C++ Allows Functions to Be Overloaded

In C++, functions can be overloaded,
◦ meaning that one function name
◦ may have multiple definitions,
◦ (overloading was mentioned briefly
  when we discussed constructors).
For each call to a function,
◦ the compiler selects and generates a call
  to one of the versions
◦ based on the number and types of arguments.
◦ Selection must be unambiguous, but the
  selection rules are somewhat complicated.

Compiler Must Be Able to Choose Unambiguously

Here’s a simple example:
// product of two 32-bit integers
int32_t prod (int32_t a, int32_t b);
// product of two doubles
double prod (double a, double b);
Which version should be called for ...
prod (42, 1.5) ?
Neither. The call is ambiguous.

Compiler Counts Automatic Conversions

Why?
Among other rules, the compiler counts
automatic conversions and disallows ties.
Given prod (42, 1.5),
◦ the compiler can either
  convert the 42 to a double,
  or convert the 1.5 to an int32_t.
◦ The compiler will not choose.
Overloading Support Operators with User-Defined Types

Overloading also extends to operators:
• most operators can be redefined in ways specific to the types of their operands.

In fact,
• “natural” use of operators with user-defined types
• was the original goal of overloading.
• Overloading is necessary to make operator redefinition useful.

Example of Code Simplification with Operator Overloading

The canonical motivating example for operator overloading is complex numbers.
Consider this task:
• given complex numbers \( P \) and \( Q \).
• calculate \( R = P^2 + Q^2 \).

Looks nice in math notation, right? Here’s C:

\[
R = \text{complex\_add}\left(\text{complex\_multiply}\left(P, P\right), \text{complex\_multiply}\left(Q, Q\right)\right);
\]

And here’s C++:

\[
R = P * P + Q * Q;
\]

Overloading Integrates Naturally with Auto-Conversion

Redefining operators should also
• fit in with the “natural” conversions among \texttt{int}, \texttt{float}, \texttt{double}, and so forth.

For example, one should not have to define all of these multiplication operators separately:
• \texttt{complex} * \texttt{int}
• \texttt{complex} * \texttt{double}
• \texttt{int} * \texttt{complex}
• \texttt{double} * \texttt{complex}
• and so on...

C++ Combines Implicit Casts with Symmetric Definitions

To simplify the definitions, C++
• allows creation of new implicit casts (auto-conversions)
• and uses friend functions for symmetry.

A friend function is
• a function outside of a class (neither a member function nor a class function)
• with full access rights to the class
• (which, of course, must appear as a friend in the class definition).
Single-Argument Constructors Create Implicit Casts

```cpp
class complex {
    // ...
public:
    complex (int32_t real_part);
    complex (double real_part);
friend complex operator*(
        const complex& a,
        const complex& b);
}
```

- Creates implicit cast from `int32_t` to `complex`.
- Creates implicit cast from `double` to `complex`.

Use `explicit` to Avoid Creating Implicit Casts

The constructors enable the following:

```cpp
complex c = 5; // cast from int32_t
double d = 42.9; // from double
```
To avoid creating implicit casts, add `explicit` before the constructor.

Be aware that **casts can be chained**.
- Eliminating the `double` cast alone affects ... almost ... nothing:
- the compiler casts from `double` to `int32_t` to `complex`!

Operators Often Return Whole Instances

```cpp
class complex {
    // ...
public:
    complex (int32_t real_part);
    complex (double real_part);
friend complex operator*(
        const complex& a,
        const complex& b);
}
```

- Friend functions are in the global namespace, not in the class, so access control has no relevance.
- Return type is the whole instance!

Instance Returned on Stack, Becomes Temporary in Caller

**Why does the operator return an instance?**

Other options?
- Pointer to automatic variable doesn’t work.
- Dynamic allocation is more expensive.

So an instance is returned on the stack.

Back in the caller,
- the result is a **temporary**.
- Eventually, a destructor must be called.*
- Usually at the end of the statement.

*Except if a new variable is constructed from the temporary.
Operators' Arguments Should be Constant References

class complex {
    // ...
    public:
        complex (int32_t real_part);
        complex (double real_part);
        friend complex operator*
             (const complex& a,
              const complex& b);
}

Arguments are constant references.

A Reference is a Pointer in Disguise

What's a reference?
- A pointer
- disguised syntactically
- as the value to which the pointer points
- (more detail coming next).

const is same as with pointers: value to which the reference refers is not changed.
Without const, implicit casts fail (the casts produce a temporary complex).

Example of Code Appearance with Pointers

Why not just use pointers as arguments?
Remember the motivating example,

\[ R = P^2 + Q^2 \]

Here's that code with pointer arguments...

\[ R = *(\&P * \&P + \&Q * \&Q); \]

But class instances can be large,
- so we want to avoid passing them as parameters and
- returning them when possible.

Reference Looks Like a Type, but Implemented as Pointer,

So what’s a reference?

A reference is
- implemented identically to a pointer,
- but is syntactically equivalent to the base type (the type to which the pointer points).
References Cannot be Modified

Ambiguity: what should happen with
\[ A = B; \]
if both \( A \) and \( B \) are references?
Copy the pointer?
Or copy the contents?
To resolve the ambiguity,
\* references are single-assignment in C++.
\* One can NOT change their value.
So the code above copies the contents.

Let's Use an Example to Illustrate Appearance

Let's do an example.
Let's write a piece of code two ways
\* one using pointers, and
\* one using references.
Both versions generate exactly
the same assembly code.
The only difference is how they look.

Use a Class with Get/Set Routines

class ALPHA {
    int32_t num;
public:
    ALPHA (int32_t val) : num (val) { }
    int32_t getNum () const {
        return num;
    }
    void setNumFromPtr (int32_t* where) {
        num = *where;
    }
};

A little odd, but useful for our purpose.

The Code Using Pointers

// a synchronization primitive...
bool compareXchgPtr (ALPHA* alpha, int32_t* compare, int32_t* newVal) {
    if (alpha->getNum () == *compare) {
        alpha->setNumFromPtr (newVal);
        return true;
    }
    return false;
}
The Code Using References

```c
// a synchronization primitive...
bool compareXchgRef (ALPHA& alpha, int32_t& compare, int32_t& newVal)
{
    if (alpha.getNum () == compare) {
        alpha.setNumFromPtr (&newVal);
        return true;
    }
    return false;
}
```

References Allow a Choice of Argument Semantics

Finally, to call the functions...

```c
compareXchgPtr (&a, &one, &two);
versus
compareXchgRef (a, one, two);
```

Notice that references allow a programmer to redefine argument semantics:
- on a per-argument basis:
- some may be call-by-value, while others are call-by-reference (pointer).

Another Round of “Help Prof. Lumetta!”

```c
I have a bug.
This loop seems to hang.
while (42 != i) {
    foo (i);
    x = bar (i);
    zap (i, x);
}
```

Can you help?

References are Easily Abused

As Stroustrup says, any language can be misused.
The worst part in my view:
- one can change argument style
- without generating warnings at call sites (code that uses the function)!
In other words,
- some code may assume that an argument doesn’t change,
- but someone “redefines” the function
- so that it can change!
Good Code is Not a Puzzle

What can be done?

Either
1. choose function names that make it obvious which arguments might change value (good luck!), or
2. mark arguments that might change in the code (as with C).

Can We Do Better for Class Instances?

What about arguments that don’t change?

Most of your data is class instances.

One rarely wants instances copied onto the stack.

And using “&” everywhere is a little clunky.

Use Pointers for Modifiable Arguments

Solution:
* use const with reference arguments!
* const was introduced for C++, then back-propagated into C.

Avoid using non-const reference arguments.

If an argument can be modified, use a pointer.

Named Return Value Optimization Reduces Overhead

One last implementation aspect:
named return value optimization
In practice,
* most C++ compilers
* transform a returned instance
* into an implicit instance pointer
* as a new first argument,
* returning either void or the instance pointer.
How Named Value Return Optimization Works

A **caller allocates space** for an instance
◦ (usually on the stack),
◦ then **passes a pointer** to the instance,
◦ and the **function fills in the bits**.
Technically,
◦ a **copy constructor should be called**,  
  ◦ but that call (or calls) is **often omitted**,  
  ◦ (even if the constructor has side effects).
The optimization is thus technically incorrect, but it’s widely used for performance.*
  
* Newer C++ standards define it to be correct.

What Should Be Called for an Addition?

As an example, assume that we have

```cpp
complex operator+ (const complex& x,  
  const complex& y);
```

and a variable

```cpp
complex a;
```
And then we **write**

```cpp
complex b = a + a;
```

What should happen?

Addition in Theory Implies Three Constructors

```cpp
complex b = a + a;
```

What should happen?

In theory, an **instance**
◦ is **constructed within operator+**, then
  ◦ **copy constructed as the return value**,  
  ◦ then **copy constructed again as b**.

Addition in Practice Usually Calls One Constructor

```cpp
complex b = a + a;
```

What should happen?

In practice,
◦ **variable b resides in the caller’s stack frame**,  
  ◦ so a **pointer to b is passed to operator+**,  
  ◦ and **operator+ constructs b**.