Pass by Value

1. `int abc(int a, int b, int c) {`
2.     return a + b * c;
3. }

4. a, b, and c are the **formal parameters**.

   5. `z = abc(2, x, y)`

6. 2, x, and y are the **actual parameters**.

At run-time, actuals are copied into the formal parameters using the copy constructor of each formal’s data type.
Pass by Value

When a function finishes, **destructors** for each data type are called to destroy the formals.

Functions do not change the original actuals.

```c++
1. int changeNothing(int a, int b, int c) {
2.     a = b + c;
3.     c = 4 + a;
4.     return a;
5. }
6. 7. cout << "A: " << a << " B: " << b << " C: " << c << endl;
8. changeNothing(a, b, c);
9. cout << "A: " << a << " B: " << b << " C: " << c << endl;
```

Output will be the same before and after the method call.
Templates

What if you want the same method for different data types?

1. template<class T>
2. T abc(T a, T b, T c)
3. {
4.    return a + b * c;
5. }

6. int result = abc<int>(5, 6, 7);

7. double result2 = abc<double>(5.4, 9.7, 3.4);

Two sets of code generated at compile-time.
Pass by Reference

Copying and destroying all those parameters can get expensive. If T were a type matrix with 1000 elements, then it would require 3000 operations to copy it in and another 3000 to destroy it.

```cpp
1. template<class T>
2. T abc(T& a, T& b, T& c)
3. {
4.   return a + b * c;
5. }
```

Pass by reference does not copy anything. Uses original actuals. Changes in the function are reflected after invocation.
Pass by Reference

1. int changeSomething(int &a, int &b, int &c) {
2.     a = b + c;
3.     c = 4 + a;
4.     return a;
5. }
6.
7. cout << "A: " << a << " B: " << b << " C: " << c << endl;
8. changeSomething(a, b, c);
9. cout << "A: " << a << " B: " << b << " C: " << c << endl;

Two output statements print two different sets of values.

Useful if you need more than one return value.
Const Reference Parameters

But what if you want to guarantee that values of parameters won’t be changed?

1. template<class T>
2. T abc(const T& a, const T& b, const T& c)
3. {
4.     return a + b * c;
5. }

Compiler insures that function does not change parameters.

C++ , if left to its own devices, is not at all helpful.

```cpp
#include <iostream>
#include <stdexcept>

using namespace std;

template<class T>
T divide(T a, T b) {
    if (b == 0) {
        throw runtime_error("Divisor cannot be zero");
    }
    return a / b;
}

int main() {
    cout << "Result of dividing 2 by 4: " << divide<int>(2, 4) << endl;
    try {
        cout << "Result of dividing 2 by 0: " << divide<int>(2, 0) << endl;
    } catch (exception& ex) {
        cerr << "The following exception was thrown: " << ex.what() << endl;
        return 1;
    }
    return 0;
}
```

Look for 0 in divisor and print something better than “Core dumped”
**Pointers**

We can store the value itself on the stack frame

```java
int num = 42;
```

Or we can store a memory address that points to the value which lives somewhere else in memory.

```java
int* numPtr = &num;
```

Pointers make it easier and more efficient to share data in a program.

In Java, anything that isn’t a primitive is a pointer — but it all happens magically behind the scenes.
Pointers

1. `int a = 1;`
2. `int b = 2;`
3. `int c = 3;`
4. `int* p;`
5. `int* q;`

```
p = &a;  // set p to refer to a
q = &b;  // set q to refer to b
```

```
c = *p;
p = q;
*p = 13;
```
The **new** operator

Use the **new** operator to allocate memory at run-time

1. `int *y = new int (10);`

This reserves memory for an integer and stores a 10 in that slot.

1. `int *y = new int;`  
2. `*y = 10;`

This does the same thing in two separate steps.

Why would you ever need this?
Variable-Length Arrays

When you know the size of the array is always going to be the same:

```cpp
char lettersInAlphabet[26];

int positiveNumbersBeforeFive[] = {1, 2, 3, 4};
```

When the size of the array will vary from one run of the program to the next, you need dynamic memory allocation.

```cpp
float *classGrades = new float [n];
```
The **delete** operator

C++ has no garbage collector!

If you don’t explicitly delete everything you created with **new**, you’ll eventually run out of memory.

```cpp
delete y;
delete [] x;
```
Two-Dimensional Arrays

When you know the dimensions at compile time:

```c
char c[7][5];
```

When only the number of columns is known:

```c
char (*c)[5];
c = new char [n][5];
```

When the number of columns is unknown:

1. `char **x;`
2. `x = new char *[numberOfRows];`
3. 
4. `for (int i = 0; i < numberOfRows; i++)`
5. `x[i] = new char [numberOfColumns];`
Deleting a Two-Dimensional Array

1. // delete the memory for each row
2. for (int i = 0; i < numberOfRows; i++)
3.     delete [] x[i];
4. 
5. // delete the row pointers
6. delete [] x;
7. x = NULL;
Exception Handling

In case your array could potentially have a few billion elements...

1. `float *x;`
2. `try {x = new float [n];}`
3. `catch (bad_alloc e)`
4. `// enter only when new fails`
5. `cerr << "Out of Memory" << endl;`
6. `exit(1);`
7. `}`
Dynamic Memory Mistakes

Easy to make mistakes.

1. `int* p;` // allocate the pointer, but not the pointee
2. `*p = 42;` // this dereference is a serious runtime error

No NullPointerException! Just weird random side-effects...

Additional Resources on Pointers:

http://cslibrary.stanford.edu/102/PointersAndMemory.pdf
http://www.gamedev.net/page/resources/_/technical/general-programming/a-tutorial-on-pointers-and-arrays-in-c-r1697
http://www.fredosaurus.com/notes-cpp/newdelete/50dynamalloc.html
Defining your own data types - Enumerations

1. `enum signType {plus, minus};`

With this, you can declare variables of type `signType` that can have a value of either plus or minus.

`signType x = plus;`
**Defining your own data types - Classes**

```cpp
1. class currency
2. {
3. 
4.   public:
5.     // constructor
6.     currency(signType theSign = plus,
7.                  unsigned long theDollars = 0,
8.                  unsigned int theCents = 0);
9.     // destructor
10.    ~currency() {}  
11.    void setValue(signType, unsigned long, unsigned int);
12.    void setValue(double);
13.    signType getSign() const {return sign;}
14.    unsigned long getDollars() const {return dollars;}
15.    unsigned int getCents() const {return cents;}
16.    currency add(const currency&) const;
17.    currency& increment(const currency&);
18.    void output() const;
19.   
20.   private:
21.     signType sign;          // sign of object
22.     unsigned long dollars;  // number of dollars
23.     unsigned int cents;     // number of cents
24. }
```
Constructors

```cpp
1. currency(signType theSign = plus,
           unsigned long theDollars = 0,
           unsigned int theCents = 0);
```

Same name as the class.

Does not return a value.

All of the following invoke the constructor:

```cpp
1. currency f, g(plus, 3, 45), h(minus, 10);
2. currency *m = new currency (plus, 8, 12);
```

Clarification: if a parameter is followed by an equals sign and value, this is a default value, used when one is not specified.
Destructors

~currency() {}

Called automatically whenever object goes out of scope.
Same name as the class.
Does not return a value.
Used to free up any dynamically allocated memory.
What's with the const?

1. unsigned long getDollars() const { return dollars; }
2. unsigned int getCents() const { return cents; }
3. currency add(const currency&) const;

These functions do not change the invoking object (the currency object). These are called constant functions.

Notice that three of the functions do not have a const:
1. void setValue(signType, unsigned long, unsigned int);
2. void setValue(double);
   currency& increment(const currency&);

The last one returns a reference to the invoking object after adding the parameter object to it.
External Implementation of Methods

Can be defined in global scope using :: operator.

1. currency::currency(signType theSign, unsigned long theDollars,
   unsigned int theCents)
3. { // Create a currency object.
4.    setValue(theSign, theDollars, theCents);
5. }
void currency::setValue(signType theSign,
                       unsigned long theDollars, unsigned int theCents)
{
    // Set currency value.
    if (theCents >= 99)
        // too many cents
    throw illegalParameterValue("Cents should be < 100");

    sign = theSign;
    dollars = theDollars;
    cents = theCents;
}

void currency::setValue(double theAmount)
{
    // Set currency value.
    if (theAmount < 0) {sign = minus;
                      theAmount = -theAmount;}
    else sign = plus;

    dollars = (unsigned long) theAmount;
    // extract integer part
    cents = (unsigned int) ((theAmount + 0.001 - dollars) * 100);
    // get two decimal digits
}
currency currency::add(const currency& x) const

{ // Add x and *this.
    long a1, a2, a3;
    currency result;
    // convert invoking object to signed integers
    a1 = dollars * 100 + cents;
    if (sign == minus) a1 = -a1;
    // convert x to signed integer
    a2 = x.dollars * 100 + x.cents;
    if (x.sign == minus) a2 = -a2;
    a3 = a1 + a2;
    // convert to currency representation
    if (a3 < 0) {result.sign = minus; a3 = -a3;}
    else result.sign = plus;
    result.dollars = a3 / 100;
    result.cents = a3 - result.dollars * 100;
    return result;
}

Does a value return – uses default copy constructor and copies result back to calling frame.
Application of class currency

```cpp
#include <iostream>
#include "currency.h"

using namespace std;

int main()
{
    currency g, h(plus, 3, 50), i, j;

    // try out both forms of setValue
    g.setValue(minus, 2, 25);
    i.setValue(-6.45);

    // do an add and output
    j = h.add(g);
    h.output();

    // do an increment and add
    j = i.increment(g).add(h);

    cout << "Attempting to initialize with cents = 152" << endl;
    try {i.setValue(plus, 3, 152);}
    catch (illegalParameterValue e) {
        cout << "Caught thrown exception" << endl;
        e.outputMessage();
    }
    return 0;
}
```
Add and Increment replaced with + and +=

1. `class` `currency`  
2. `{`  
3. `    public:`  
4. `        // constructor`  
5. `        currency(signType theSign = plus,`  
6. `            unsigned long theDollars = 0,`  
7. `            unsigned int theCents = 0);`  
8. `        // destructor`  
9. `        ~currency() {}`  
10. `        void setValue(signType, unsigned long, unsigned int);`  
11. `        void setValue(double);`  
12. `        currency operator+(const currency&) const;`  
13. `        currency& operator+=(const currency& x)`  
14. `            {amount += x.amount; return *this;}`  
15. `        void output(ostream&) const;`  
16. `    private:`  
17. `        long amount;`  
18. `};`
Operator Overloading

```cpp
1. currency currency::operator+(const currency& x) const
2. {// Add x and *this.
3.    currency result;
4.    result.amount = amount + x.amount;
5.    return result;
6. }
7. 8.
8. void currency::output(ostream& out) const
9. {// Insert currency value into stream out.
10.    long theAmount = amount;
11.    if (theAmount < 0) {out << '-';
12.       theAmount = -theAmount;}
13.    long dollars = theAmount / 100; // dollars
14.    out << '$' << dollars << '.';
15.    int cents = theAmount - dollars * 100; // cents
16.    if (cents < 10) out << '0';
17.    out << cents;
18. }
19. 20.
20. // overload <<
21. ostream& operator<<(ostream& out, const currency& x)
22. {x.output(out); return out;}
```
```cpp
#include <iostream>
#include "currencyOverload.h"

using namespace std;

int main()
{
    currency g, h(plus, 3, 50), i, j;
    // try out both forms of setValue
    g.setValue(minus, 2, 25);
    i.setValue(-6.45);
    // do an add and output
    j = h + g;
    cout << h << " + " << g << " = " << j << endl;
    // do two adds in a sequence
    j = i + g + h;
    // do an increment and add
    j = (i += g) + h;
    return 0;
}
```
Friend Statements

By default, private members are not accessible outside a class. But sometimes, we want to give certain classes and functions access. We do this with friend statements.

1. `class currency {`
2. `    friend ostream& operator<<(ostream&, const currency&);`
3. `    public: `

Friend Statements

1. // overload <<
2. ostream& operator<<(ostream& out, const currency& x)
3. {// Insert currency value into stream out.
4.     long theAmount = x.amount;
5.     if (theAmount < 0) {out << '-';
6.         theAmount = -theAmount;}
7.     long dollars = theAmount / 100; // dollars
8.     out << '$' << dollars << '.';
9.     int cents = theAmount - dollars * 100; // cents
10.    if (cents < 10) out << '0';
11.    out << cents;
12.    return out;
13.}
Preprocessor Directives

At the top of your header file containing your currency class

1. ifndef currency_
2. define currency_

And at the bottom

endif

This prevents the compiler from trying to compile your currency class more than once.
class illegalParameterValue
{
    public:
    illegalParameterValue() :
        message("Illegal parameter value") {}
    illegalParameterValue(char * theMessage)
    {message = theMessage;}
    void outputMessage() {cout << message << endl;}
    private:
    char * message;
};

int abc(int a, int b, int c)
{
    if (a <= 0 || b <= 0 || c <= 0)
        throw illegalParameterValue ("All parameters should be > 0");
    return a + b * c;
Catching your Custom Exception

```cpp
int main()
{
  try { cout << abc(2, 0, 4) << endl; }
  catch (illegalParameterValue e) {
    cout << "The parameters to abc were 2, 0, and 4" << endl;
    cout << "illegalParameterValue exception thrown" << endl;
    e.outputMessage();
    return 1;
  }
  return 0;
}
```