Dynamic Allocation

Instead of malloc() and free(), use:

```
Cat *c = new Cat();    // Goes on the heap
…
delete c;              // Remove it manually
```

- Variable must be a pointer.
- **No Garbage Collector!** If you forget to delete, you have a memory leak.
- If not careful, can have stale pointers, double deletions, etc.

Allocating Variables Summary

Two ways:

1. Declare an object variable: `Cat x;`

   Object is created. It is automatically destroyed when variable goes out of scope

   Safe and PREFERRED.

2. Pointer to object: `Cat *x = new Cat();`

   You must remember to delete it!

   Can lead to memory problems! Only use when necessary.

Basics of **const**

Can be used to make **anything** immutable!

```
const int y = 7;
y = 12;          // won't compile

const string s("huh");
s = "duh";   // won't compile
```

Literals like 25.9 or "hello" are always considered const. You can't mutate a literal.

const with Pointers

- **pointer to constant object:**

```
string const *x  = new string("foo");
```

- **constant pointer to an object:**

```
string * const x = new string("foo");
```

- **constant pointer to constant object:**

```
string const * const x = new string("foo");
```

Hint: Read these right-to-left!

const with References

```
const int y = 12;    // const data
int x = 7;           // non-const data
```

- A **const reference** can refer to anything, but cannot be used to modify the data:

```
int const &z = x;    // fine
int const &z = y;    // fine
z = 15;              // doesn't compile!
```

- A **non-const reference** cannot refer to const data, but can be used to modify the data:

```
int &z = x;     // fine
int &z = y;     // doesn't compile!
z = 15;         // fine
```

Recall: Passing Arguments in **Java**

Java: Everything is passed by value – Either a primitive gets copied or a reference gets copied.

```
void foo(int x, Cat y) {
…
}

int a = 4;
Cat c = new Cat();
foo(a, c);
```

Both arguments are copied to the parameters. That makes **x** a separate copy of 4, and **y** becomes a separate reference to the same Cat object. (We say **c** and **y** are "aliases".)

4 Ways to Pass Arguments in C++…

1. **Pass by Value**

```
void foo(string b)
{
    …
}

String y("hi");
foo(y);
```

A **copy** of the string is made!!! HOW???

Important Concept: When passing an object by value, the copy constructor for the class in implicitly called!

If you don't provide a copy constructor for a class, one is provided for you that makes shallow copies.

4 Ways to Pass Arguments in C++…

2. **Use a pointer.** (Really just passing a pointer by value.)

```
void foo(string *b)
{
    …
}

String y("hi");
foo(&y);
```

- Used frequently in C when we want the function to modify our object.
- Not used often in C++ because… Pointers are ugly and bug-prone – in C++ we have reference variables!

4 Ways to Pass Arguments in C++…

3. **Pass by (non-const) reference**

```
void foo(string &b)
{
    …
}
String y("hi");
foo(y);
```

b is now a reference to the same string.

Important: You cannot pass a const argument to a non-const reference parameter!

4 Ways to Pass Arguments in C++…

By the way…

Pass by (non-const) reference works with primitives too.
Try doing **this** in Java:

```
void triple(int &x)
{
   x *= 3;
}

int y = 7;
triple(y);      // y is now 21
```

4 Ways to Pass Arguments in C++…

4. **Pass by const reference**

```
void foo(string const &b)
{
    …
}

String y("hi");
foo(y);
```

- Function cannot modify the object. (Safe.)
- Pass by value is also "safe", but makes a copy of the object which is a waste of time.

Since this method is both **safe** and **efficient**, it is the most popular one!

Rule of Thumb for Parameters

If (function **must** modify the original copy)
{
use (non-const) reference
}
else
{
if (data is really small)
pass by value
else
use const-reference
}

Example: passingParameters.cpp

Which of the following are “safe” and which allow the function to modify the original copy of the data?

- Pass by value
- Pass using a pointer
- Pass by (non-const) reference
- Pass by const reference

Other ideas:

- Function can’t modify const reference parameter
- You can’t pass const data to non-const reference parameter

3 Ideas for Return Values

1. Return by value:

```
string f() {…}
```

Important concept: Returning an object by value makes a copy by implicitly invoking the copy constructor!

Completely safe, but wastes time.

3 Ideas for Return Values

2. Return a reference

```
string & foo() {…}
```

- Are you supposed to delete the object you get back? Sometimes yes, sometimes no!

Aside… Weird Example

```
string & f(string &a)
{
   …
  return a;
}

String x("whatever");
f(x) = "hi";    // Huh?? It works!!
```

3 Ideas for Return Values

3. Return a pointer

```
String * f() {…}
```

- Again, are you supposed to delete the thing?
- Also… other memory errors are possible. Very common error: Returning a pointer to an object that gets deleted when the function ends!