Control Statements and Functions

EECS 211

Winter 2017
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

- How to talk about syntax
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.
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:

```c
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:

```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;
```

The usual rules of precedence apply:

```
int 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>bool</th>
<th>int</th>
<th>double</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></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>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&amp;&amp;,</td>
<td></td>
<td></td>
<td>and, or</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Concise operators

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

- \( a += c \) means \( a = a + c \)
- \( a *= \text{scale} \) means \( a = a \times \text{scale} \)
- \( ++a \) means \( a += 1 \)
  or \( a = a + 1 \)

Use them when they make your code clearer
Syntax of Expressions

In BNF:

\[
\langle expr \rangle \ ::= \langle \langle \text{numeric-literal} \rangle \rangle \\
\quad | \langle \langle \text{string-literal} \rangle \rangle \\
\quad | \langle \langle \text{variable} \rangle \rangle \\
\quad | \langle expr \rangle \langle \langle \text{op} \rangle \rangle \langle expr \rangle \\
\quad | \langle expr \rangle \langle \langle \text{expr-list} \rangle \rangle \\
\quad | \langle expr \rangle \\
\langle expr-list \rangle \ ::= \\
\quad | \langle expr \rangle \langle expr-cont \rangle \\
\langle expr-cont \rangle \ ::= \\
\quad | , \langle expr \rangle \langle expr-cont \rangle
Syntax of Expressions

In BNF:

```
⟨expr⟩  ::=  ⟨⟨numeric-literal⟩⟩
          |  ⟨⟨string-literal⟩⟩
          |  ⟨⟨variable⟩⟩
          |  ⟨expr⟩ ⟨⟨op⟩⟩ ⟨expr⟩
          |  ⟨expr⟩ ( ⟨expr-list⟩ )
          |  ⟨expr⟩ ? ⟨expr⟩ : ⟨expr⟩
          |  ( ⟨expr⟩ )

⟨expr-list⟩ ::=  
                |  ⟨expr⟩ ⟨expr-cont⟩

⟨expr-cont⟩ ::=  
                |  , ⟨expr⟩ ⟨expr-cont⟩
```
Statements

A statement is one of:

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

A statement is one of:

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

Examples:

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

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

\[
\langle\text{type}\rangle := \text{int} \mid \text{double} \mid \text{string} \mid \ldots
\]

\[
\langle\text{decl}\rangle := \langle\text{type}\rangle \langle\text{variable}\rangle = \langle\text{expr}\rangle \\
| \langle\text{type}\rangle \langle\text{variable}\rangle
\]

\[
\langle\text{stmt}\rangle := \langle\text{expr}\rangle; \\
| \langle\text{decl}\rangle; \\
| \text{if} (\langle\text{expr}\rangle) \langle\text{stmt}\rangle \text{else} \langle\text{stmt}\rangle \\
| \text{if} (\langle\text{expr}\rangle) \langle\text{stmt}\rangle \\
| \text{while} (\langle\text{expr}\rangle) \langle\text{stmt}\rangle \\
| \text{for} (\langle\text{decl}\rangle; \langle\text{expr}\rangle; \langle\text{expr}\rangle) \langle\text{stmt}\rangle \\
| \text{return} \langle\text{expr}\rangle; \\
| \{\langle\text{stmt-list}\}\}
\]

\[
\langle\text{stmt-list}\rangle := \\
| \langle\text{stmt}\rangle \langle\text{stmt-list}\rangle
\]
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;
```
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
⟨stmt⟩ := if (⟨expr⟩) ⟨stmt⟩ else ⟨stmt⟩
```
Sequencing

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

Use a compound statement:

```c
if (a < b) {
    fmax = b;
    min = a;
} else {
    fmax = a;
    min = b;
}
```

The syntax is:

```
⟨stmt⟩ ::= { ⟨stmt-list⟩ }
⟨stmt-list⟩ ::= | ⟨stmt⟩ ⟨stmt-list⟩
```
Sequencing

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

Use a compound statement:

```c
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:

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

The syntax is

```
⟨stmt⟩  ::=  {  ⟨stmt-list⟩  }

⟨stmt-list⟩ ::=  ∅  |  ⟨stmt⟩  ⟨stmt-list⟩
```
Iteration (while)

```cpp
int i = 0;
while (i < 100) {
    cout << i << 't' << square(i) << '
';
    ++i;
}
```
int i = 0;

while (i < 100) {
    cout << i << ' ' << square(i) << '\n';
    ++i;
}

The syntax is

\(\langle \text{stmt} \rangle := \text{while} (\langle \text{expr} \rangle) \langle \text{stmt} \rangle\)
Iteration (for)

```cpp
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:

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

```
int i = 0;       // initialization

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

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

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

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

for (init-decl; cond-expr; step-expr)
  body-stm
Syntax of for

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

means

\[
\text{init-decl;}
\]
\[
\text{while } (\text{cond-expr}) \{ \\
\text{body-stm} \\
\text{step-expr;}
\}
\]
Functions

But what did 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;
}
```
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}

The syntax is:

\begin{verbatim}
⟨fun-decl⟩ := ⟨type⟩ ⟨⟨variable⟩⟩ ( ⟨args⟩ ) { ⟨stmt-list⟩ }
⟨args⟩ :=
    | ⟨type⟩ ⟨⟨variable⟩⟩ ⟨more-args⟩
⟨more-args⟩ :=
    | , ⟨type⟩ ⟨⟨variable⟩⟩ ⟨more-args⟩
\end{verbatim}
Why define a function?

We want to separate and name a computation because it...

...is logically separate.

...make the program clearer.

...can be reused.

...eases testing, distribution of labor, and maintenance.
Why define a function?

We want to separate and name a computation because it…

- …is logically separate.
Why define a function?

We want to separate and name a computation because it…

- …is logically separate.
- …make the program clearer.
Why define a function?

We want to separate and name a computation because it…

- …is logically separate.
- …make the program clearer.
- …can be reused.
Why define a function?

We want to separate and name a computation because it…

- …is logically separate.
- …make the program clearer.
- …can be reused.
- …eases testing, distribution of labor, and maintenance.
A function example

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

int main () {
    cout << sqrt(square(3) + square(4)) << '\n';
}
```
int square(int n) {
    return n * n;
}

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

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

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

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

double sqrt(double);
```