Control Statements and Functions

EECS 230

Spring 2016
Agenda

- **Computation**
  - What is computable? How best to compute it?
  - Abstractions, algorithms, heuristics, data structures

- **Language constructs and ideas**
  - Sequential order of execution
  - Expressions and statements
  - Selection
  - Iteration
  - Functional abstraction
  - Vectors
You already know most of this

- You know how to do arithmetic:
  - $d = a + b \times c$
You already know most of this

- You know how to do arithmetic:
  - $d = a + b \times c$

- You know how to sequence:
  - “Open the door, then walk through.”
You already know most of this

- You know how to do arithmetic:
  - $d = a + b \times c$

- You know how to sequence:
  - “Open the door, then walk through.”

- You know how to select:
  - “If it’s raining, take an umbrella; otherwise take sunglasses.”
You already know most of this

- You know how to do arithmetic:
  - \( d = a + b \times c \)

- You know how to sequence:
  - “Open the door, then walk through.”

- You know how to select:
  - “If it’s raining, take an umbrella; otherwise take sunglasses.”

- You know how to iterate:
  - “Do 20 reps.”
  - “Stir until no lumps remain.”
You already know most of this

- You know how to do arithmetic:
  - $d = a + b \times c$

- You know how to sequence:
  - “Open the door, then walk through.”

- You know how to select:
  - “If it’s raining, take an umbrella; otherwise take sunglasses.”

- You know how to iterate:
  - “Do 20 reps.”
  - “Stir until no lumps remain.”

- You know how to do function calls (sort of):
  - “Go ask Alice and report back to me.”
You already know most of this

- You know how to do arithmetic:
  - $d = a + b \times c$
- You know how to sequence:
  - “Open the door, then walk through.”
- You know how to select:
  - “If it’s raining, take an umbrella; otherwise take sunglasses.”
- You know how to iterate:
  - “Do 20 reps.”
  - “Stir until no lumps remain.”
- You know how to do function calls (sort of):
  - “Go ask Alice and report back to me.”

So what I’ll be showing you is mainly syntax for things you already know.
Computation: the big picture

- Input: from keyboard, files, mouse, other input devices, the network, other programs
- Code: consumes the input and does something to produce the output
- Output: to the screen, files, printer, other output devices, the network, other programs
Expressing computation

Our job is to express computations

- simply,
- correctly, and
- efficiently.

Tools:

- Divide and conquer
  - Break a big computation into several smaller ones
- Abstraction
  - Use a higher-level concept that hides detail
- Data organization (often key to good code)
  - Input/output formats
  - Communication protocols
  - Data structures

Note the emphasis is on structure and organization
Expressing computation

Our job is to express computations

- simply,
- correctly, and
- efficiently.

Tools:
Expressing computation

Our job is to express computations

- simply,
- correctly, and
- efficiently.

Tools:

- Divide and conquer
  - Break a big computation into several smaller ones
Expressing computation

Our job is to express computations

- simply,
- correctly, and
- efficiently.

Tools:

- Divide and conquer
  - Break a big computation into several smaller ones
- Abstraction
  - Use a higher-level concept that hides detail
Expressing computation

Our job is to express computations

- simply,
- correctly, and
- efficiently.

Tools:

- **Divide and conquer**
  - Break a big computation into several smaller ones
- **Abstraction**
  - Use a higher-level concept that hides detail
- **Data organization (often key to good code)**
  - Input/output formats
  - Communication protocols
  - Data structures
Expressing computation

Our job is to express computations

- simply,
- correctly, and
- efficiently.

Tools:

- Divide and conquer
  - Break a big computation into several smaller ones
- Abstraction
  - Use a higher-level concept that hides detail
- Data organization (often key to good code)
  - Input/output formats
  - Communication protocols
  - Data structures

Note the emphasis is on structure and organization
Programming language features

Each language feature exists to express a fundamental idea:

- `+` addition
- `*` multiplication
- `{ stm stm ... }` sequencing
- `if (expr) stm else stm` selection
- `while (expr) stm` iteration
- `f(x);` function call
Programming language features

Each language feature exists to express a fundamental idea:

+  addition
*  multiplication
{ stm stm ... }  sequencing
if (expr) stm else stm  selection
while (expr) stm  iteration
f(x);  function call

The meaning of each feature is simple, but we combine them into programs of arbitrary complexity.
Expressions

An expression computes a value:

```java
int length = 20; // simplest expression is a literal
int width = 40;
```
Expressions

An expression computes a value:

```plaintext
int length = 20; // simplest expression is a literal
int width = 40;
int area = length * width; // multiplication
```
Expressions

An expression computes a value:

```c
int length = 20; // simplest expression is a literal
int width = 40;

int area = length * width; // multiplication
```

// as in algebra, you can compose operations
```
int average = (length + width) / 2;
```
Expressions

An expression computes a value:

```java
int length   = 20;  // simplest expression is a literal
int width    = 40;

int area     = length * width;  // multiplication

// as in algebra, you can compose operations
int average  = (length + width) / 2;
```

The usual rules of precedence apply:
```
a * b + c / d means (a * b) + (c / d), not ((a * b) + c) / d
```
Expressions

An expression computes a value:

```java
int length = 20; // simplest expression is a literal
int width = 40;

int area = length * width; // multiplication
```

// as in algebra, you can compose operations
int average = (length + width) / 2;

The usual rules of precedence apply:
a * b + c / d means (a * b) + (c / d), not ((a * b) + c) / d

When in doubt, parenthesize (but don’t overdo it)
What expressions are made of

Operators and operands

- operators specify what to do
- operands specify the data to do it to
What expressions are made of

Operators and operands

- operators specify what to do
- operands specify the data to do it to

Some common operators:

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Meaning</th>
<th>&lt;bool&gt;</th>
<th>&lt;int&gt;</th>
<th>&lt;double&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, −, *, /</td>
<td>arithmetic</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>%</td>
<td>remainder</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>comparisons</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&amp;&amp;,</td>
<td></td>
<td></td>
<td>and, or</td>
<td></td>
</tr>
</tbody>
</table>
Concise operators

For many binary operators, there are (roughly) equivalent more concise versions:

\[
\begin{align*}
    a & += c \quad \text{means} \quad a = a + c \\
    a & *= \text{scale} \quad \text{means} \quad a = a \times \text{scale} \\
    ++a & \quad \text{means} \quad a += 1 \\
        & \quad \text{or} \quad a = a + 1
\end{align*}
\]

Use them when they make your code clearer
Statements

A statement is one of:

- an expression followed by a semicolon,
- a declaration, or
- a *control* statement that determines control flow.

I don't expect you to recognize all of these…yet.
Statements

A statement is one of:

- an expression followed by a semicolon,
- a declaration, or
- a control statement that determines control flow.

Examples:

- \( a = b; \)
- \( \text{double } d2 = 2.5; \)
- \( \text{if } (x == 2) y = 4; \)
- \( \text{while } (\text{cin } >> \text{number}) \text{ numbers.push_back(number);} \)
- \( \text{int } \text{average} = (\text{length} + \text{width}) / 2; \)
- \( \text{return } x; \)
Statements

A statement is one of:

- an expression followed by a semicolon,
- a declaration, or
- a control statement that determines control flow.

Examples:

- \( a = b; \)
- \( \text{double } d2 = 2.5; \)
- \( \text{if } (x == 2) y = 4; \)
- \( \text{while } (\text{cin } >> \text{number}) \text{ numbers.push_back(number);} \)
- \( \text{int average } = (\text{length } + \text{width}) \div 2; \)
- \( \text{return } x; \)

I don’t expect you to recognize all of these…yet.
Selection

Sometimes we must choose between alternatives.

For example, suppose we want to identify the larger of two numbers. We can use an `if` statement:

```java
if (a < b)
    max = b;
else
    max = a;
```
Selection

Sometimes we must choose between alternatives.

For example, suppose we want to identify the larger of two numbers. We can use an `if` statement:

```plaintext
if (a < b)
    max = b;
else
    max = a;
```

The syntax is

```plaintext
if (condition)
    statement-if-true
else
    statement-if-false
```
Sequencing

What if you want to do more than one thing in an if?

Use a compound statement:

```
if (a < b)
    f_max = b; min = a;
g else
    f_max = a; min = b;
```

The syntax is

```
first-statement second-statement // etc.
```
Sequencing

What if you want to do more than one thing in an `if`?

Use a compound statement:

```java
if (a < b) {
    max = b;
    min = a;
} else {
    max = a;
    min = b;
}
```
Sequencing

What if you want to do more than one thing in an `<if>`?

Use a compound statement:

```java
if (a < b) {
    max = b;
    min = a;
} else {
    max = a;
    min = b;
}
```

The syntax is

```java
{  
    first-statement
    second-statement
    // etc.
}
```
Iteration (while)

```
int i = 0;

while (i < 100) {
    cout << i << ' ' << square(i) << '
';
    ++i;
}
```
Iteration (while)

```cpp
int i = 0;

while (i < 100) {
    cout << i << 't' << square(i) << '
';
    ++i;
}
```

The syntax is

```cpp
while (condition) statement
```
Iteration (for)

```c
int i = 0;    // initialization

while (i < 100) {
    cout << i << '\t' << square(i) << '\n';
    ++i;    // step
}
```

This pattern—a loop with initialization and step—is so common that there’s special syntax for it:

```c
for (int i = 0; i < 100; ++i)
    cout << i << '\t' << square(i) << '\n';
```
Iteration (for)

```cpp
int i = 0; // initialization

while (i < 100) {
    cout << i << ' ' << square(i) << '
';
    ++i;   // step
}
```

This pattern—a loop with initialization and step—is so common that there’s special syntax for it:

```cpp
for (int i = 0; i < 100; ++i)
    cout << i << ' ' << square(i) << 'n';
```

'dfor' loops are the idiomatic way to count in C++
Syntax of for

\[
\text{for} \ (\text{init-expr}; \ \text{cond-expr}; \ \text{step-expr}) \\
\text{body-stm}
\]
Syntax of for

for (init-expr; cond-expr; step-expr)
  body-stm

means

init-expr;

while (cond-expr) {
  body-stm
  step-expr;
}

15
Functions

But what did \( \text{square}(i) \) mean?
Functions

But what did square(i) mean?

A call to the function square(int), which might be defined like

```c
int square(int x)
{
    return x * x;
}
```
Functions

But what did `square(i)` mean?

A call to the function `square(int)`, which might be defined like

```c
int square(int x)
{
    return x * x;
}
```

Why define a function?
Functions

But what did square(i) mean?

A call to the function square(int), which might be defined like

```c
int square(int x)
{
    return x * x;
}
```

Why define a function? We want to separate and name a computation because it…

- …is logically separate.
Functions

But what did `square(i)` mean?

A call to the function `square(int)`, which might be defined like

```c
int square(int x)
{
    return x * x;
}
```

Why define a function? We want to separate and name a computation because it…

- …is logically separate.
- …make the program clearer.
Functions

But what did square(i) mean?

A call to the function square(int), which might be defined like

```c
int square(int x)
{
    return x * x;
}
```

Why define a function? We want to separate and name a computation because it…

- …is logically separate.
- …make the program clearer.
- …can be reused.
Functions

But what did \texttt{square(i)} mean?

A call to the function \texttt{square(int)}, which might be defined like

\begin{verbatim}
int square(int x)
{
    return x * x;
}
\end{verbatim}

Why define a function? We want to separate and name a computation because it…

\begin{itemize}
  \item …is logically separate.
  \item …make the program clearer.
  \item …can be reused.
  \item …eases testing, distribution of labor, and maintenance.
\end{itemize}
A function example

```cpp
int square(int n) {
    return n * n;
}

int main {
    cout << sqrt(square(3) + square(4)) << '\n';
}
```
A function example

```cpp
int square(int n) {
    return n * n;
}

int main {
    double a2 = square(3);
    double b2 = square(4);
    double c2 = a + b;
    double c  = sqrt(c2);
    cout << c << '\n';
}
```
A function example

```c++
int main {
    double a2 = square(3);
    double b2 = square(4);
    double c2 = a + b;
    double c = sqrt(c2);
    cout << c << 'n';
}

int square(int n) {
    return n * n;
}
```
A function example

```cpp
int main {
    double a2 = square(3);

    double b2 = square(4);

    double c2 = a + b;
    double c  = sqrt(c2);

    cout << c << '\n';
}
```

```cpp
template<int n> double square(int n) {
    return n * n;
} 
```
Function definition syntax

Our function

```c
int square(int x)
{
    return x * x;
}
```

is an example of

```
return-type function-name(param-type param-name, …)
{
    // code, which can use parameter(s) param-name, etc.
    return some-value;
}
```