University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Implications of Overloading

Overloading Implies Rules for Matching Calls to Functions

Since functions can be overloaded,
- a **compiler must have rules**
- **for choosing** amongst functions
- with the same name.

Let's first ask: **how different**
- **do** two **definitions** of a function **need to be**
- for the compiler **to distinguish them?**

C++ Supports Extremely Minor Variations

High-level answer:
- **C++ allows for extremely minor variations**;
- use them at your own risk.

Variations on functions
- are not much different
- than naming two variables
- VaRiAbLe and vArIaBlE.

C++ Distinguishes Between Similar Types

For example,
- C's **default conversions** are **not assumed**,
- so the following can be differentiated:

```
int operator+= (int i);
int operator+= (char c);
```

One can also distinguish
- between signed and unsigned values,
- between **`const`** and non-**`const`** values,
- and so forth.

Matching Happens on a Per-Argument Basis

How does a compiler choose?

The **basic rule**:
- pick the **"most derived" class**
- **for each argument**.

The result is **not always unique**, as you already know.

Compilers report **errors if** the **results** are **ambiguous**.

Pitfall: Overloading too Finely Can Confuse Users

Exact rules exist, but don't seem to be widely known, understood, portable, and so forth.*

My advice: **as with precedence,**
- **if you don't know the answer,**
- **don't try to look it up**:
- instead, avoid using it.

Matching overloaded calls is much more dangerous—one can 'steal' calls from existing code by creating new functions that provide better matches.

*See ECE409 notes L7P5-6 for more detailed comments, or the C++ ARM or standards for a definitive version.

Some Operators Cannot be Overloaded

C++ allows most, but not all, operators to be overloaded.

Operators than cannot be overloaded include
- member access ("`.`"),
- pointer to member function invocation ("`.*`"),
- conditional expressions ("`? :`"), and
- scope identification ("`::`").

C Equivalences May Not Be Valid in C++

In C++, **C's equivalences may no longer hold**.

For example, **pointer-like and array-like objects are not necessarily the same**
- `array[10]` (calls `operator[]`)
- may not be the same as
- `*(array + 10)` (calls `operator+` and `operator*`).

These operators **can be defined to be consistent**, however.

Some Expressions Cannot Be Made Equivalent

Other equivalences **cannot be made consistent** if overloading is used to substantially change some operators.

For example, pointer dereference:

- `inst->member`
- `(*inst).member`
- `inst[0].member`

The latter two expressions use "`.`", which **cannot be overloaded**.

Assignment Requires First Destroying the Old Instance

Similarly, given

```
ALPHA a;
```

the **following are not equivalent**:

```
ALPHA b = a;  // copy constructor
b = a;    // operator= (assignment)
```

For a new variable, the first version is better:

- **work** may be **required to destroy b**
- (for example, removing **b** from a sorted list or a binary search tree).
- **before copying** the new value **from a**.

Copy Constructor and Assignment not Equivalent in C++

Note that

- the **copy constructor** and
- **assignment** (operator=)
- are **not equivalent in C++**.

The **default versions are the same** (copy constructor detailed in earlier slides), but

- overriding one does NOT override the other,
- which continues to use the default version
- and compilers will NOT warn you.

Treating Instances as C Variables Generates Useless Work

One last topic: variable declarations and single-assignment.

C++ instances are "always" valid:

- constructed when they are declared, and
- destructed when they leave scope (and can no longer be accessed).

Treating them as C variables generates useless work by forcing initialization before information for initialization is available.

C++ Allows Variable Declarations Between Statements

To address the problem, **C++**
- **allows variable declarations**
- to be **interleaved with statements**
- (later, this feature was back-propagated into **C**).

`for` loop iteration variables
- of a single base type
- can also be declared in the init clause
- (not supported for other loops).

Avoiding Extra Work with Conditionals Still Tricky

The problem is not completely solved, but the following approach may help for conditionals:

```
int choice = some_calculation ();
if (choice) {
    ALPHA a = someWork ();
} else {
    BETA b = someOtherWork ();
}
SomeType var =
  (choice ? SomeType (a) : SomeType (b));
// use var ...
```


Op-Assign Operators Should Be Used When Possible

Use of op-assign operators
- +=, -=, *=, and so forth
- is also **encouraged in C++**.

Why?

Are these all the same?

```
A = A + B
A = B + A
A += B
```

One answer: not if + is string concatenation.

Compilers May Fail to Perform "Obvious" Optimizations

Even restricting our attention to **`A = A + B`**, proving equivalence may be challenging.

A **compiler**
- **must consider aliasing**:
- **can A be changed before** the **sum operation is complete**?

Compiler can analyze the problem
- if both functions, and
- may be able to transform into **`A += B`**.

In contrast, **programmer must handle aliasing** when implementing **`operator+=`**.

