Statement and Control Structure

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Expression

Expression A combination of values, variables, and operators. The purpose is to *have values*.

Statement A complete line of code that performs some action. The purpose is their *side-effects*.

```
int main() {
  int a = 3;
  a + 4;
  ++a;
}
```

The distinction is important only for theorist.

Basics

In PL's, arithmetic expressions consist of operators, operands, parentheses, and function calls. Operators are special functions.

```
int main() {
  int a = 1 + 2;
  int b = operator+(1, 2);  // DOES NOT COMPILE
}
```

Unary function takes single operand, e.g., negative sign `-`.

Binary function takes two operands, e.g., plus, minus.

Ternary function takes three operands., e.g., `a ? b : c`.

Notation

Infix Operators are between operands. Parentheses surrounding groups of operands and operators are necessary. More difficult to parse albeit more natural. E.g., $2 + 2$, 3×3, $(2 + 2) \times 3$.

Prefix Operators are before operands. Fancy name: normal Polish notation (NPN). If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. E.g., $+\ 2\ 2$, $\times + 2\ 2\ 3$

Postfix Operators are places after operands. Fancy name: reverse Polish notation (RPN). No parentheses are needed either. E.g., $n!$, $3\ 2\ 2 + \times$, $2\ 2 + 3 \times$. Shunting-yard algorithm converts infix to postfix notation.

Operator Evaluation Order

In NPN or RPN, operator evaluation is always *unambiguous*. Not so for infix notation.
To solve Infix notation ambiguity, two rules in PL are needed

- operator *precedence* and
- operator *associativity*

Operator Precedence

Given the following expression

$$3 + 4 \times 5$$

- Evaluated from left to right, i.e., $+$ first, the result is 35.
- Evaluated from right to left, i.e., $\times$ first, the result is 23.

Mathematicians place operators in a *hierarchy of evaluation priorities* and base the evaluation order of expressions partly on this hierarchy.

Operator Precedence Rule Partially defines the order in which the operators of different precedence levels are evaluated. The operator precedence rules for expressions are based on the hierarchy of operator priorities.

C++ Operator Precedence

http://en.cppreference.com/w/cpp/language/operator_precedence

Operator with higher precedence is evaluated first. What if they are of same precedence? E.g., $2 + 3 + 4$.

Operator Associativity

Operator Associativity Rule Partially defines the order in which the operators of the same precedence levels are evaluated. This rule is based on *convention*.

Two possible associativity

- Left to right. E.g., $2 + 3 + 4$ in `C++` is equivalent to $(2 + 3) + 4$.
- Right to left. E.g., $2 * *2 * *3$ in `Python` is equivalent to $2 * *(2 * *3)$.

Associativity in Real World

Floating point math is *not associative* due to accuracy. E.g. the non-representability of 0.1 and 0.01 in binary means that 0.1^2 is neither 0.01 nor the closet representable number. In single precision, 0.1 is represented as

`0.100000001490116119384765625 exactly`

Squaring it with single-precision floating-point hardware gives

`0.010000000707805156707763671875 exactly`

But the representable number closest to 0.01 is

`0.009999999776482582092285156250 exactly`

Non-associativity Example

Using 7-digit significand decimal arithmetic.
Given $a = 1234.567$, $b = 45.67834$, $c = 0.0004$. The symbol $\rightarrow$ denotes rounding.

$(a + b) + c$

	1234	.	567	a
$+$	45	.	67834	b
	1280	.	24534	$a + b$
$\rightarrow$	1280	.	245	
$+$	0	.	0004	c
	1280	.	2454	
$\rightarrow$	1280	.	245	

$a + (b + c)$

	45	.	67834	b
$+$	0	.	0004	c
	45	.	67874	$b + c$
$\rightarrow$	45	.	67874	
$+$	1234	.	567	a
	1280	.	24574	
$\rightarrow$	1280	.	246	

Optimization Based on Associativity

Compiler may *reassociate expressions* if possible.
In GCC, left linearization (switch on by `-O3`) is used. E.g.,

$$a + b + c + d + e \rightarrow ((((a + b) + c) + d) + e)$$

In LLVM, it uses the flag `-reassociate`. This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, GCSE, LICM, PRE, etc.

$$4 + (x + 5) \rightarrow x + (4 + 5)$$

- Constants has rank 0
- function parameters rank 1 and
- other values ranked corresponding to the reverse post-order traversal of current function (starting at 2) which effectively gives values in deep loops higher rank than values not in loops.

Operand Evaluation Order

Mathematically, it does not matter. For programs, it does.

```
int foo(int &a) { return a++; }
int main() {
  int a = 3;
  int b = a + a;              // DOES IT MATTER?
  int c = foo(a) + foo(a); // DOES IT MATTER?
}
```

More precisely, it matters only when the evaluation of an operand has *side effects*.

Side Effect

Side Effect The function (e.g., operator) changes either one of its parameters or a global variable.

```
int a = 0;

int fun() {
  a = 100;
  return 10;
}

int main() {
  b = a + fun();  // 10 or 110?
}
```

C++ Evaluation Order

Almost all operands, including function parameters are unspecified. There are two kinds of *optional* evaluations performed by the compiler:

- value computation, e.g., calculation of the value that is returned by the expression.
- side effect, e.g., access (read or write) an object designated by a volatile glvalue, and etc.

Order

Since C++11, *sequenced-before* rules are used instead of *sequence point* rules in C++0x.
"sequenced-before", denote as $\prec$ (precede), is an asymmetric, transitive, pair-wise relationship between evaluations within the same thread.

- If $A \prec B$, then evaluation of A will be complete before evaluation of B begins and vice versa.
- If $A \not\prec B$ and $B \not\prec A$, then two possibilities exist:
  - evaluations of A and B are unsequenced
  - they may be performed in any order but may not overlap

Undefined Behavior

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

void f(int a, int b) { }
int main() {
  int i = 0;
  i = ++i + i++;      // undefined behavior
  i = i++ + 1;        // undefined behavior
  i = ++i + 1;        // well-defined in C++11
  ++ ++i;             // well-defined in C++11
  f(++i, ++i);        // undefined behavior
  f(i = -1, i = -2); // undefined behavior
  cout << i << i++;
  int a[3];
  a[i] = i++;
}
```

Referential Transparency

Referential Transparency If any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program, the program is said to have referential transparency.

```
int foo() {
  return 10;
}

int bar(int a, int b) {
  return a + b;
}

int main() {
  int a = foo() * SIZE;
  bar(++a, ++a);
}
```

Operator Overloading

Similar to functions, Arithmetic operators are often used for more than one purpose.

```cpp
#include <string>
using namespace std;

int main() {
  int a = 3 + 4;
  double b = 1.3 + 4.5;
  string s = "abc" + string();
}
```

Operator Overloading – Good

User-defined overloading may improve the readability.

```
class Mat {
 public:
  Mat() { }
};

Mat& operator+(const Mat& a, const Mat& b)
{ return Mat(); }

Mat& addMat(const Mat& a, const Mat& b)
{ return Mat(); }

int main() {
  Mat a, b, c;
  Mat m0 = addMat(addMat(a, b), c);
  Mat m1 = a + b + c;
}
```

Operator Overloading – Evil

Overloading may also harm the readability or cause bugs.

```
int main() {
  int a = 1, b = 2;
  if (a & b) { }
  if (& b) { }
}
```

Relational Expression

Relational Operator Operator that compares the values of its two operands, which together make a relational expression. The value of a relational expression is *Boolean*.

In most PL's, relational operator includes $<$, $\leq$, $>$, $\geq$, $=$ and $\neq$.

```
int main() {
  int a = 3, b = 4;
  bool c = a > b;
}
```

Boolean Expression

Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators (e.g., `AND`, `OR` and `NOT` operation).
Mathematically, `OR` and `AND` have *equal precedence*. However, in most `C`-based PL's, `AND` has higher precedence over `OR`.

```
int main() {
  bool a{true};
  bool b{false};
  bool c{true};
  a || b && c;
  a > b > c;
}
```

Short-Circuit Evaluation

Short-circuit evaluation The result of the expression is determined without evaluating all of the operands and/or operators.

> *Every value computation and side effect of the first (left) argument of the built-in logical `AND` operator `&&` and the built-in logical `OR` operator `||` is sequenced before every value computation and side effect of the second (right) argument. (`C++11` and above)*

```
int main() {
  int a = 1, b = 3;
  (a >= 0) || (++b);
  (a < 0) && (++b);
}
```

Why Short-Circuit?

> *C guarantees that && and || are evaluated left to right – we shall soon see cases where this matters. – K&R*

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

int find_key(const vector<int> &v, int key) {
  int p = 0;
  for (; p < v.size() && v[p] != key; ++p) ;  // <- short-circuit!
  if (v.size() == p) p = -1;
  return p;
}

int main() {
  vector<int> v{1, 2, 3, 4};
  cout << find_key(v, 1) << endl;
  cout << find_key(v, 10) << endl;
}
```

Outline

Introduction

Flow of control is governed by

- precedence and associativity within expression
- control statements at statement level
- concurrency rules at program unit level

Control Structure

Control Structure A control statement and the collection of statements whose execution it controls.

- how many different control statement are needed?
- Multiple entries or not?

Conditional Goto

A selectable goto, `if (...) goto`, is minimally sufficient. The following code is called *Spaghetti Code*, since program flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled.

```c
int main() {
  int a = 0;

  if (a) {
    a = 3;
  }

  do {
    ++a;
  } while (a < 10);
}
```

```c
int main() {
  int a = 0;

  if (!a) goto end;
  a = 3;
end:

start:
  ++a;
  if (a < 10) goto start;
}
```

Goto Statement *Considered Harmful*

By Edsger Dijkstra, who advocates *structured programming*.

Structured Programming A programming paradigm aimed at improving the *clarity*, *quality*, and *development time* of a computer program by making extensive use of subroutines, block structures and for and while loops.

It's proven that *all* algorithms that can be expressed by flowcharts can be coded in PL with two control statements:

- choosing between two control flow paths and
- logically controlled iterations

Actually one more, the sequence structure, taken for granted.

How Many Needed?

Programmers don't care about theories, only *readability* and *writability*.

- Too many choices may hinder the readability
- Too few choices may hinder the writability

E.g., logically-controlled loops vs counter-based loops, lower-level `goto` statement.

Multiple Entries

- Add little to flexibility
- Hard to read

```
#include <random>
#include <iostream>
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<> dis(0, 1);
  int sum = 0;
  int i = 5;
  if (dis(gen) > 0.5) goto loop;
  for (i = 0; i < 10; ++i) {
 loop:
    sum += i;
  }
}
```

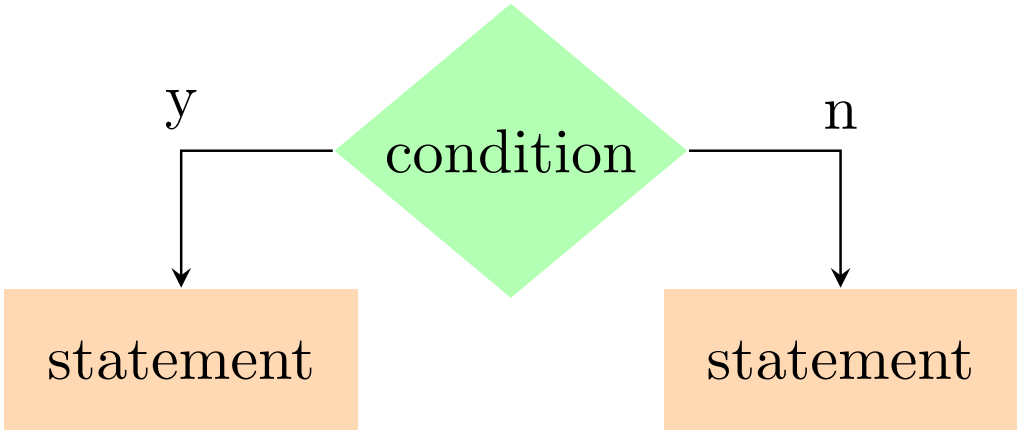
Multiple Exits

Most of the time, multiple exits is allowed and prefered (e.g. ?)

- Exits to the first statement, or
- Jump to (i.e., `goto`) anywhere in the program.

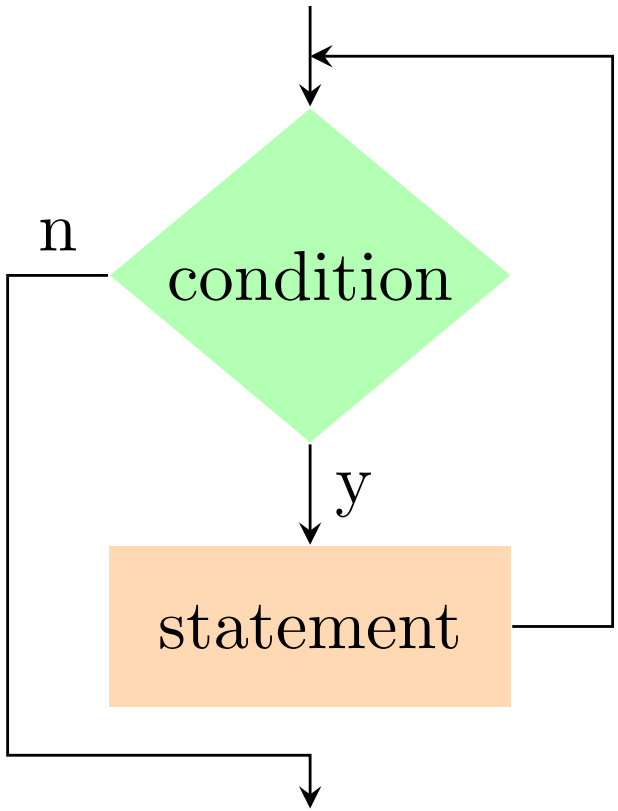
Outline

Selection Statements



- 2-way selection
  - Control expression form and type
  - Nested selection
  - Then and Else statement
- n-way selection
  - Control expression form and type
  - How electable segments specified
  - Single segment executed?
  - How case value specified

Iterative Statements



- How iteration controlled
- Control expression position
- Logically-controlled loop
- Counter-based loop
- etc.

Outline

Summary

- Expression
- Statement-level control structure
- Design issues and example