C/C++ Programming Language

CS205 Spring

Feng Zheng

Weak 1

南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

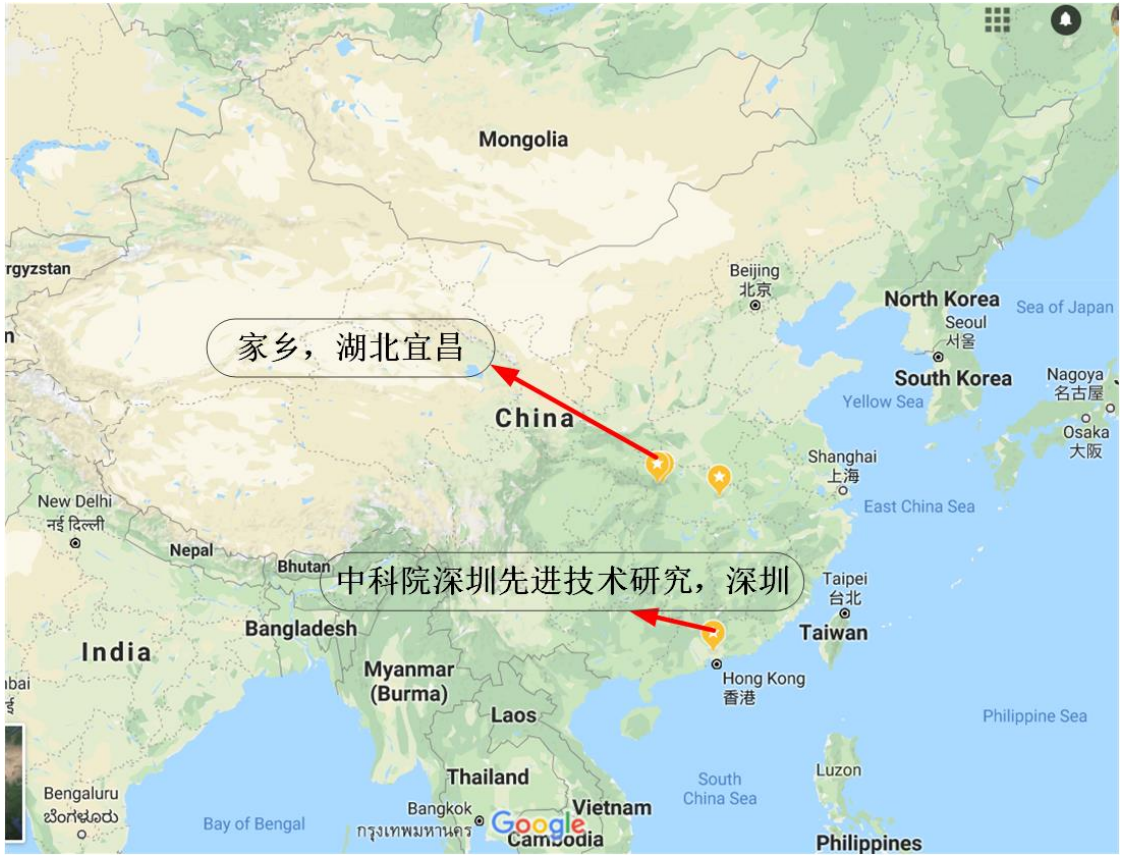
Content

- Brief Biography
- About This Course
- Getting Started with C++
- Setting Out to C++

Brief Biography

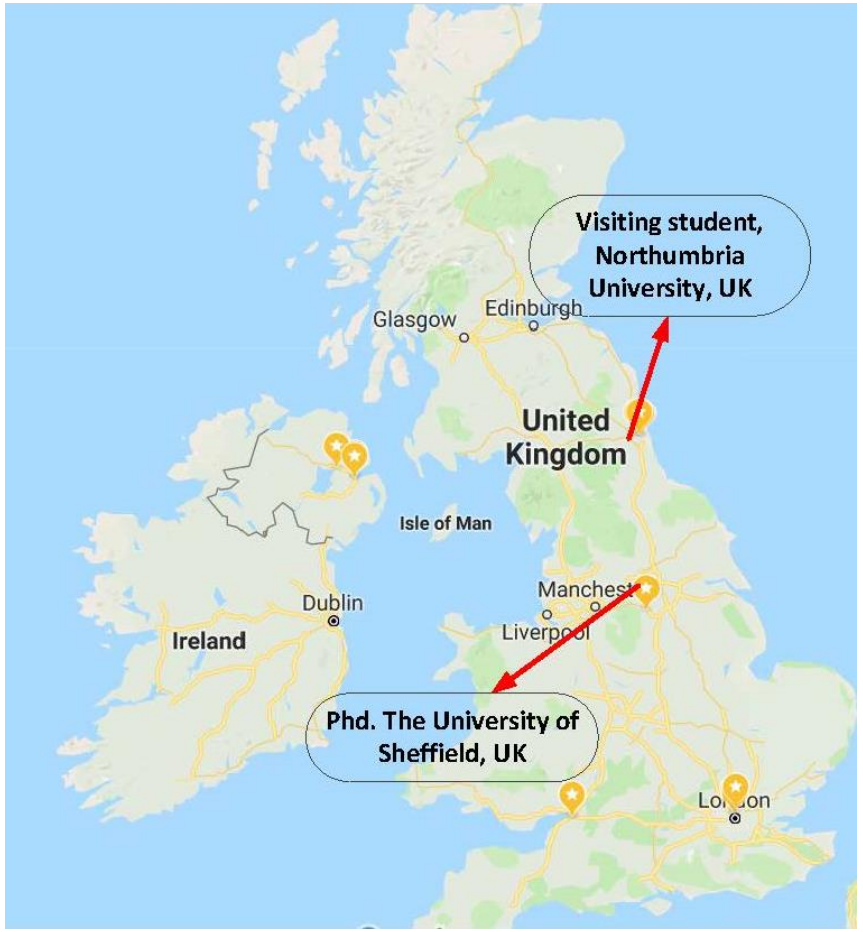
China

- Shenzhen Institutes of Advanced Technology (SIAT), CAS, Jul. 2009 - Sep. 2012



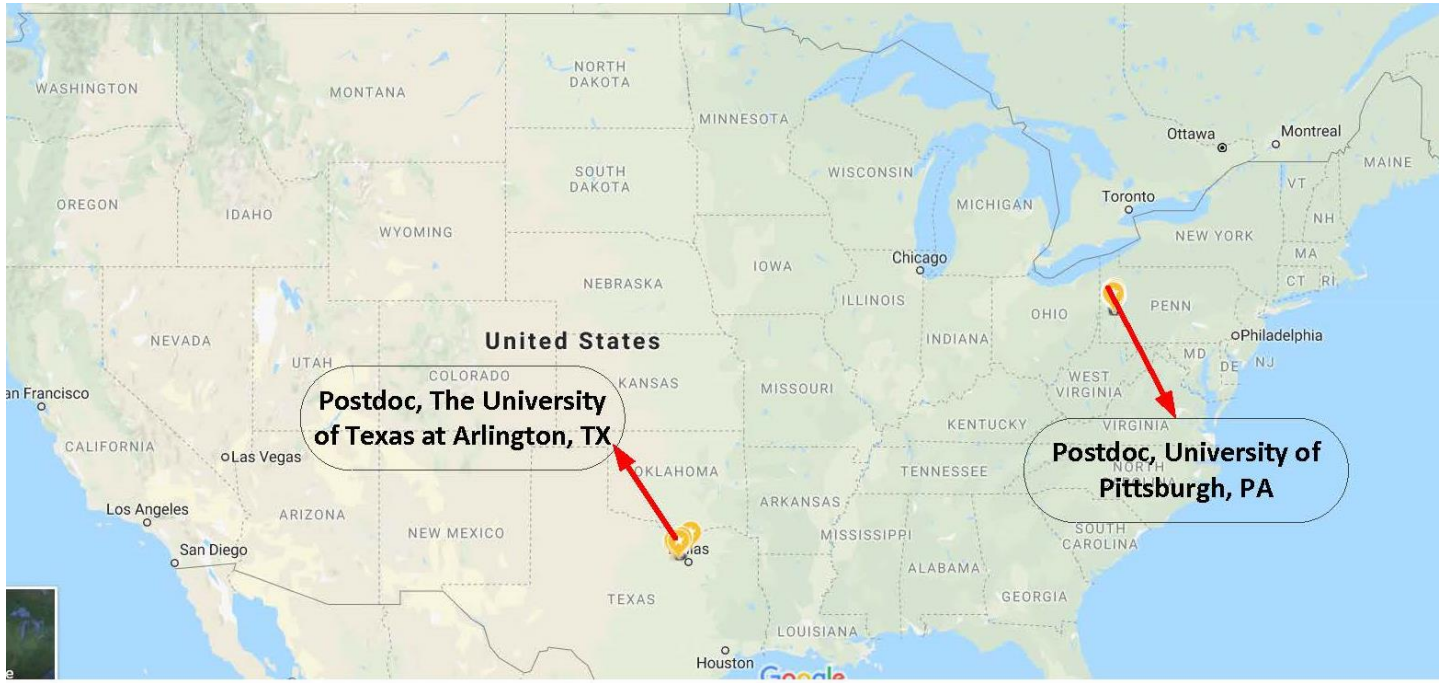
United Kingdom

- The University of Sheffield, UK, Oct. 2012 - Otc. 2016



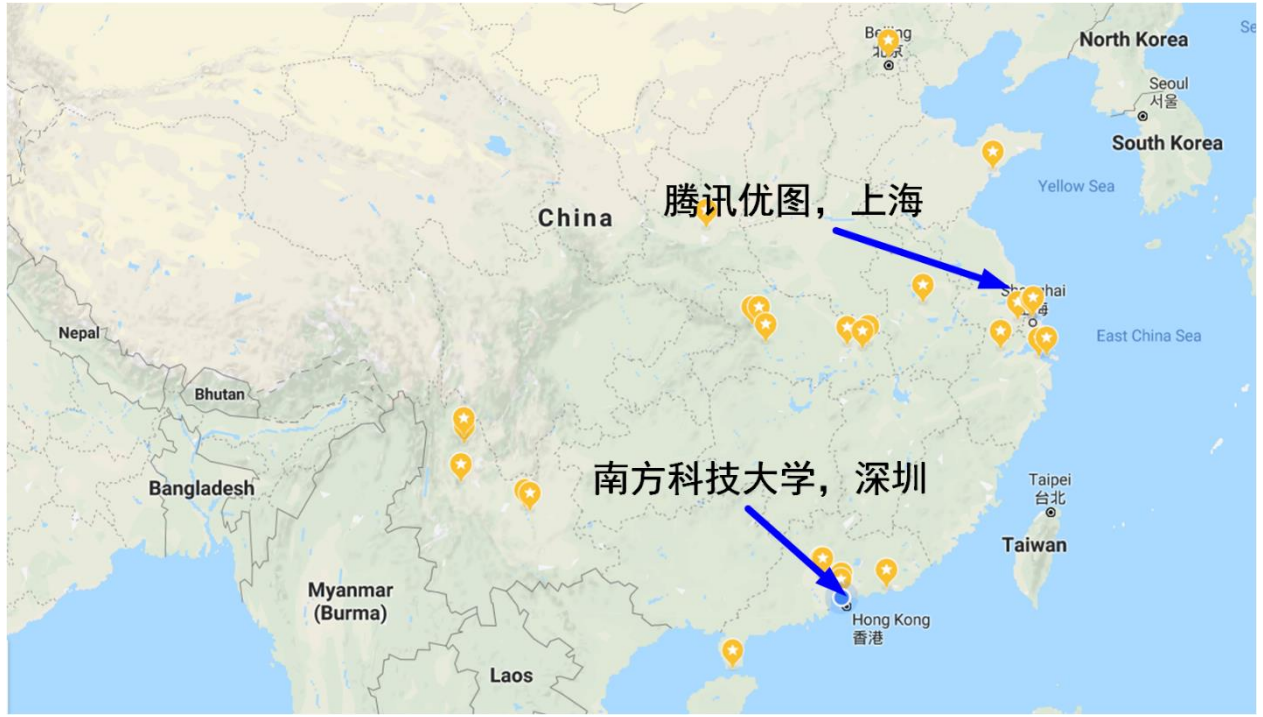
United States

- The University of Texas, at Arlington, Texas, USA, Dec. 2016 - Aug. 2017
- University of Pittsburgh, Pittsburgh, USA, Sep. 2017 - July 2018



China

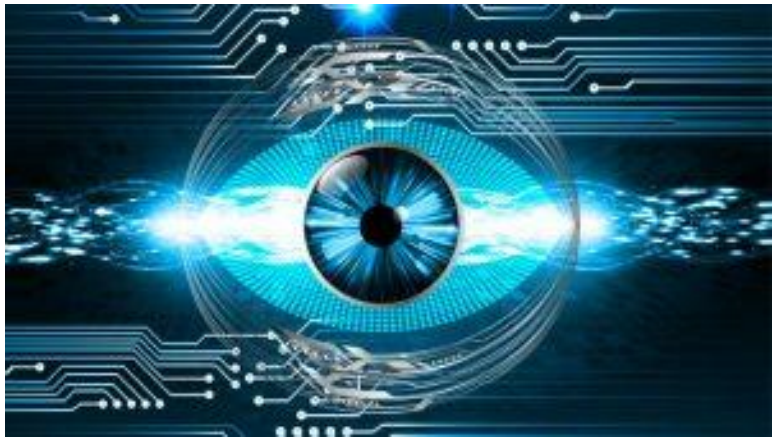
- Youtu Lab, Tencent, Shanghai, China, Aug. 2018 - Oct. 2018
- Southern University of Science and Technology, Shenzhen, China, Nov. 2018 - present



Research Interests

- Research Area
  - Computer Vision
  - Machine Learning
  - Human-Computer Interaction

http://faculty.sustech.edu.cn/fengzheng/



Services

- **Associate Editor:**
  - IET Image Processing

- **Program Committee for Conference :**
  - CVPR, ICLR, AAAI, IJCAI, ICML, NIPS, KDD, UAI.

- **Journal Reviewer:**
  - IEEE TNNLS, IEEE TCSVT, IEEE TMM, Neurocomputing, Information Sciences, IET Computer Vision, IET Image Processing etc..

Office

- **Office :** Room 513, South Building, College of Engineering
- **Phone:** 0755-88015178
- **Email :** zhengf@sustech.edu.cn

About This Course

Why we need to learn C/C++?

- Almost all other modern programming languages and popular libraries are built by C/C++
  - Java: The core of Java Virtual Machine hotspot is implemented in C++.
  - Python: The Python interpreter is implemented in C.
  - Javascript: The popular Javascript engine V8 is implemented in C++.
  - Numpy: The core is implemented in C. It is widely used in AI and ML.
- C/C++ powers the world
  - Most operating system kernels are written in C, including but not limited to Windows, Linux, Mac, iOS, Android and so on.
  - Modern browsers are also written in C/C++. like Chrome, Firefox etc.
  - Modern game engines are written in C/C++, like Unity3D, Unreal Engine, cocos2d-x etc.

https://simpliv.wordpress.com/2019/08/06/7-reasons-why-you-should-keep-learning-c-c/?utm_campaign=News&utm_medium=Community&utm_source=DataCamp.com

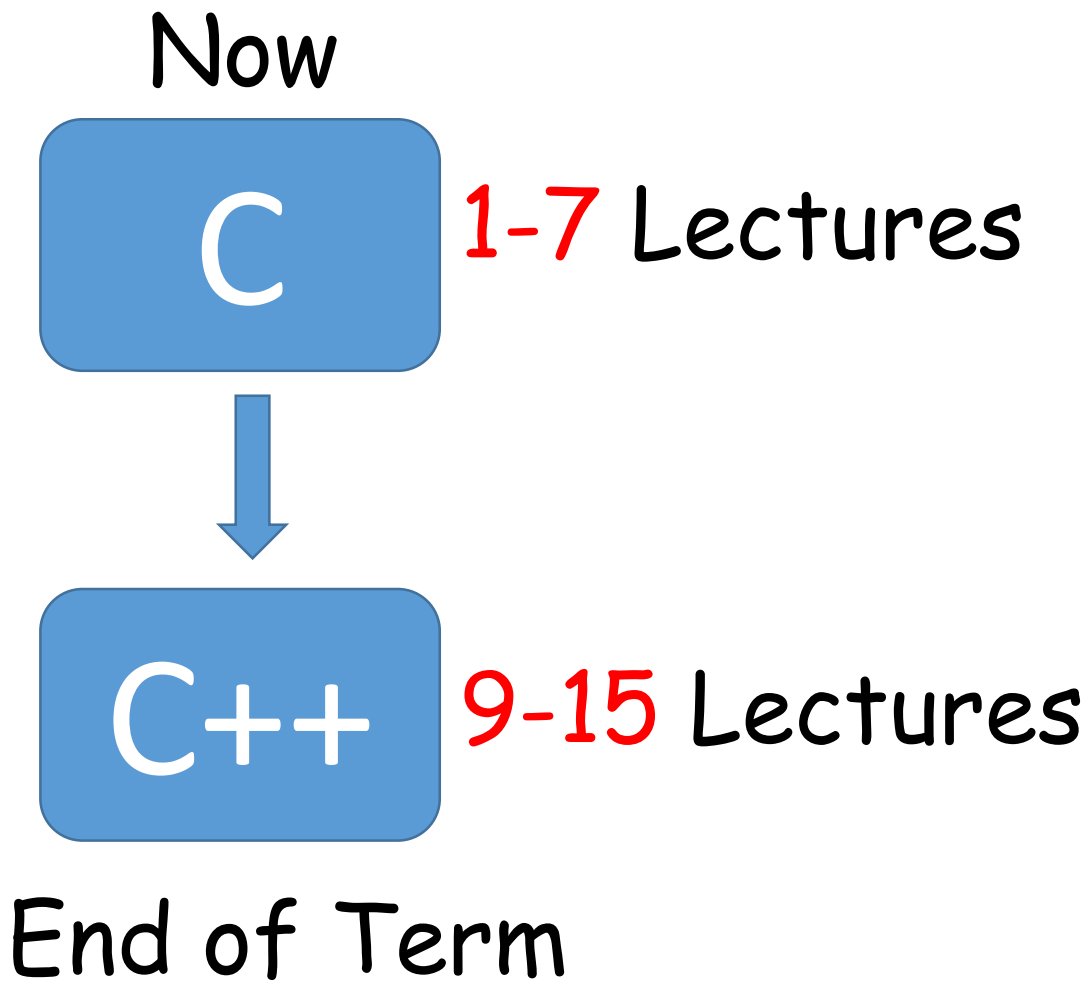
Why we need to learn C/C++?

Efficient machine code produced by C++ compilers

Structure

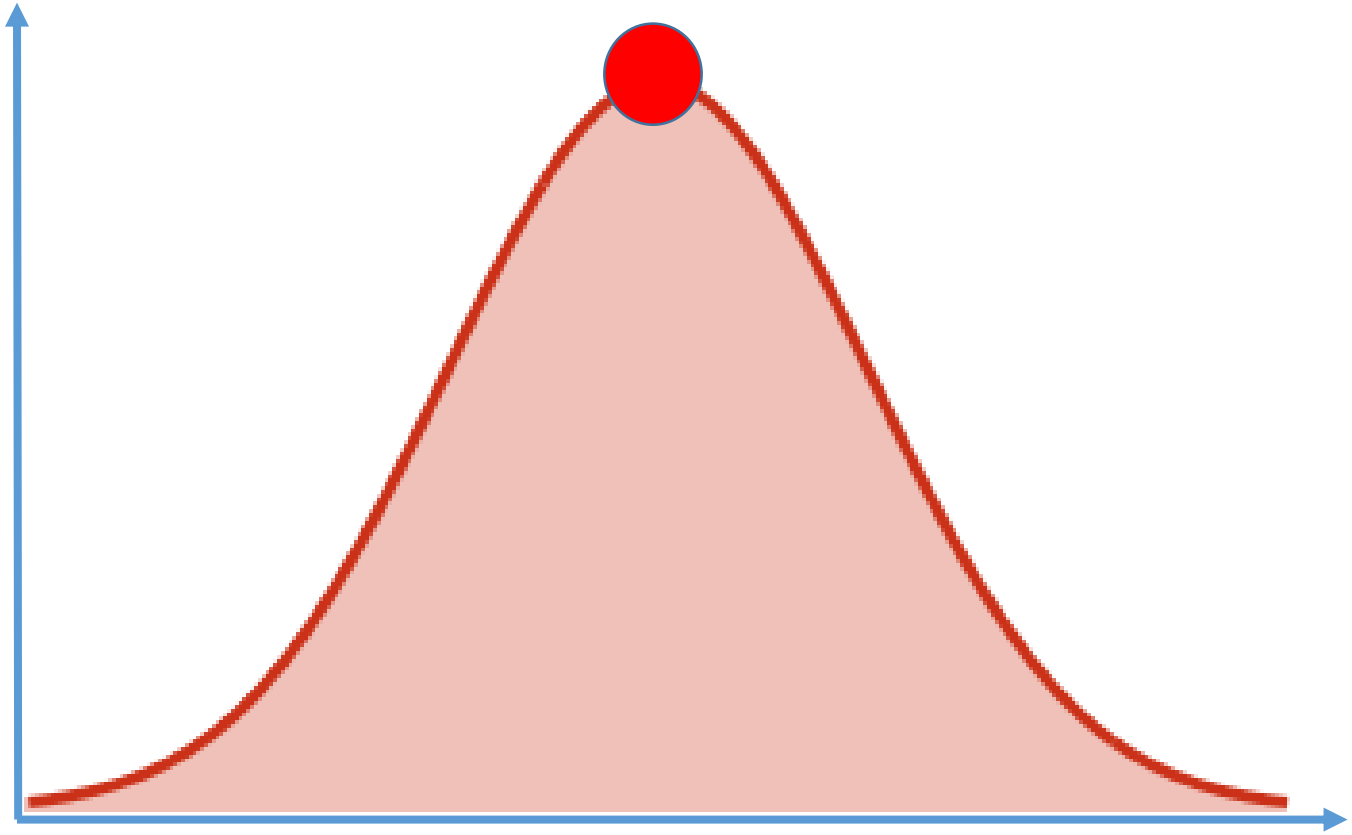
- C related part in C++
  - Pointer (指针)
  - Reference (引用)

- Class types related part



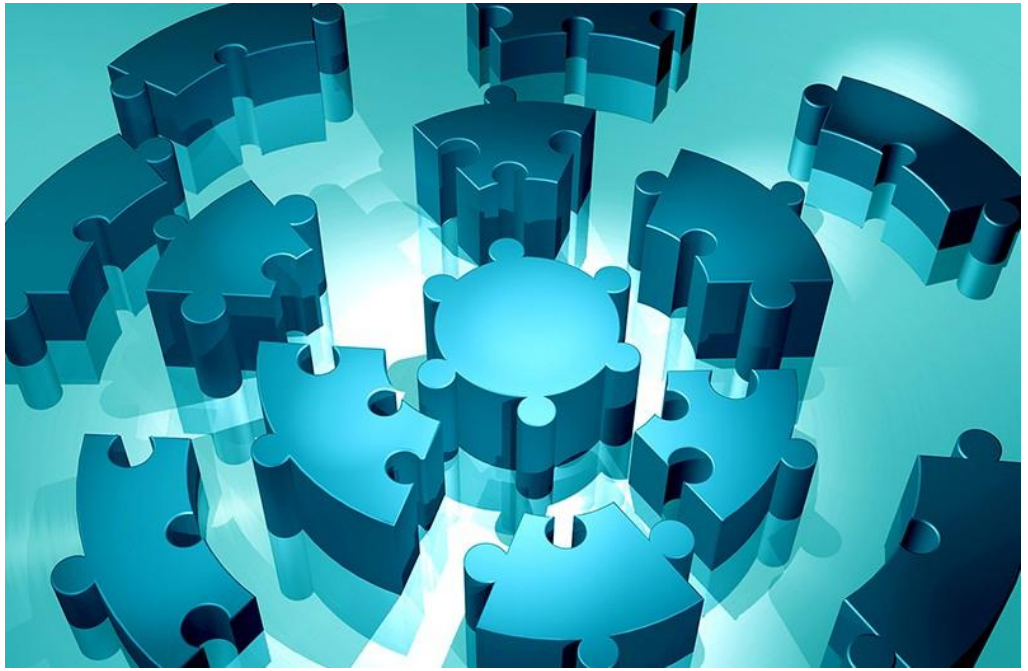
Target Student

- Average ability of programming



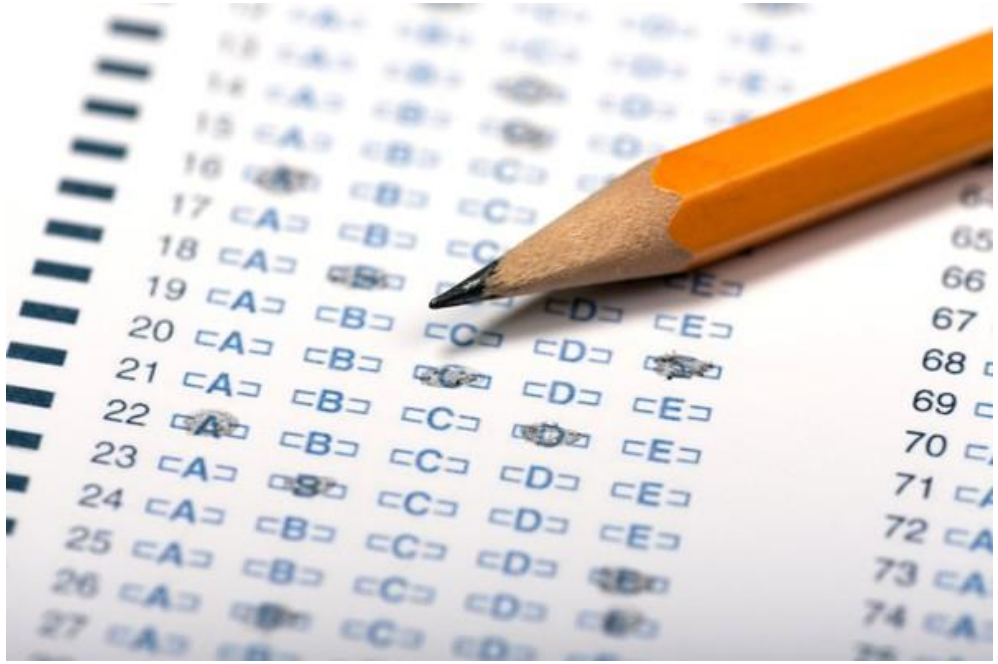
Expectations

- Good understanding of C/C++
- Ability to write reasonably complex programs
- Professional attitude and habits
- Programming thinking



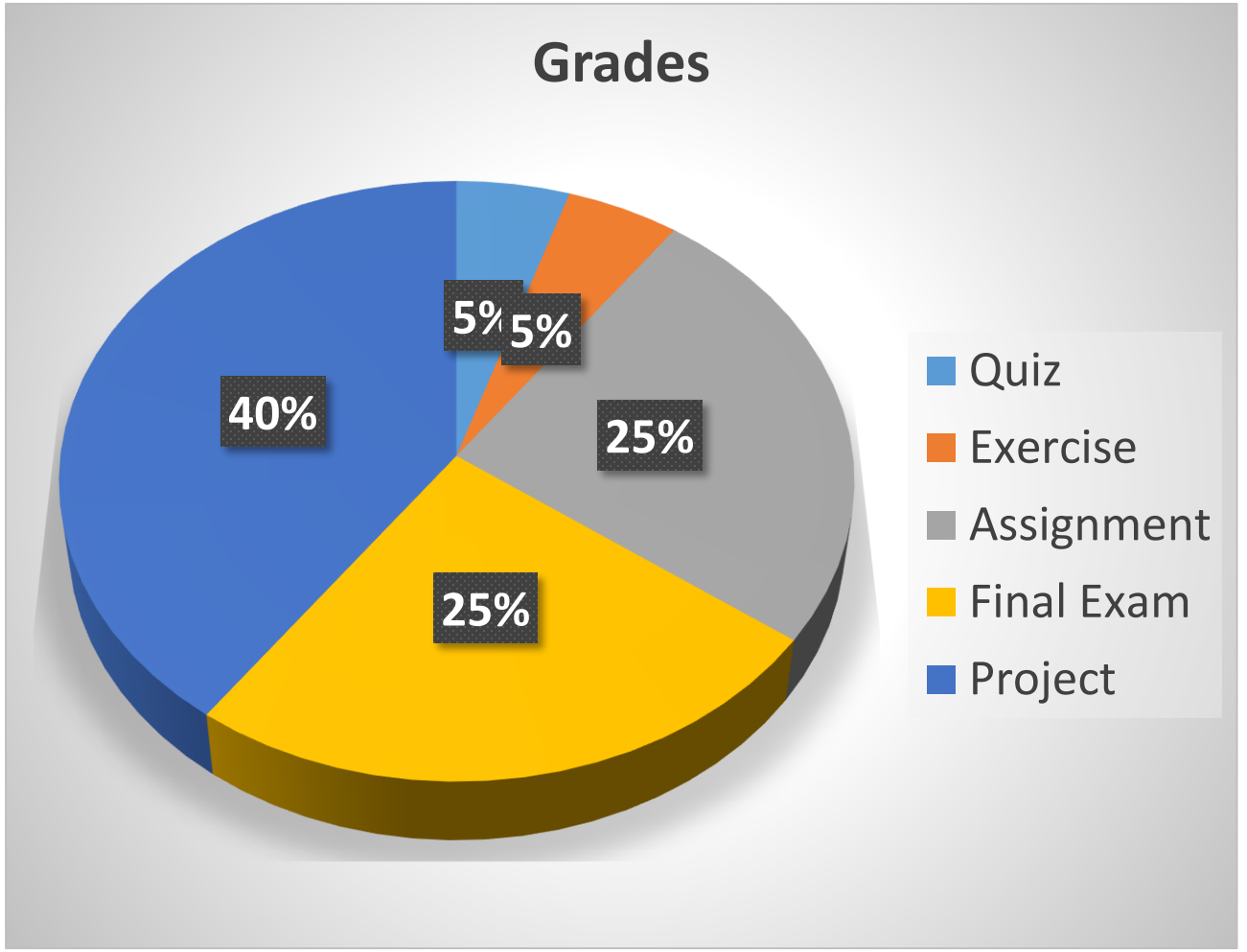
Exams test you on

- General knowledge (why) about C/C++
- Being able to tell (read) what a program does
- Finding errors in a program
- Ability to write codes or pseudo-codes for a moderately complex algorithm



Grade Component

- Quiz: 5%
- Projects I (individual): 20%
- Projects II (group): 20%
- LABS
  - 5 Assignments: 25%
  - Exercises: 5%
- Final Exam 25%
- Projects are VERY IMPORTANT



Honesty

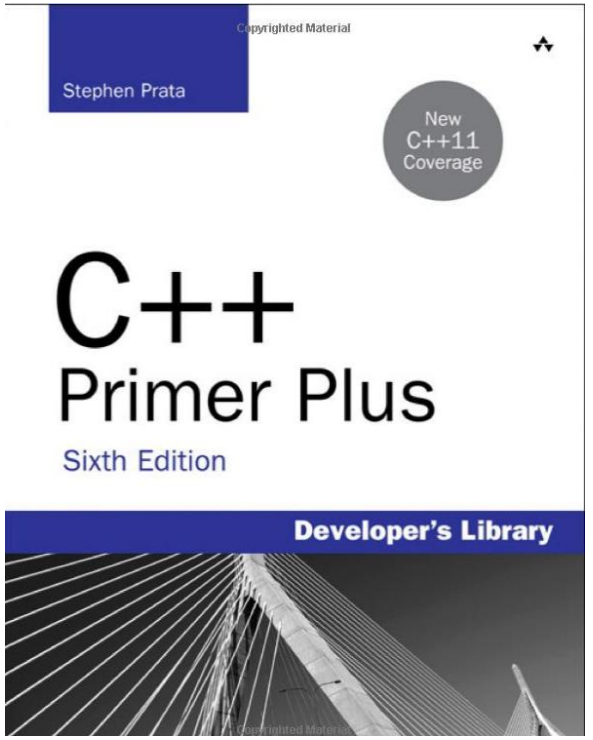
- Get code from the internet for labs/assignments is perfectly **OK**
  - When you borrow, just say it.
  - You don't need to reinvent the wheel

- **DON'T** pretend that you are the author of something that you didn't write

Resources

- **Quick Response Code**
- **Sakai**
  - C/C++ CS205 Spring2021
- **Useful websites:**
  - http://cpp.sh/
  - https://www.onlinegdb.com/

群名称：CS205-C/C++ 2021 Spring
群　号：866138891



Getting Started with C++

What We Need to Know

- The history and philosophy of C and of C++
- Procedural versus object-oriented programming
- How C++ adds object-oriented concepts to the C language
- The mechanics of creating a program

Computer Languages

- Machine language
  - Only computer understands; Defined by hardware design; Strings of numbers (01); Cumbersome for humans
  - Instruct computers to perform elementary operations;
  - **Example:**

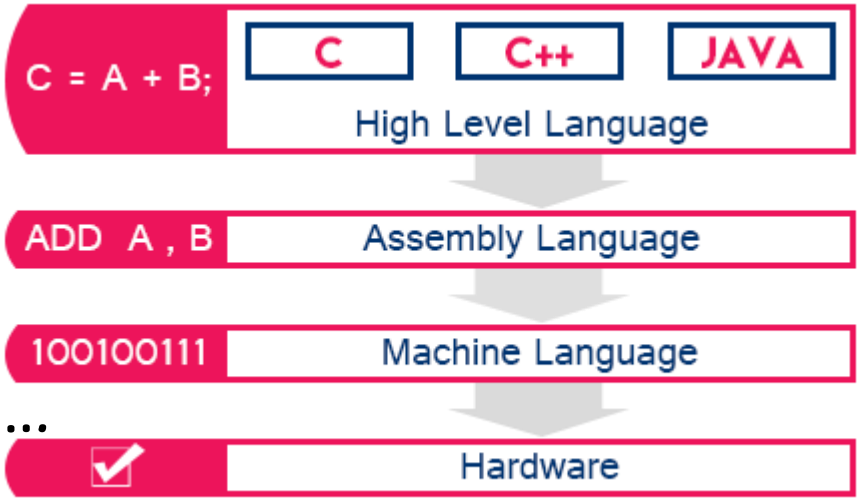
- Assembly language
  - English-like abbreviations representing elementary computer operations; Clearer to humans; Incomprehensible to computers
  - **Example: LOAD BASEPAY**

High-level Languages

Programmer = translator

- High-level languages
  - Similar to English, use common mathematical notations
  - Single statements accomplish substantial tasks: Assembly language requires many instructions to accomplish simple tasks
  - Translator programs (compilers): Convert to machine language
  - Interpreter programs: Directly execute it
  - Example:

grossPay = basePay + overTimePay



- C/C++, JAVA, PYTHON, MATLAB,……

Difference between Compiler and Interpreter

- Interpreter
  - Translate just one statement of the program at a time into machine code.
  - Take very less time to analyze the source code. However, the overall time to execute the process is much slower.
  - Does not generate an intermediary code.
- Compiler
  - Scan the entire program and translates the whole of it into machine code at once.
  - Take a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.
  - Generate an intermediary object code.

History of C

- Evolved from two other programming languages
  - BCPL and B: "Typeless" languages
- Dennis Ritchie (Bell Laboratories)
  - Added data typing, other features
- Development language of UNIX
- Hardware **independent**
  - Portable programs

Year	C Standard[9]
1972	Birth
1978	K&R C
1989/1990	ANSI C and ISO C
1999	C99
2011	C11
2017/2018	C18

BCPL: **Basic Combined Programming Language → C**

https://www.unixmen.com/dennis-m-ritchie-father-c-programming-language/

C Programming Philosophy

- Branching statements
  - Hard: earlier procedural programming
  - Easy: structured programming
- Top-down
  - Divide large tasks into smaller tasks

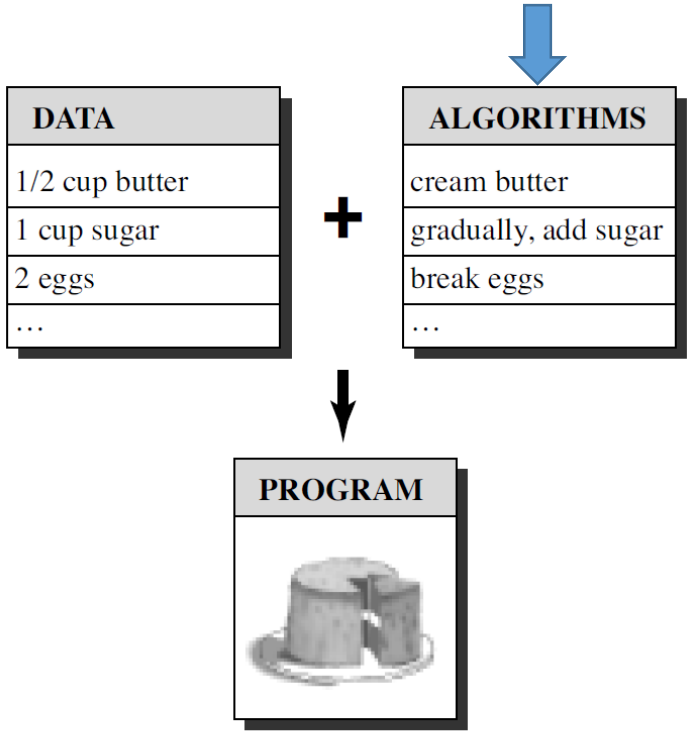
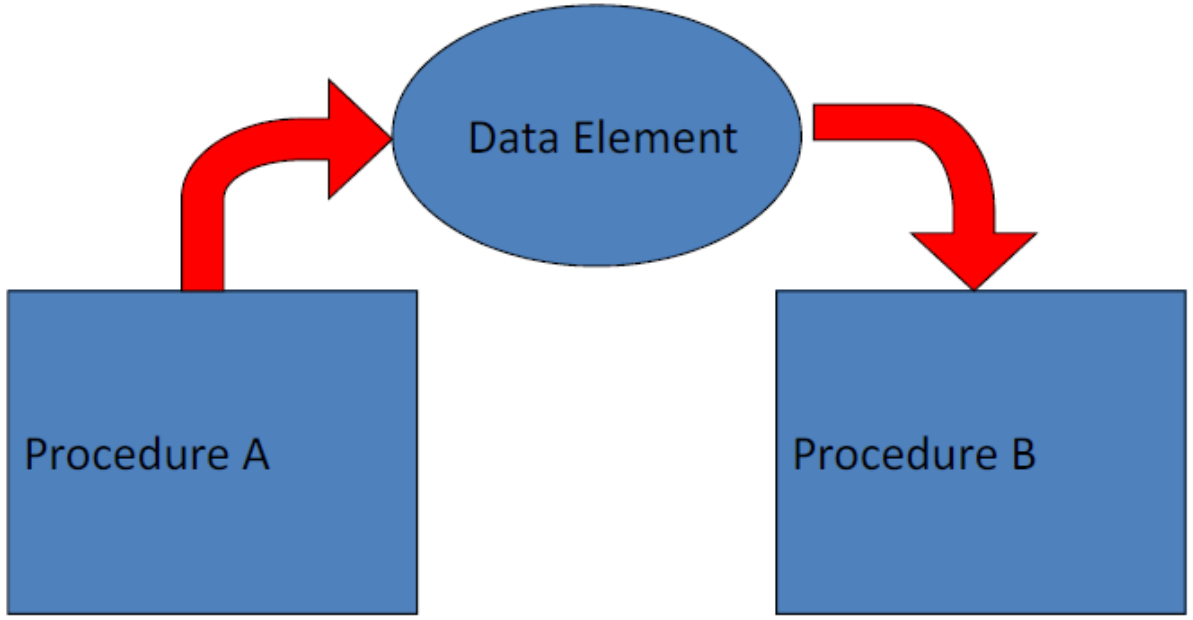


Figure 1.1 Data + algorithms = program.

C Programming Philosophy

- Procedural programming --- Compared to OOP
  - More algorithms but data
  - Interaction between procedures



History of C++

- Extension of *C*
- Early 1980s: Bjarne Stroustrup (Bell Laboratories)
- Provides capabilities for Object-Oriented Programming
  - Objects: reusable software components: Model items in real world
  - Object-oriented programs: Easy to understand, correct and modify
- Hybrid language
  - C-like style
  - Object-oriented style

C++ standards

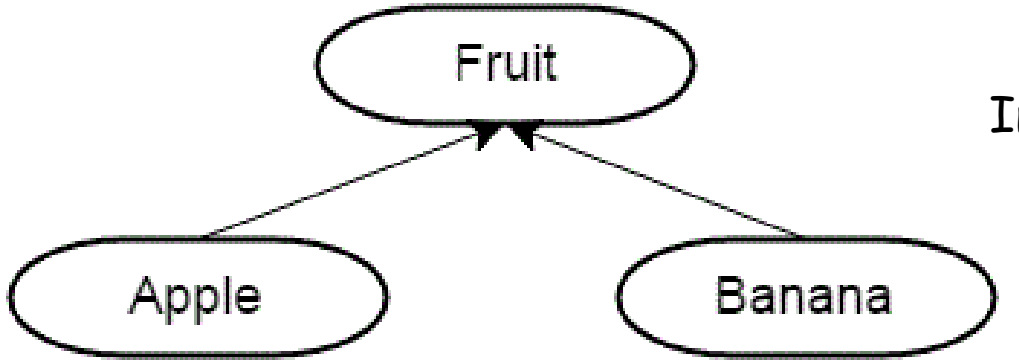
Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998[29]	C++98
2003	ISO/IEC 14882:2003[30]	C++03
2011	ISO/IEC 14882:2011[31]	C++11, C++0x
2014	ISO/IEC 14882:2014[32]	C++14, C++1y
2017	ISO/IEC 14882:2017[33]	C++17, C++1z
2020	ISO/IEC 14882:2020[12]	C++20, C++2a

C++ Philosophy

- Fit the language to the problem
- A class is a specification describing such a new data form
  - What data is used to represent an object
  - The operations that can be performed on that data
- An object is a particular data constructed according to that plan
- Emphasizes the data
- Bottom-up programming
  - Class definition to program design

Features of C++

- Binding
- Reusable (可重用的)
- Protectability (可保护的)
- Polymorphism (多态性)- multiple definitions for operators and functions
- Inheritance (继承性)
- Portable (可移植性)

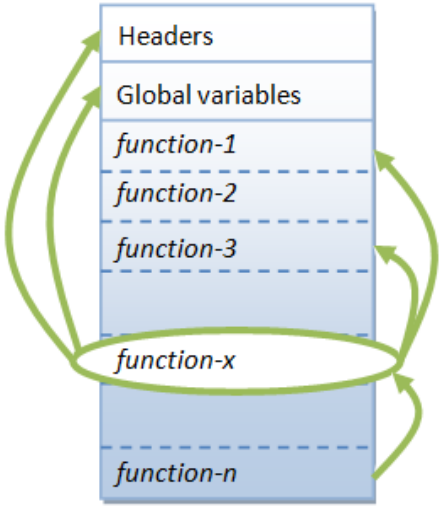


Inheritance

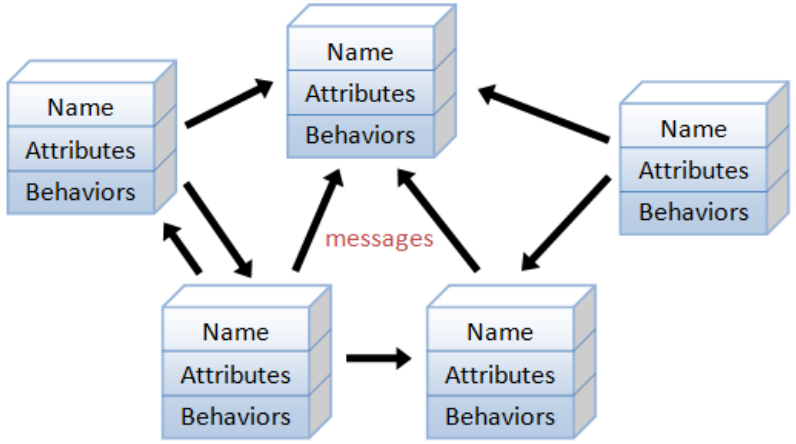
Protectability

Comparison

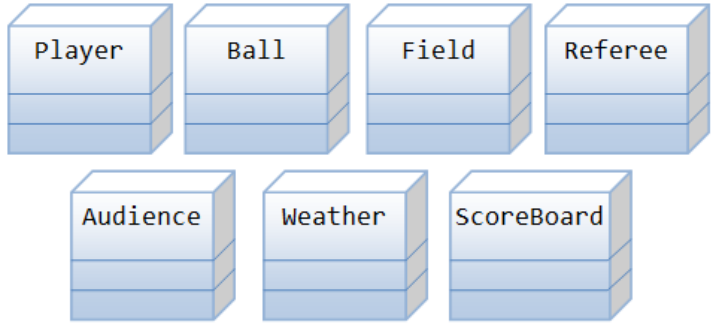
- Procedural versus Object-oriented (Encapsulated: 封装的)



A function (in C) is not well-encapsulated



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages



Classes (Entities) in a Computer Soccer Game

Class definition

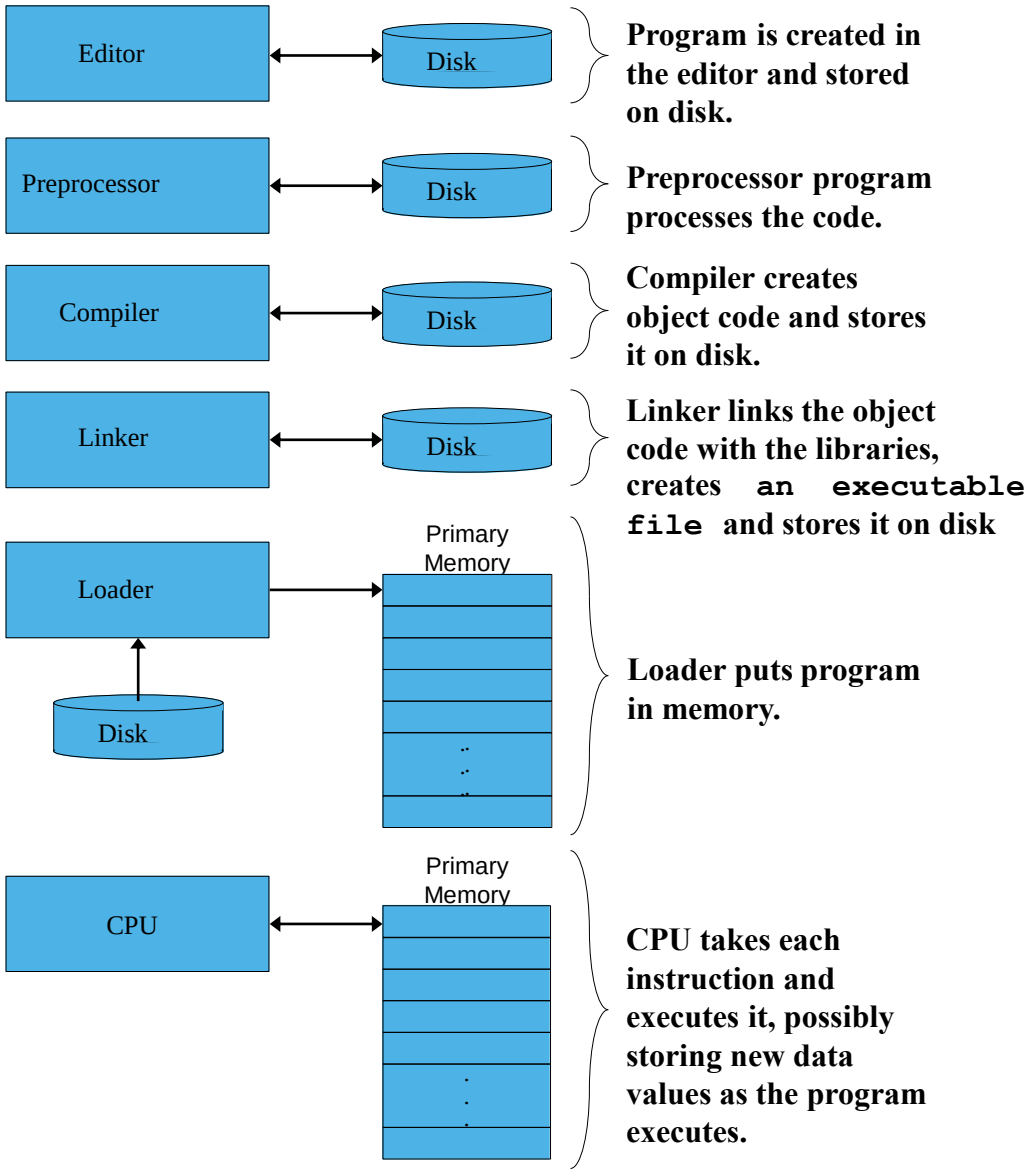
Object declaration

Program Phase

- Edit

- Preprocess (how to organize)
- Compile
- Link

- Load
- Execute



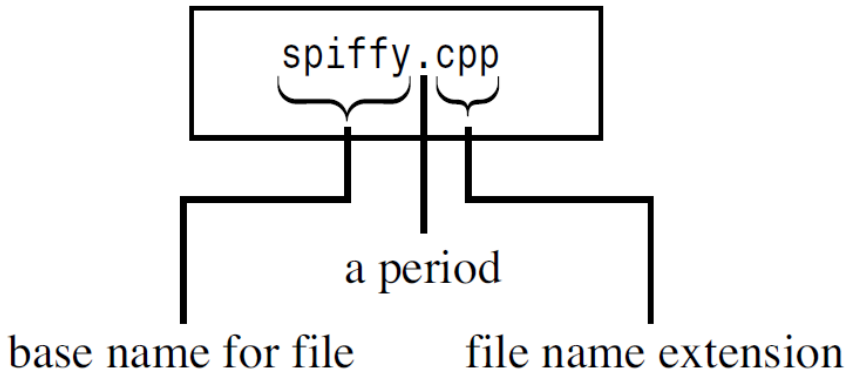
Creating the Source Code File

- Integrated development environments
  - VSCODE, Microsoft Visual C++
  - QT
  - Apple Xcode

- Any available text editor
  - Debuggers: GDB: The GNU Project Debugger
  - Command prompt
  - Compiler

Proper Extensions

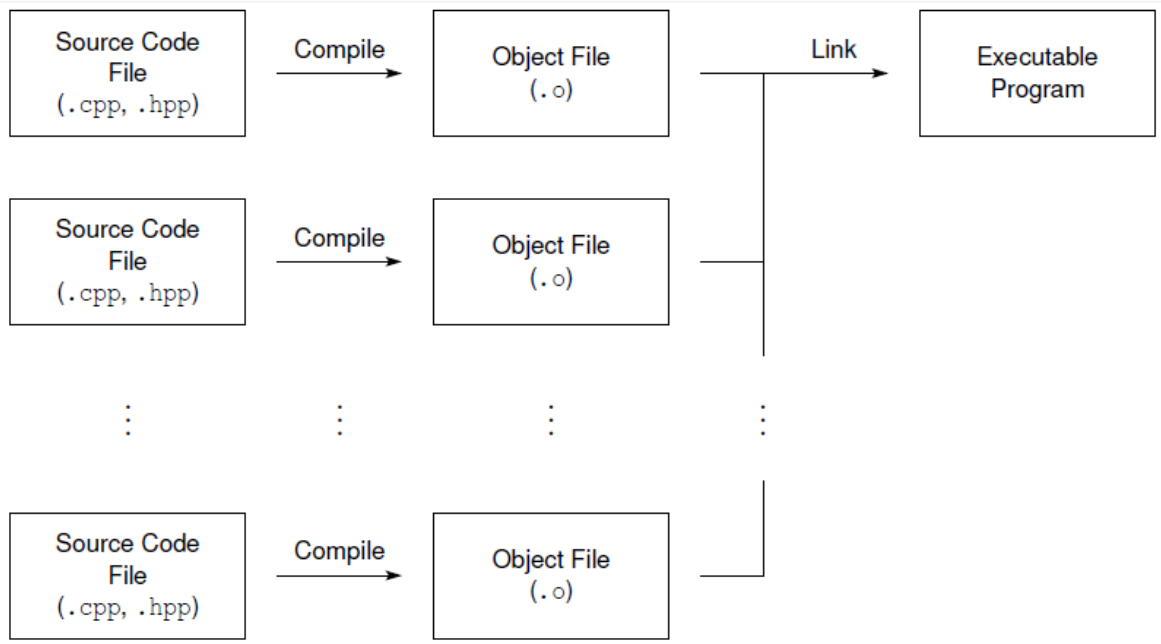
- Suffix



C++ Implementation	Source Code Extension(s)
Unix	`C, cc, cxx, c`
GNU C++	`C, cc, cxx, cpp, c++`
Digital Mars	`cpp, cxx`
Borland C++	`cpp`
Watcom	`cpp`
Microsoft Visual C++	`cpp, cxx, cc`
Freestyle CodeWarrior	`cpp, cp, cc, cxx, c++`

Software Build Process

- Start with C++ source code files (.cpp, .hpp)
- Compile: convert code to object code stored in object file (.o)
- Link: combine contents of one or more object files (and possibly some libraries) to produce executable program



GNU Compiler Collection (GCC)
C++ Compiler (编译器)

- g++ command provides both compiling and linking functionality
- Command-line usage:

  g++ [*options*] *input file . . .*

- Compile C++ source file *file.cpp* to produce object code file *file.o*:

  g++ *-c file.cpp*

- Link object files *file 1.o*, *file 2.o*, . . . to produce executable file *executable_name*:

  g++ *-o executable_name file 1.o file 2.o . . .*

- Tools for windows: Windows Subsystem, MinGW, MSYS2, Cygwin

Common g++ Command-Line Options

- Web site: http://www.gnu.org/software/gcc
- C++ standards support in GCC: https://gcc.gnu.org/projects/cxx-status.html

- `-c`
  - compile only (i.e., do not link)
- `-o` *file*
  - use file *file* for output
- `-g`
  - include debugging information
- `-O`*n*
  - set optimization level to *n* (0 almost none; 3 full)
- `-std=c++17`
  - conform to C++17 standard
- `-I`*dir*
  - specify additional directory *dir* to search for include files
- `-L`*dir*
  - specify additional directory *dir* to search for libraries
- `-l`*lib*
  - link with library *lib*
- `-pthread`
  - enable concurrency support (via pthreads library)
- `-pedantic-errors`
  - strictly enforce compliance with standard
- `-Wall`
  - enable most warning messages
- `-Wextra`
  - enable some extra warning messages not enabled by `-Wall`
- `-Wpedantic`
  - warn about deviations from strict standard compliance
- `-Werror`
  - treat all warnings as errors
- `-fno-elide-constructors`
  - in contexts where standard allows (but does not require) optimization that omits creation of temporary, do not attempt to perform this optimization

Windows Compilers

- **Choose windows applications:** MFC Windows application, dynamic link library, ActiveX control, DOS or character-mode executable, static library, or console application
- **Choose 64-bit or 32-bit versions**
- **Actions:** Compile, Build, Make, Build All, Link, Execute, Run, and Debug
  - Compile: the code in the file that is currently open
  - Build or Make: all the source code files in the project.
  - Build All: all the source code files from scratch
  - Link: combining the compiled source code with the necessary library code
  - Execute or Run: running the program (may do the earlier steps)
  - Debug: containing extra code that increases the program size, slows program execution, but enables detailed debugging features
- http://en.wikipedia.org/wiki/List_of_compilers

Summary

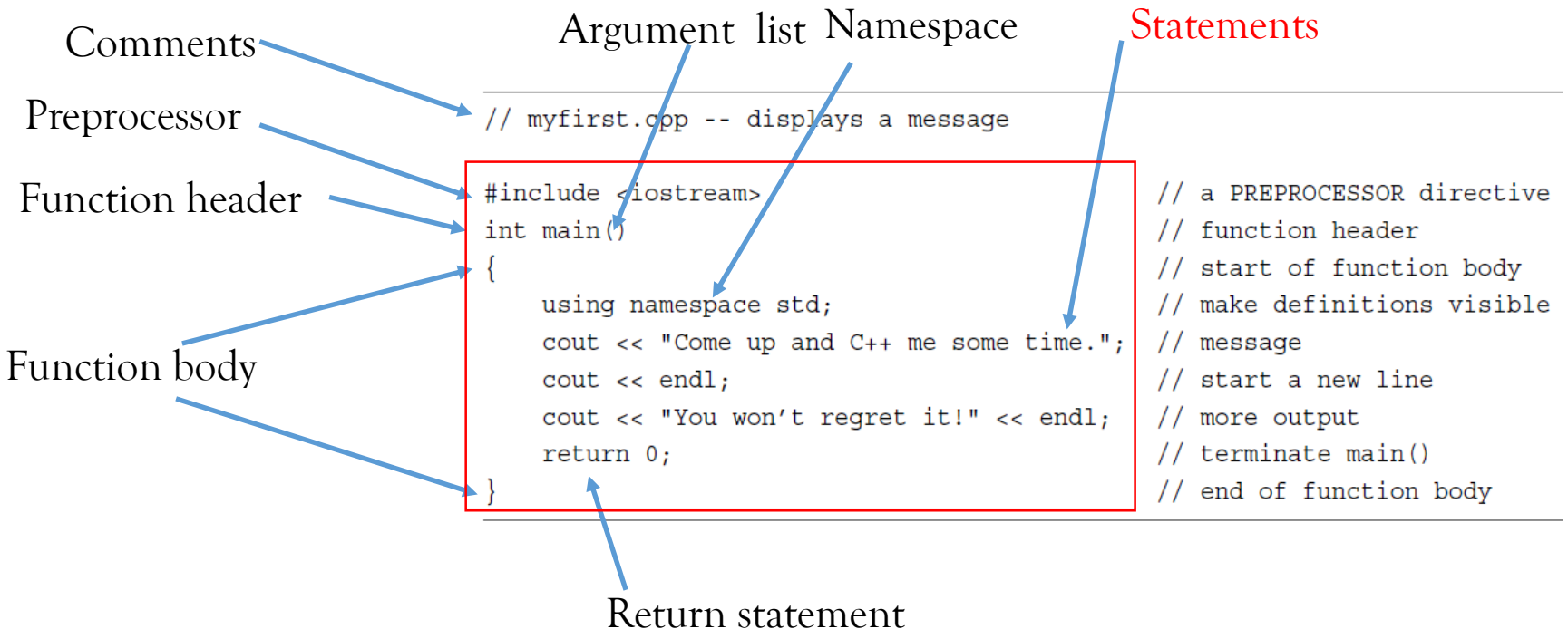
- C Programming Philosophy
- C++ Programming Philosophy
- Tools
- Compilers

Setting Out to C++

C++ Program Sample

Run myfirst.cpp

- A program example
  - Noting: C++ is **CaSe SenSiTiVe**



Comments (注释)

- Two styles of comments provided
  - Comment starts with // and proceeds to end of line
  - Comment starts with /* and proceeds to first */

```
// This is an example of a comment.
/* This is another example of a comment. */
/* This is an example of a comment that
   spans
   multiple lines. */
```

- The compiler ignores comments

Identifiers (标识符)

- Identifiers: used to name entities such as: types, objects (i.e., variables), and functions
  - Valid identifier: sequence of one or more letters, digits, and underscore "_" characters that does not begin with a digit
  - Identifiers are case sensitive
  - Identifiers cannot be any of reserved keywords
  - **Scope** of identifier is context in which identifier is valid (e.g., block, function, global)

```
event_counter
eventCounter
sqrt_2
f_o_o_b_a_r_4_2
```

Keywords (关键字)

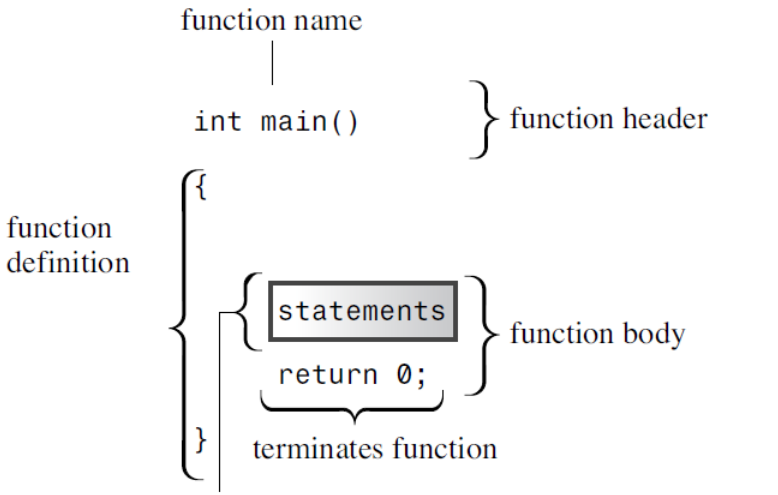
- Keywords are the vocabulary of a computer language

alignas	default	noexcept	this
alignof	delete	not	thread_local
and	do	not_eq	throw
and_eq	double	nullptr	true
asm	dynamic_cast	operator	try
auto	else	or	typedef
bitand	enum	or_eq	typeid
bitor	explicit	private	typename
bool	export	protected	union
break	extern	public	unsigned
case	false	register	using
catch	float	reinterpret_cast	virtual
char	for	return	void
char16_t	friend	short	volatile
char32_t	goto	signed	wchar_t
class	if	sizeof	while
compl	inline	static	xor
const	int	static_assert	xor_eq
constexpr	long	static_cast	override*
const_cast	mutable	struct	final*
continue	namespace	switch	
decltype	new	template	

https://doc.bccnsoft.com/docs/cppreference_en/keywords/index.html

Definition of the main() Function

- Function definition
  - ➢ Function header - a summary of the function's interface
  - ➢ Function body {}
  - ① Statement - each complete instruction + semicolon [;]
  - ② Return statement



Statements are C++ expressions terminated by a semicolon.

Features of the main() Function

- main() functions are called by startup code - mediate between the program and the operating system
- Function header - describe the interface between main() and the operating system

```
int main()

main()      // original C style
```

- Standalone program - does need a main()
  ① Main() or MAIN() or mane()
  ② WinMain() or _tmain()

```
int main(void)      // very explicit style
return 0;
```

- Otherwise
  ① A dynamic link library (DLL)
  ② A controller chip in a robot

```
void main()
```

#pragma comment(lib, "emapi")
#pragma comment(lib, "emapi")

C++ Preprocessor (预处理)

- Preprocessor transforms source code, prior to compilation
  - Preprocessor passes the output to compiler for compilation
  - Preprocessor behavior can be controlled by directives
  - Directive occupies **single line** of code
  - No semicolon (;→\)

- Consists of:

1. hash character (i.e., "#")
2. preprocessor instruction (i.e., `define`, `undef`, `include`, `if`, `ifdef`, `ifndef`, `else`, `elif`, `endif`, `line`, `error`, and `pragma`)
3. arguments (depending on instruction)
4. line break

#pragma comment(lib, "emapi")
#pragma comment(compiler)

- Can be used to:

- conditionally compile parts of source file
- define macros and perform macro expansion
- include other files

Preprocessor: Source-File Inclusion

- Include contents of another file in source using preprocessor

```
#include <path_specifier>
or
#include "path_specifier"
```

  - Angle brackets used for system header files
  - Double quotes used otherwise
  - Path specifier is pathname (which may include directory) identifying file whose content is to be substituted in place of include directive

- Examples

```
#include <iostream>
#include <boost/tokenizer.hpp>
#include "my_header_file.hpp"
#include "some_directory/my_header_file.hpp"
```

Preprocessor: Defining Macros (宏)

Run glue.cpp

- Define macros using #define directive
- When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement
- This replacement can be an expression, a statement, a block or simply anything

```
#define getmax(a,b) a>b?a:b
```

- Function macro definitions accept two special operators: #, ##(concatenate)
- Less readable

```
#define glue(a,b) a ## b
glue(c,out) << "test";   →   cout<< "test";
```

Preprocessor: Conditional Compilation

- Conditionally include code through use of if-elif-else directives
- Conditional preprocessing block consists of:
  1. **#if**, **#ifdef**, or **#ifndef** directive
  2. optionally any number of **#elif** directives
  3. at most one **#else** directive
  4. **#endif** directive
- Example:

```
#if DEBUG_LEVEL == 1
// ...
#elif DEBUG_LEVEL == 2
// ...
#else
// ...
#endif
```

Header Filenames

- Reason of using header files
  - As programs grow larger (and make use of more files), it becomes increasingly tedious to have to forward declare every function you want to use that is defined in a different file.

Kind of Header	Convention	Example	Comments
C++ old style	Ends in `.h`	`iostream.h`	Usable by C++ programs
C old style	Ends in `.h`	`math.h`	Usable by C and C++ programs
C++ new style	No extension	`iostream`	Usable by C++ programs, uses `namespace std`
Converted C	`c` prefix, no extension	`cmath`	Usable by C++ programs, might use non-C features, such as `namespace std`

Namespaces

- Reasons of using namesapce
  - To simplify the writing of large programs
  - To help organize programs that combine pre-existing code from several companies
- Example: indicate which wanda you want, using double colon ::

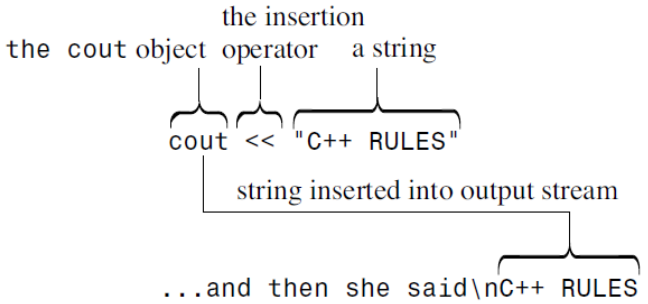
```
Microflop::wanda("go dancing?");       // use Microflop namespace version
Piscine::wanda("a fish named Desire"); // use Piscine namespace version
```

- A namespace example: std
  - Standard component of C++ compilers

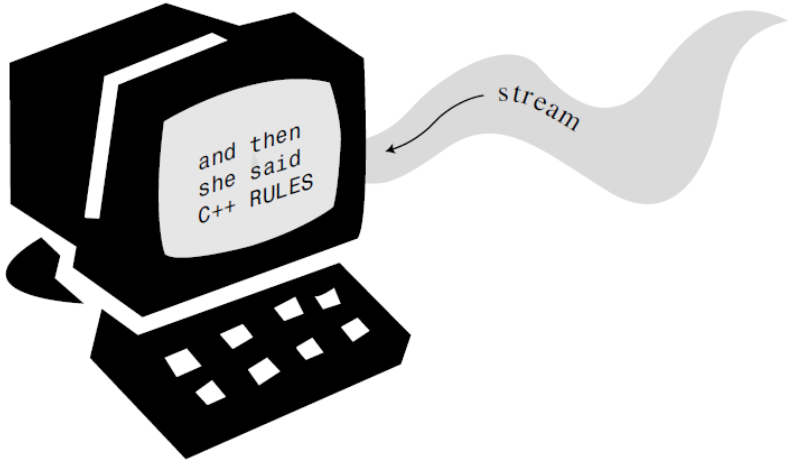
```
std::cout << "Come up and C++ me some time.";
std::cout << std::endl;

using namespace std;  // lazy approach, all names available

using std::cout;    // make cout available
using std::endl;    // make endl available
using std::cin;     // make cin available
```

C++ Output with cout

- An example to print a string
  - cout: is an **object** defined in stream
  - String: double quotation marks “ ”
  - Insertion operator: <<
  - Two ways: endl, \n




```
cout << "Pluto is a dwarf planet.\n";          // show text, go to next line
cout << "Pluto is a dwarf planet." << endl;    // show text, go to next line
```



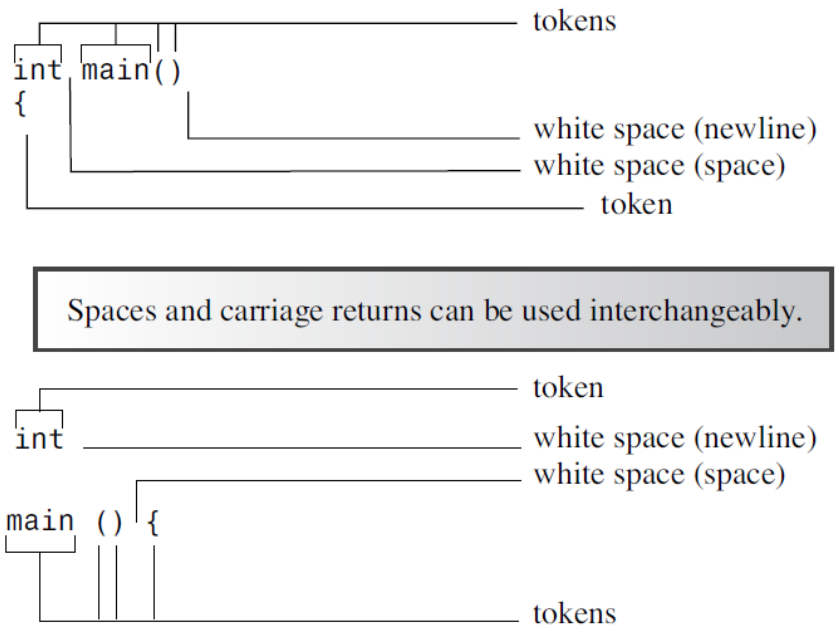
C++ Source Code Formatting

- Source code contains three components
  - Tokens - indivisible elements in a line of code
  - White space - a space, tab, or carriage return
  - Semicolon marks the end of each statement

① Spread a single statement over several lines

② Place several statements on one line

```
#include <iostream>
     int
main
() {   using
    namespace
         std; cout
            <<
"Come up and C++ me some time."
;    cout <<
endl; cout <<
"You won't regret it!" <<
endl;return 0; }
```



C++ Source Code Formatting

Make program more readable

- Observe these rules:
  - One statement per line
  - An opening brace and a closing brace for a function, each of which is on its own line
  - Statements in a function indented from the braces{}
  - No whitespace around the parentheses() associated with a function name

```
return0;            // INVALID, must be return 0;
return(0);          // VALID, white space omitted
return (0);         // VALID, white space used
intmain();          // INVALID, white space omitted
int main()          // VALID, white space omitted in ()
int main ( )        // ALSO VALID, white space used in ( )
```

Program: C++ Statements

Run carrots.cpp

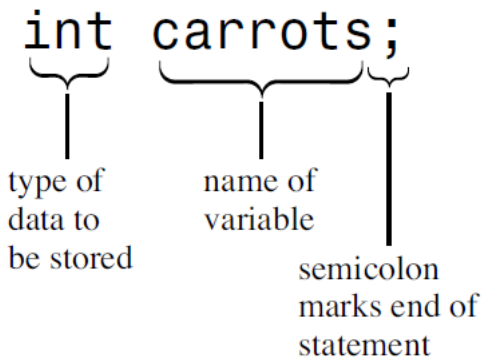
- A program is a collection of functions
- Each function is a collection of statements
  - A declaration statement creates a variable of certain type
  - An assignment statement provides a value for that variable

```
// carrots.cpp -- food processing program
// uses and displays a variable

#include <iostream>

int main()
{
    using namespace std;

    int carrots;            // declare an integer variable

    carrots = 25;            // assign a value to the variable
    cout << "I have ";
    cout << carrots;        // display the value of the variable
    cout << " carrots.";
    cout << endl;
    carrots = carrots - 1;  // modify the variable
    cout << "Crunch, crunch. Now I have " << carrots << " carrots." << endl;
    return 0;
}
```

What is the definition of type?

Declaration Statements

- Variable: Identify both the storage location and how much memory space to store an item
  - Declaration statement: to provide **a label for the location** and to indicate the type of storage
  - Complier: to allocate the memory space



Assignment Statements

- An assignment statement assigns a value to a storage location
- Assignment operator: =
- Two examples:
  - Assign serially (from copy)
  - Arithmetic expression: +-*/ (form CPU)

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

```
int carrots;

carrots = 25;

carrots = carrots - 1;  // modify the variable
```

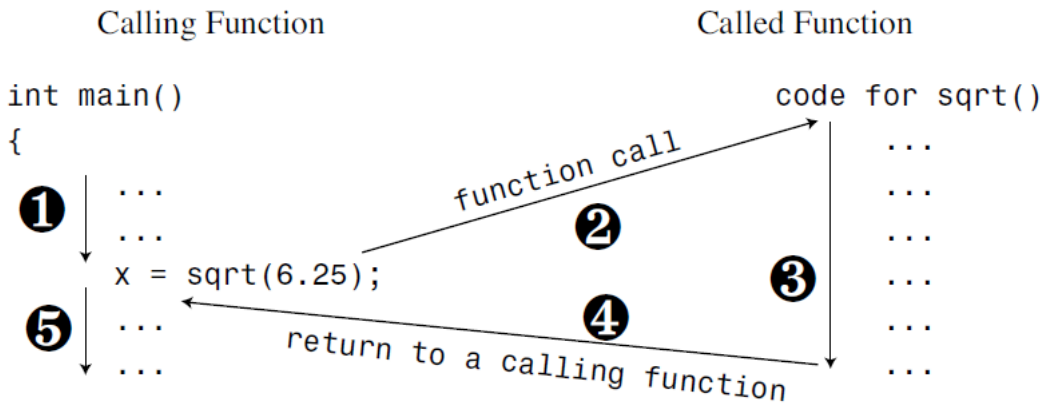
Assignment: cin

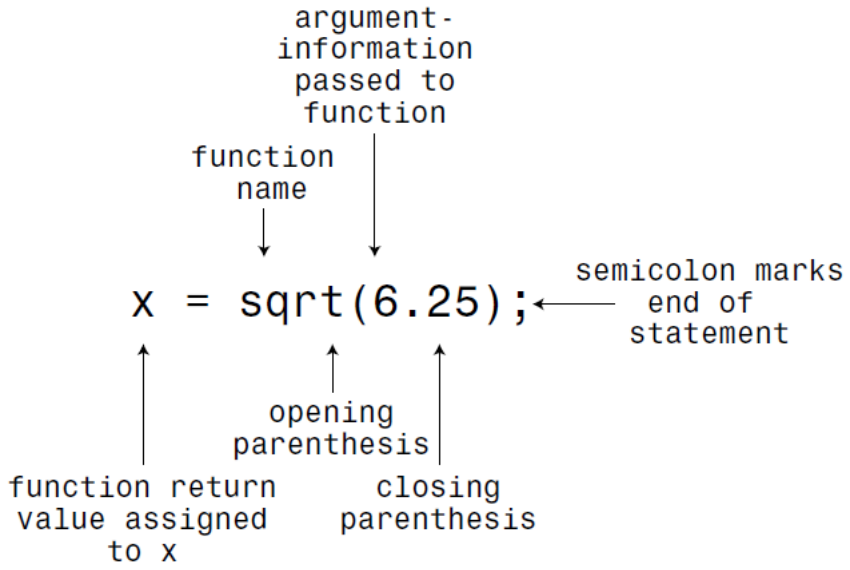
- An **object** of input istream (a class)
  - >> operator: extract characters from the input stream
  - The value typed from the keyboard is eventually assigned to the variable: carrots

```
// getinfo.cpp -- input and output
#include <iostream>

int main()
{
    using namespace std;

    int carrots;

    cout << "How many carrots do you have?" << endl;
    cin >> carrots;                 // C++ input
    cout << "Here are two more. ";
    carrots = carrots + 2;
// the next line concatenates output
    cout << "Now you have " << carrots << " carrots." << endl;
    return 0;
}
```

Assignment: Called Functions

- The example of called functions
  - 6.25 in parentheses is the input, called an argument or parameter
  - This example assigns the return value to the variable x





Function prototype of functions

- What a function prototype does for functions is the **same** to that a variable declaration does for variables

```
double sqrt(double);   // function prototype
```

  - You can type the function prototype into your source code file yourself
  - You can include the cmath (math.h on older systems) header file, which has the prototype in it
- The terminating semicolon in the prototype
  - Identifies it as a statement
  - Makes it a prototype instead of a function header

Basic characteristics of functions

- Don't confuse the function prototype with the function definition
  - Prototype describes the function interface
  - Definition includes the code for the function's workings

- Place a function prototype ahead of where you first use the function
- Using #include directive
  - The header files contain the prototypes.
  - The library files contain the compiled code for the functions,

Math functions in cmath

Run sqrt.cpp

- An example shows the use of the library function sqrt()
- It provides a prototype by including the cmath file.

```
// sqrt.cpp -- using the sqrt() function

#include <iostream>
#include <cmath>    // or math.h

int main()
{
    using namespace std;

    double area;
    cout << "Enter the floor area, in square feet, of your home: ";
    cin >> area;
    double side;
    side = sqrt(area);
    cout << "That's the equivalent of a square " << side
         << " feet to the side." << endl;
    cout << "How fascinating!" << endl;
    return 0;
}
```

Function Variations

- Some functions require more than one item of information

```
double pow(double, double);  // prototype of a function with two arguments
answer = pow(5.0, 8.0);      // function call with a list of arguments
```

- Other functions take no arguments

```
int rand(void);        // prototype of a function that takes no arguments
myGuess = rand();      // function call with no arguments
```

- There also are functions that have no return value

```
void bucks(double);  // prototype for function with no return value
bucks(1234.56);      // function call, no return value
```

User-Defined Functions

Run ourfunc.cpp

- The standard C library: more than 140 predefined functions
- Two examples
  - main() is a user-defined function
  - simon() is another user-defined function

```
// ourfunc.cpp -- defining your own function
#include <iostream>
void simon(int);    // function prototype for simon()

int main()
{
    using namespace std;
    simon(3);       // call the simon() function
    cout << "Pick an integer: ";
    int count;
    cin >> count;
    simon(count);   // call it again
    cout << "Done!" << endl;
    return 0;
}

void simon(int n)   // define the simon() function
{
    using namespace std;
    cout << "Simon says touch your toes " << n << " times." << endl;
}                   // void functions don't need return statements
```

User-Defined Function Form

- A function header
- Comes the function body
- Enclosed in braces

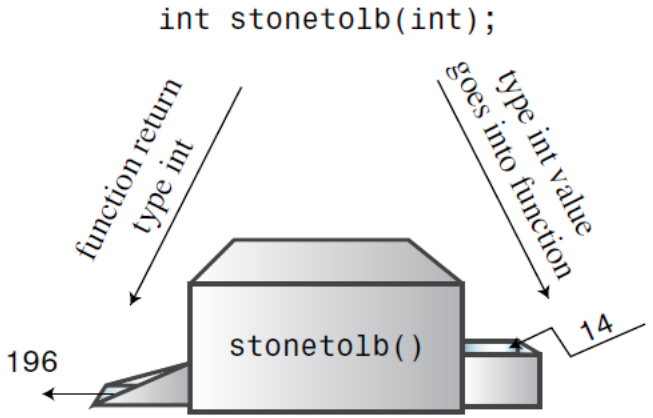
```
type functionname(argumentlist)
{
     statements
}
```

```
#include <iostream>
using namespace std;

function prototypes
void simon(int);
double taxes(double);

function #1
int main()
{
   ...
  return 0;
}

function #2
void simon(int n)
{
  ...
}

function #3
double taxes(double t)
{
  ...
  return 2 * t;
}
```

User-Defined Function Form

- Give the return type in the function header and use return at the end of the function body
- Two return ways

```
// convert.cpp -- converts stone to pounds
#include <iostream>
int stonetolb(int);     // function prototype
int main()
{
    using namespace std;
    int stone;
    cout << "Enter the weight in stone: ";
    cin >> stone;
    int pounds = stonetolb(stone);
    cout << stone << " stone = ";
    cout << pounds << " pounds." << endl;
    return 0;
}

int stonetolb(int sts)
{
     return 14 * sts;
}
```




```
int stonetolb(int sts)
{
      int pounds = 14 * sts;
      return pounds;
}
```

Overview of Function

- Definition
  - It has a header
  - It has a body
    - ✓Multiple statements
  - It accepts an argument
  - It returns a value

- It requires a prototype

Review C++ statement types

- After introducing function definition
  - Declaration statement
  - Assignment statement
  - Function call – generally included in expression
  - Function prototype
  - Return statement

Summary

- What are the modules of C++ programs called? functions
- What does the preprocessor directive do?
- What does the namespace do?
- What does the header do?
- What is the structure of function?
- Where does the prototype put?
- Where does the program start to run?

……

Summary

- C++ provides two predefined objects: cout, cin
  - They are objects of the istream and ostream classes, in the iostream file
  - These classes view input and output as streams of characters
  - The insertion operator (<<), which is defined for the ostream class, lets you insert data into the output stream
  - The extraction operator (>>), which is defined for the istream class, lets you extract information from the input stream
- C++ can use the extensive set of C library functions
  - Inclusion statement: #include <cmath>
  - Power function: pow(double); pow(double, double);
  - Square root: sqrt(int);

Operators Overloading

Thanks

zhengf@sustech.edu.cn

http://faculty.sustech.edu.cn/fengzheng/