Lecture #04

Coding Style Guide, Basic Computer Architecture

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides from Prof. Shin at DGIST

Short Notice

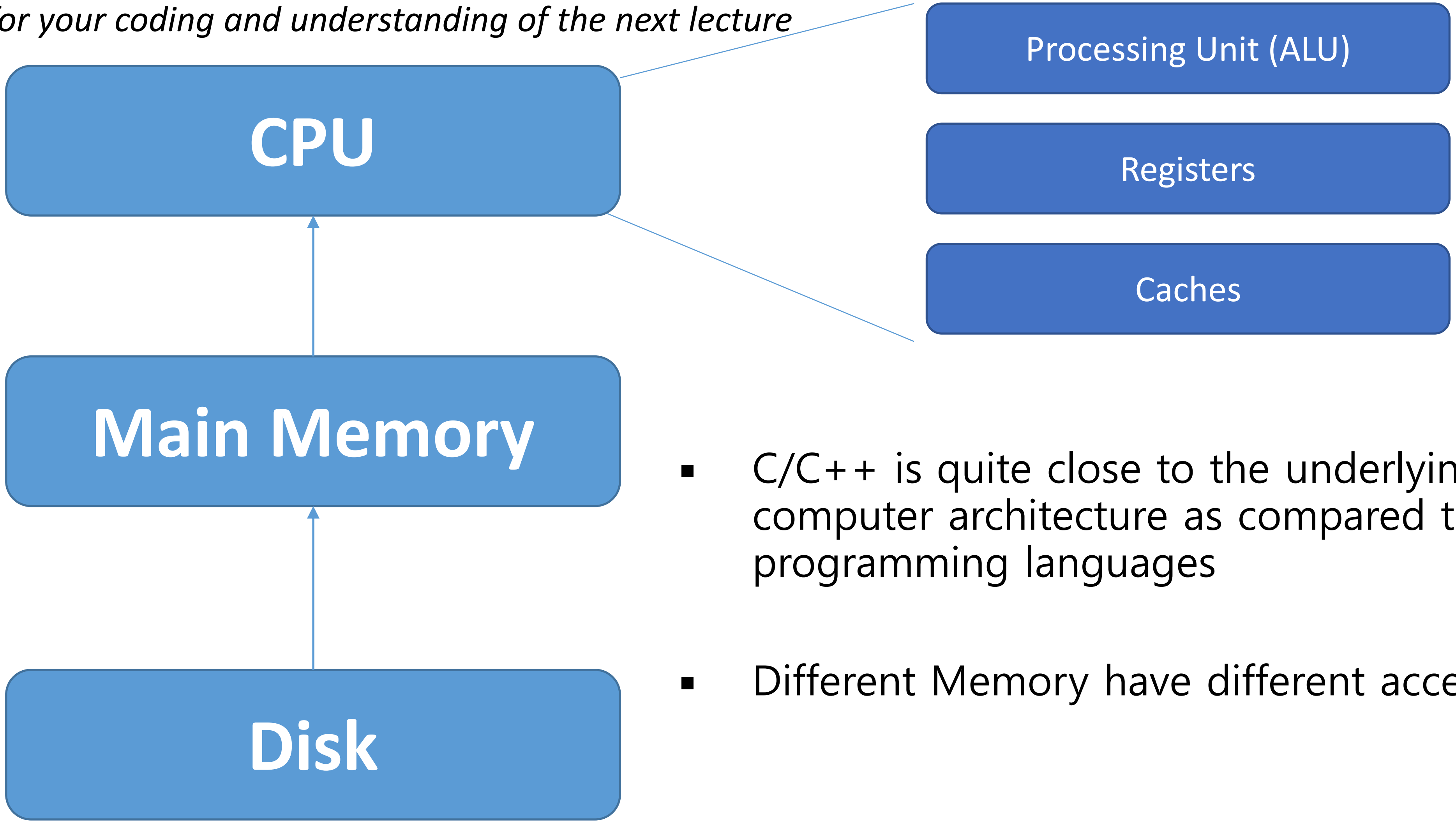
- This lecture is provided with a recorded video
- The first homework will be released next week, on Monday

Today’s Topic

- Basic Computer Architecture
  - For your understanding of the next lecture “Array and Pointer”
- Coding Style
  - Google Style Guide
  - Practice with your first homework

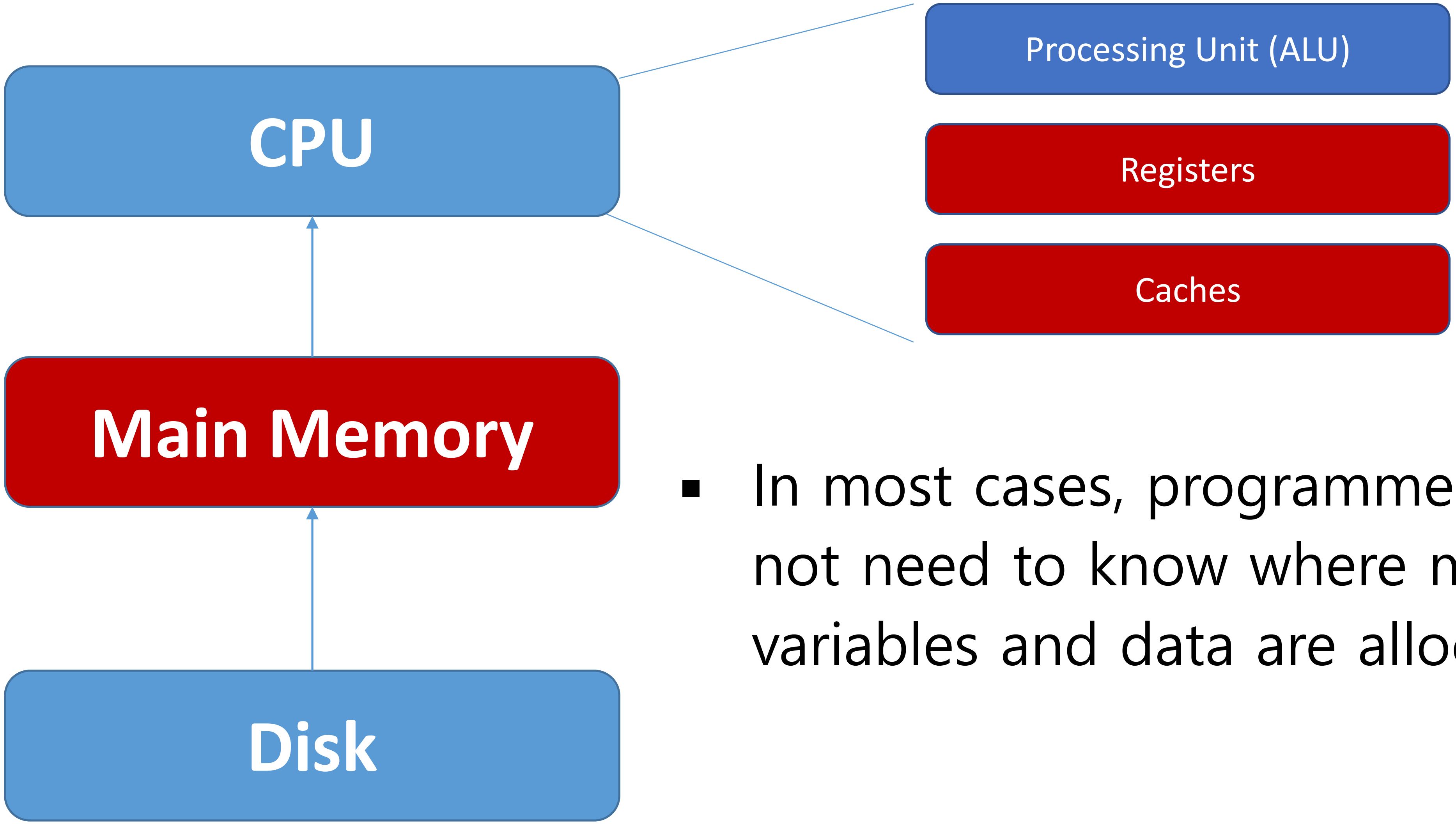
Basic Computer Architecture

Not to be included in your exam/homework; but I will promise:
it will be useful for your coding and understanding of the next lecture



- C/C++ is quite close to the underlying computer architecture as compared to other programming languages
- Different Memory have different access time

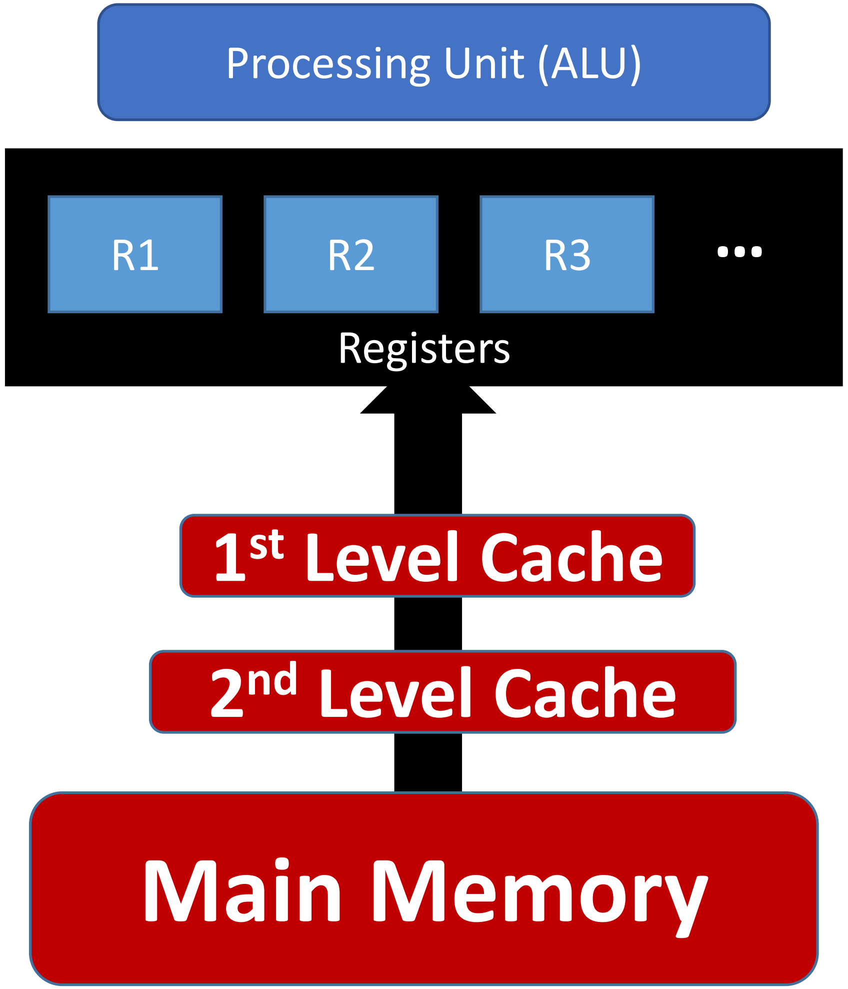
Compiler Decide Where to Use



- In most cases, programmers do not need to know where my variables and data are allocated

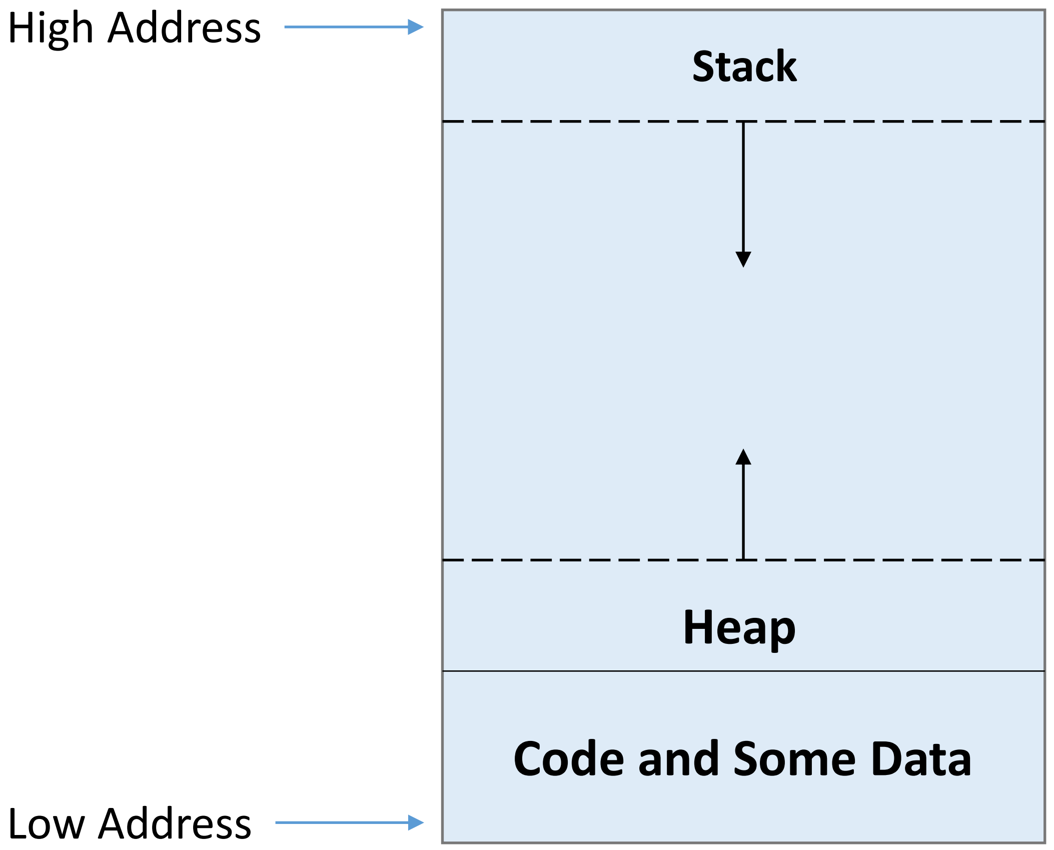
But You Should Know Boxes and a Tape

- Processor computes numbers stored in registers (boxes)
- If variables are stored in the cache/memory (a tape), it should be loaded first
  - e.g., due to insufficient boxes
- Memory is a tape, some of them are cached in caches



Stack, Heap, and Address

- Memory is a tape, and computer saves changeable data in two directions
  - Access the bytes in the tape using “address”



Stack stores the data that
we may know at the *compile-time*

Heap stores the data that
we may know while the program is running

Remember at least this:
Your data and variable has its own address!

Why we should consider coding style

- Functioning != Readability
  - You are not alone in the world
- Error-free code
  - e.g., avoid including duplicated header files
- Maintainability
  - Help others for collaboration
  - cf. Refactoring
- Popular coding guides exist
  - We will discuss Google's C++ coding guide

Good Code and Bad Code

- What makes the difference?

```
#include <iostream>
using namespace std;
void swap(int a, int b)
{
int temp{a};
a = b;
b = temp;
cout << a << “ “ << b << endl;
}
```

```
#include <iostream>
void swap_int(int num1, int num2) {
    int temp{num1};
    num1 = num2;
    num2 = temp;
    std::cout << num1 << “ “ << num2 << std::endl;
}
```

- Google Style Guide:
  - https://google.github.io/styleguide/cppguide.html

Naming

- The most important consistency rules are those that govern naming
  - Immediately inform what sort of thing the named entity is
    - A type, a variable, a function, a constant, a macro, etc.
- To avoid requiring search for the declaration of that entity
  - The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences

Filename

- Filenames should be all lowercase and can include underscores (_) or dashes (-).
  - Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "_".

```
myusefulclass.cc
my-useful-class.cc
my_useful_class.cc
```

- Do not use filenames that are too common, e.g., already exist in /usr/include, such as db.h.
- Make your filenames very specific.
  - e.g., use http_server_logs.h rather than logs.h.
  - A very common case is to have a pair of files called, e.g., foo_bar.h and foo_bar.cc, defining a class called FooBar.

Type and Variable Name – Most Important & Frequent

- Type names start with a capital letter and have a capital letter for each new word, with no underscores (CamelCase)
  - e.g., classes, structs, type aliases, enums, and type template parameters

```
// classes and structs
class UrlTable { …
class UrlTableTester { …
struct UrlTableProperties { …
```

- The names of variables (including function parameters) and data members are all lowercase, with underscores between words.

```
std::string table_name;   // OK - lowercase with underscore
std::string tableName;   // Bad - mixed case
```

Hungarian Notation – Mostly Deprecated

- An identifier naming convention in computer programming
  - Charles Simonyi suggested at MS
- The name of a variable or function indicates its intention or kind: **its type.**

> lAccountNum : variable is a long integer ("l");
>
> arru8NumberList : variable is an array of unsigned 8-bit integers ("arru8");
>
> bReadLine(bPort,&arru8NumberList) : function with a byte-value return code.
>
> strName : Variable represents a string ("str") containing the name, but does not specify how that string is implemented.
>
> * Source: wikipedia

- Mostly deprecated:
  - Turns out it's rather hard to understand code immediately
  - Hard to remember the variable name
  - When the data type changes, should change the variable name
    - It's difficult – without good IDE
  - Modern IDE already supports showing the type of variables!

Header Guard & Forward Declarations

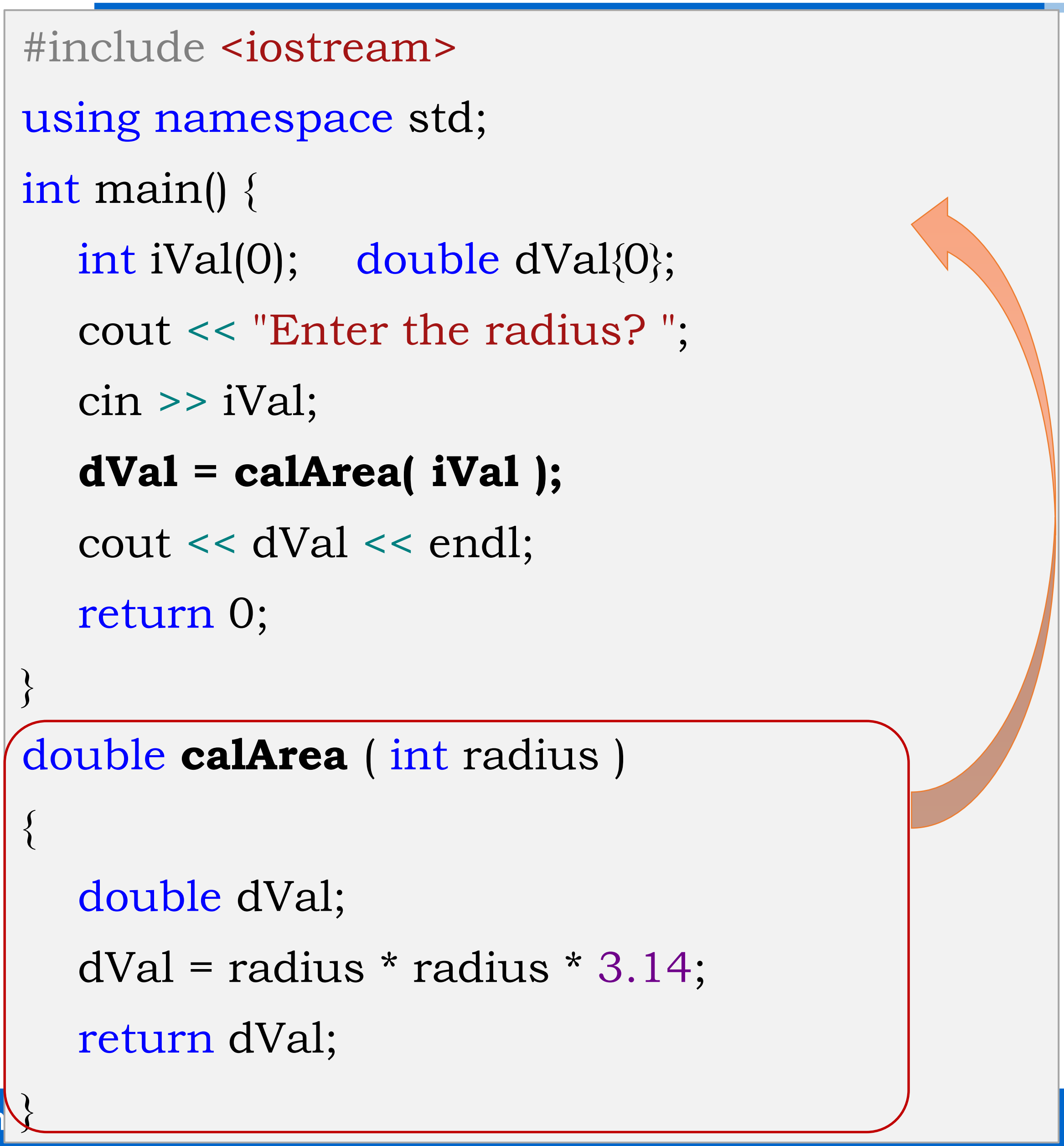
- Use #define guard instead of #pragma once

```
#ifndef THIS_HEADER_FILE_NAME_H_
#define THIS_HEADER_FILE_NAME_H_

…  header body …

#endif
```

```
#pragma once

…  header body …
```

  – Note: Pragma once is complier-dependent

Use #include instead of forward declarations



```
#include <iostream>
using namespace std;
int main() {
    int iVal(0);    double dVal{0};
    cout << "Enter the radius? ";
    cin >> iVal;
    dVal = calArea( iVal );
    cout << dVal << endl;
    return 0;
}
double calArea ( int radius )
{
    double dVal;
    dVal = radius * radius * 3.14;
    return dVal;
}
```

```
#include <iostream>
using namespace std;
double calArea ( int radius ) ;
int main() {
    int iVal(0);    double dVal{0};
    cout << "Enter the radius? ";
    cin >> iVal;
    dVal = calArea( iVal );
    cout << dVal << endl;
    return 0;
}
double calArea ( int radius )
{
    double dVal;
    dVal = radius * radius * 3.14;
    return dVal;
}
```

Header File Order

- For example, In dir/foo.cc, whose main purpose is to implement or test the stuff in dir2/foo.h, order your includes as follows:

- dir2/foo.h.
- A blank line
- C system headers (more precisely: headers in angle brackets with the .h extension), e.g., <unistd.h>, <stdlib.h>.
- A blank line
- C++ standard library headers (without file extension), e.g., <algorithm>, <cstddef>.
- A blank line
- Other libraries' .h files.
- Your project's .h files.

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>

#include <string>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

Function

- Prefer small and focused functions.
  - If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.
- Define functions inline only when they are small, say, 10 lines or fewer.
  - Inlining a function can generate more efficient object code, as long as the inlined function is small.

```cpp
#include <iostream>

// function declaration
inline int Multiply ( int , int  = 1);

int main() {
    std::cout << Multiply (10) ;
    std::cout << Multiply (10, 20) ;
    return 0;
}

// function definition
int Multiply ( int  iNum1, int  iNum2) {
    return iNum1 * iNum2;
}
```

Comments

- Comments are absolutely vital to keeping our code readable
  - Use either the // or /* */ syntax, as long as you are consistent.
  - You can use either the // or the /* */ syntax; however, // is much more common.

- Header file
  - Start each file with license boilerplate (if any)
  - If a .h declares multiple abstractions, the file-level comment should broadly describe the contents of the file

Function Comments

- Declaration comments describe use of the function
  - When it is non-obvious
  - Comments at the definition of a function describe operation.
- Describe what the inputs and outputs are.
  - For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.
  - If the function allocates memory that the caller must free.
  - Whether any of the arguments can be a null pointer.

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

Implementation Comments

- In your implementation you should have comments in *tricky, non-obvious, interesting, or important* parts of your code.
  - Tricky or complicated code blocks should have comments before them.

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); ++i) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

- Lines that are non-obvious should get a comment at the end of the line.
- However, do not state the obvious.

```
++counter; // increase the counter
```

Code Lint

- Google's cpplint tool
  - Developed with Python
  - Not inclusive
  - Let's go back to the example

- Use Visual Studio for simple indentation
  - Practice with its shortcut: Ctrl+K, Ctrl+F

ANY QUESTIONS?