Practical Programming Methodology

(CMPUT-201)

Michael Buro

Lecture 5

- Expressions
- Assignment operators
- Type conversion
- Operator associativity, precedence, arity
- Flow control
- C++ I/O Basics
- Functions

Expressions

```
(a+b) * (a-b)          // OK
)a+b(                  // not OK
(a2*x + a1)*x + a0     // OK
a + b + c              // OK, a + b first
a + b * c              // OK, * first
(a >= b) || (c != 1)   // Boolean expression
```

- Built from variables, constants, operators, and ()
- infix notation
- () used for explicit evaluation order, must be balanced
- Operators have fixed arity, associativity & precedence

Assignment Operators

```
int a,b,c;
float d;

a = a + 4;       a += 4;   // equivalent
b = b >> x;      b >>= x;  // equivalent
c = c | 3;       c |= 3;   // equivalent
d = d * (a+1);   d *= a+1; // equivalent
```

- Set/change value of variable
- Syntax: <variable> = <expression> ;
- `i OP= c` equivalent to `i = i OP (c)`, where `OP` is one of `+ - * / % << >> & ^ |`

Type Conversions

Types of variable and expression must be compatible

Value is converted to type of variable

```
int a; double b;

// Implicit type casting
b = a;    // OK, int fits into double
a = b;    // not OK, warning should be issued

// Explicit type casting
a = (int)b;                 // oldest C style
a = int(b);                 // older C++ style
a = static_cast<int>(b);    // new C++ style
```

Explicit casts supress warnings, but precision may be lost!

Floating point numbers are truncated when converted to integers, not rounded!

Demonstration: starting g++ from within emacs, float-to-int conversion

Associativity and Precedence

```
() [] -> .                                    ltr high
! ~ ++ -- +(1) -(1) *(1) &(1) (type) sizeof   rtl
*(2) / %                                      ltr
+(2) -(2)                                     ltr
<< >>                                         ltr
< <= > >=                                     ltr
== !=                                         ltr
&(2)                                          ltr
^                                             ltr
|                                             ltr
&&                                            ltr
||                                            ltr
?:                                            rtl
= += -= *= /= %= &= |= <<= >>=                rtl
,                                             ltr low
```

rtl: right to left, ltr: left to right, (1)(2): arity
unary +-* have higher precedence than binary ops.

Precedence Examples

```
a = b + c * d;          // * before + before =

a = b >= 5 && c <= 6;   // >= (before <=) before =

a = b = c+1;            // right to left evaluation
                        // c+1 before b= before a=
```

Program Flow Control

- if-then-else
- switch
- goto
- loops
- functions

if-then-else Statement

```
if (y > x) x = y; // x = max(x,y)

if (x < 0) {
  sign = -1;
} else if (x > 0) {
  sign = +1;
} else
  sign = 0;
```

Obvious semantics

If then/else part consists of more than one statement, block is required: `{ <statements> }`

switch Statement

Multi-way switch dependent on integer value

```
char c; cin >> c;
switch (c) { // integer expression
  case '+':  // integer constant
    result = x + y;
    break; // <- beware of "fall-through" if missing!
  case '-':
    result = x - y;
    break;
  case 'q','Q','x','X':
    exit(0);
  default:
    cerr << "illegal input" << endl;
    exit(10);
}
```

Goto Statement

```
...
goto label;

...

label: ;  // resume execution here
```

- Control flow resumes at a specific location marked by a label (identifier)
- Use rarely! goto code is hard to understand and maintain ⇝ "Spaghetti code"

Loops

- Repeat execution of statements until a condition is met
- Three forms:

  `while` (<test-expr>) <statement>

  `do` <statement> `while` (<test-expr>) ;

  `for` (<init> ; <test-expr> ; <update>)
  <statement>

while Loop

- while (<test-expr>) <statement>
- while expression evaluates to true execute statement

```
// add values 1..100

int s = 0, i = 1;

while (i <= 100) { s += i; i++; }
```

do Loop

- first execute statement and loop if expression evaluates to true
- do <statement> while (<test-expr>) ;

```
int s = 0, i = 1;

do { s = s+i; i = i+1; } while (i <= 100);
```

for Loop

for (<init> ; <test-expr> ; <update>)
  <statement>

is equivalent to:

<init> ;
while (<test-expr>) { <statement> ; <update> ;}

```
int s = 0;

for (int i=1; i <= 100; ++i) s += i;
```

Local variables can be defined in the <init> part

Loop Control

- break; : exits loop immediately
- continue; : skips loop body

```
while (...) {
  ...
  break;
  // equivalent to
  // goto break_loc;
  ...
}
break_loc: ;
```

```
while (...) {
  ...
  continue;
  // equivalent to
  // goto cont_loc;
  ...
  cont_loc: ;
}
```

In for loops, continue resumes with the update

C++ Input/Output Introduction

```
#include <iostream>  // required
using namespace std; // required (or else: std::cout, std::endl etc)

int main() {
  int n;
  cout << "n=?\n";
  cin >> n;
  cout << "2*n=" << (2*n) << endl;
  return 0;
}
```

- Input via input-stream cin (”standard input”)
- Syntax: cin >> <variable> >> . . . >> <variable> ;
- Output via output-stream cout (”standard output”)
- Syntax: cout << <expr> << . . . << <expr> ;
- cin/cout is defined in standard C++ header file <iostream>

Standard Error

- Another predefined output stream: cerr
- Used for error messages
- Same output operator: <<
- Output is also sent to the console
- It is not redirected when using > or |

```
cerr << "division by zero" << endl; exit(10);
```

Visit http://www.cplusplus.com/ref/iostream to get more information on iostreams. More on them in a lab.

Example

```
// copy stdin to stdout
// description of cin functions
// @ http://www.cplusplus.com/ref/iostream/istream/

#include <iostream>
using namespace std;

int main()
{
  while (1) {                // iterate
    char c = cin.get();      // get one character from stdin
    if (!cin.good()) break;  // exit loop if error or eof
    cout << c;               // if good, append c to stdout
  }

  if (!cin.eof()) { cerr << "read error" << endl; }
  return 0;
}
```

Functions

- Modular programming
- Breaking down tasks into smaller sub-tasks
- Increases readability
- Eases debugging and program maintenance because program pieces can be tested individually
- Faster project development: work on separate functions simultaneously