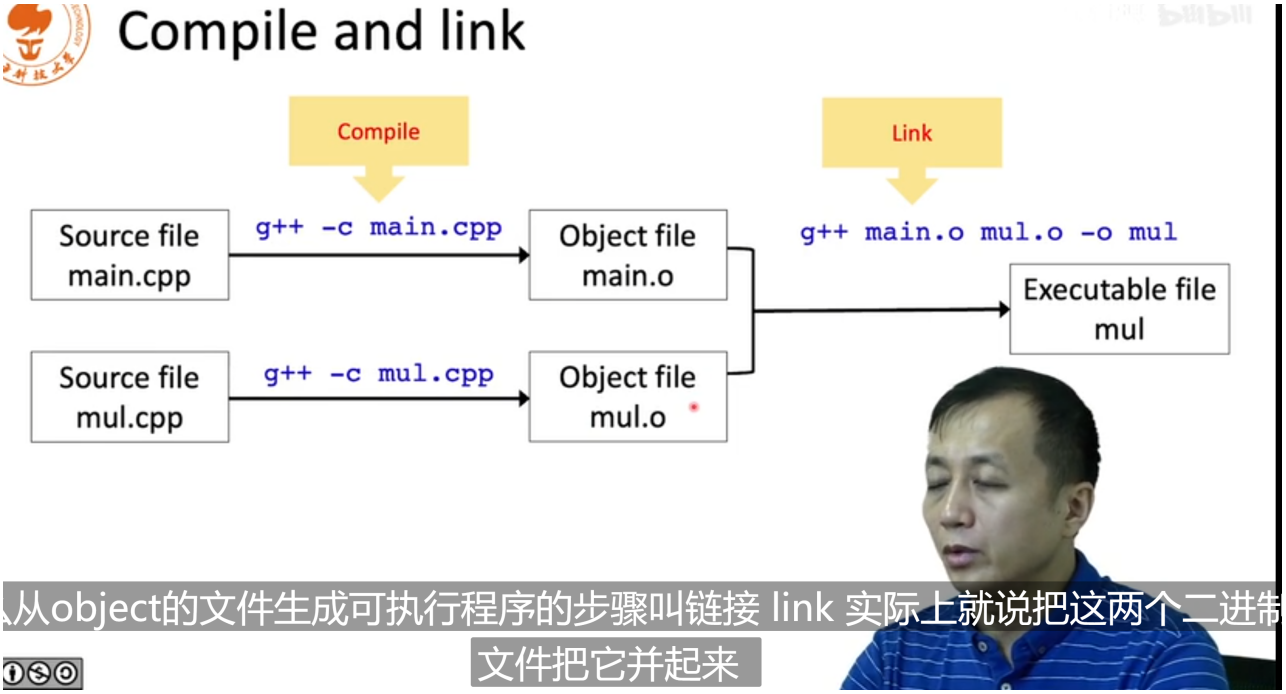
- Java: 1995
  - I hate memory management in C/C++!
  - I want "Write once, run anywhere", not "write once, compile anywhere".
  - Grammar is similar with C++.
  - A Java compiler generates *.class files, not executable files.

就编译成一个字解码文件 中间文件 然后在所有的平台上运行 当然

Java不用程序员申请内存，释放内存了

Advantages of C/C++

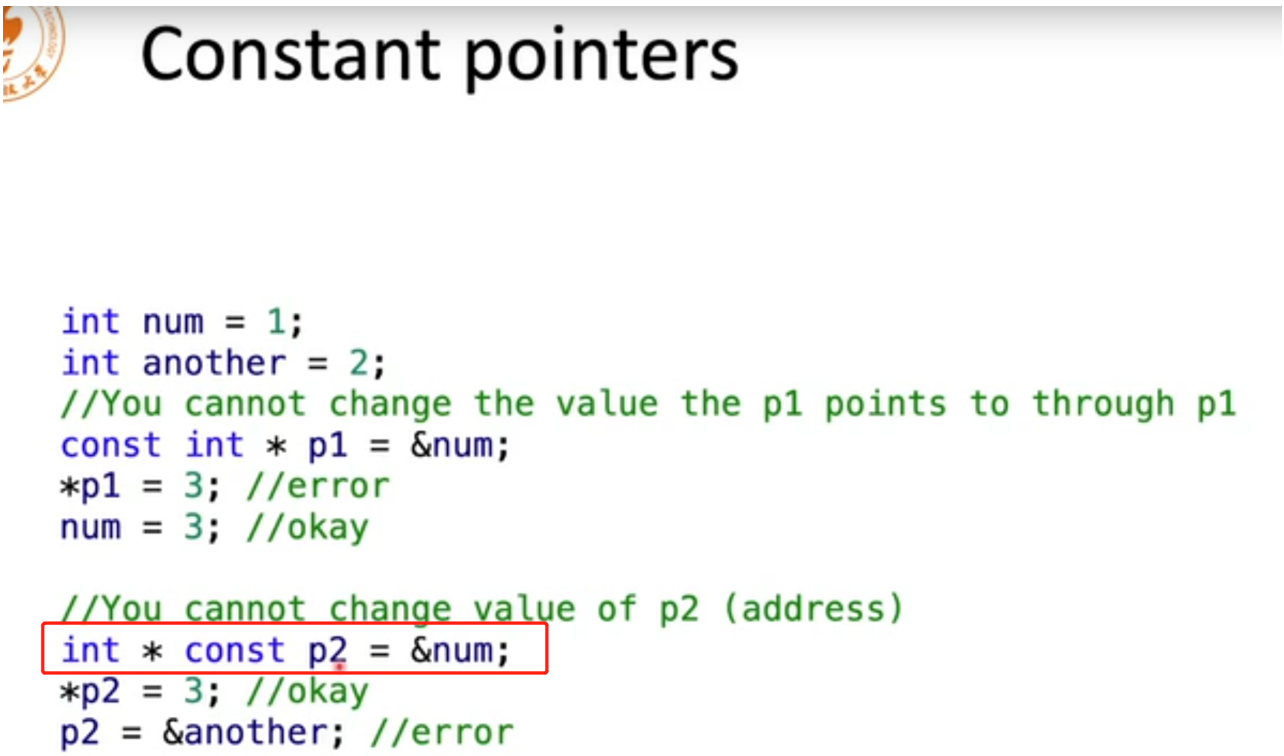
- Development language of most fundamental computer systems
  - Linux
  - MySQL
  - OpenCV
  - Backend of TensorFlow, PyTorch
  - ...
- High efficiency
  - Widely optimized compilers
  - Access memory directly
  - Excellent on computing
  - Important language for AI algorithm implementation



-c：只编译，不链接

g++ main.cpp mul.cpp -o mul 可直接从源码编译得到可执行文件

三大类错误：编译错误、链接时错误、运行时错误



红框处让 p2 始终指向 num 的地址

const int* const p3 = #

两处 const 使既不可以修改指针指向的内容，也不可以修改指针本身

64 / 32 bit 下，sizeof 分别返回 unsigned long long 和 unsigned int 类型，占 8 / 4 字节。

指针类型占用的内存大小是固定的（无论该指针指向哪种数据类型），在 32 位系统中为 4 字节；在 64 位系统中为 8 字节

Memory deallocation

- The dynamically allocated memory must be deallocated explicitly!

  void free(void* ptr) ;

- Question:

```
p = (int *) malloc(4 * sizeof(int));
// ...
p = (int *) malloc(8 * sizeof(int));
// ...
free (p);
```

```
void foo()
{
    int* p = (int *) malloc( sizeof(int));
    return;
} //memory leak
```

申请过的内存没有释放，如果变量作用域消失就再也不能释放那块内存了！

Operator new and new[ ]

- Operator `new` is similar with `malloc()` but with more features.

```
//allocate an int, default initializer (do nothing)
int * p1 = new int;
//allocate an int, initialized to 0
int * p2 = new int();
//allocate an int, initialized to 5
int * p3 = new int(5);
//allocate an int, initialized to 0
int * p4 = new int{};//C++11
//allocate an int, initialized to 5
int * p5 = new int {5};//C++11

//allocate a Student object, default initializer
Student * ps1 = new Student;
//allocate a Student object, initialize the members
Student * ps2 = new Student {"Yu", 2020, 1}; //C++11
```

Operator new and new[ ]

- Operator new is similar with malloc() but with more features.

```
//allocate 16 int, default initializer (do nothing)
int * pa1 = new int[16];
//allocate 16 int, zero initialized
int * pa2 = new int[16]();
//allocate 16 int, zero initialized
int * pa3 = new int[16]{}; //C++11
//allocate 16 int, the first 3 element are initialized to 1,2,3, the rest 0
int * pa4 = new int[16]{1,2,3}; //C++11

//allocate memory for 16 Student objects, default initializ
Student * psa1 = new Student[16];
//allocate memory for 16 Student objects, the first two are      ly
initialized
Student * psa2 = new Student[16]{{"Li", 2000,1}, {"Yu", 2001                C++11
```

- Destroys object/objects allocated by new and free memory

```
//deallocate memory
delete p1;
//deallocate memory
delete ps1;

//deallocate the memory of the array
delete pa1;
//deallocate the memory of the array
delete []pa2;

//deallocate the memory of the array, and
call the destructor of the first element
delete psa1;
//deallocate the memory of the array, and
call the destructors of all the elements
delete []psa2;
```

所以写 C++ 代码需要释放数组时，一律写 `delete []arr;`

Where should a function be? Option 3

```
// draw.h
#ifndef __DRAW_H__
#define __DRAW_H__
bool drawLine(int x1, int y1, int x2, int y2);
bool drawRectangle(int x1, int y1, int x2, int y2);
#endif
```

```
// draw.cpp
#include <draw.h>

bool drawRectangle(int x1, int y1, int x2, int y2)
{
    // some calculation here
    drawLine(...);
    drawLine(...);
    drawLine(...);
    drawLine(...);

    return true;
}
// define it later
bool drawLine(int x1, int y1, int x2, int y2)
```

```
// main.cpp
#include <draw.h>

int main()
{
    // ...
    drawRectangle(10,
    // ...
}
```

How are functions called?

- A call stack can store information about the active functions of a program
  - Store the address the program returns after the function call
  - Store the registers
  - Store the local variables
  - //do some work of the called function
  - Restore the registers
  - Restore the local variables
  - Store the function returned result
  - Jump to the return address

The cost to call a function!

Pass by value: the param is a copy of the original variable

if param is a pointer, it's still passing by value (a copy of the address)

- Pass by value:
  - Fundamental types: the value of a constant/variable is copied
  - Pointers: the address is copied
  - Structures: the whole structure is copied

A reference is an alias to an already-existing variable/object.

A reference must be initialized after its declaration.

No data copying in the reference version; Better efficiency (can be chosen if param is a struct)