C/C++ Programming Language

CS205 Spring

Feng Zheng

Lecture 1

南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Content

- Self-Introduction
- About This Course
- Getting Started with C++
- Setting Out to C++

Self-Introduction

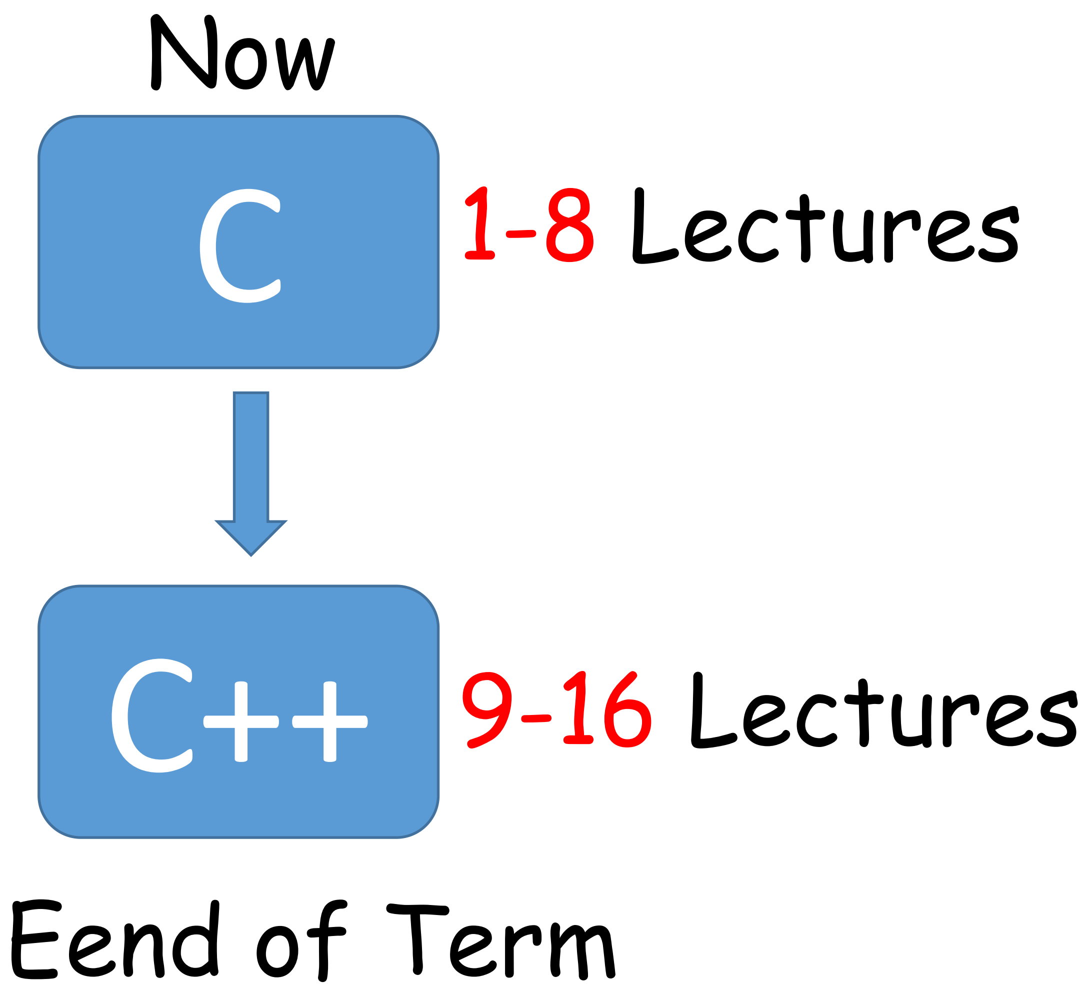
Office

- **Office :** #513 South Building of Engineering
- **Phone:** 0755-88015178
- **Email :** zhengf@sustech.edu.cn
- Hamepage: http://faculty.sustech.edu.cn/fengzheng/

About This Course

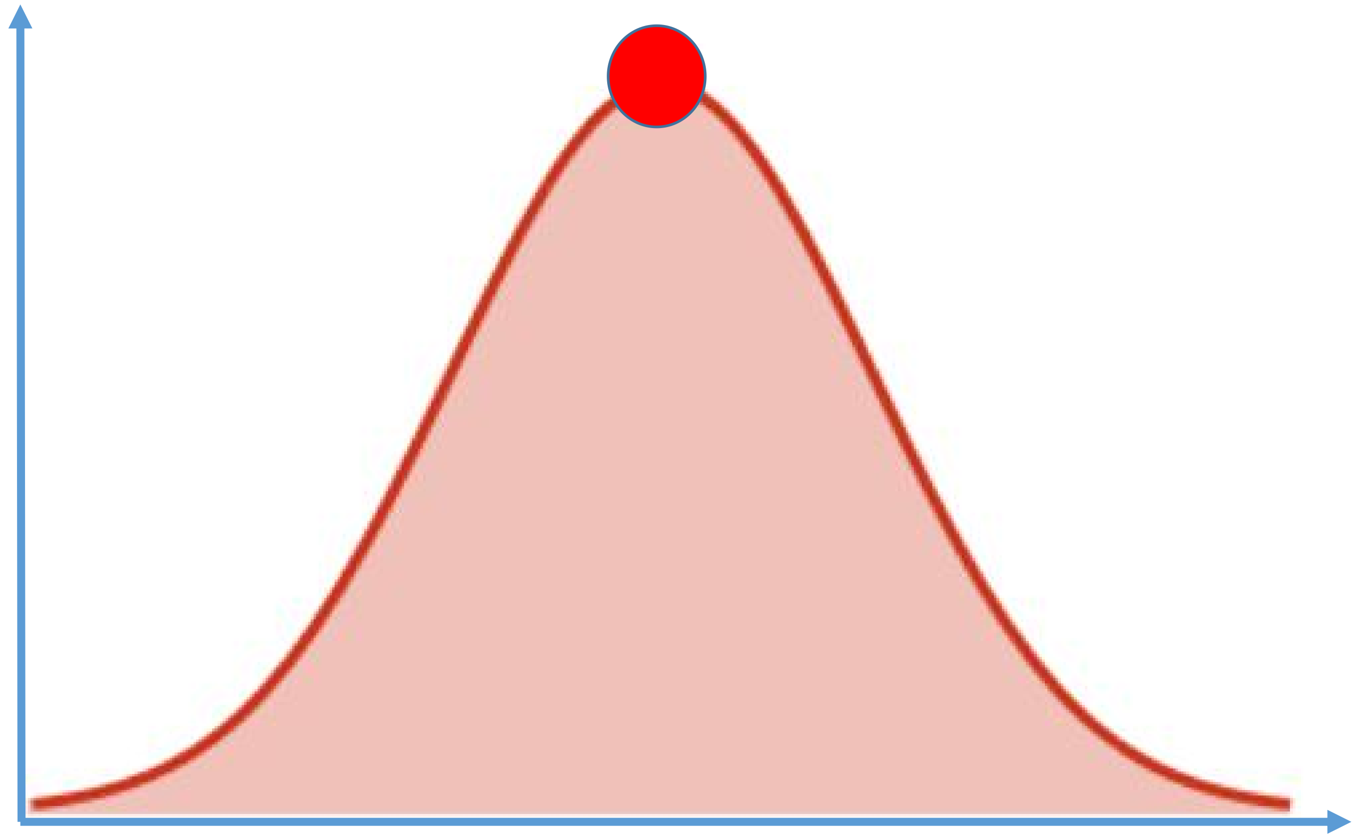
Structure

- C related part in C++.
- Class types related part.



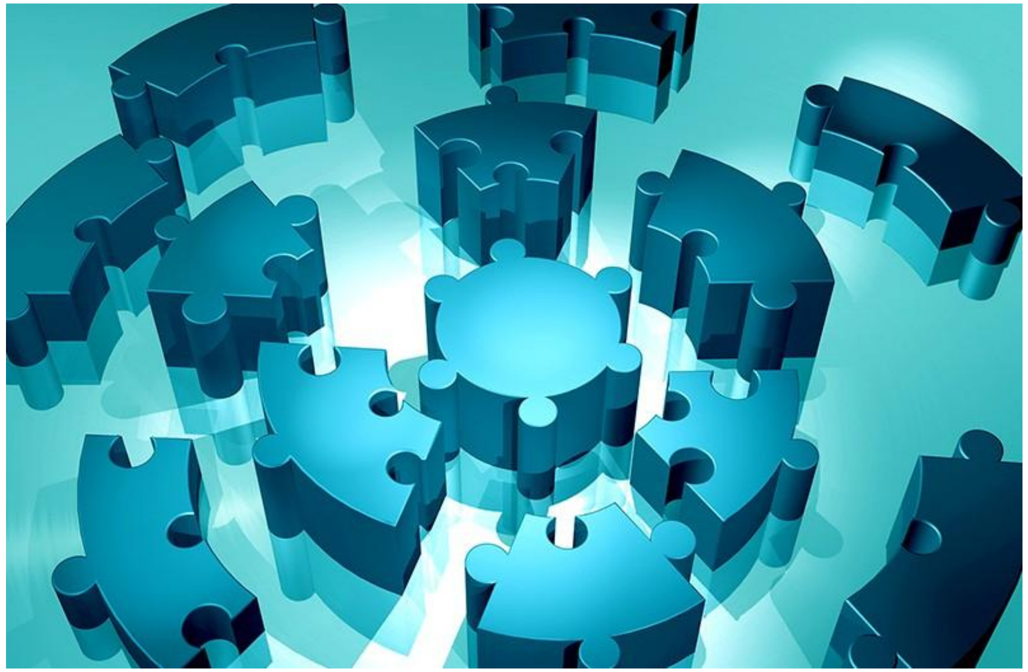
Target Student

- Average ability of programming



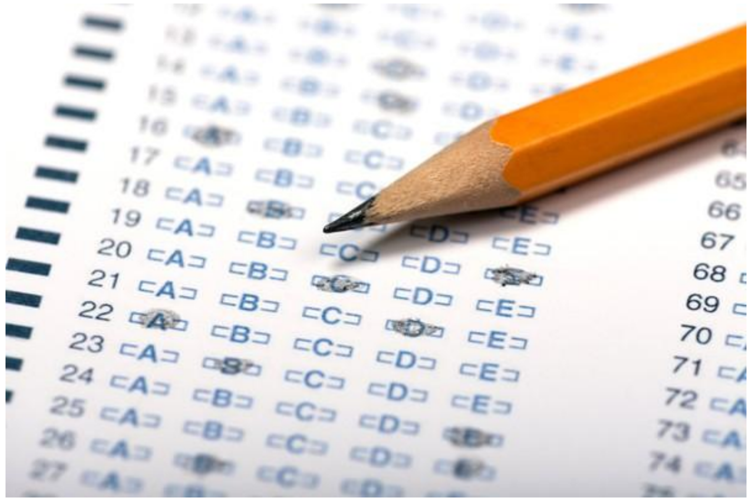
Expectations

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



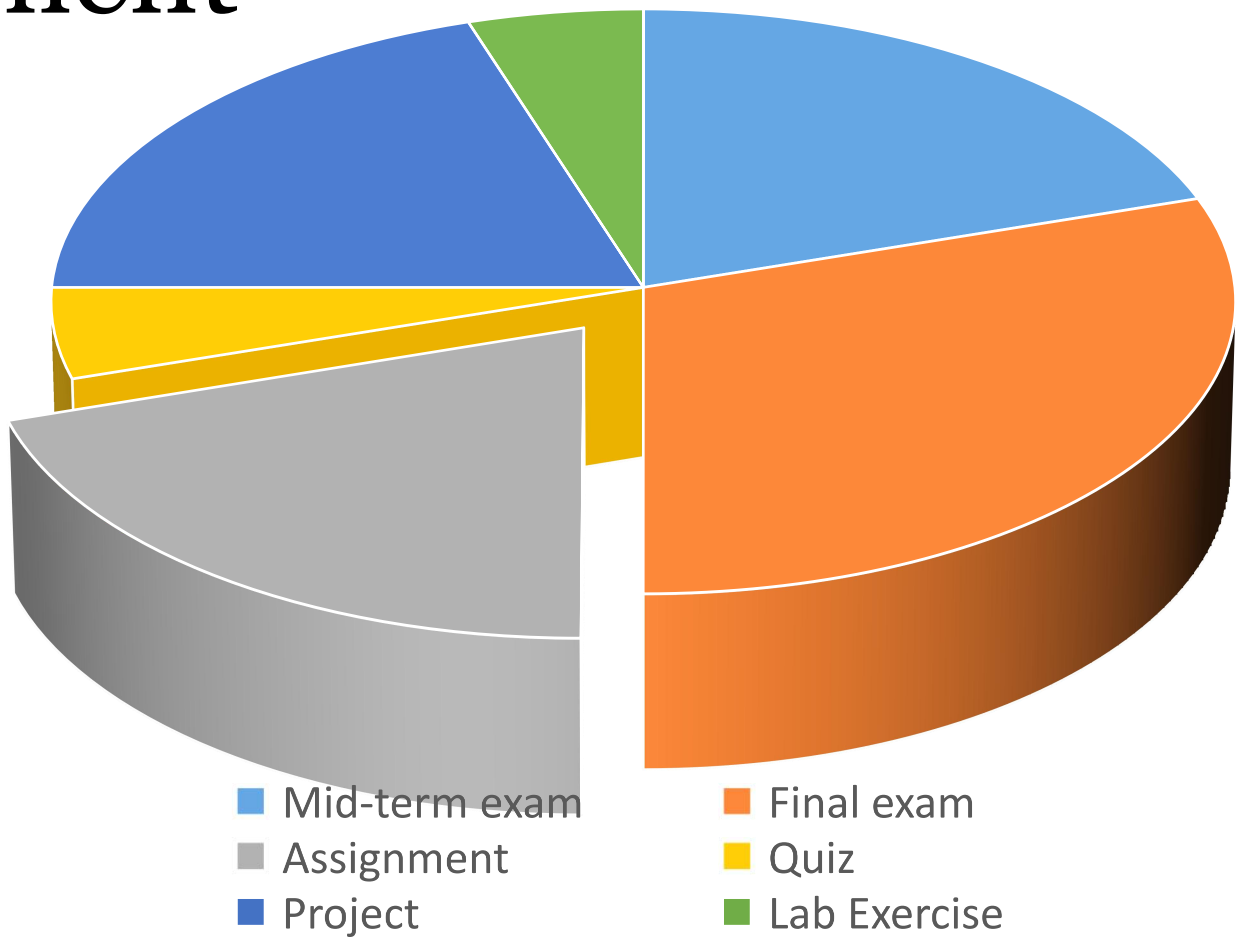
Exams test you on

- General knowledge about C/C++
- Ability to write pseudo-code for a moderately complex algorithm
- Being able to tell what a program does
- Finding errors in a program



Grade Component

- Mid-Term Exam: 20%
- Final Exam: 30%
- Assignment: 20%
- Project: 20%
- Lab Exercise: 5%
- Quiz: 5%



- Projects and assignments 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 or suggest that you are the author of something that you didn't write.

Group for CS219

Getting Started with C++

Content

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

Computer Languages

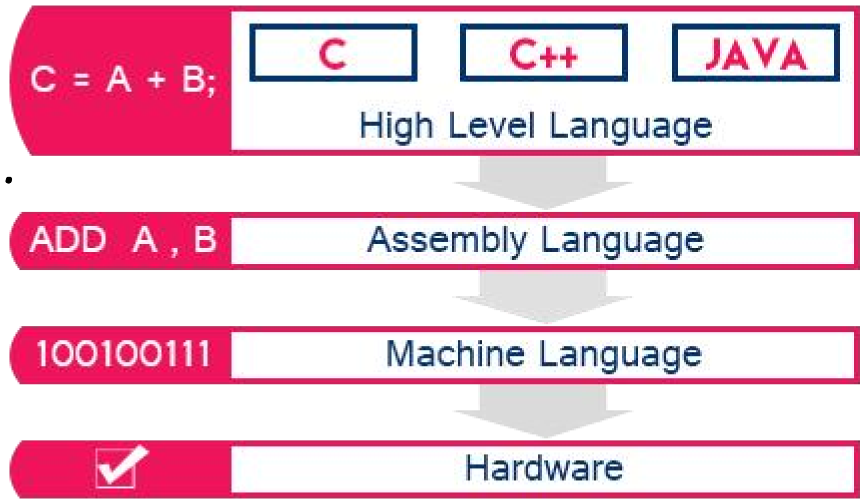
- Machine language
  - Only computer understands; Defined by hardware design; Strings of numbers; Instruct computers to perform elementary operations; Cumbersome for humans
  - **Example:**

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

Computer Languages

- 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:

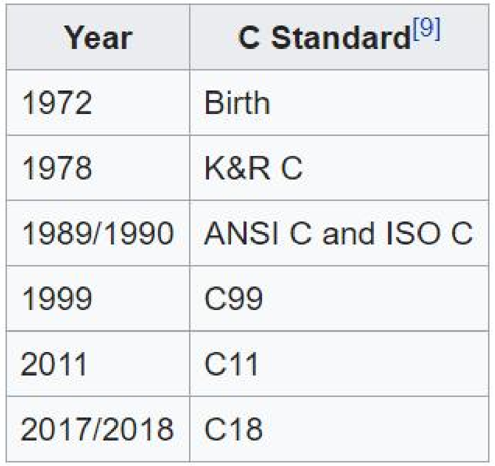
    grossPay = basePay + overTimePay
  - ➢ C/C++, JAVA, PYTHON, MATLAB,……



- Natural Language

History of C

- Evolved from two other programming languages
  - ➢ BCPL and B: "**Type**less" 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

C
Language

C Programming Philosophy

- Structured programming
  - Earlier Procedural programming
  - Branching statements
- Top-down
  - Divide large tasks into smaller tasks

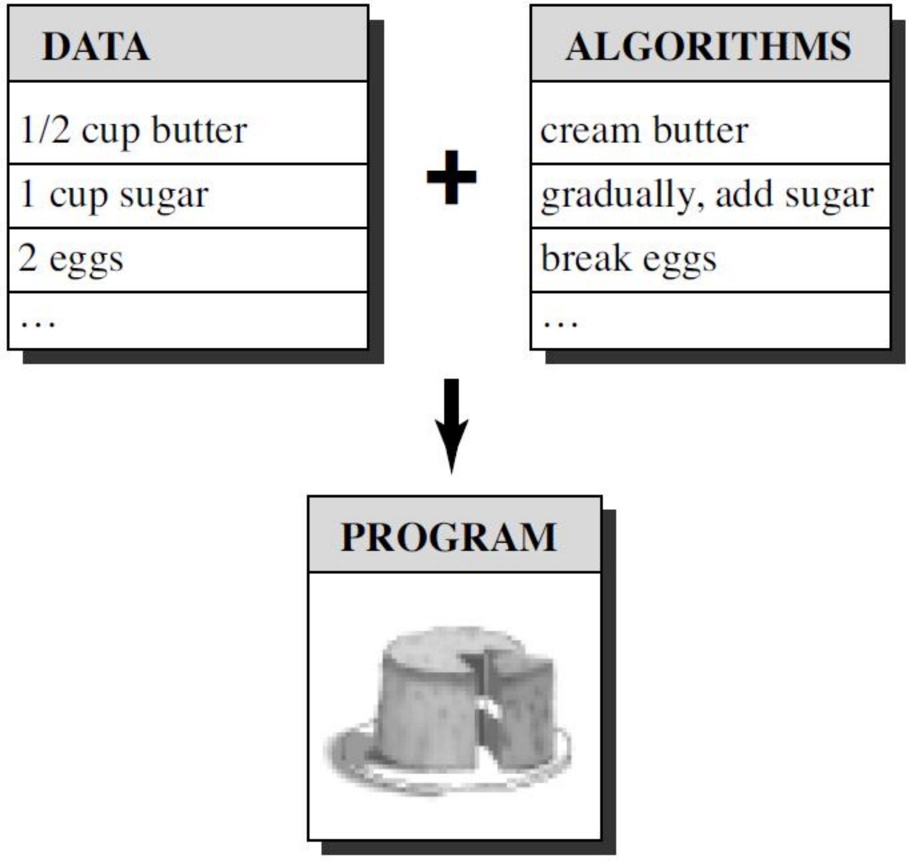
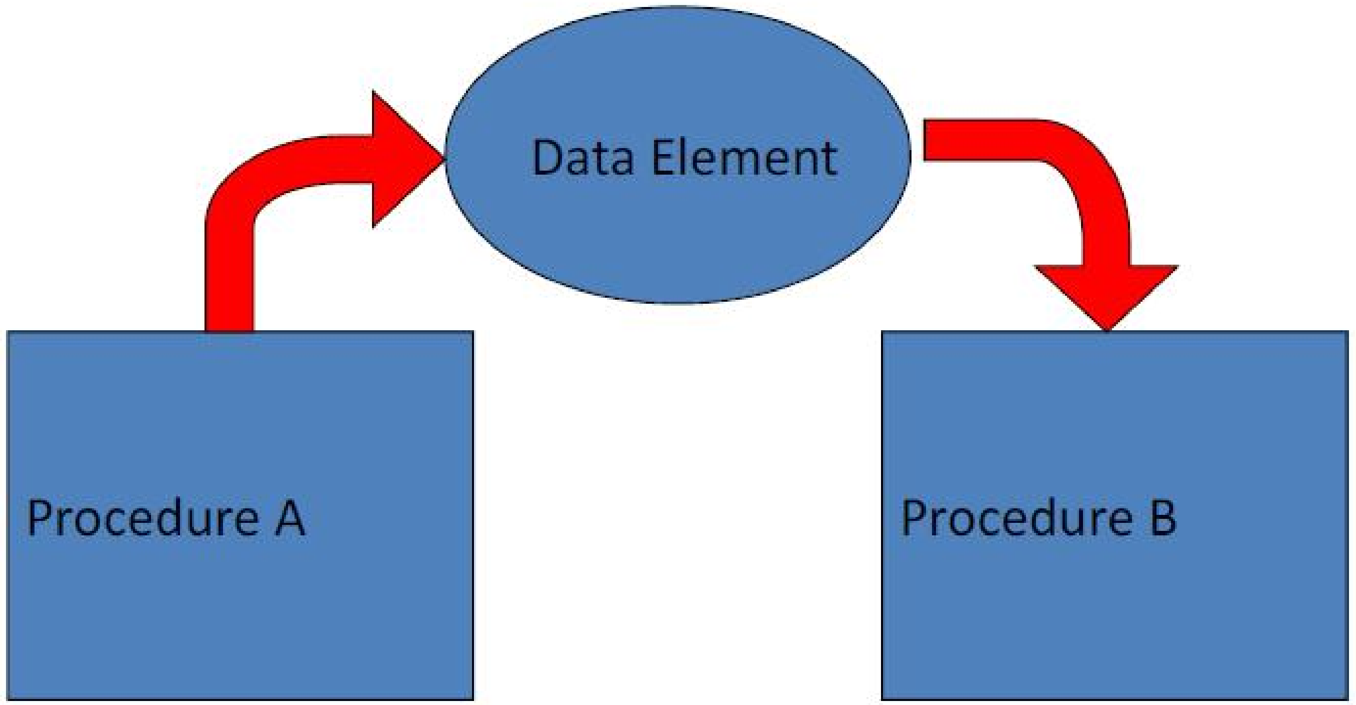


Figure 1.1 Data + algorithms = program.

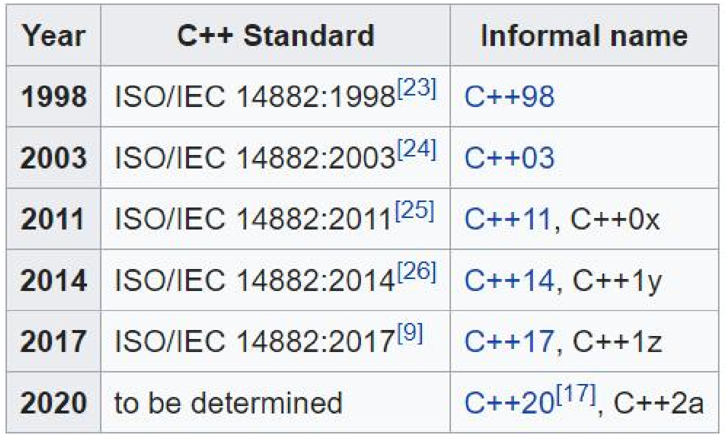
C Programming Philosophy

- Procedural programming --- Compared to OOP
  - Data and algorithms



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



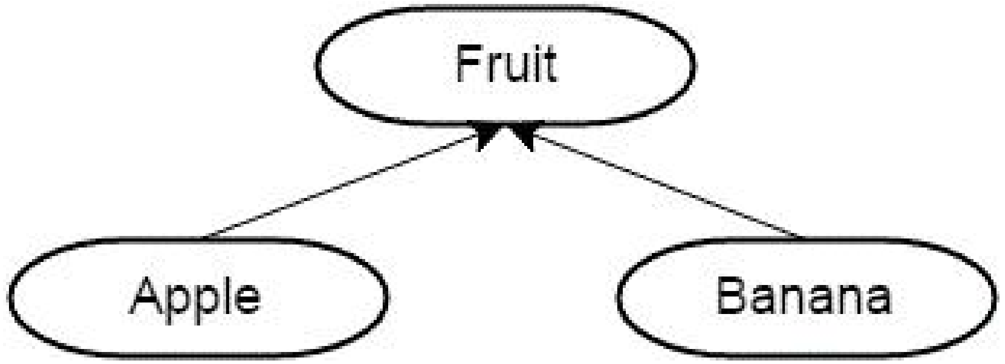
Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998[23]	C++98
2003	ISO/IEC 14882:2003[24]	C++03
2011	ISO/IEC 14882:2011[25]	C++11, C++0x
2014	ISO/IEC 14882:2014[26]	C++14, C++1y
2017	ISO/IEC 14882:2017[9]	C++17, C++1z
2020	to be determined	C++20[17], C++2a

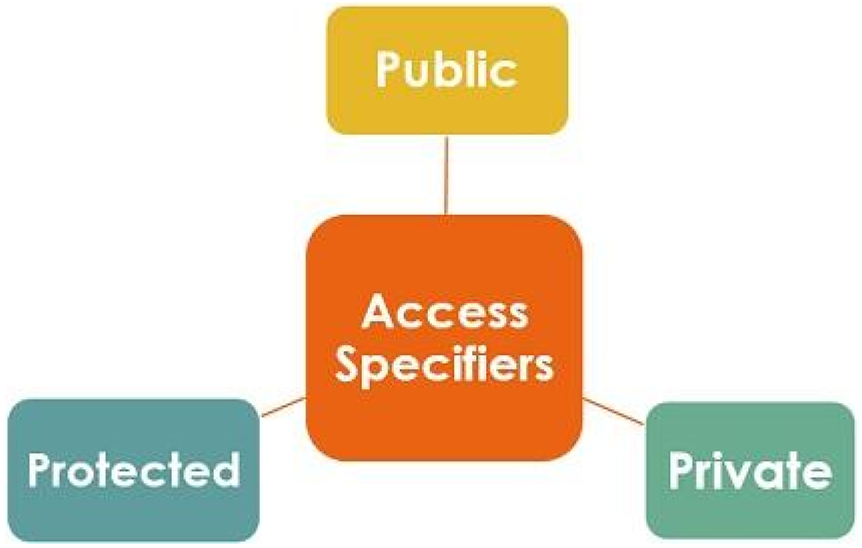
C++ Philosophy

- 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++

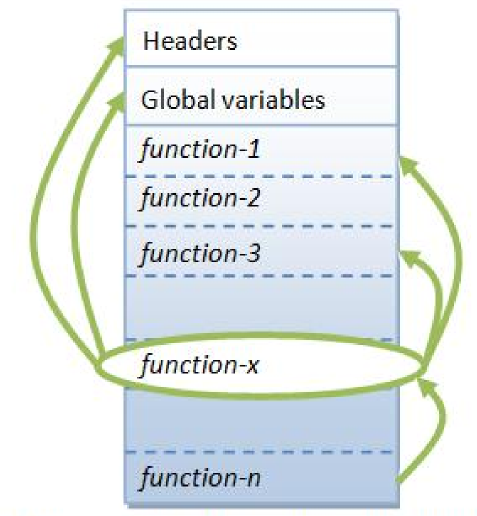
- Binding
- Reusable (可重用的)
- Protectability (可保护的)
- Polymorphism (多态性)
- Inheritance (继承性)
- Portable (可移植性)



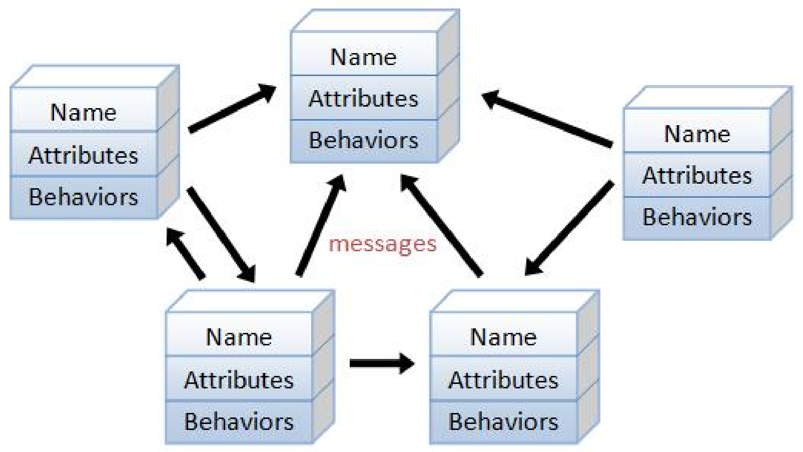


Comparison

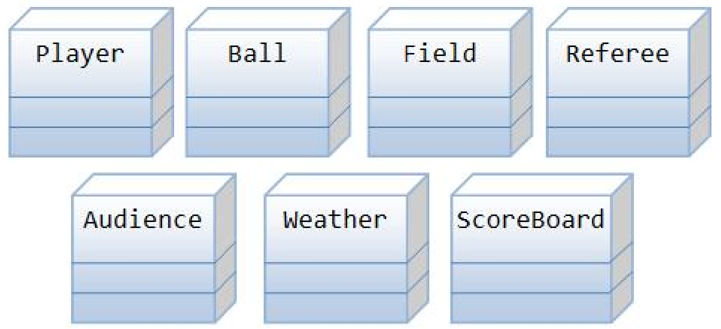
- Procedural versus Object-oriented (Encapsulated: 封装的)



A function (in C) is not well-encapsulated



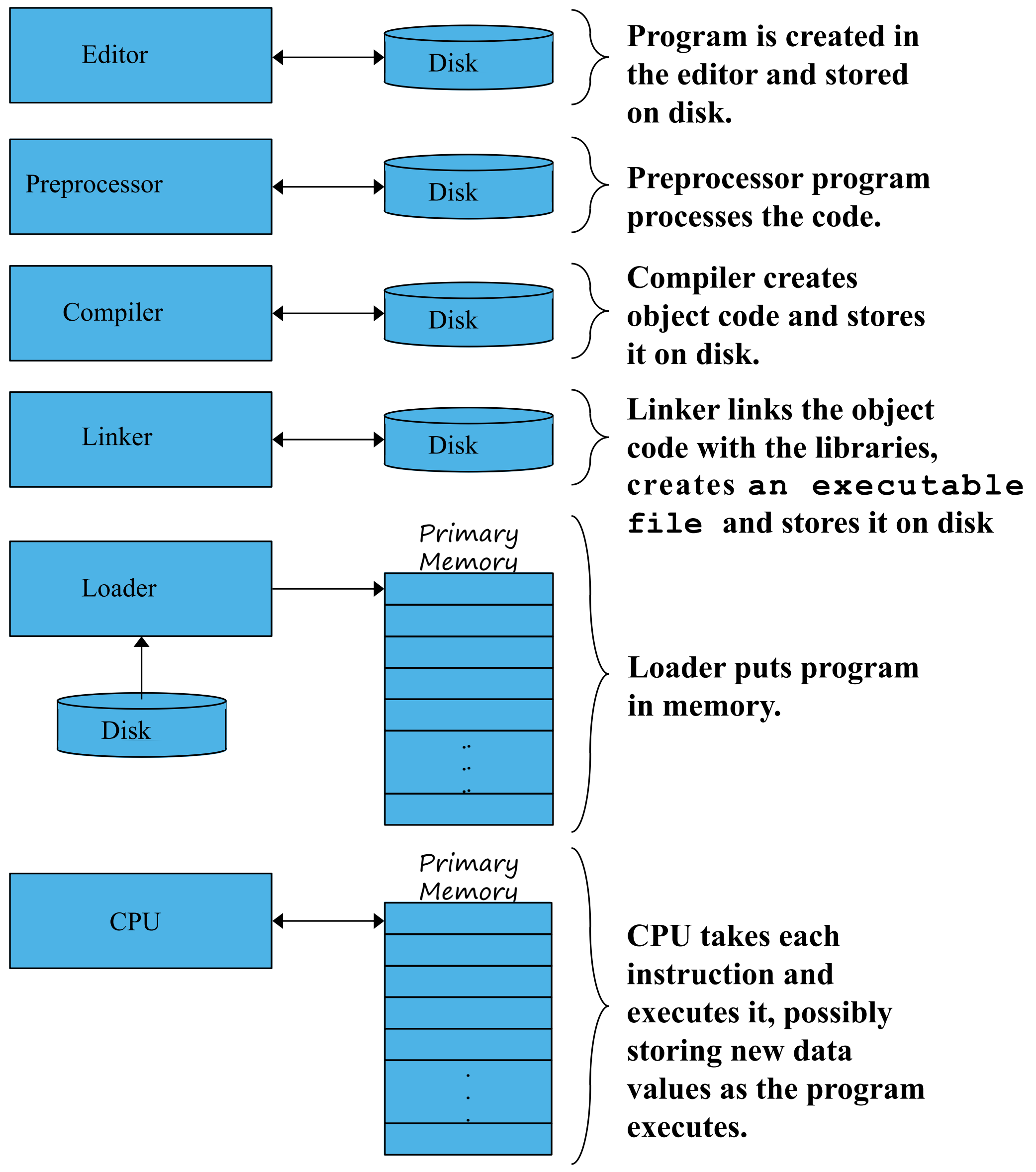
An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages



Classes (Entities) in a Computer Soccer Game

Program Phases

- Edit
- Preprocess
- Compile
- Link
- Load
- Execute

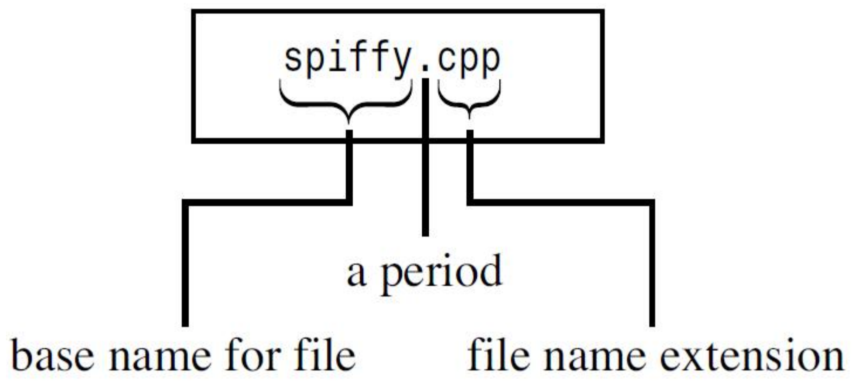

Editor
Disk
Program is created in the editor and stored on disk.
Preprocessor
Disk
Preprocessor program processes the code.
Compiler
Disk
Compiler creates object code and stores it on disk.
Linker
Disk
Linker links the object code with the libraries, creates an executable file and stores it on disk
Loader
Disk
Primary Memory
Loader puts program in memory.
CPU
Primary Memory
CPU takes each instruction and executes it, possibly storing new data values as the program executes.

Creating the Source Code File

- Integrated development environments
  - ➢ Microsoft Visual C++
  - ➢ QT
  - ➢ Apple Xcode
- Any available text editor
  - ➢ Debuggers: GDB: The GNU Project Debugger
  - ➢ Command prompt
  - ➢ Compiler

Proper Extensions

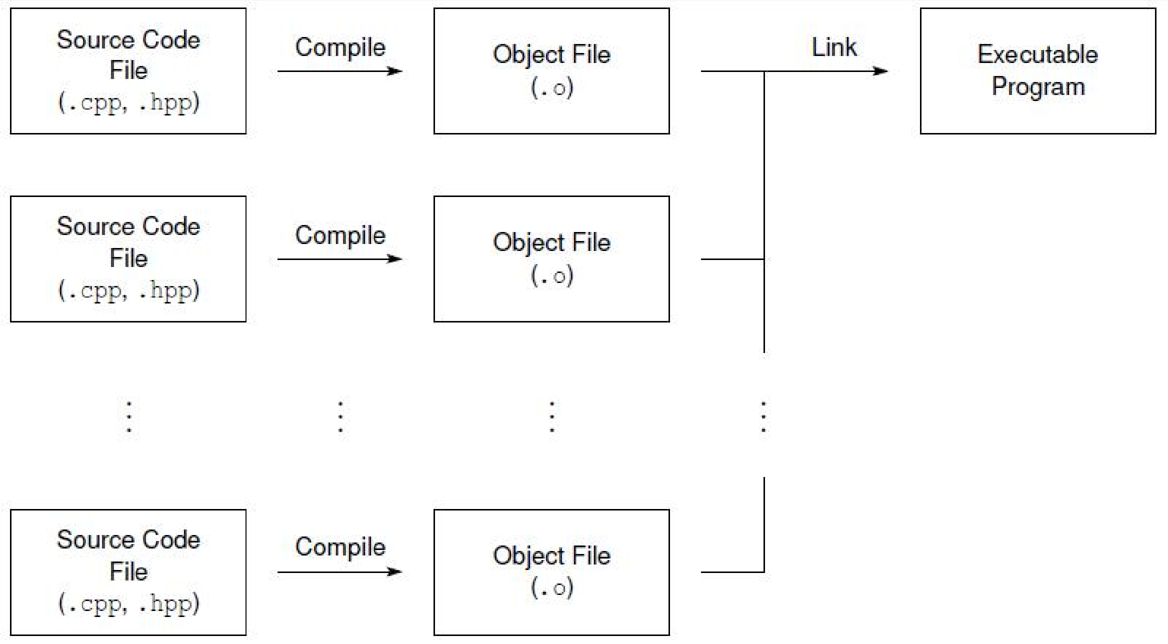
- Suffix



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
```

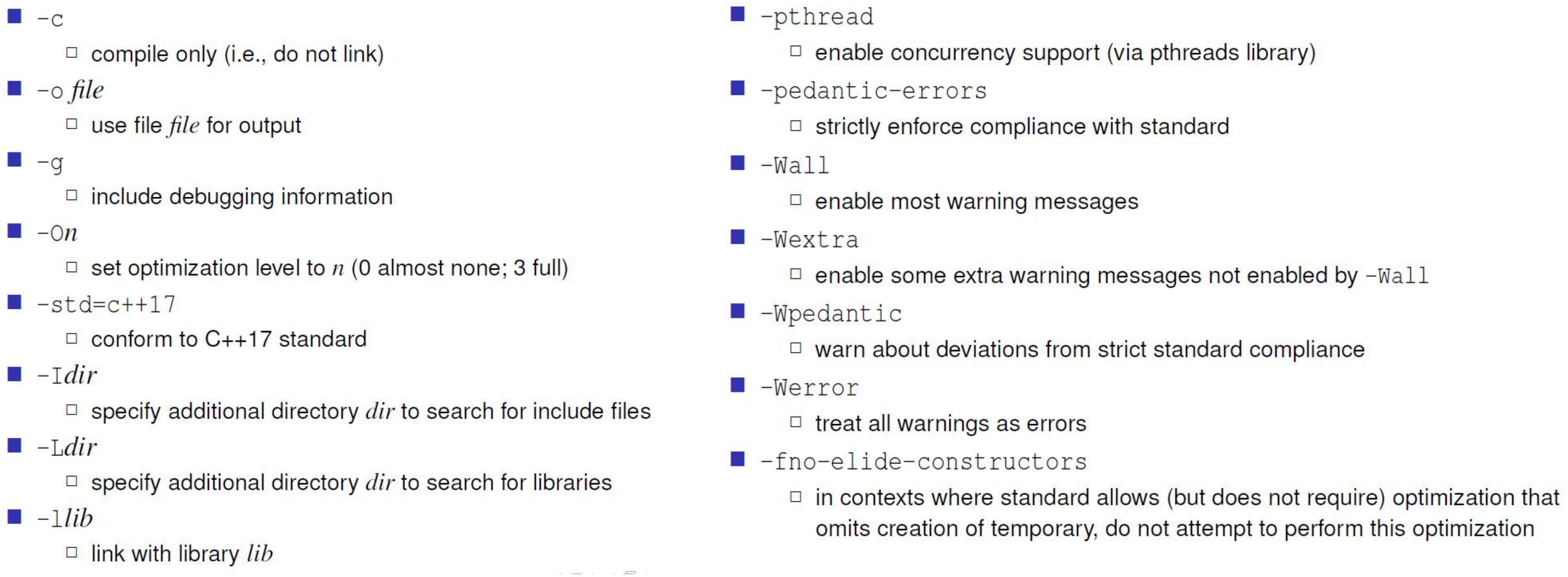
- Link object files *file 1.o*, *file 2.o*, . . . to produce executable file *executable_name*:

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

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

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

- Windows application: MFC Windows application, dynamic link library, ActiveX control, DOS or character-mode executable, static library, or console application
- Both 64-bit and 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

Setting Out to C++

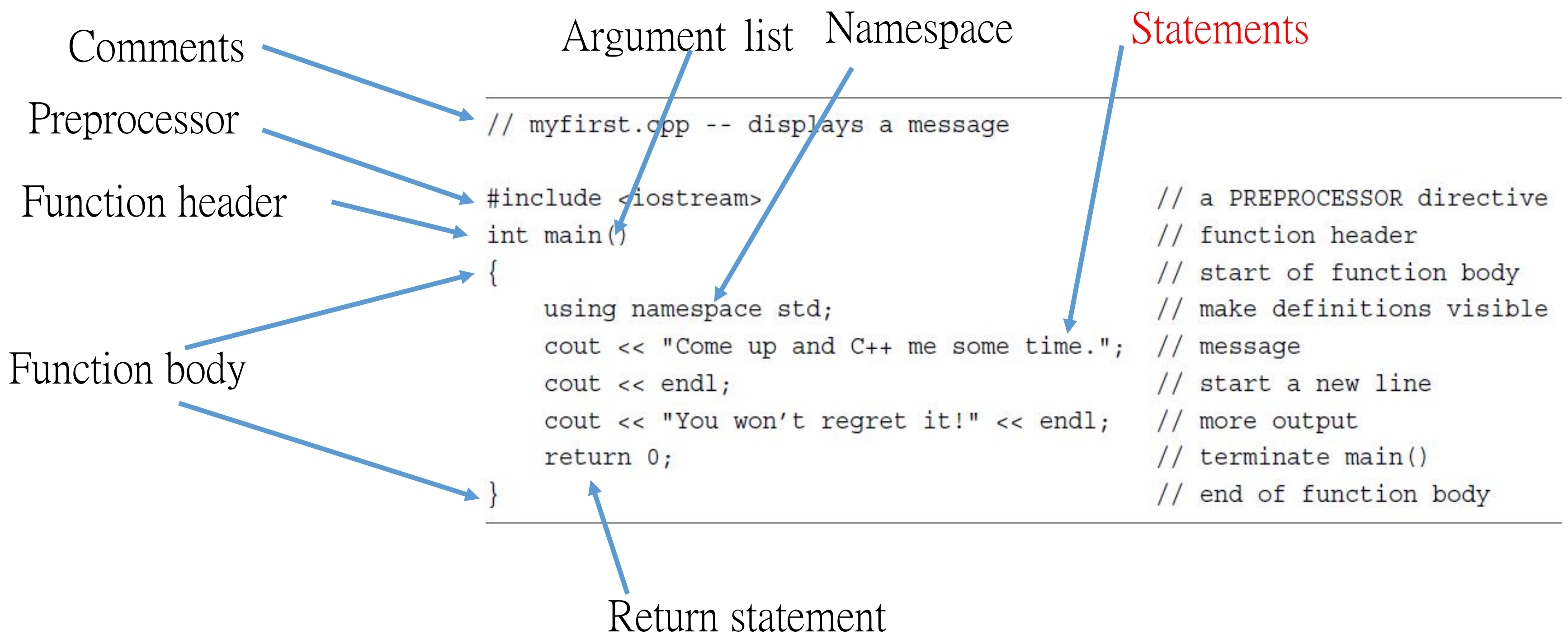
Content

- Creating a C++ program
- The general format for a C++ program
- The #include directive
- The main() function
- Using the cout object for output
- Placing comments in a C++ program
- How and when to use endl
- Declaring and using variables
- Using the cin object for input
- Defining and using simple functions

C++ Program Sample

- C++ is case sensitive

Header



```
// myfirst.cpp -- displays a message

#include <iostream>                            // a PREPROCESSOR directive
int main()                                     // function header
{                                              // start of function body
    using namespace std;                       // make definitions visible
    cout << "Come up and C++ me some time.";   // message
    cout << endl;                              // start a new line
    cout << "You won't regret it!" << endl;    // more output
    return 0;                                  // terminate main()
}                                              // end of function body
```

Comments (注释)

- The compiler ignores comments
- 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. */
```

Identifiers (标识符)

- Identifiers used to name entities such as: types, objects (i.e., variables), and functions
- Valid identifier is 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

```
☐ event_counter
☐ eventCounter
☐ sqrt_2
☐ f_o_o_b_a_r_4_2
```

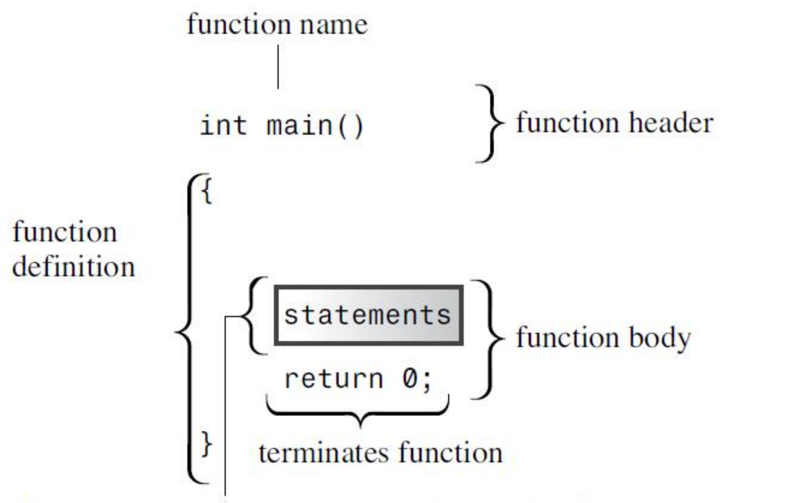
Keywords

- 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	

Features of the main() Function

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



```
int main()
{
    statements
    return 0;
}
```

Statements are C++ expressions terminated by a semicolon.

Features of the main() Function

- 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()
```

C++ Preprocessor(预处理)

- Source code transformed by preprocessor, prior to compliation
- Preprocessor output then passed to compiler for compilation
- Behavior can be controlled by preprocessor directives
- Preprocessor directive occupies single line of code
- 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
- Can be used to:
  - ☐ conditionally compile parts of source file
  - ☐ define macros and perform macro expansion
  - ☐ include other files
  - ☐ force error to be generated
- No semicolon (;→\)

Preprocessor: Source-File Inclusion

- Include contents of another file in source using preprocessor

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

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

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

Preprocessor: Defining Macros(宏)

- 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: #, ##
- Less readable

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

Preprocessor: Conditional Compilation

- Include code through use of if-elif-else construct
- Conditional preprocessing block consists of:

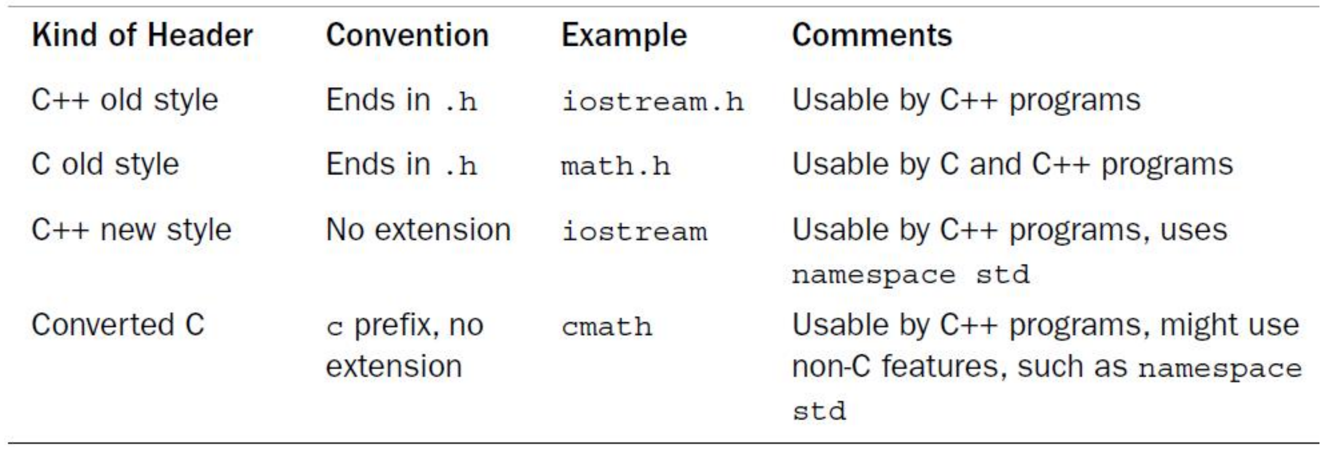
1. **#if**, **#ifdef**, or **#ifndef** directive
2. optionally any number of **#elif** directives
3. at most one **#else** directive
4. **#endif** directive

- Example:

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

Header Filenames

- Reason
  - 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

- Reason
  - To simplify the writing of large programs and of programs that combine pre-existing code from several vendors and to help organize programs
  - To indicate which vendor's product (wanda) you want

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

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

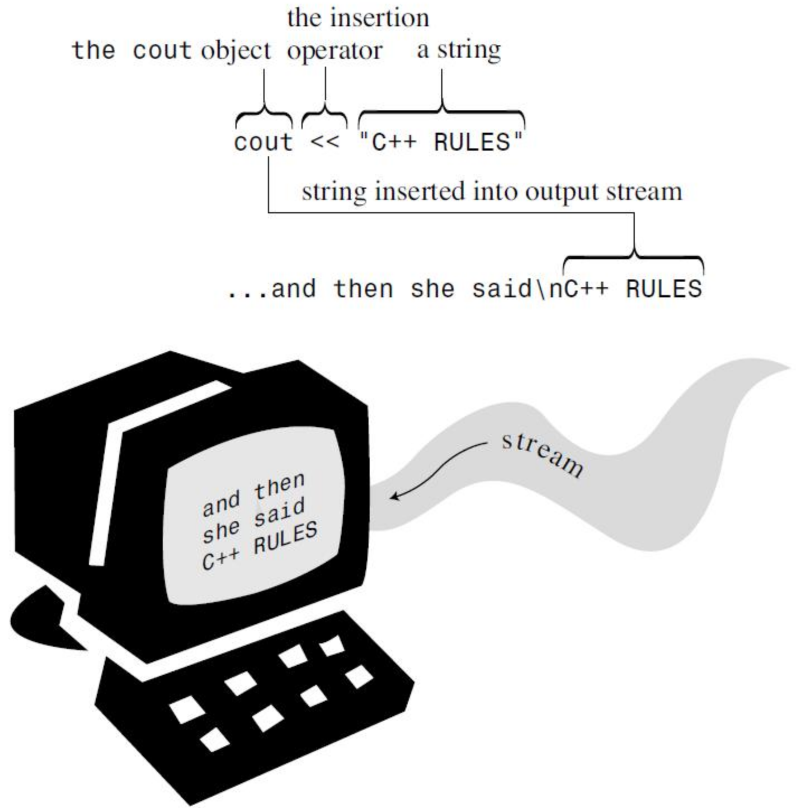
```
std::cout << "Come up and C++ me some time.";
std::cout << std::endl;
```

```
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

- cout: is an object defined in stream
- String: double quotation marks
- Insertion operator: <<
- Manipulator: endl \n



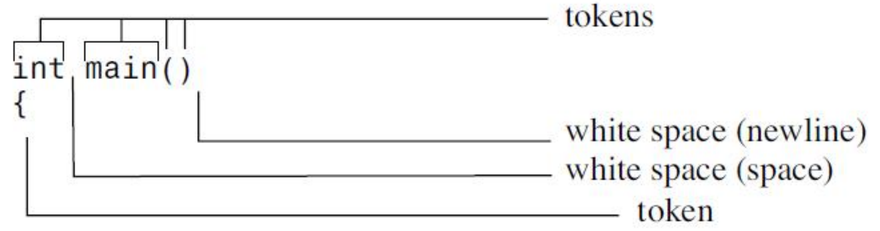
the cout object
the insertion operator
a string
cout << "C++ RULES"
string inserted into output stream
...and then she said\nC++ RULES
and then she said C++ RULES
stream

```
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
```

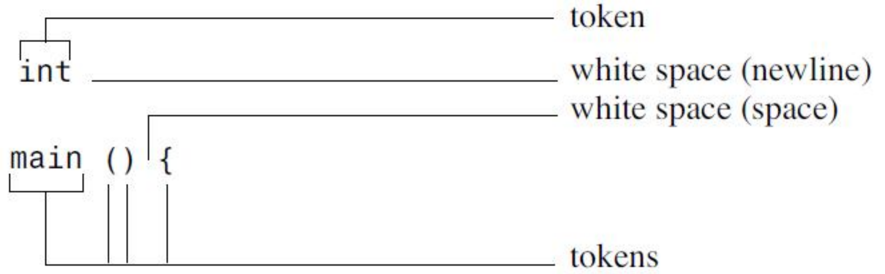
C++ Source Code Formatting

- Semicolon marks the end of each statement
  - Spread a single statement over several lines
  - Place several statements on one line
- Tokens – indivisible elements in a line of code
- White space – a space, tab, or carriage return

```
#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; }
```



Spaces and carriage returns can be used interchangeably.



C++ Source Code Formatting

- 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 ( )
```

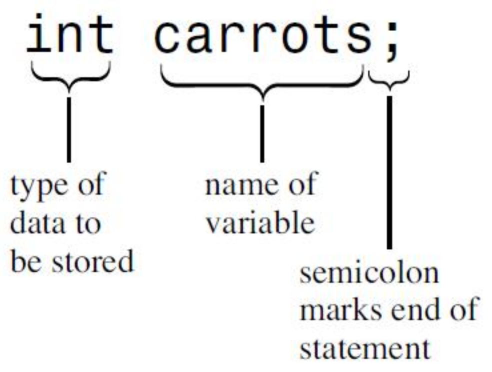
C++ Statements

- A program is a collection of functions
- Each function is a collection of statements
  - A declaration statement creates a variable
  - 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;
}
```

Declaration Statements and Variables

- Identify both the storage location and how much memory storage 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: =
- Arithmetic expression: -

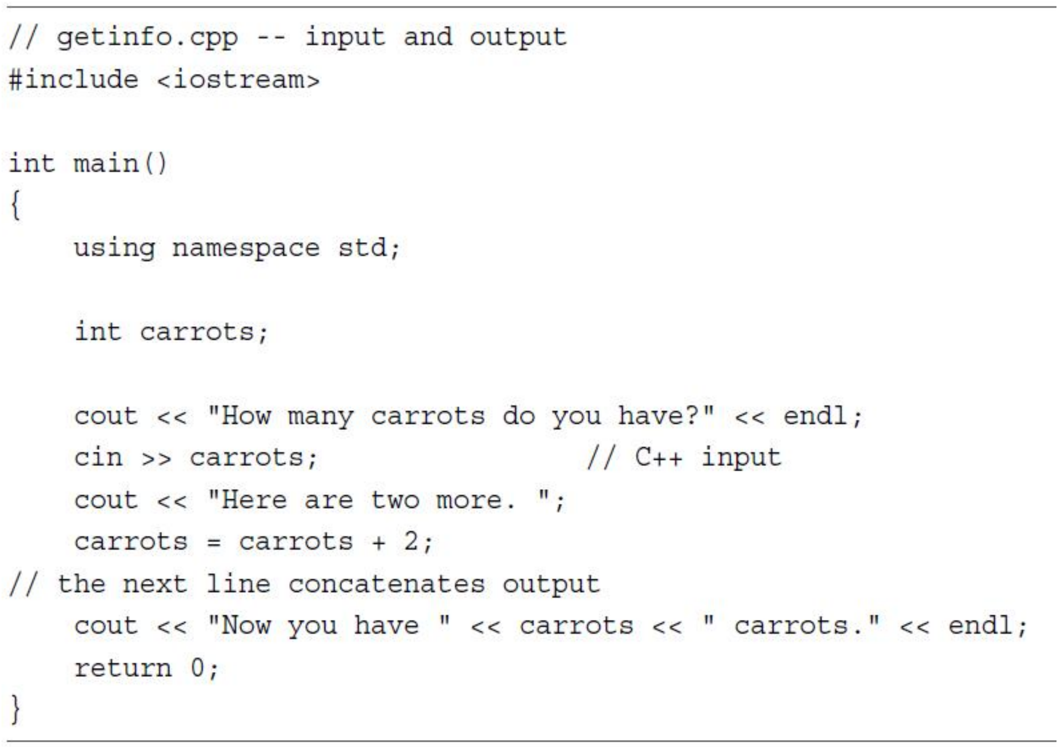
```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

```
int carrots;

carrots = 25;

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

- Can be assigned by a returned value by a function

Assignment: cin

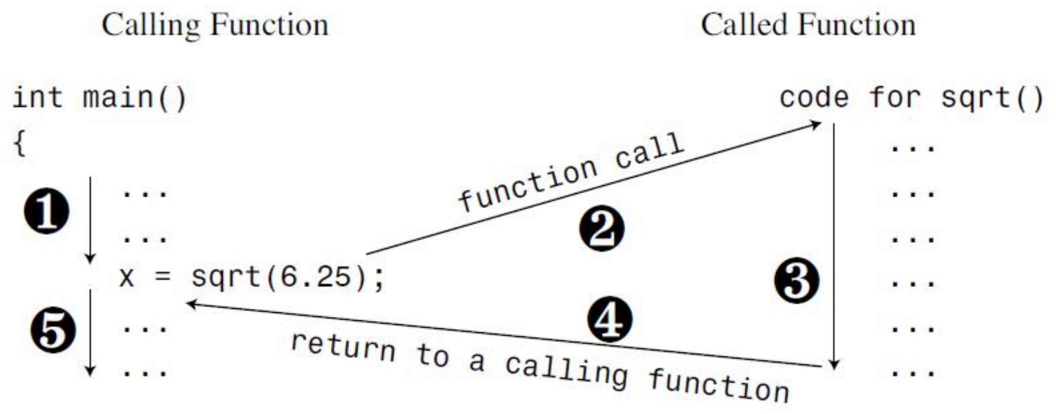
- >> operator: extract characters from the input stream
- The value typed from the keyboard is eventually assigned to the variable carrots

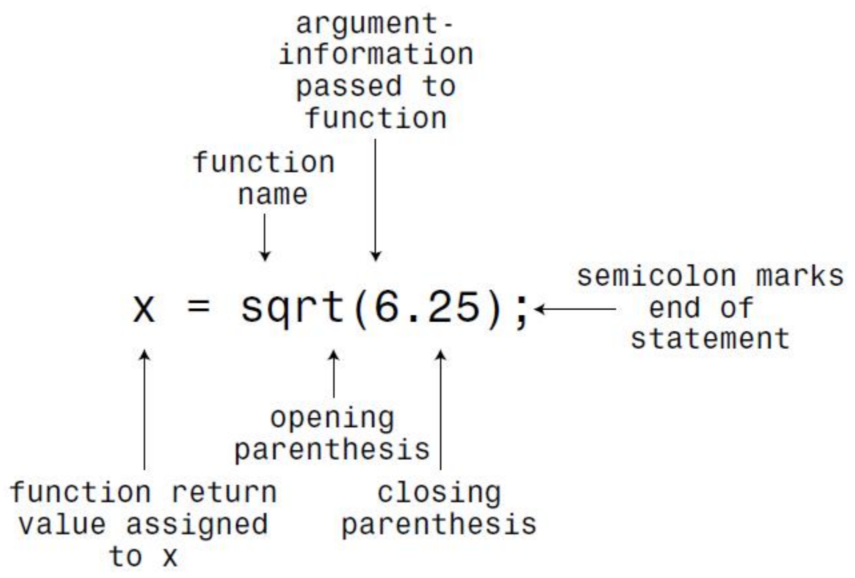


```
// getinfo.cpp -- input and output
#include <iostream>

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;
}
```

Basic characteristics of functions

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




argument-information passed to function
function name
x = sqrt(6.25);
semicolon marks end of statement
opening parenthesis
function return value assigned to x
closing parenthesis

Basic characteristics of functions

- A function prototype does for functions what a variable declaration does for variables

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

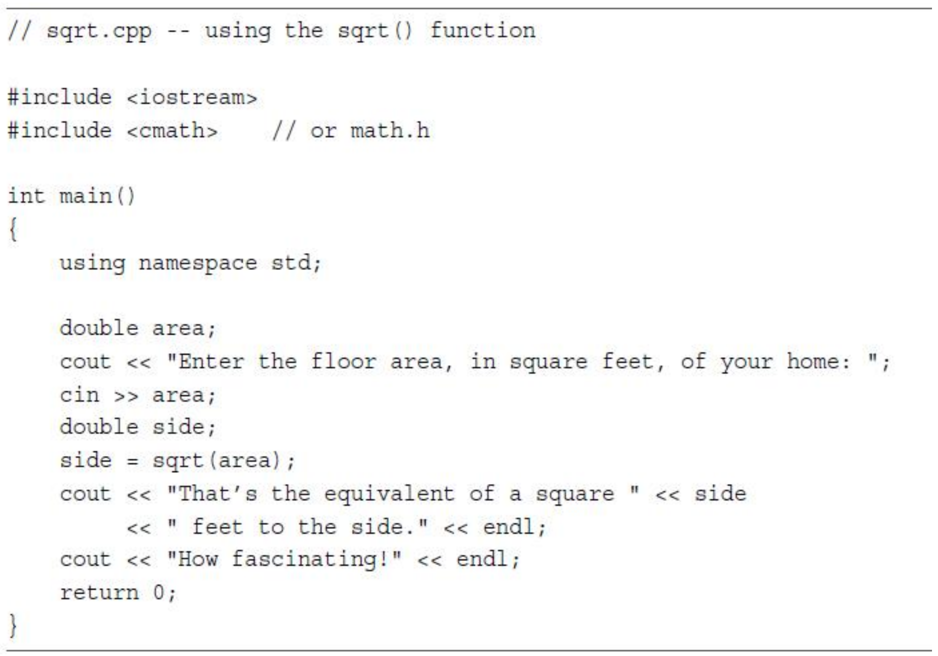
- The terminating semicolon in the prototype identifies it as a statement and thus makes it a prototype instead of a function header.
  - You can type the function prototype into your source code file yourself.
  - You can include the cmath (math.h on older systems) header file, which has the prototype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

Basic characteristics of functions

- Don't confuse the function prototype with the function definition
  - Function prototype describes the function interface
  - The definition includes the code for the function's workings
- Place a function prototype ahead of where you first use the function
- The library files contain the compiled code for the functions, whereas the header files contain the prototypes.

Basic characteristics of functions

- Demonstrate 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

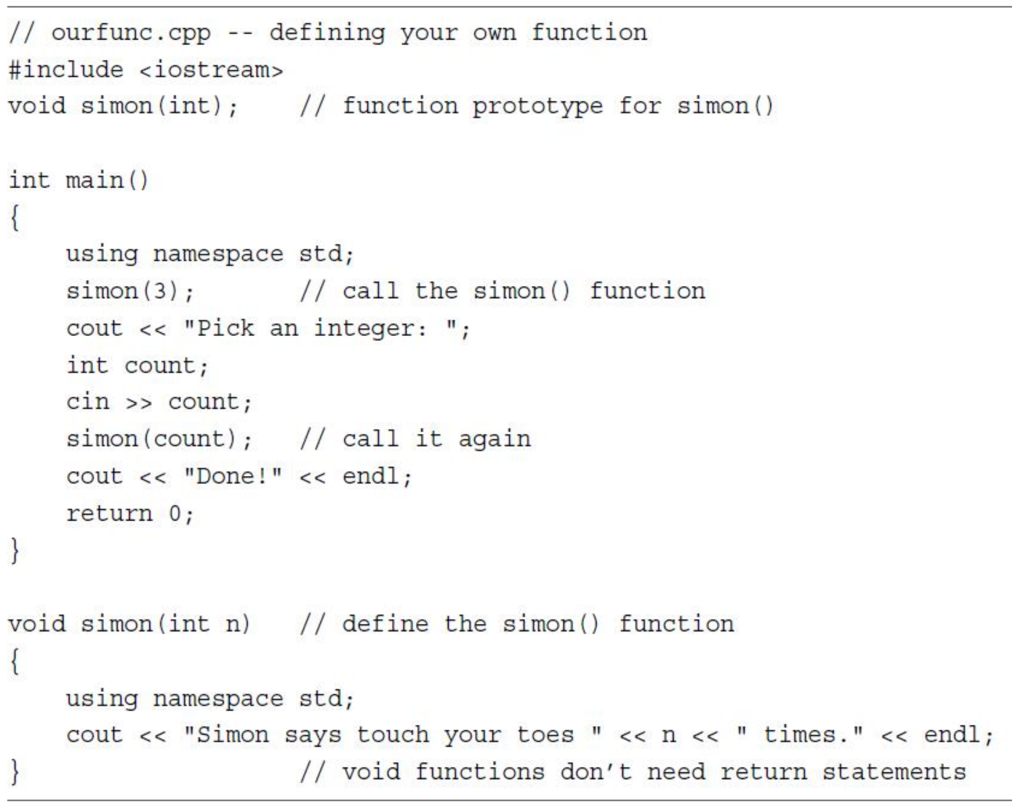
```
int rand(void);          // prototype of a function that takes no arguments
myGuess = rand();         // function call with no arguments
```

- 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

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

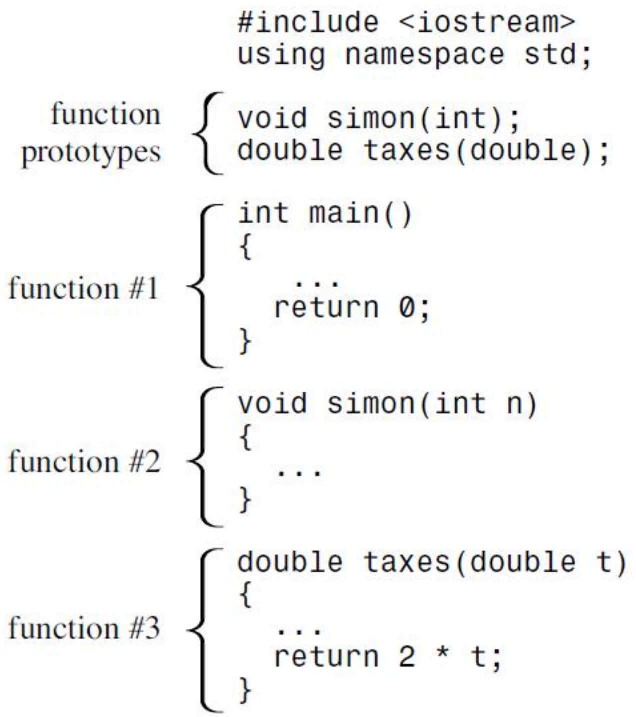


```
// ourfunc.cpp -- defining your own function
#include <iostream>
void simon(int);    // function prototype for simon()

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
```

Function Form

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

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



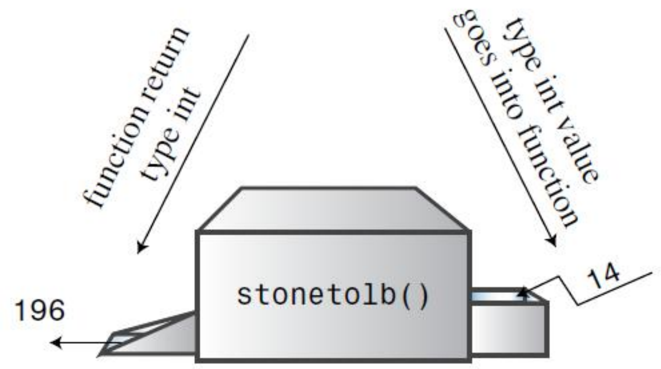
```
#include <iostream>
using namespace std;

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

int main()
{
   ...
   return 0;
}

void simon(int n)
{
   ...
}

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

User-Defined Function That Has a Return Value

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

```
// 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);
```



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

Function

- It has a header and a body
- It accepts an argument
- It returns a value
- It requires a prototype

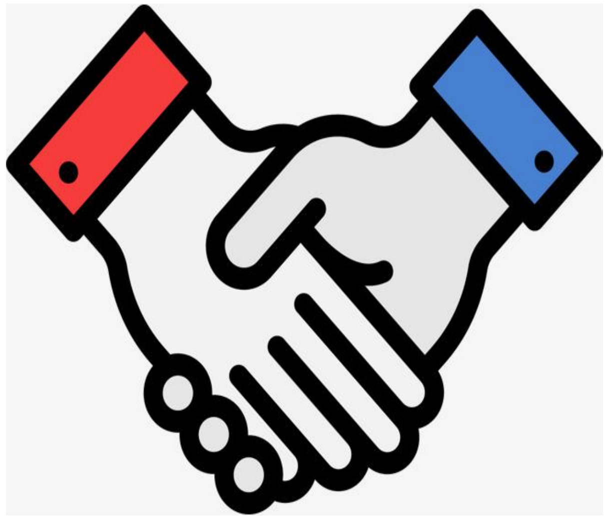
C++ statement types

- Declaration statement
- Assignment statement
- Message statement
- Function call
- Function prototype
- Return statement

Summary

- What are the modules of C++ programs called?
- 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?

……



Thanks

zhengf@sustech.edu.cn