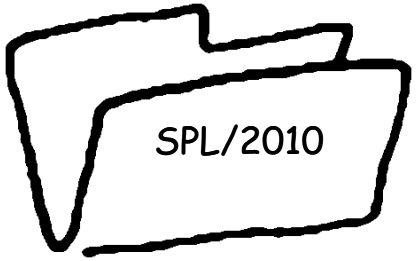
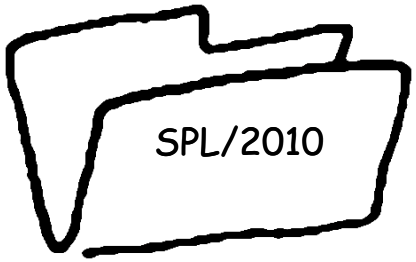
Pointers and Parameter Passing in C++



In Java

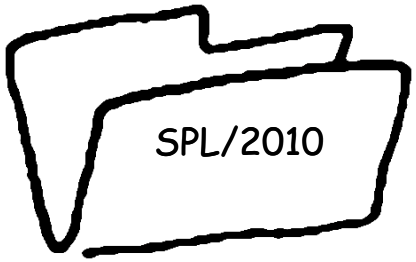
- Primitive types (byte, short, int…)
  - allocated on the stack
- Objects
  - allocated on the heap



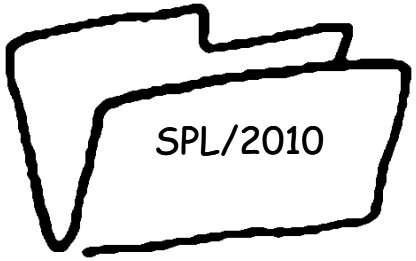
Parameter passing in Java

- Myth: "Objects are passed by reference, primitives are passed by value"
- Truth #1:
  Everything in Java is passed by value.

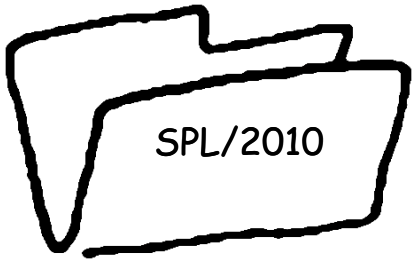
  (Objects, are never passed *at all*)
- Truth #2: The values of variables are always **primitives** or **references**, never objects



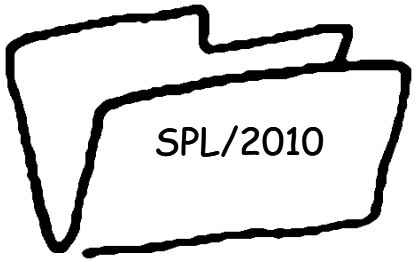
- **Pass-by-value**
  - actual parameter is fully evaluated and the resulting value is *copied* into a location being used to hold the formal parameter's value during method/function execution.
  - location is typically a chunk of memory on the runtime stack for the application



- **Pass-by-reference**
  - formal parameter merely acts as an *alias* for the actual parameter.
  - anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter

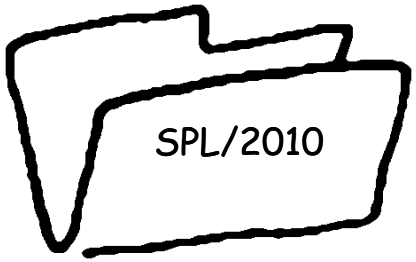


```
public void foo(Dog d)
{
d = new Dog("Fifi"); // creating the "Fifi" dog
}

Dog aDog = new Dog("Max"); // creating the "Max" dog
// at this point, aDog points to the "Max" dog
foo(aDog);
// aDog still points to the "Max" dog
```



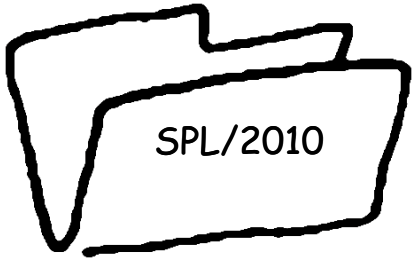
foo(d);

- passes the ***value of d*** to foo; it does not pass the object that d points to!
- The value of the pointer being passed is similar to a memory address.
- The value uniquely identifies some object on the heap.



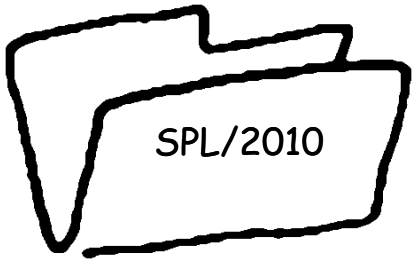
passing a reference by value

```
Object x = null;
giveMeAString (x);
System.out.println (x);


void giveMeAString (Object y)
{
 y = "This is a string";
 }
```

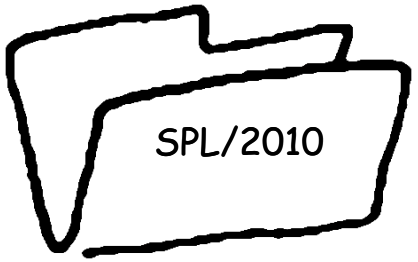


passing a reference by value

```
int x = 0;
giveMeATen (x);
System.out.println (x);

void giveMeATen (int y)
{
 y = 10;
 }
```



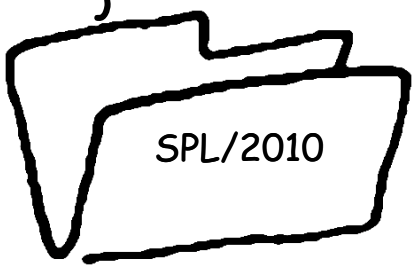
- The primitive value of a parameter is set to the value of another parameter
- the value "0" was passed into the method giveMeTen, not the variable itself.
- same is true of reference variables - value of reference is passed in, not the variable itself



```
Dog myDog = new Dog("Rover");
foo(myDog);
```

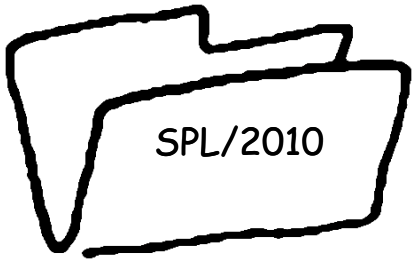
Suppose the Dog object resides at memory address 42. This means we pass 42 to foo().

```
public void foo(Dog someDog)
{
    someDog.setName("Max"); // AAA
    someDog = new Dog("Fifi"); // BBB
    someDog.setName("Rowlf"); // CCC
}
```



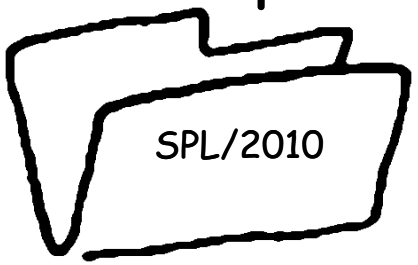
In C++

- both primitives and objects may be allocated on stack or heap.
- anything allocated on the heap can be reached by using a **reference** to its location
- reference = pointer - holds a memory address

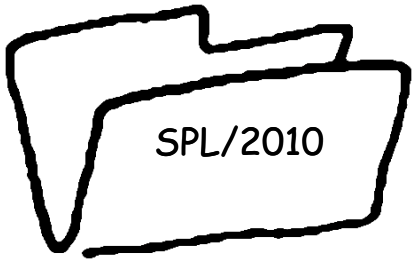


Pointers

- pointers are primitive types themselves
- hold memory address of a primitive or an object which resides either on the heap or on the stack
- pointer to type type_a is of type type_a *
  - type_a * is a primitive type
  - we can have a pointer to any type
  - pointer to a pointer to type_a: type_a **.

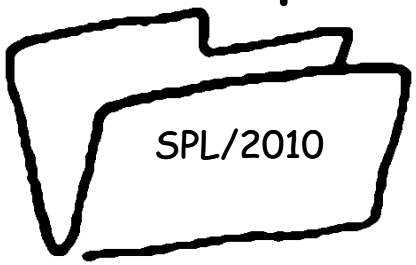


```cpp
#include <iostream>

int main()
{
    int *i_ptr = new int(10);

    std::cout << *i_ptr << std::endl;
    return 0;
}
```



Code analysis

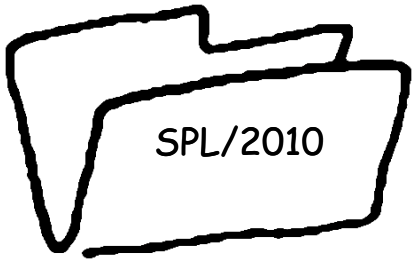
1. space for primitive of type ***int**** allocated on activation frame of main function
2. space allocated is associated with variable i_ptr
3. space for primitive of type ***int*** is allocated on heap (using new), initialized to 10
4. address of newly allocated integer is saved in i_ptr.
5. operator << is passed **content** (by value) i_ptr points to.



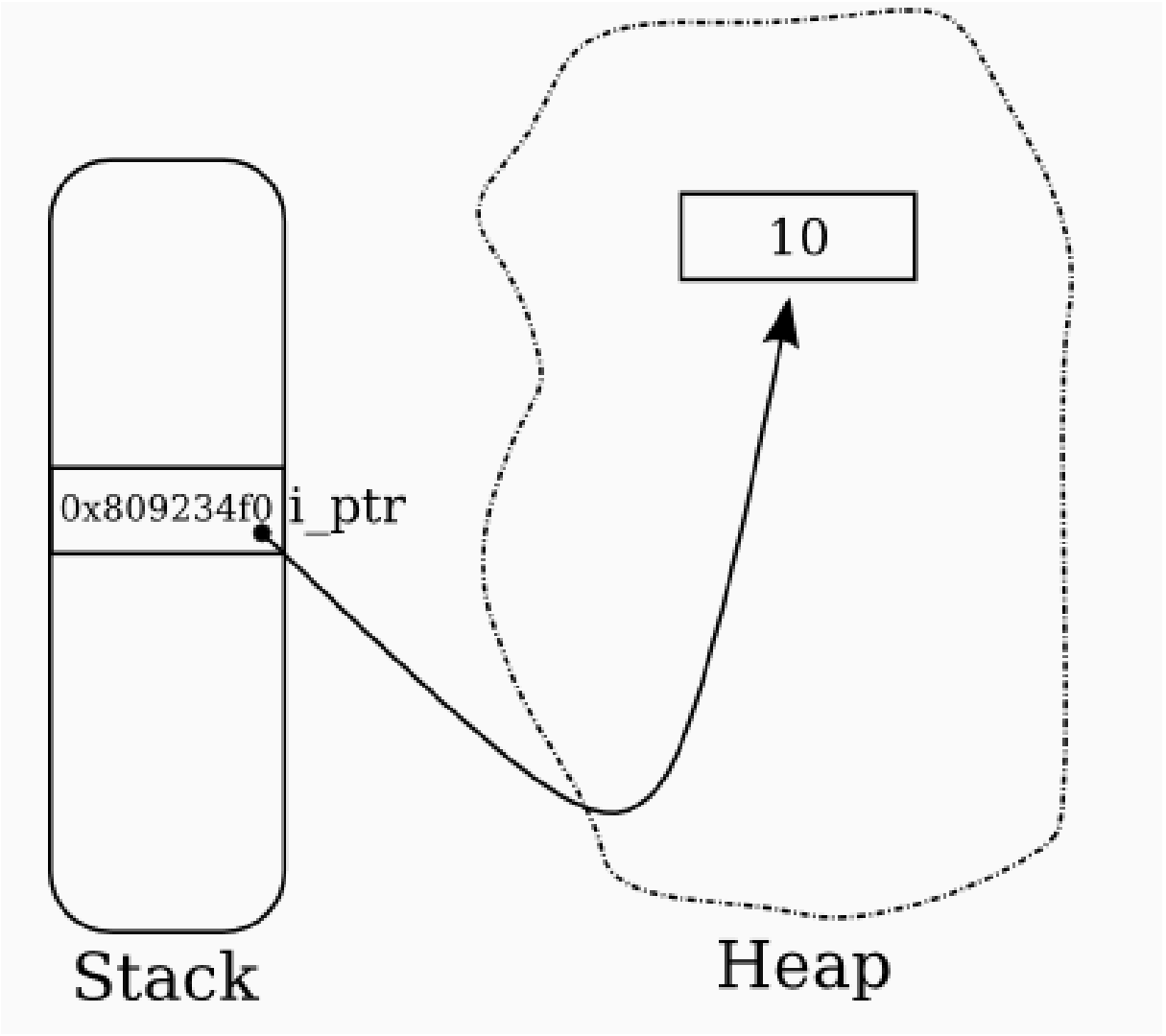
Operator *

- operator *: whose value is the content of the memory to which the pointer points

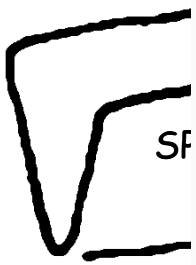
```
std::cout << *i_ptr << std::endl;
```



Process memory

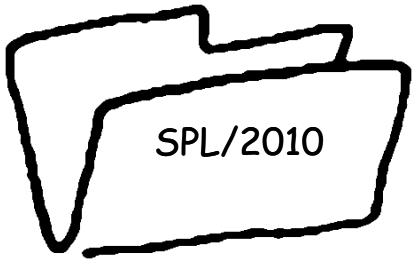


```
1.  class Cow {
2.  private:
3.          int _id;
4.  public:
5.          Cow(int id) {
6.                  this->_id = id;
7.          }
8.
9.          int getId() const {
10.                 return this->_id;
11.         }
12.
13.         void setId(int newId) {
14.                 this->_id = newId;
15.         }
16.
17.         void moooo() const {
18.                 std::cout << "moooo: " << this->_id << std::endl;
19.         }
20. };
```



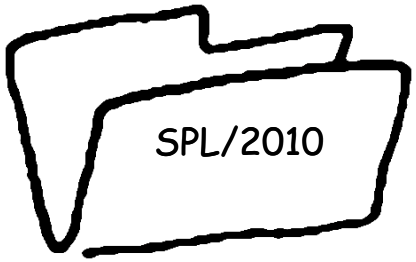
Access through pointers

- member methods of class T are executed always as T *- instance location in memory
- method of an object does not change the object's internal state, method is const
- operator -> access members and methods via pointer to object
- access members / methods of an object not through pointer using (.) dot-operator



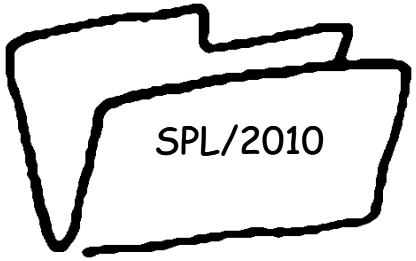
instantiate some cows

```
1.  int main()
2.  {
3.          Cow bety(482528404);
4.          Cow *ula = new Cow(834579343);
5.
6.          bety.moooo();
7.          ula->moooo();
8.          return 0;
9.  }
```



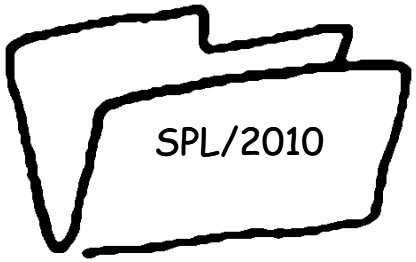
Analysis - bety

- space for Cow is allocated on activation frame of main function.
- constructor of Cow is called with 482528404
- ***this*** points to the address of the space allocated on the stack.
- space allocated is associated with variable bety

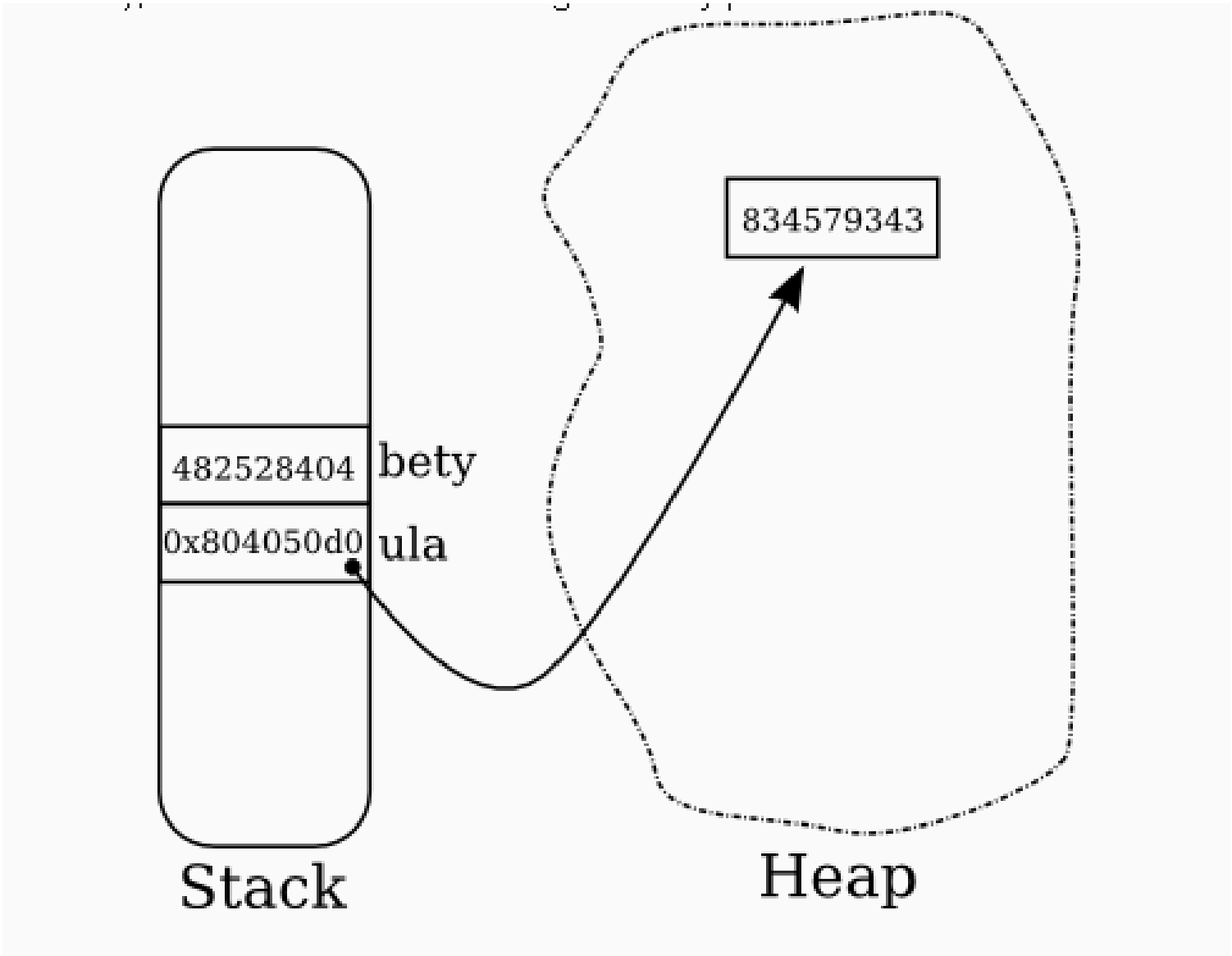


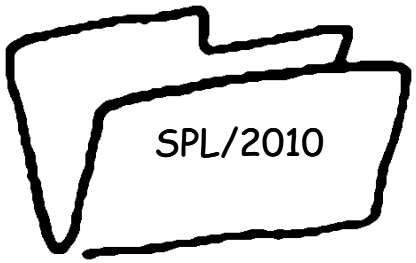
Analysis - ula

- space for a pointer Cow * is allocated on the activation frame of the main function.
- space allocated is associated with the variable ula.
- space for a Cow is allocated on the heap (using new operator)
- constructor is called with 834579343
- ***this*** points to the address of the space allocated on the heap.
- address of allocated Cow is saved in ula.



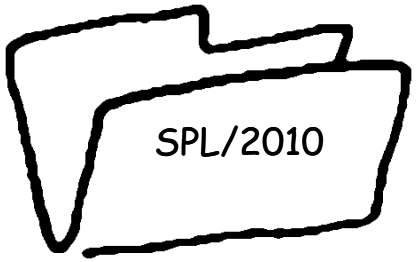
Process memory



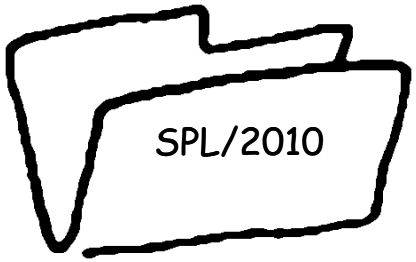


Dereferencing /"Address Of"

- dereference a pointer (* operator):
  - (*ula).moooo();
- take the address of something (& operator):
  - int i = 10;
  - int *i_ptr = &i;
  - i_ptr holds the address in which i is stored on the stack



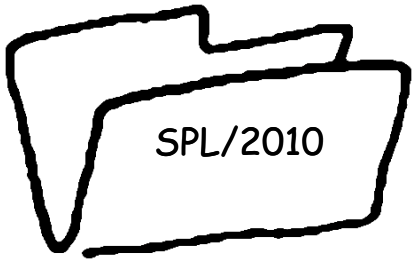
pass pointer arguments to functions



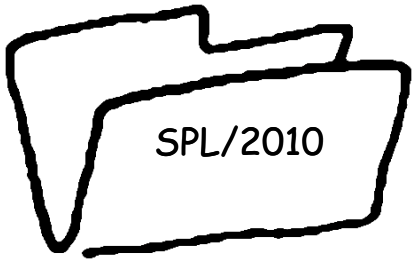

```
1.  void inc(int *i_ptr)
2.  {
3.      (*i_ptr)++;
4.  }
5.
6.  ...
7.  ...
8.  ...
9.
10. int i = 0;
11. inc(&i);
12. std::cout << i << endl;
```

Reference

- is basically a const pointer without using any pointer notations.
- may only be assigned once, when it is declared (initialization of reference)
- may not be altered to reference something else later.

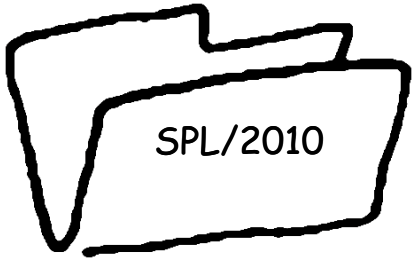


```
1.  int i=0;
2.  int &i_ref = i;
3.
4.  i_ref++;
5.  std::cout<<i<<std::endl;
```



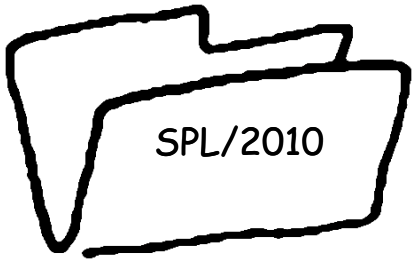
const pointers

- Any type in C++ can be marked as const, which means that its value cannot be changed
- const pointer is declared by adding the const keyword after the type
  - int *const i_ptr: a const pointer to an int
- cannot have references to references (this is explicitly illegal)



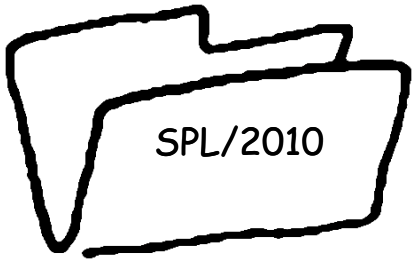
Parameter Passing

- all parameters are either 'in' or 'out'
  - 'in' parameters - information passed to the function, which the function does not change.
  - operation on 'in' parameter - not visible outside
  - 'out' parameters are a side-channel for function to return information, in addition to return value.
  - changes made to 'out' parameters are visible outside the scope of the function.



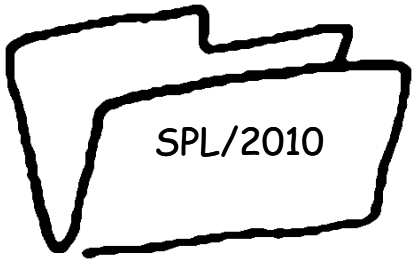
Parameter Passing - Java

- 2 forms of passing parameters to methods,
  - primitives are passed by value
  - Objects by reference (possible 'out' parameters).



Parameter Passing – C++

- 3 forms for parameter passing
  - By value, for 'in' parameters.
  - By pointer, for 'out' parameters.
  - By reference, for 'out' parameters.

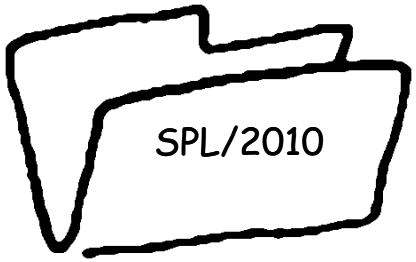


By Value

- outputs = 20

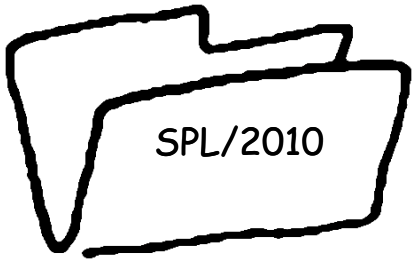
```
1.  void byVal(int i, Cow mooo){
2.          mooo.setId(i);
3.  }
```

```
1.  Cow hemda(20);
2.  byVal(30, hemda);
3.  std::cout << hemda.getId() << std::endl;
```



By Value

- call byVal - both 30 and the entire content of hemda are copied
- placed on the activation frame of byVal
- byVal performs all of its operations on these local copies - no changes to hemda

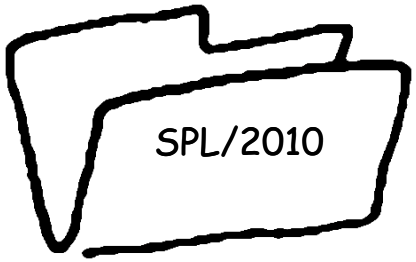


By Pointer

- output =30

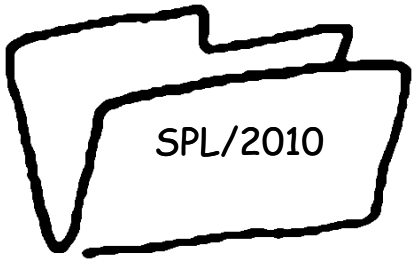
```
1.  void byPointer(int i, Cow *mooo){
2.          mooo->setId(i);
3.  }
```

```
1.  Cow hemda(20);
2.  byPointer(30, &hemda);
3.  std::cout << hemda.getId() << std::endl;
```



By Pointer

- byPointer received a pointer to location of hemda on activation frame of calling function
- changed its id

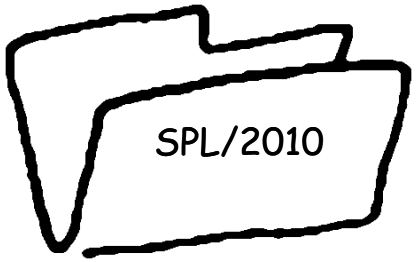


By Reference

- output =30

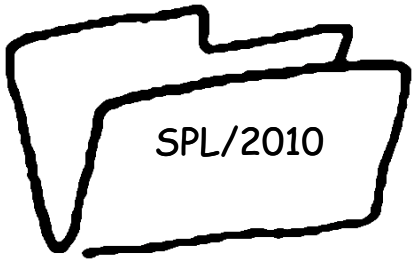
```
1.  void byReference(int i, Cow &mooo){
2.          mooo.setId(i);
3.  }
```

```
1.  Cow hemda(20);
2.  byReference(30, hemda);
3.  std::cout << hemda.getId() << std::endl;
```



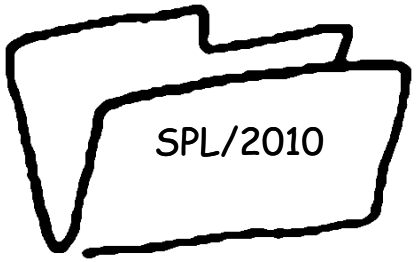
By Reference

- refrain from using pointers
  - inherently unsafe - easily cast to other types
- compiler is allowed to optimize the reference beyond the "const pointer" abstraction



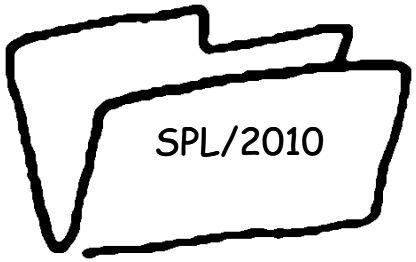
When to Use Each Form of Parameter Passing?

- passing parameters by value comes with a cost - copying and constructing a new object
- change a parameter outside the scope of the local function: by-reference or by-pointer



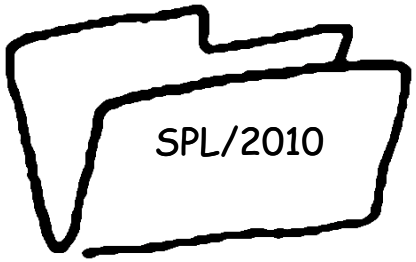
Recommendations

- For a function that uses passed data without modifying it (In parameter):
  - If data object is small (built-in data type or a small structure) - pass it by value.
  - If data object is an array, use a pointer because that's your only choice. Make the pointer a pointer to const.
  - If the data object is a good-sized structure, use a const reference.
  - If the data object is a class object, use a const reference



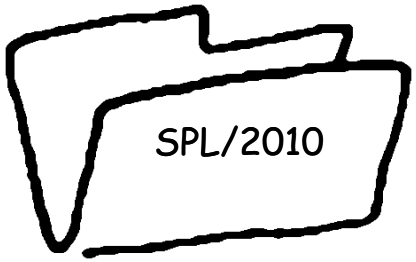
Recommendations

- For a function that modifies data in the calling function (Out parameter):
  - If the data object is a built-in data type, use a pointer or a reference, prefer the later.
  - If the data object is an array, use your only choice, a pointer.
  - If the data object is a structure, or a class object, use a reference



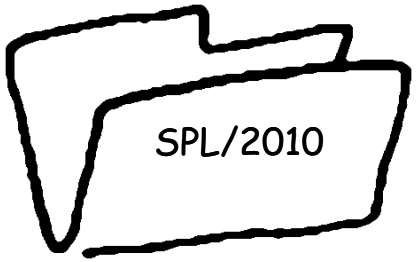
Recommendations

- When receiving a pointer check pointer for nullity. (A reference cannot be null.)



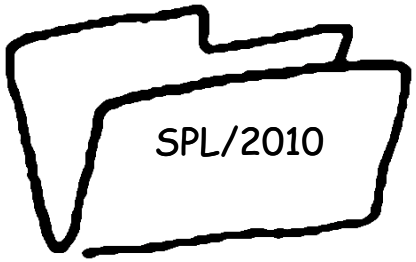
Returning Values From Functions

- values can be returned:
  - either by value (copy)
  - by reference
  - by pointer
- when returning something by reference or pointer care should be taken
  - Is it inside to be demolished activation frame?

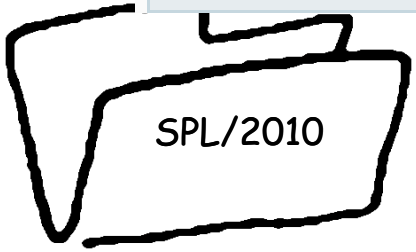


```
1.  Cow& f(int x) {
2.          Cow c(x);
3.          return c;   // THIS IS A TRAGIC MISTAKE
4.                      // c would be undefined as soon as the function returns.
5.      }
```

- Returning a reference / pointer to an invalid address on the stack is one of the main pitfalls of C++ beginners.

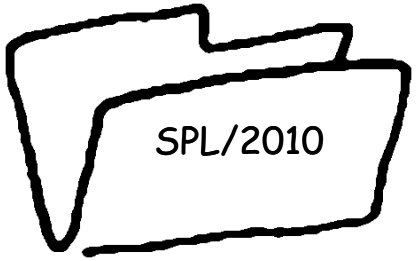


```
1.  #include <iostream>
2.  int *f()
3.  {
4.      int i = 1;
5.      cout << &i << endl;
6.      return &i;
7.  }
8.  void g()
9.  {
10.     int k = 2;
11.      cout << &k << endl;
12. }
13. void main()
14. {
15.     int *i = f();
16.     cout << *i << endl;
17.     g();
18.     cout << *i << endl;
19. }
```

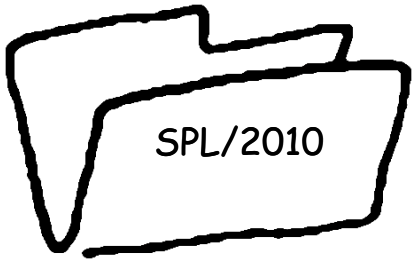


different level of compiler optimizations

- g++ 1.cpp; ./a.out
- 0xbffff564 134513864 0xbffff564 134513864
- g++ 1.cpp -O1; ./a.out
- 0xbffff564 1 0xbffff564 2
- g++ 1.cpp -O2 ; ./a.out
- 0xbffff560 1 0xbffff564 134519000
- g++ 1.cpp -O3 ; ./a.out
- 0xbffff574 1 0xbffff570 1

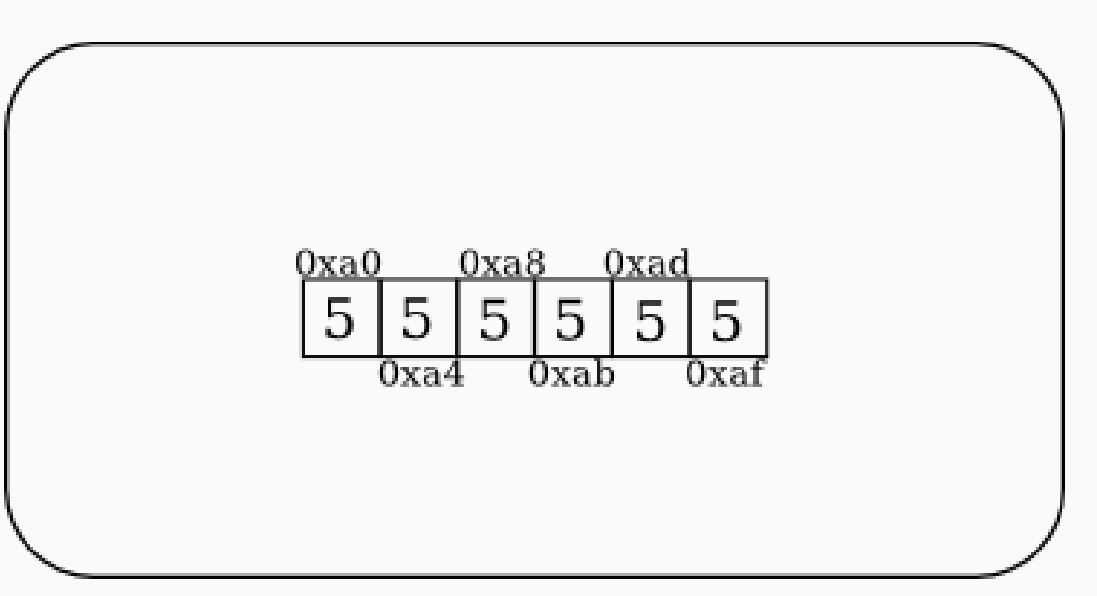


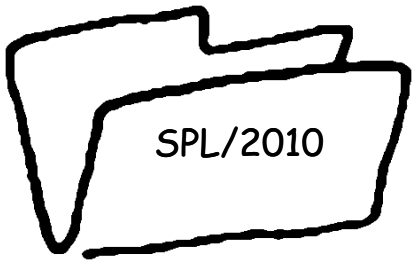
- This is totally bad as we can not predict how our program will work! No flag is lifted for us,a.k.a no exception, no segmentation fault. It works every time differently.



C++ Arrays

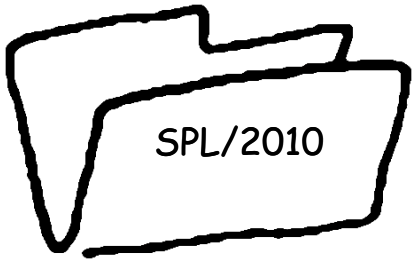
- blocks of continuous memory
- store data of the same type.
- memory image of an array of integers which holds the number 5 in each cell
- cell is of size 4 bytes





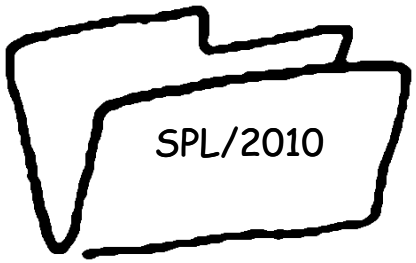
C++ Arrays

- Accessing individual cells - dereferencing a pointer to the specific cell.
- assume int *arr_ptr:
  - access fourth cell: arr_ptr[3] = *(arr_ptr+3)
  - pointer arithmetic:
    arr_ptr+3 = arr_ptr+3 x sizeof(int)
  - add/subtract numbers from pointer - implicitly multiplied by size of data the pointer points



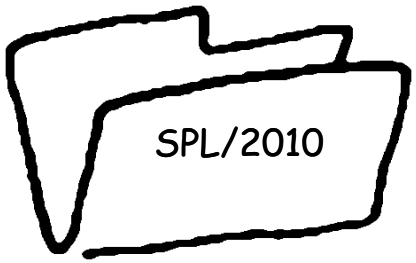
Arrays on the Heap

- everything in C++ may be allocated on Stack or Heap.
- allocate array on heap: new [] operator
- deallocating an array: delete [] operator




```
1.  int *arr = new int[100];
2.
3.  std::cout << arr[2] << std::endl;
4.  ...
5.  ...
6.  delete [] arr;
```

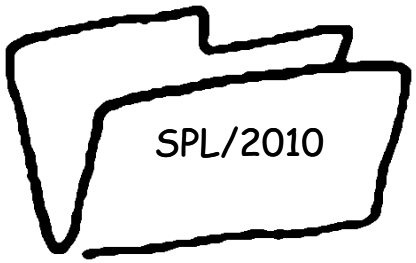
- output = 0
- new [] operator initialize array's elements by calling their default constructor (int - 0).




```
1.  int *arr = new int[100];
2.
3.  std::cout << arr[2] << std::endl;
4.  ...
5.  ...
6.  delete [] arr;
```

array of pointers

```
1.  Cow **cow_arr = new Cow*[100];
2.
3.  for (int i=0; i<100; i++)
4.        cow_arr[i] = new Cow(i);
5.
6.  ...
7.  ...
8.
9.  delete [] cow_arr;
```



array of pointers

- allocate a new Cow object on the heap, and store a pointer to it in a cell of Cow*
- delete [] calls destructor of elements in array
  - each element in the array is a pointer - destructor of a pointer is a nop
  - individual Cows will not be deleted
- delete [] deallocates memory allocated by new []
- delete each Cow we allocated manually – before deleting the array!

Arrays on the Stack

- array's size must be known in advance:
  - Cow cow_arr[5];
- initialize individual Cows:
  - Cow cow_arr[5] = {Cow(1), Cow(21), Cow(454), Cow(8), Cow(88)};
- accessing cells of the array on the Stack same as through a pointer
- cow_arr is basically a pointer to the beginning of the array of Cows on the Stack.