07. if Pitfalls, Type Conversion, Constants

CPSC 120: Introduction to Programming
Kevin A. Wortman ~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Q&A
2. if Pitfalls
3. Type Conversion
4. Constants

1. Q&A

Q&A

Let’s hear your questions about...

- This week’s Lab
- Linux
- Any other issues

Reminder: write these questions in your notebook during lab

2. if Pitfalls

Recap: Syntax: if statement

statement:

if (*condition-expr* **)** *true-statement*

else-clause(optional)

else-clause:

else *false-statement*

Semantics:

1. Evaluate *condition-expr* and convert result to `bool`
2. If result is true: execute *true-statement*
3. Otherwise, execute *false-statement* if it exists

Examples:

```
if (lives == 0)
  std::cout << "Game over";
```

```
if (age >= 18)
  std::cout << "legal adult";
else
  std::cout << "legal minor";
```

Recap: Syntax: Compound Statement

statement:

{ *inner-statement...* **}**

Semantics:

- Execute *inner-statement...* in top-to-bottom order

Example:

```
if (health == 0) {
  std::cout << "You lost a life\n";
  --lives;
}
```

Needless compound statement:

```
{
  std::cout << "Enter a number: ";
}
```

Recap: Relational Operators

Operator	**Semantics**	**Example** (x and y are same type)
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`<`	Less than	`x < y`
`>`	Greater than	`x > y`
`<=`	Less than or equal to	`x <= y`
`>=`	Greater than or equal to	`x >= y`

Pitfall: = versus ==

- = is **assignment** operator;
  `x = 3;`
  x *changes* to become 3
- == is equality comparison operator;
  `x == 3`
  produces `true` when x is 3, `false` otherwise, *leaving x unchanged*
- Easy mixup
  - Unfortunate!
  - `if (x = 3)` // should be ==
  - `x == 0;` // should be =

Pitfall: = in if

Logic error: write = in `if` expression instead of ==

If statement on right:

1. **Assigns** (changes) `choice` to 1
2. `choice` is converted to bool
3. 1 is nonzero which **always** counts as `true`

So this **always** prints “you chose 1”, even if the user input something other than 1!

```
int choice{ 0 };
std::cin >> choice;
if (choice = 1) {
  std::cout << “you chose 1”;
}

// if should be:

if (choice == 1) {
  std::cout << “you chose 1”;
}
```

Pitfall: Stray Semicolon After if Expression

Review: if syntax:

if (*condition-expr* **)** *true-statement*

else-clause(optional)

```
if (x > 0)
  std::cout << “positive”;
```

Logic error:

if (*condition-expr* **) ;**

true-statement

stray semicolon

```
if (x > 0);
  std::cout << “positive”;
```

Pitfall: Stray Semicolon After if Expression

Logic error:

1. As usual, whitespace is ignored
2. The `;` counts as the *true-statement* of the if
3. If *condition-expr* is true, execute `;` (do nothing)
4. Then, always, execute `cout << “positive”`

```
if (x > 0);
  std::cout << “positive”; // always prints, regardless of x
```

Pitfall: Unexpected Expression After Else

Compile error:

```
if (x < 0) {
  std::cout << “negative”;
} else (x >= 0) {
  std::cout << “non-negative”;
}
```

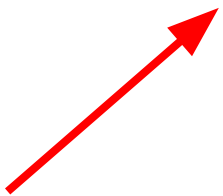
- Highlighted `(x >= 0)` is invalid syntax
- **Remember**: `else` means “otherwise” aka “in all other cases”
  - Doesn’t make sense to limit when `else` happens

Problem: Choose Between 3+ Alternatives

```
// do one thing, or nothing
if (count == 1) {
  std::cout << “once”;
}

// choose between two alternatives
if (count == 1) {
  std::cout << “once”;
} else {
  std::cout << “more than once”;
}
```

```
// choose between four alternatives
if (count == 1) {
  std::cout << “once”;
} else {
  if (count == 2) {
    std::cout << “twice”;
  } else {
    if (count == 3) {
      std::cout << “thrice”;
    } else {
      std::cout << count << “ times”;
    }
  }
}
```



works, but hard to read

Chaining If Statements

To decide between 3+ alternatives:

- chain together `if`s and `else`s
- Omit **{** between else and if
- Indent all the compound statements the same amount
- Still plain if syntax; nothing new

```
// choose between four alternatives
if (count == 1) {
  std::cout << “once”;
} else if (count == 2) {
  std::cout << “twice”;
} else if (count == 3) {
  std::cout << “thrice”;
} else {
  std::cout << count << “ times”;
}
```

Floating Point Imprecision

- A floating point type (`double`) uses scientific notation

$$mantissa \times 10^{exponent}$$

$$4.732 \times 10^{4} = 47{,}320$$

- Limited number of digits in mantissa
- May be no effect from adding/subtracting a small number with a big one
- **Floating point imprecision:** when arithmetic on floating point types produces mathematically-incorrect values

Demo: Floating Point Imprecision

```
#include <iostream>

int main(int argc, char* argv[]) {

double big{47320};
double small{.001};
std::cout << "big number: " << big << "\n";
std::cout << "little bigger: " << big + small
          << "\n";

return 0;
}
```

```
$ ./a.out
big number: 47320
little bigger: 47320
```

floating point imprecision

3. Type Conversion

Implicit Semantics

- **Explicit** (adj): expressly stated
  - Ex: in
    ```
    bool winning{false};
    ```
    data type is `bool`
- **Implicit** (adj): implied; not explicit
  - Ex.: when a barista calls your name, implicity you should pick it up
- Some semantics are implicit
  - Automatically happen even if you don't write code for it
- Automates tedious programming tasks
  - Division of labor

Mixed Expressions

- **Mixed expression***:* involves values of different types

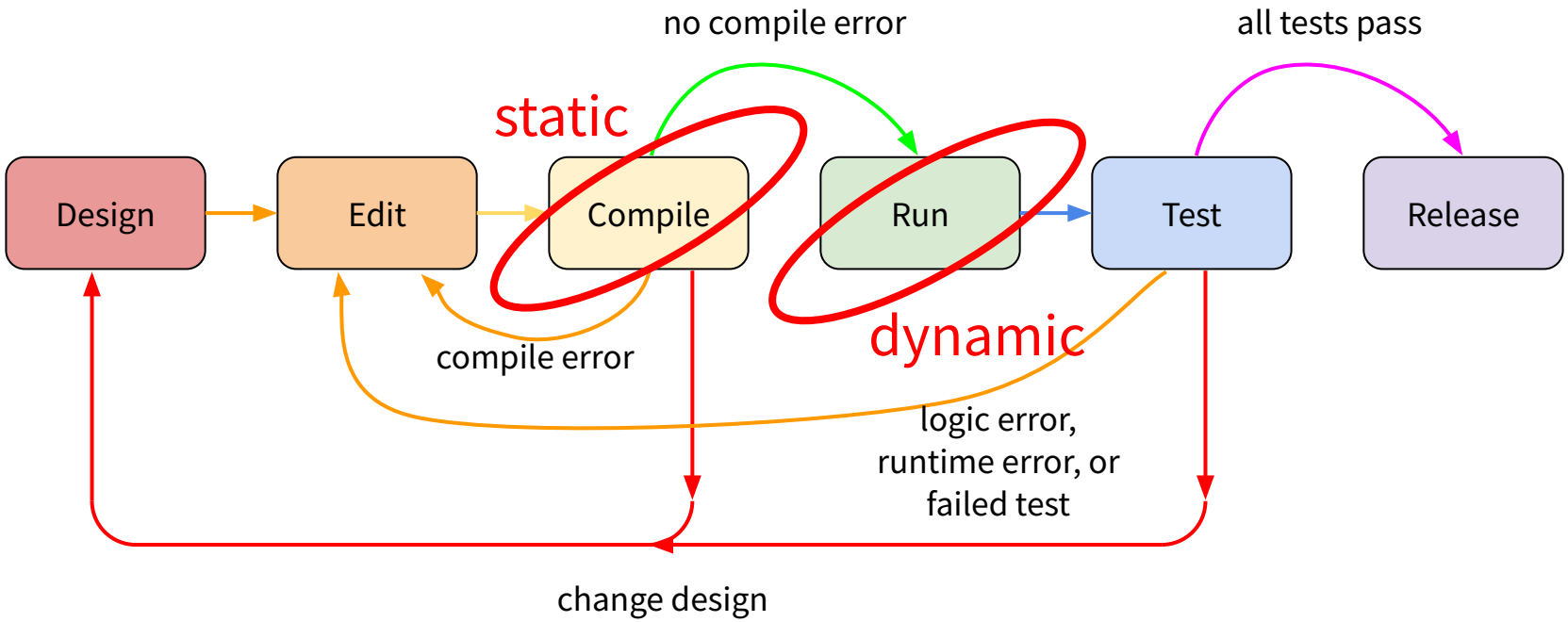
```
int x{3};
double y{2.5};
std::cout << x+y;
```

- **Implicit Type Promotion:** in a mixed expression, *the narrower type is implicitly converted to the wider type*
  - `double` is wider than `int`
  - `double`s are implicitly promoted to `int`s
  - `cout` above prints 5.5, not 5

Type Casting

- *Type cast:* expression to explicitly convert a value to a different data type
- Several alternatives
  - C-style cast
  - Functional cast
  - **static_cast** (preferred)

Static versus Dynamic



Static versus Dynamic

Static or Dynamic?	Happens When?	Aspects of Code Below
Static	**Compile-time** = while code is being compiled	• Header check, format check • Compile error checks ○ Ex: `0.0` is right type for `price`
Dynamic	**Runtime** = while program is running	• Printing `"Enter price: "` • Initializing `price` to 0.0 • User typing in new value

```
double price{0.0};
std::cout << "Enter price: ";
std::cin >> price;
```

static_cast

- static_cast: function that converts a value to a different data type
- Built-in function
  - Doesn't need `#include`
- Creates and returns a new different object
- Compiler determines how to convert statically
  - (also a `dynamic_cast`)

Syntax: static_cast function call

expression:

static_cast<*target-type*>(*expression*)

Semantics:

1. Returns a value of type *target-type*

Example:

```
double a{2.3};
int b{5};
std::cout << static_cast<int>(a * b);
```

Output:

11

(not 11.5)

4. Constants

Principle of Least Privilege

- **Principle of least privilege**: only grant access that is truly necessary
  - “Need to know basis”
  - Evident in iOS and Android apps
  - Ex. only let an app access your location if there is a legitimate need
- Prevents
  - Bugs causing undue harm
  - Spyware

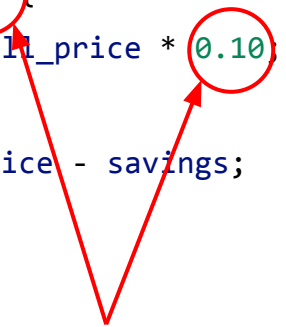
Principle: Detect Bugs at Compile Time

- Humans are imperfect
- Bugs are inevitable
- **Ideal division of labor:** better for a program to check source code, than a human
- Some kinds of errors are easier to debug
  - Compile error: easiest (told cause and location)
  - Runtime error: harder (told cause, not location)
  - Logic error: hardest (told neither)
  - Given the choice, want errors to be compile errors
- Principle: **help the compiler to detect bugs and report them as compile errors**

Understand the Problem: Magic Numbers

- **Magic number:** numeric literal that represents a business logic concept
- Unclean
  - What does 55 mean?
  - What does 0.10 mean?
- Labor-intensive to change
  - Policies are likely to change someday
  - Hard work to find and change all 55, 0.10 occurrences
  - (Division of labor)

```
double PriceAfterSeniorDiscount(
  double full_price, int age) {
  double savings{0.0};
  if (age >= 55) {
    savings = full_price * 0.10;
  }
  return full_price - savings;
  }
```



magic numbers

Recap: Single Point Of Truth (SPOT)

- **Single Point Of Truth (SPOT):** an idea is represented in **only one place**
  - aka **Don't Repeat Yourself (DRY)**
- General principle
- In programming:
  - define a "magic number" **once** in a **constant variable**
  - define an algorithm **once** in a **function**
- **Ideal Division of Labor** principle:
  - **humans** should not copy-paste the same idea
  - tedious, error-prone
  - **computer** should do that by looking at the **SPOT**

Constant Variables

- Data type may be preceded by `const`
- const variables
  - must be initialized
  - cannot be re-assigned
- **Best practice:** magic numbers in constant variables, not literals
- Cleaner
- Easier to change
- Principles of
  - least privilege
  - detect bugs at compile time
- Example: chromium::cc::layers::Viewport::kPinchZoomSnapMarginDips

```
const int kMinimumAgeForSeniorDiscount{55};
const double kSeniorDiscountPercentage{10.0};

double PriceAfterSeniorDiscount(
  double full_price, int age) {
  double savings{0.0};
  if (age >= kMinimumAgeForSeniorDiscount) {
  savings = full_price *
            kSeniorDiscountPercentage / 100.0;
  }
  return full_price - savings;
}
```