Friends of if

Abhiram Ranade

Outline

- Remarks on if statement
- The switch statement
- Using Boolean Variables
- Program to determine if a number is prime
- while statement

if(a>0) if(b>0) c = 2; else c = 3;

```
if(a > 0){
   if(b > 0) c = 2;
   else c = 3;
}
```

a,b>0 : c = 2

a>0, b<=0 : c = 3

```
if(a > 0){
   if(b > 0) c = 2;
}
else c = 3;
```

a,b>0 : c = 2;

a<=0 : c = 3;

Remarks

- C++ chooses left interpretation.
- Use parenthesization.
- Do not remember such tricky rules.
- Do not expect others to remember them.
- If you compile using s++, you get a warning if you don’t parenthesize.

if(a = 2) c = 3;

- Quite likely: programmer wrote = instead of ==.
- s++ will give warning
- Why not error?
- a = 2 is assignment expression, of value 2.
- C++ tries to convert 2 to bool type.
  - Expression == 0 : false
  - Expression != 0 : true
  - So c=3 will always execute.
- If you mean if(a = 2) c = 3; write if((a = 2)) c = 3;
  - s++ will not give warning. () signals you mean expression.

The switch statement

Turtle controller revisited

```
main_program{
  turtleSim();
  repeat(100){
     char command; cin >> command;
     if(command == 'f') forward(100);
     else if(command == 'r') right(90);
     else if(command == 'l') left(90);
     else cout << “Invalid command.\n”;
  }
} // command determines what happens.
```

Another program

```
main_program{
  turtleSim();
  repeat(100){
    char command; cin >> command;
    switch(command){
    case 'f' : forward(100); break;
    case 'r' : right(90); break;
    case 'l'  : left(90; break;
    default : cout << "Invalid command.\n";
  }
} // stresses importance of command
```

General form

```
switch (exp){
  case v1: statements
  ...                                    // vi : constant
  case vn: statements
  default: statements                    // optional
}
```

- exp equals vi, then execution starts after case vi:
- does not equal any vi: execute from default:
- break -- ignore subsequent statements.
- vi : values known at compile time.

Remark

- Usually the statements after each case vi: end with break;
- If break; is omitted, next set of statements is also executed.
- Called fall-through
- High possibility of “forgetting” break;
- So statement considered error-prone.

Number of days in a month

```
main_program{
  int month;   cin >> month;
  switch(month){
    case 1: case 3: case 5: case 7: case 8: case 10:  case 12:
                   cout << 31 << endl; break;
    case 2:        cout << 28 << endl; break;
    case 4: case 6: case 9: case 11:
                    cout << 30 << endl; break;
    default:        cout << ``Invalid input.\n'';
  }
} // fall through is useful.
```

Logical data

```
float income; cin >> income;
bool highincome = (income >
   800000);
```

- Value of condition can be stored.
- And used later

```
if(highincome)
   tax = 92000 + (income -80000) * 0.3
```

More Examples

```
char c; cin >> c;
bool capital = (‘A’ <= c) && (c <= ‘Z’);
if(capital) ...

bool x = (y % 2 == 0) || (y % 3 == 0);
```

When is x true if y is an integer?

Is a given number n prime?

- Algorithm idea: Is there at least one number between between 2 and n-1 that divides n without leaving a remainder?
  - at least one divides perfectly: n is composite.
  - All leave a remainder: n is prime.
- between 2 and n-1 : ?
- divides perfectly : ?
- At least one : ?

> sequence generation
> (n % divisor == 0)
> OR should be true

Primality testing program

```
main_program{
  int n; cin >> n;
  int divisor = 2;  bool composite = false;
  repeat(n-2){
     composite = composite || (n%divisor == 0);
     divisor = divisor + 1;
  }
  if(composite) cout <<“Composite.\n”;
  else cout <<“Prime.\n”;
}
```

A better program, suggested by a student

```
main_program{
  int n; cin >> n;
  int divisor = 2;  bool composite = false;
  repeat(n-2){
     if(n%divisor == 0) composite = true;
     divisor = divisor + 1;
  }
  if(composite) cout <<“Composite.\n”;
  else cout <<“Prime.\n”;
}
```

Invariants
(for both programs)

At the beginning of t th iteration:

divisor = 1+t

composite = true if some number in the range 2..t divides n

Can you prove the invariant?

Does it imply correctness?

Is the program efficient?

- Once a factor is detected, need not check subsequent divisors.

while

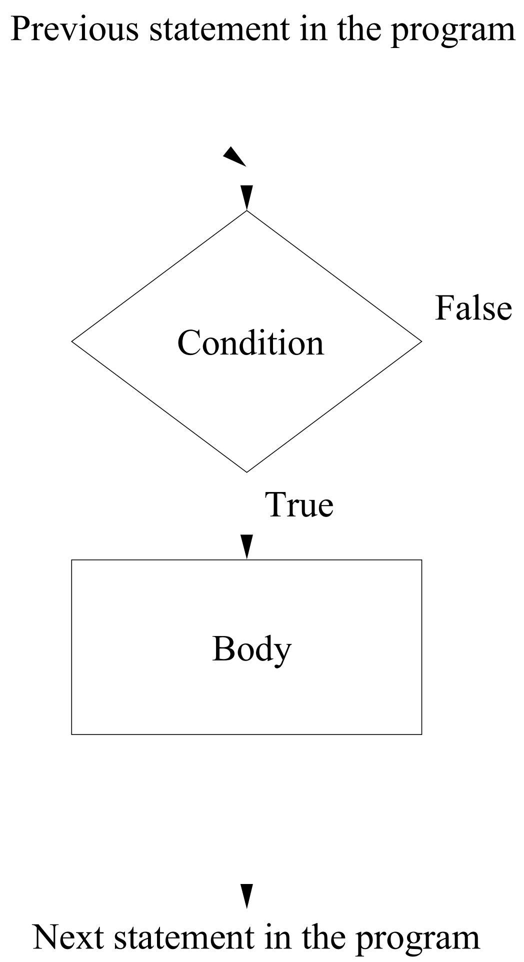
while (condition) body

condition: boolean expression

body: statement

1. Evaluate condition.
2. If false, execution of statement ends.
3. If true, execute body. Then go back and execute from step 1.

While flowchart



Primality testing program

```
main_program{
  int n; cin >> n;
  int divisor = 2;  bool composite = false;
  repeat(n-2){
     composite = composite || (n%divisor == 0);
     divisor = divisor + 1;
  }
  if(composite) cout <<“Composite.\n”;
  else cout <<“Prime.\n”;
}
```

Primality testing program

```
main_program{
  int n; cin >> n;
  int divisor = 2;  bool composite = false;
  while(!composite && divisor < n){
     composite = composite || (n%divisor == 0);
     divisor = divisor + 1;
  }
  if(composite) cout <<“Composite.\n”;
  else cout <<“Prime.\n”;
}
```

Arguing correctness

- In general, a program containing while may not terminate.
  - condition in while may never become false.

  i= 0; while(i >= 0){ i++; ….}

  - Programs with repeat always terminate
- May not terminate correctly.
- Must argue termination and correctness.

Invariant

- At the beginning of the t th iteration:
  - divisor = t + 1
  - composite = false if 2..t do not divide n. true otherwise.
- Will loop terminate?
  - As t increases, divisor = t+1 will equal n. Might terminate even earlier.

Proof of correctness

- What happens on termination?
  - Loop condition must be false. Either composite == true, or divisor == n
- Argue separately for the case when program printed “Composite” and for the case program printed “Prime”.

Proof of correctness(sketch)

- Case: program printed “Composite”.
  - composite must have been true.
  - composite starts true, so must have become true in some iteration.
  - in that iteration some factor must have been discovered.
  - Hence correct.

(Contd.)

- Case: program printed “Prime”.
  - composite == false at the end.
  - divisor == n must be true.
  - loop executed for all values of divisor from 2 to n-1.
  - n% divisor == 0 was never true.
  - Hence n must be prime.
  - Hence correct.