CSI 310: Slides on Reference and Value Parameters
Section 4.2

Various Kinds of Value and Reference Parameters — See
void fun(int VP, int& RP, const int& CRP) {
    int CV;
    CV = 98; // Changes CV
    RP = CV; //caller's argument variable!
    //inside called fun's body...
    //value parameter.
    //const ref. parm.
    //const int CRP
    //int RP
    //reference parm.
    //value parameter.
    //void fun (int VP,'
The parameter class does copy constructor is run if there is any:

This happens when the function is called but before its body runs.

The corresponding local variable and those variables are initialized by copying each argument into parameter, and those variables are initialized by copying for each value parameter.

In general, don't declare with & before parameter:

```c
void make jint int* j, ptr, int cnt, bge parm;  // Value Parameters
```

Value Parameters Examples:

Reference:
The function declaration determines the kind of each parameter: value or
Value parameter that is a pointer.

```c

void Make42(int *w, int *u)
{
    assert (w == u);
    assert (w == u);
    Make42(w, u);
    // Prints GARBAGE
    cout »w »u;
    w = new int;
    int w;
    int *w;
    } 
}

(int main
}

}* = 42; *p2 = 42;

*P1 = 42; *P2 = 42;

// P1 and P2 are 42 // vars with addresses
// Post: The values of int.
// addresses of int vars. // pre: Both P1 and P2 are legal
// void Make42(int *P1, int *P2)
```cpp
main() { int* m_ptr; int m_int;
  m_ptr = new int;
  Make42(m_ptr, &m_int);
}

void Make42(int *P1, int *P2) {
  *P1 = 42; *P2 = 42; return;
}

same
variable
different
times
past

future

void Make42(int* m_ptr, int* m_ptr) {
  m_ptr = new int;
  m_ptr = new int;
}

same
variable
past

future

different
variable

past

future

same
variable

past

future

same
variable

past

future

same
variable

past

future

same
variable

past

future

same
variable

past

future

same
variable

past

future

same
variable

past
```
reference \( \text{int \& \ q} \) from \text{value \& int}.

warning: initialization of non-const

\text{& prints the warning:}

and the reference parameters refer to THEM.

Answer: Temporary variables are created, initialized to the argument values,

\[
\begin{cases}
R_{p1} = 0 ; & R_{p2} = 0 ; \\
\end{cases}
\]

\text{void \ fun(\text{int \& \ p1, \ int \& \ p2})}

\text{where \ fun() \ is}

\text{fun(38, \ x+39) ;}

\text{We call}

\text{parameter?}

What if a constant or a computed value is used as an argument for a reference

An annoying detail:
PRINT ANY WARNINGS.

Rule for CS1310: All compilations must be done with -Wall and must NOT

(iii) (not 39) (not 0

complies and might print

You get a warning about a conversion that discards const; BUT the program

cout << TINTE << endl;

fun(TINTE, TINTE);

const int TINTE = 39;

What if? void caller()
void makeAll42(double data[], size_t n) {
    data[0] = 42.0;
    for (size_t i = 0; i < n; i++) {
        data[i] = 42.0;
    }
}

main() {
    double A[4] = {42.0, 0.0, 42.0, 0.0};
    makeAll42(A, 4);
}

variables belonging to \texttt{main} are automatic (local) variables.
\texttt{A} is an automatic (local) variable belonging to \texttt{main}.

\texttt{void makeAll42(doble *data, size_t n)} is a value parameter that is a pointer.
\texttt{data} here is a value parameter that is a pointer.
\texttt{void makeAll42(doble [n], size_t n)}

is treated as a pointer to the first array element.

\texttt{void makeAll42(doble data[n], size_t n)}

like data in "surprising twist" unique to \texttt{C++}. An array parameter.
callee’s pointer variable.

into the callee’s size_t variable \textbf{AND} writes the array’s address into the
This function DECIDES how big an array to allocate, and writes that size back

{ 
    p = new double[n]; // allocate array
    cout << "Enter " << n << " elements: 
";
    for (int i = 0; i < n; ++i) {
        cout << "Enter element " << i + 1 << " : 
";
        p[i] = get_double();
    }
}

A reference parameter can have any type including \textbf{POINTER}.

\textbf{Avoid COPYING} the argument.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{avoid MODIFY} the argument.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

\textbf{const &} parameters \textit{are} \textbf{not} a variable.

Reference parameters: like in
try. `cx: t` cannot declare pointers to references

++-`wait try.cxx` `cxx`

"pointer to Reference" doesn't exist in C++.

"pointer" applies to type of data.

"Reference" / "Value" applies to parameter passing.

What about `fun (int & x)`?

<table>
<thead>
<tr>
<th>Reference Parameter</th>
<th><code>fun (int &amp; x, PId)</code></th>
<th>a pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Parameter That Is</td>
<td><code>fun (int &amp; x, PId)</code></td>
<td>not a pointer</td>
</tr>
<tr>
<td>Pointer Type</td>
<td><code>fun (int * PId)</code></td>
<td>a pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference Parameter</th>
<th><code>fun (int &amp; x, PId)</code></th>
<th><code>fun (int &amp; x, PId)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Parameter That Is</td>
<td><code>fun (int &amp; x, PId)</code></td>
<td>not a pointer</td>
</tr>
<tr>
<td>Pointer Type</td>
<td><code>fun (int * PId)</code></td>
<td>not a pointer</td>
</tr>
</tbody>
</table>

Non-Pointer Type

Orthogonality?