

CS241 Principles and Practice of Problem Solving

Lecture 1: Review on C++ (I)

Yuye Ling, Ph.D.

Shanghai Jiao Tong University
John Hopcroft Center for Computer Science

September 10, 2019

Copyright Notice

A large portion of the contents in the following pages come from Prof. Bjarne Stroustrup's slides.
The original contents could be found [here](#).

Programs

- Introduction

- GCC and makefile

Computation

- Definition, objectives and tools

- Codes: expressions, statements, and functions

Errors

- Error types

- Error handling: conventional vs exception

- Debugging

- Pre-conditions and post-conditions

Programs

Introduction

GCC and makefile

Computation

Definition, objectives and tools

Codes: expressions, statements, and functions

Errors

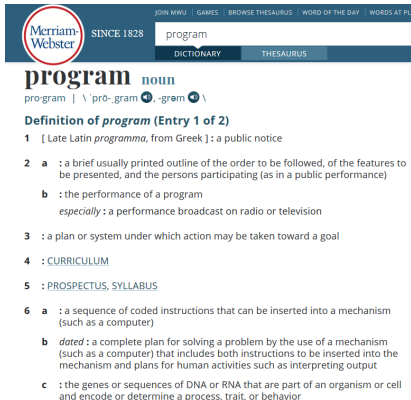
Error types

Error handling: conventional vs exception

Debugging

Pre-conditions and post-conditions

What is a program?



Merriam-Webster SINCE 1828

program

DICTIONARY THESAURUS

program *noun*

pro·gram | \ 'prō-,gram, -grəm \

Definition of *program* (Entry 1 of 2)

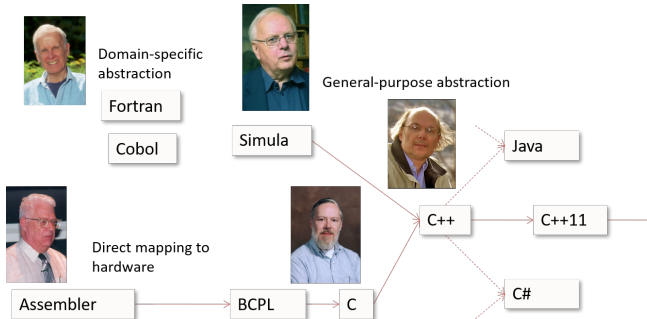
- [Late Latin *programma*, from Greek]: a public notice
- a brief usually printed outline of the order to be followed, of the features to be presented, and the persons participating (as in a public performance)
 - the performance of a program
especially : a performance broadcast on radio or television
- : a plan or system under which action may be taken toward a goal
- : [CURRICULUM](#)
- : [PROSPECTUS](#), [SYLLABUS](#)
- a sequence of coded instructions that can be inserted into a mechanism (such as a computer)
 - dated* : a complete plan for solving a problem by the use of a mechanism (such as a computer) that includes both instructions to be inserted into the mechanism and plans for human activities such as interpreting output
 - the genes or sequences of DNA or RNA that are part of an organism or cell and encode or determine a process, trait, or behavior

What does that mean in CS

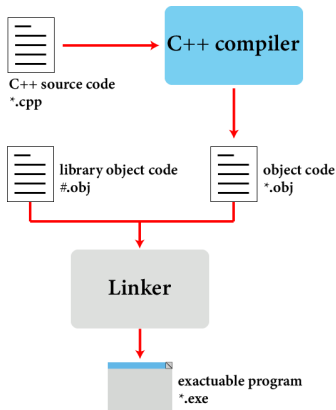
- ▶ Telling a very fast moron (aka computer) exactly what to do
- ▶ A plan for solving a problem on a computer
- ▶ Specifying the order of a program execution
 - ▶ Millions of lines of code
 - ▶ Data manipulation is key

What is programming languages?

We want to let the computer understand the “program” scripted by the people



How does the computer digest the program?



1. You write C++ source code
 - ▶ Source code is (in principle) human readable
2. The compiler translates what you wrote into object code
3. The linker links your code to system code needed to execute
4. The result is an executable program
 - ▶ E.g., a .exe file on windows or an a .out file on Unix

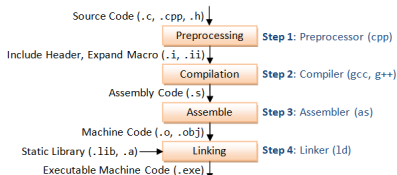
GNU Compiler Collection (GCC)

- Pre-processing
`cpp hello.c > hello.i`

- Compilation
`gcc -S hello.i`

- Assembly
`as -o hello.o hello.s`

- Linker
`ld -o hello.exe hello.o
...libraries...`



GNU Make

The compilation will become crazily complicated and tedious if the project grows large

1. You have to type the commands every time whenever you want to build the program
2. Recompiling the whole project could be time-consuming

```
1 //hello.c
2 #include <stdio.h>
3
4 int main(){
5     printf("Hello , world!\n");
6     return 0;
7 }
8
```

```
all: hello.exe
hello.exe: hello.o
    gcc -o hello.exe hello.o
hello.o: hello.c
    gcc -c hello.c
clean:
    rm hello.o
```

makefile

Syntax

- ▶ `$@`: the target filename.
- ▶ `$*`: the target filename without the file extension.
- ▶ `$<`: the first prerequisite filename.
- ▶ `^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- ▶ `+`: similar to `^`, but includes duplicates.
- ▶ `?`: the names of all prerequisites that are newer than the target, separated by spaces.

```
all: hello.exe
hello.exe: hello.o
    gcc -o hello.exe hello.o
hello.o: hello.c
    gcc -c hello.c
clean:
    rm hello.o
```

```
all: hello.exe
hello.exe: hello.o
    gcc -o $@ $<
hello.o: hello.c
    gcc -c $<
clean:
    rm hello.o
```

Programs

Introduction

GCC and makefile

Computation

Definition, objectives and tools

Codes: expressions, statements, and functions

Errors

Error types

Error handling: conventional vs exception

Debugging

Pre-conditions and post-conditions

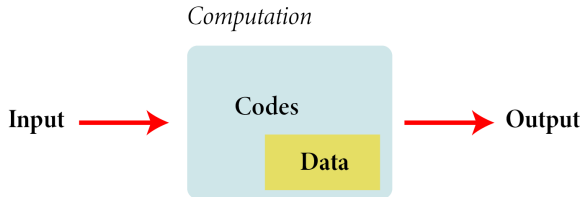
What is computation?



How can we “use or operate” a computer?

- ▶ Input
- ▶ Output

The composition of computation



Computation – what our program will do with the input to produce the output

- ▶ Codes
 - ▶ Expressions, statements, functions
- ▶ Data

Objectives and tools

Our job is to express computations

- ▶ correctly
- ▶ simply
- ▶ and efficiently

Tools available

- ▶ Abstraction: provide a higher-level concept that hides detail
- ▶ Divide and conquer: break up big computations into many little ones
- ▶ External libraries: Do not reinvent the wheel

Expressions

Expressions are made out of operators and operands

1. **Operators** specify what is to be done
2. **Operands** specify the data for the operators to work with

```
1 // compute area:
2 int length = 20;
3 int width = 40;
4 int area = length * width;
5 int average = (length + width) / 2;
6
```

Concise operators:

- ▶ $a += c$ means $a = a + c$
- ▶ $a *= \text{scale}$ means $a = a * \text{scale}$
- ▶ $++a$ means $a += 1$
- ? Which one is better?

Statements

A statement is

1. an expression followed by a semicolon, or
2. a declaration, or
3. a “control statement” that determines the flow of control

```
1 a = b;  
2 double d2 = 2.5;  
3 if (x == 2) y = 4;  
4 while (cin >> number) numbers.push_back(number);  
5 int average = (length+width)/2;  
6 return x;  
7
```


Selection

Sometimes we must select between alternatives

- ▶ if-else

```
1 if (condition-1)
2     statement-1 // if the condition-1 is true, do statement-1
3 else if (condition-2)
4     statement-2 // if the condition-2 is true, do statement-2
5 else
6     statement-3 // if not, do statement-3
7
```

- ? Whatelse

- ▶ case-switch

Iteration

Question: how can we politely ask a robot to do 100 push-ups?

- ▶ Divide and conquer \Rightarrow the atomic operation: do one push-up
a time \Rightarrow we can make this operation a “function”
- ▶ We need a counter:
 1. Reset, initialize the counter
 2. Start, increment the counter
 3. Stop, set the termination criterion
- ▶ Two implementations
 1. while
 2. for

Functions

Why do we need functions?

We define a function when we want to separate a computation because it

- ▶ is logically separate
- ▶ makes the program text clearer (by naming the computation)
- ▶ is useful in more than one place in our program
- ▶ eases testing, distribution of labor, and maintenance

Programs

Introduction

GCC and makefile

Computation

Definition, objectives and tools

Codes: expressions, statements, and functions

Errors

Error types

Error handling: conventional vs exception

Debugging

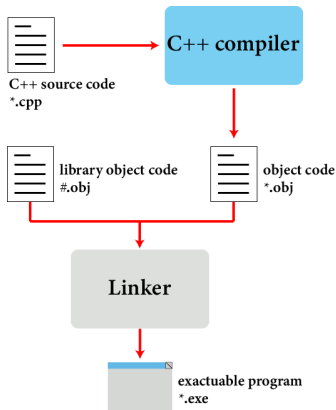
Pre-conditions and post-conditions

Errors

When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?

1. Organize software to minimize errors.
2. Eliminate most of the errors we made anyway. Through Debugging and Testing.
3. Make sure the remaining errors are not serious.

Error types



1. Compile-time errors
2. Link-time errors
3. Run-time errors
4. Logic errors

Error types: examples

Consider that we want to write a program to compute the area of a rectangular from the inputs of width and length

```
1 int area(int length, int width)
2 {
3     return length*width;
4 }
5
```

```
1 int x1 = area(7);
2 int x2 = area("seven", 2);
3 int x3 = area(7, 10);
4 int x5 = area(7.5, 10);
5 int x6 = area(7.5, 0);
6 int x = area(10, -7);
7 double ratio = double(x3) / x6;
8
```

Error handling

Now we have a negative rectangular area and a terminated ratio calculation. What should we do?

Solutions:

1. The caller deals with errors
2. The callee deals with errors

Question: Which one is better?

Caller vs callee

Caller

- ▶ Has to know all the details about the function
- ▶ Input or output?
- ▶ Has to check every time the function is called

Callee

- ▶ What if we can't modify the callee
- ▶ The callee doesn't know what to do
- ▶ The callee doesn't know where it was called from
- ▶ Performance

Conventional error reporting I

► Return an “error value”

```
1 int area(int length, int width)    // return a negative value for bad
   input
2 {
3     if (length <= 0 || width <= 0) return -1;
4     return length*width;
5 }
6
```

► So, “let the caller beware”

```
1 int z = area(x,y);
2 if (z<0) error("bad area computation");
3
```

► Issues:

- What if I forget to check that return value?
- For some functions there isn't a “bad value” to return (e.g., `max()`)

Conventional error reporting II

► Set an error status indicator

```
1 int errno = 0; // used to indicate errors
2 int area(int length, int width)
3 {
4     if (length <= 0 || width <= 0) errno = 7;
5     return length * width;
6 }
7
```

► So, “let the caller check”

```
1 int z = area(x, y);
2 if (errno == 7) error("bad area computation");
3
```

► Issues:

- What if I forget to check **errno**?
- How do I pick a value for **errno** that's different from all others?
- How do I deal with that error?

Exception

- ▶ We might forget to handle the error and we might not even notice the error
- ▶ Need a mechanism that is dedicated to error reporting
 - ▶ an exception is **thrown** whenever an error is encountered

```
1 class Bad_area { }; // a class is a user defined type
2                       // Bad_area is a type to be used as an exception
3 int area(int length, int width)
4 {
5     if (length <= 0 || width <= 0) throw Bad_area {};
6     return length * width;
7 }
8
```

- ▶ caller can **catch** it and specify what to do

```
1 try {
2     int z = area(x,y); // if area() doesn't throw an exception
3 }                       // make the assignment and proceed
4 catch(Bad_area) {       // if area() throws Bad_area {}, respond
5     cerr << "oops! Bad area calculation fix program\n";
6 }
7
```

Example: out of range

```
1 vector<int> v(10); // a vector of 10 ints,  
2 // each initialized to the default value, 0,  
3 // referred to as v[0].. v[9]  
4 for (int i = 0; i<v.size(); ++i) v[i] = i; // set values  
5 for (int i = 0; i<=10; ++i) // print 10 values (???)  
6     cout << "v[" << i << "] = " << v[i] << endl;  
7
```

vector's operator[](subscript operator) reports a bad index (its argument) by throwing a **out_of_range** if you use

```
1 #include "std-lib-facilities.h"  
2
```

Example: out of range (Cont'd)

For now, just use exceptions to terminate programs gracefully, like this

```
1  int main()
2  try
3  {
4      // ...
5  }
6  catch (out_of_range) { // out_of_range exceptions
7      cerr << "oops some vector index out of range\n";
8  }
9  catch (...) { // all other exceptions
10     cerr << "oops some exception\n";
11 }
12
```

How to look for errors?

When you have written a program, it'll have errors (commonly called “bugs”)

- ▶ It'll do something, but not what you expected
- ▶ How do you find out what it actually does?
- ▶ How do you correct it?
- ▶ This process is usually called “debugging”

Program structure

Make the program easy to read so that you have a chance of spotting the bugs

1. Comments
2. Use meaningful names
3. Indent
4. Break code into small functions
5. Avoid complicated code sequences
 - ▶ Try to avoid nested loops, nested if-statements, etc.
6. Use library facilities

First get the program to compile

- Is every string literal terminated?

```
1 cout << "Hello , << name << '\n'; // oops!  
2
```

- Is every character literal terminated?

```
1 cout << "Hello , " << name << '\n; // oops!  
2
```

- Is every block terminated?

```
1 if (a>0) { /* do something */  
2 else { /* do something else */ } // oops!  
3
```

- Is every set of parentheses matched?

```
1 if (a // oops!  
2 x = f(y);  
3
```

First get the program to compile (Cont'd)

- ▶ Is every name declared?
 - ▶ Did you include needed headers?
- ▶ Is every name declared before it's used?
 - ▶ Did you spell all names correctly?

```
1 int count; /* */ ++Count; // oops!  
2 char ch; /* */ Cin>>c; // double oops!  
3
```

- ▶ Did you terminate each expression statement with a semicolon?

```
1 x = sqrt(y)+2 // oops!  
2 z = x+3;  
3
```

Debugging

- ▶ Carefully follow the program through the specified sequence of steps
 - ▶ Pretend you're the computer executing the program
 - ▶ Does the output match your expectations?
 - ▶ If there isn't enough output to help, add a few debug output statements

```
1 cerr << "x = " << x << ", y = " << y << '\n';  
2
```

Debugging (Cont'd)

- ▶ When you write the program, insert some checks (“sanity checks”) that variables have “reasonable values”
 - ▶ Function argument checks are prominent examples of this

```
1 if (number_of_elements < 0)
2     error("impossible: negative number of elements");
3 if (largest_reasonable < number_of_elements)
4     error("unexpectedly large number of elements");
5 if (x < y) error("impossible: x < y");
6
```

Debugging (Cont'd)

- ▶ Pay special attention to “end cases” (beginnings and ends)
 - ▶ Did you initialize every variable?
To a reasonable value
 - ▶ Did the function get the right arguments?
Did the function return the right value?
 - ▶ Did you handle the first element correctly?
The last element?
 - ▶ Did you handle the empty case correctly?
No elements
No input
 - ▶ Did you open your files correctly?
More on this in chapter 11
 - ▶ Did you actually read that input?
Write that output?

Pre-conditions

“A condition that must always be true just **prior to** the execution of some section of code or **before** an operation” (From Wikipedia).

```
1 int area(int length, int width) // calculate area of a rectangle
2 // length and width must be positive
3 {
4     if (length <= 0 || width <= 0) throw Bad_area{};
5     return length * width;
6 }
7
```

Post-conditions

“A condition that must always be true just **after** the execution of some section of code or **after** an operation” (From Wikipedia).

```
1 int area(int length, int width) // calculate area of a rectangle
2 // length and width must be positive
3 {
4     if (length <= 0 || width <= 0) throw Bad_area{};
5     // the result must be a positive int that is the area
6     // no variables had their values changed
7     return length*width;
8 }
9
```

Quick summary on pre- and postconditions

- ▶ Use `assert()` to check
- ▶ Always think about them
- ▶ If nothing else write them as comments
- ▶ Check them “where reasonable”
- ▶ Check a lot when you are looking for a bug

```
1 int area(int length, int width) // calculate area of a rectangle
2 // length and width must be positive
3 {
4   if (length <= 0 || width <= 0) throw Bad_area{};
5   // the result must be a positive int that is the area
6   // no variables had their values changed
7   return length * width;
8 }
9
```

- ▶ **Question:** Is it possible to find a set of inputs that satisfies pre-condition but fails to pass the post-condition?