If while and for

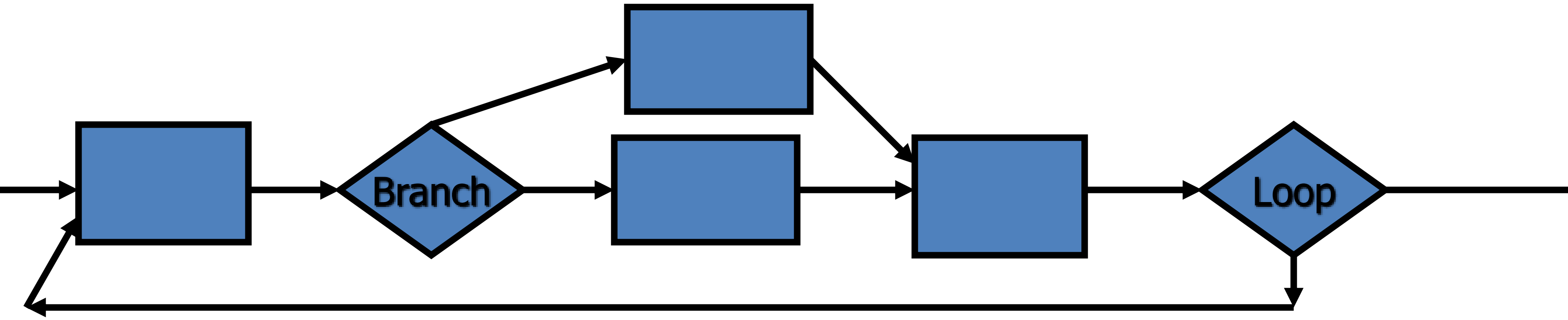
PHYS2G03

© James Wadsley,

McMaster University

Program Flow

- In general a program flows linearly from start to finish
- There can be occasional branching or loops (repetition) …



Controlling Program Flow

- Changes to the flow are essentially decisions based on logical expressions:
- Typically they are binary:

```
if (true)   do_this();
else        do_that();
```

The **if** statement

- if statements conditionally execute code if a logical expression is true

- `if (expression)   statement;`

-
  ```
  if (expression)  {
        many statements …
    }
  ```

if only performs one statement or, equivalently, one block { }

if statement example

```
float period;

std::cout <<  “Enter the period of the orbit\n”;
std::cin >> period;

if (period < 0)  {
    std::cout << “The period cannot be negative\n”;
    return 1;
}
```

if statement example

```
float period;

std::cout <<  “Enter the period of th
std::cin >> period;

if (period < 0)  {
    std::cout << “The period cannot be negative\n”;
    return 1;
}
```

If the inputs are bad a good idea is to
1) Tell the user
2) Exit the program

Independent if statements

Each test is independent. Depending on the expressions any or none the actions are done.

```
if (expression1)  { actions1; }
if (expression2)  { actions2; }
if (expression3)  { actions3; }
```

if statement example

```
if (day_of_month == 15)  pay_people();

if (day_of_week == Friday) shut_down_factory();

if (year == 2017) celebrate_20_years in business();
```

Dependent if statements: **else**

Each test is related. Tests following else only occur if prior tests fail. Only one actions will be performed. If expressions are related, this is more efficient.

```
if (expression1)  { actions1; }
else if (expression2) { actions2; }
else if (expression3) { actions3; }
else { default action; }
```

if else example

```
float period;
std::cout <<  “Enter the period of the orbit\n”;
std::cin >> period;

if (period < 0)  {
    std::cout << “The period cannot be negative\n”;
    return 1;
} else if (period > max_period) {
    std::cout << “This program can’t handle periods longer
    than “ << max_period << “\n”;
    return 2;
}
```

Nested if statements

Any block of code can contain if statements. Thus an if can lead to more ifs

```
if (expression1)  {
   if (exp2) { actions2; }
   else { action2fail; }
}
```

Nested if

```
if (a<b) {
   if (answer == true)
      std::cout <<" a<b, answer == true";
   else
      std::cout << "a<b but answer false";
}
else
    std::cout << "Nothing is working today";
```

Nested if

```
if (a<b) {
    if (answer == true)
        std::cout <<" a<b, answer == true";      1
    else
        std::cout << "a<b but answer false";            2
}
else
    std::cout << "Nothing is working today";   3
```

Note 3 possible outcomes!
Only one of these 3 will happen:

Nested if – Indentation required!

```
if (a<b) {
    if (answer == true)
        std::cout <<" a<b, answer == true";
    else
        std::cout << "a<b but answer false";
}
else
     std::cout << "Nothing is working today";
```

Note the **indentation** (spaces at the front of each line)
You should do this consistently
e.g. Move over 4 spaces when inside an if or else

Nested if (BAD)

```
if (a<b) {
if (answer == true)
std::cout <<” a<b, answer is true”;
else
std::cout << “a<b but answer false”;
}
else
std::cout << ”Nothing is working today”;
```

No indentation is hard to follow! You will lose marks for work that looks like this!

Nested if (WORSE)

```
if (a<b) { if (answer == true) std::cout <<” a<b, answer == true”;
    else std::cout << “a<b but answer false”; } else std::cout <<
    ”Nothing is working today”;
```

Use whitespace – remember C/C++ compiler doesn’t care (this code would compile) but we do!
I think you agree the above is horrible for a human to read and understand!

Nested if

```
if (a<b) {
   if (answer == true) {
      std::cout <<" a<b, answer == true";
   }
   else {
      std::cout << "a<b but answer false";
   }
}
else {
    std::cout << "Nothing is working today";
}
          ;
```

You can add braces if you feel it is clearer

Logical Expressions

Any comparison or other operator with a logical (bool) result is ok -- that is **true** or **false**

e.g.

```
(a < b)
(103.57 >= x)
(true)
```

Preview

Combining logical results: Logical operators

and &&   or || not !

((a<b) && (c<d))   Only true if a<b and c<d

((a<b) and (c<d))

(a>2.0 || i)   True if a>2.0 or the integer i not 0

(a>2.0 or i)

(!(a>1))   True if a <= 1

(not is_nan())   True if is_nan() returns false

Loops!

Repetition

- Many algorithms rely on repeated iterations to achieve a result

- There are two main cases:
- 1. A fixed number of iterations

  e.g. Listing the two times table
- 2. Iterations until a tolerance or convergence criterion is met

  e.g. Calculating pi to 6 figures using a series

Simple loop: **while** statement

Syntax:

while (expression) statement;

The expression should a logical true/false thing, e.g.

while (i>2)

while (3.0 > 4.0) (never does it)

while (true) (never ends!)

Simple loop: **while** statement

Syntax:

while (expression)

```
while (truething)  statement;

while (truething) {
        statement1;
        statement2;
 }
```

You can do one statement or many with braces {}

While

```
int money = 10;
while (money>0) {
    // Spend money!
    std::cout << “Money left = “ << money << “\n”;
    money=money-2;
}
```

Program

```
Money left = 10
Money left = 8
Money left = 6
Money left = 4
Money left = 2
```

Output

When money==0 it leaves the loop

While using break

```
int money = 10;
while (true) {
    // Earn money!
    std::cout << “Money total = “ << money << “\n”;
    money=money+2;
    if (money > 100) break;    // holiday time
}
```

Program

If the while expression is always true the loop would go forever
Sometimes you may wish to leave immediately
The break statement leaves the current loop

While loop – common format

```
int money = 10;


while (money>0) {
    // Spend money!
    std::cout << “Money
    left = “ << money <<
    “\n”;
    money=money-2;
}
```

Initial value for thing

Test if thing is still true

Do stuff

Change thing

General loop: **for** statement

Syntax:

```
for (initial;  condition;  change)   repeated_code;
```

They call be empty if you like: `for ( ; ; )`

You can also repeat a block of code:

```
for ( ; ; )  {
        statement1;
        statement2;
 }
```

for statement

initial statements are optional. They occur BEFORE the loop once.

The condition is an expression, it must be true or the loop ends immediately. If it is not true the first time the repeated_code is never done

The change statement is executed after the repeated_code just before the condition is tested again

Basic **for** loop: count 1 to 10

```
int i;

for (i=1; i<=10; i=i+1) {
  std:cout << "I counted to " << i << "\n";
}

// here i=11
```

Note use i= i+1 to add one to i

i is the only thing that changes the next time the loop is repeated. At the end, i=11

equivalent **while** loop: count 1 to 10

```
int i;
i=1;
while (i<=10) {
  std:cout << "I counted to " << i << "\n";
  i=i+1;
}

// here i=11
```

while loop is longer to write than **for** loop
Still perfectly good code
Will produce same program when compiled.

Note use i= i+1 to add one to i
i is the only thing that changes the next time the loop is repeated. At the end, i=11

Compact **for** loop: count 1 to 10

```
for (i=1; i<=10; i++) {
  std:cout << “I counted to “ << i << “\n”;
}

// here i=11
```

Note use of i++ (C/C++ shorthand) to add one to I

i++ is equivalent to i=i+1

Increment operator ++

```
i=i+1;
OR
i++;
```

Counting up is very common in programming

C/C++ have a shorthand for “add one to this” which is ++

x++	add 1 to x	x+=2	add 2 to x
x--	subtract 1 from x	x-=3	subtract 3 from x

C++ **for** loop: count 1 to 10

```
for (int i=0; i<=10; i++) {
  std::cout << “I counted to “ << i << “\n”;
}

// here i doesn’t exist!
```

C++ ONLY: If the variable int i is declared *inside the for* like this it’s scope is only the loop. It doesn’t exist when the loop is finished