南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

CS205

Week 7

Prof. Shiqi Yu (于仕琪)
<yusq@sustech.edu.cn>
Prof. Feng (郑锋)
<zhengf@sustech.edu.cn>

The slides are based on the book <Stephen Prata, C++ Primer Plus, 6th Edition, Addison-Wesley Professional, 2011>

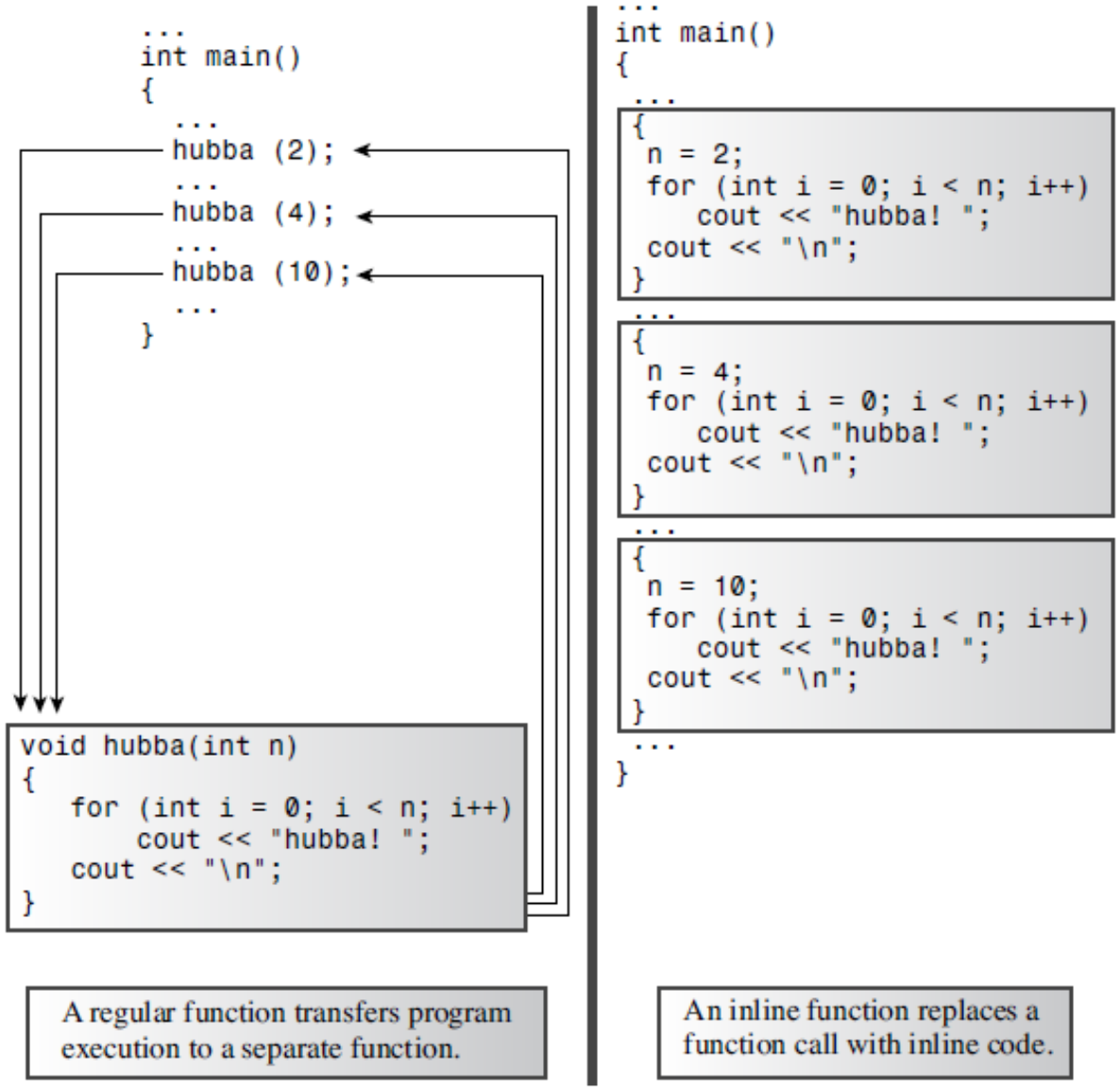
Adventures in Functions

Mechanism of Function Call

- Codes are the data as well
  - The product of the compilation process is an executable program
    - ✓ Consist of a set of machine language instructions
  - The operating system loads these instructions into the memory
    - ✓ Each instruction has a particular memory address
    - ✓ Jump backward or forward to a particular address (loop or branching)
- Normal function: Jump forth and back
  - Store the memory address of the instruction immediately following the function call
  - Copy function arguments to the stack
  - Jump to the memory location that marks the beginning of the function
  - Execute the function code
  - Jump back to the instruction whose address it saved

C++ Inline Functions

- Compiler replaces the function call with the corresponding function code
  - Run a little faster than regular functions
  - Come with a memory penalty
  - Be selective about using inline functions
- Two steps
  - Preface the declaration with the keyword inline.
  - Preface the function definition with the keyword inline
- Inline versus macros
  - Macros don't pass by value



github.com/opencv/opencv/search?q=inline

inline / Pull requests Issues Marketplace Explore

- Code 735
- Commits 52
- Issues 559
- Discussions Beta 0
- Packages 0
- Wikis 9

Languages

C++	562
C	124
OpenCL	20
CMake	7
Python	4
CSS	1

735 code results in opencv/opencv or view all results on GitHub

Sc

3rdparty/carotene/src/vtransform.hpp

```
94  template <> struct VecTraits<f32, 4> { typedef float32x4x4_t vec128; typede
    float32x2x4_t vec64; typedef VecTraits< u32, 3> unsign; };
95
96  ////////////////////////////// vld1q ////////////////////////
97
98  inline  uint8x16_t vld1q(const u8  * ptr) { return  vld1q_u8(ptr); }
99  inline   int8x16_t vld1q(const s8  * ptr) { return  vld1q_s8(ptr); }
100 inline  uint16x8_t vld1q(const u16 * ptr) { return vld1q_u16(ptr); }
```

C++ Showing the top three matches Last indexed on 26 Jun 2018

modules/imgproc/src/fixedpoint.inl.hpp

```
18      fixedpoint64(int64_t _val) : val(_val) {}
19      static CV_ALWAYS_INLINE uint64_t fixedround(const uint64_t& _val) { ret
    ((1LL << fixedShift) >> 1)); }
...
24      typedef int64_t raw_t;
25      CV_ALWAYS_INLINE fixedpoint64() { val = 0; }
26      CV_ALWAYS_INLINE fixedpoint64(const fixedpoint64& v) { val = v.val; }
```

Default Arguments

- How do you establish a default value?
  - You must use the function prototype
- When you use a function with an argument list, you must add defaults from right to left

```
int harpo(int n, int m = 4, int j = 5);          // VALID
int chico(int n, int m = 6, int j);              // INVALID
int groucho(int k = 1, int m = 2, int n = 3);    // VALID
```

- The actual arguments are assigned to the corresponding formal arguments from left to right; you can't skip over arguments
- Run program example left.cpp

Function Overloading(重载)

- Function overloading:
  - Let you use **multiple** functions sharing the **same** name
  - Comparison: in default argument, it can call the **same** function by using varying numbers of arguments
- C++ enables you to define two functions by the same name, provided that the functions have **different** signatures
  - Function signature: function's argument list
  - Signature can differ in the number of arguments or in the type of arguments, or both
- Run program example leftover.cpp


```
#include <cmath>

float       round ( float arg );
double      round ( double arg );
long double round ( long double arg );
```

Function Templates

- Define a function in terms of a generic type
  - A specific type, such as int or double, can be substituted
  - The process is termed generic programming
  - The keywords **template** and **typename** (class)

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Run program example funtemp.cpp
  - Apply the same algorithm to a variety of types

More Templates

- Overloaded templates
  - Solved problem: not all types would use the same algorithm in templates
  - Non-template functions take precedence over template functions
  - Need distinct function **signatures** (overloading)
  - Run program example twotemps.cpp

- Template limitations
  - It's easy to write a template function that cannot handle certain types
  - In some cases, require **different** codes but the arguments would be the same

Specializations (特化)

- Explicit specializations
  - If the compiler finds a specialized definition that exactly matches a function call, it uses that definition without looking for templates.
- Third-Generation Specialization
  - A function name can have a non template function, a template function, and an explicit specialization template function, along with all overloaded versions
  - The prototype and definition for an explicit specialization should be preceded by template <> and should mention the specialized type by name
  - A specialization overrides the regular template, and a non template function overrides both
  - Run program example twoswap.cpp

Instantiations(实例化) and Specializations(特化)

- Instantiation
  - Template: merely a plan for generating a function definition
  - Instantiation: use the template to generate a function definition
    - ✓ Implicit: the compiler deduces the necessity for making the definition
    - ✓ Explicit: using the <> notation to indicate the type and prefixing the declaration with the keyword template

```
...
template <class T>
void Swap (T &, T &);  // template prototype
template void Swap<char>(char &, char &); // explicit instantiation for char
template <> void Swap<job>(job &, job &);   // explicit specialization for job
int main(void)
{

  short a, b;
  ...
  Swap(a,b);    // implicit template instantiation for short
  job n, m;
  ...
  Swap(n, m);   // use explicit specialization for job
  char g, h;
  ...
  Swap(g, h);  // use explicit template instantiation for char
  ...
}
```

Speedup your program

Principle for programming

Simple is Beautiful !

Short

Simple

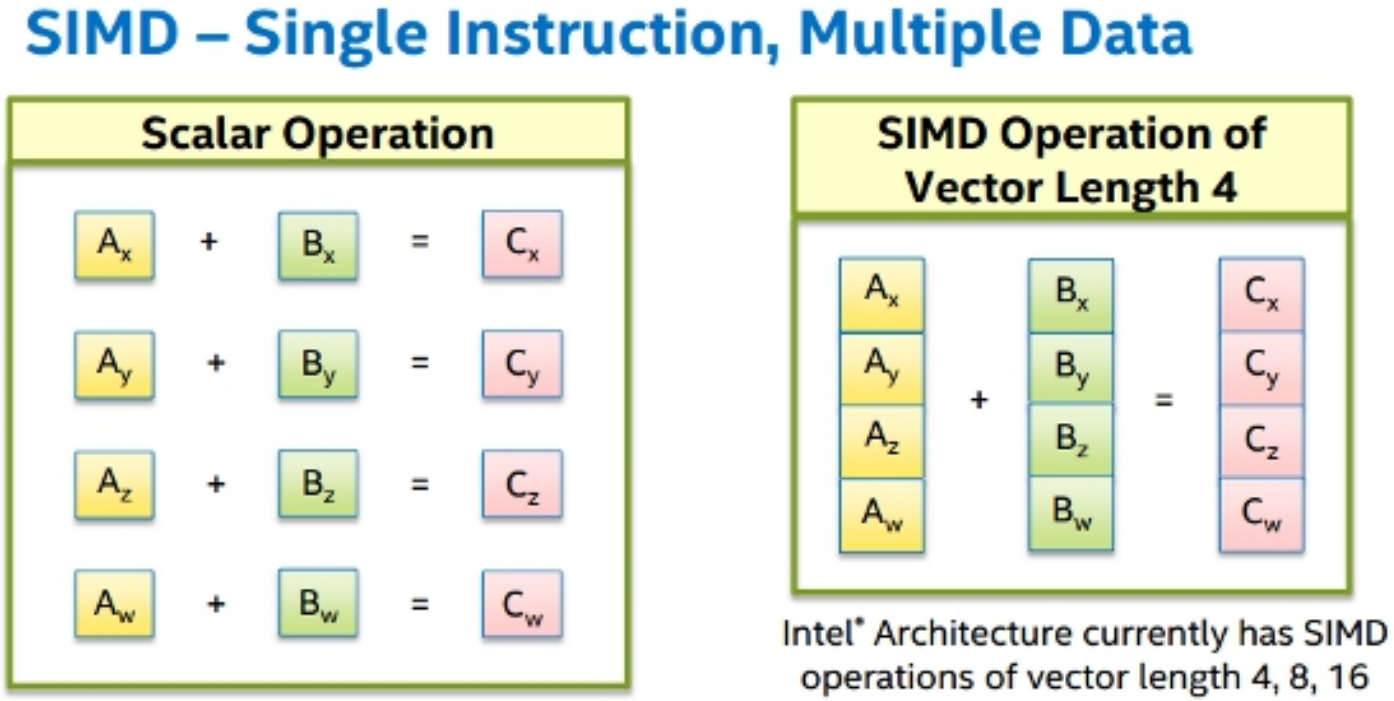
Efficient

An example: libfacedetection

- Face detection and facial landmark detection in **1700 lines** of source code
  - `facedetectcnn.h` :
    - ✓ 400 lines
    - ✓ CNN APIs
  - `facedetectcnn.cpp`:
    - ✓ 900 lines
    - ✓ CNN function definitions
  - `facedetectcnn-model.cpp`:
    - ✓ 400 lines
    - ✓ Face detection model
  - `facedetectcnn-int8data.cpp`
    - ✓ CNN model parameters in static variables

https://github.com/ShiqiYu/libfacedetection/

SIMD: Single instruction, multiple data



- Intel: MMX, SSE, SSE2, AVX, AVX2, AVX512
- ARM: NEON
- RISC-V: RVV(RISC-V Vector Extension)

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

SIMD in OpenCV

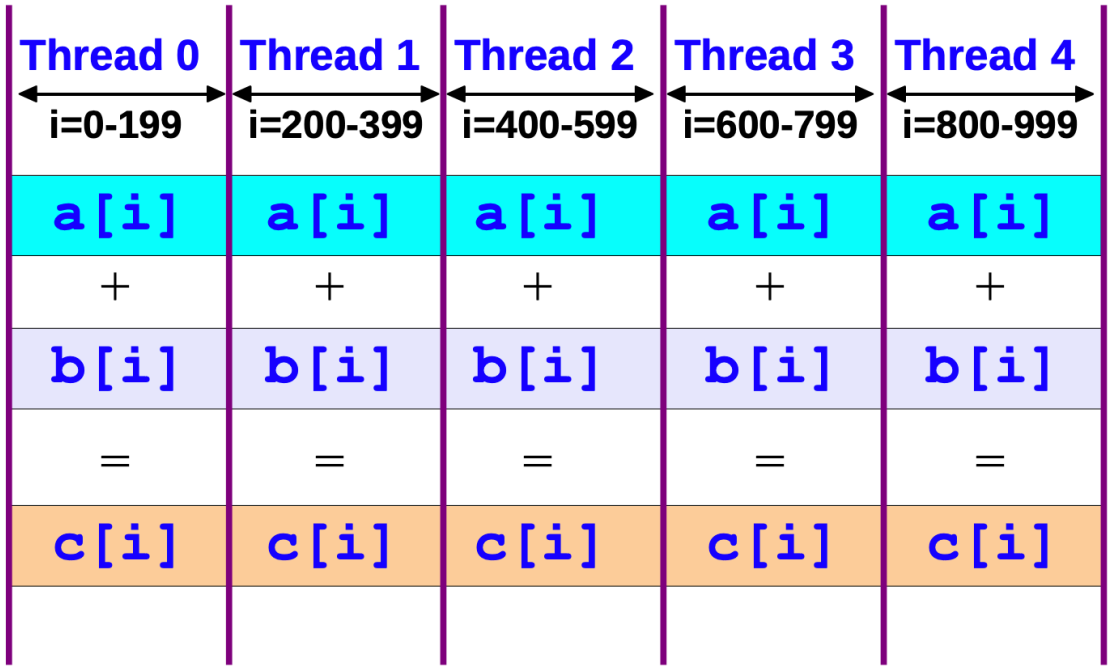
- "Universal intrinsics" is a types and functions set intended to simplify vectorization of code on different platforms.

- https://docs.opencv.org/master/df/d91/group__core__hal__intrin.html

- 使用OpenCV中的universal intrinsics为算法提速(1)(2)(3)
  - https://mp.weixin.qq.com/s/_dFQ9lDu-qjd8AaiCxYjcQ
  - https://mp.weixin.qq.com/s/3UmDImwlQwGX50b1hvz_Zw
  - https://mp.weixin.qq.com/s/XtV2ZUwDq8sZ8HlzGDRaWA

An Overview of OpenMP – Ruud van der Pas – Sun Microsystems https://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf

OpenMP




```
#include <omp.h>


#pragma omp parallel for
for (size_t i = 0; i < n; i++)
{
   c[i] = a[i] + b[i];
}
```

OpenMP

Where should #pragma be? The 1st loop or the 2nd?

```
#include <omp.h>

#pragma omp parallel for
for (size_t i = 0; i < n; i++)
{
    //#pragma omp parallel for
    for (size_t j = 0; j < n; j++)
    {
        //...
    }
}
```