Intro. C-strings Sec. 4.5

Pointers to structures:

Copying structures with pointers:

Static vs. Dynamic Bases: Sec. 4.3-4.4

Sec. 4.2: Function parameters, C/C++ arrays and pointers.

Some CH. 3 bag details

CSI 310: Lecture 6 (No lab assignment: Use lab time for TA help with Proj. 1.)
In the scope of class `bag`, value-type, size-type, capacity, data and
used have local meanings.

```cpp
#include <cstdlib>  // Provides size_t

class bag {
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type CAPACITY = 30;

    NOTE: No other public data members!

    value_type data[CAPACITY];  // The array to store items
    size_type used;  // How much of array is used

    private:
        static const size_type CAPACITY = 30;
        typedef const size_type size_type;
        typedef int value_type;

    public:
        class bag {
            #include <cstdlib>

            public:
                // CLASS PROVIDED: bag
                FILE: bag.h
        }

    public:
        class bag
        ...

        #include <cstdlib>

        class bag
        ...

        // CLASS PROVIDED: bag
        FILE: bag.h
        ...
```

University at Albany Computer Science Dept.
class bag {
    ... 
    static const size_type CAPACITY = 30;
    ... 
    private:
        value_type data[CAPACITY]; // The array to store items
        size_type used;            // How much of array is used
    ... 
};

NOTE: No other public data members!

static const size_type CAPACITY = 30;

just ONE, named bag::CAPACITY;

Every const variable must have an initializer.

Note: No other public data members!

static const size_type CAPACITY = 30;

Cannot be changed (directly).

Just ONE, named bag::CAPACITY;

class bag
Within the scope of class bag,
say, within a member function,
Within the scope of class bag,

```cpp

class bag {
  public:
    // some (public) member functions
    static const size_type CAPACITY = 30;
    // NOTE: No other public data members!

  private:
    size_type used; // How much of array is used
    value_type data[CAPACITY]; // The array to store items
}
```

Outside the scope of class bag,

```

Within the scope of class bag,

```

You must code the scope

```

Within the scope of class bag,

```

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,

```

CAPACITY

ref: type bag::CAPACITY

Outside the scope of class bag,

```

Within the scope of class bag,
Section 4.2

Various Kinds of Value and Reference Parameters — See
Reference parameters are great if
(1) sizeof parameter is big
(2) caller function must modify
and/or
(3) const/reference parameters ditto,
which of the XXX's is made 98??

Changes RP = 98, Changes CV
the caller's argument variable!

void fun(CV, CV, CV);
fun(98, 98, 98); //inside called fun's body... //called fun's body

int RP = 98;

Reference parameter refers to
const ref. parm.

Which of the XXX's is made 98??

When function is called,

void caller()

Reference parameter refers to
value parameter.

Each value parm.

Is a local variable!

When function is called,

int RP = 98;

void fun(int VP, RP, CRP);

//value parameter.

//const ref. parm.

//const int crp

//value parameter.

=int RP, //reference parm.

int CV, CV, CV;

fun(cv, cv, cv);

//parameter variable.

//reference parameter.

//value parameter.
The parameter class's copy constructor is run if there is any.

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

The corresponding local variable.

value parameter, and those variables are initialized by copying each argument into

The function has its own automatic (local) variable for each value

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

void make(int* int_ptr, int cut, int &param);

Value Parameters Examples:

reference parameter.

The function declaration determines the kind of each parameter: value or
```c
assert (m->int == 42);
assert (*m->ptr == 42);
Make42(m->ptr, &m->int);

// prints Garbage


cout << *m->ptr << m->int;

int m->int;
int *m->ptr;
}
}
}
}
}
}
}
}
}
}
}

void Make42(int *m1, int *m2);
main() { int* m_ptr; int m_int;
m_ptr = new int;
garbage
m_int = 42;
Make42(m_ptr, &m_int);
}

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

same
variable
different
variable
past
times
future
void makeAll42(double data[], size_t n) {
    for (size_t i = 0; i < n; i++)
        data[i] = 42.0;
}

void makeAll42(double *data, size_t n) {
    data = &data[0];
}

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

A is an automatic (local) variable belonging to main

variables belonging to main and data

Surprising twist: unlike to C++, an array parameter is treated as a pointer to the first array element.

Like data in variables belonging to main, A is an automatic (local) variable.
caller's pointer variable.

into the caller's size_t variable 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
        cin >> n;
    cout << "How many elements would you like to allocate?";
    double* p = allocate<double>(size_t n);
    void allocate-double* (double* p, size_t n);

A reference parameter can have any type including POINTER.

Avoid COPYING the argument.

const & parameters, or when argc isn't a variable.

Allow the called function to MODIFY the argument variable (except

void allocate-double* (double* p, size_t n);
Reference parameters: like in
try: `cx.x` cannot declare pointers to references

++- wait `cx.x` `cx`

"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>Reference Parameter</th>
<th>Reference Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun(int &amp; x)</code>;</td>
<td><code>fun(int &amp; x)</code>;</td>
<td><code>fun(int &amp; x)</code>;</td>
</tr>
<tr>
<td>a pointer.</td>
<td>not a pointer.</td>
<td>a pointer.</td>
</tr>
<tr>
<td>that is a pointer.</td>
<td>that is not a pointer.</td>
<td>not a pointer.</td>
</tr>
<tr>
<td>Value Parameter that is</td>
<td>Value Parameter that is</td>
<td>Value Parameter</td>
</tr>
<tr>
<td><code>fun(int * &amp; x)</code>;</td>
<td><code>fun(int * &amp; x)</code>;</td>
<td><code>fun(int * &amp; x)</code>;</td>
</tr>
<tr>
<td>Pointer type</td>
<td>Pointer type</td>
<td>Pointer type</td>
</tr>
<tr>
<td>Non-Pointer type</td>
<td>Non-Pointer type</td>
<td>Non-Pointer type</td>
</tr>
</tbody>
</table>
Bigger array.

When it fills up, the insertion algorithm runs new to allocate a
allocated partially dynamically allocated partially
unlimited. The implement data structure is a structure is a fixed size partially fixed array.
fixed size partially fixed array.

chapter 4 improves this bag class, so the number of items is practically
how many can be stored up to

Roadmap:
size() > bag::CAPACITY
 precondition for insert():

bt1.size() + bt2.size() >= bag::CAPACITY :
 precondition for operator+(bag bt1, bag bt2):

(We don’t care about the rest of data []).

1. The number of items in the bag is the value of used.
2. For an empty bag, used = 0.

Invariant for the Primitive Bag Class


}

// How much of the array is used.
size_type used;

// The array to store items
value_type data[CAPACITY];

private:
    ... public:
}

class bag

FILE: bag1.h //
3. The total size of this dynamic array is the value of capacity.

2. The actual items in the bag are stored in a partially filled array. It is a

1. The number of items in the bag is the value of used.

Invariant for the Revisited Bag Class

```cpp
;

size_type capacity; // Current capacity of the bag.
size_type used; // How much of the array is used.
value_type* data; // Pointer to dynamic array.

private:
    ...

public:
    class Bag
    
    // FILE: bag2.h
```
The improved bag uses a dynamically allocated array:

```
MyBag1: insert (9);
MyBag2: insert (4);
```

```
MyBag1: used
MyBag2: used
```

```
MyBag1: MyBag2
```

```
MyBag2: MyBag1
```

```
size - type capacity;
size - type used;
value - type used;
value - type data;
```

```
private:
static const size - type CAPACITY = 30;
```

```
public:
```

```
class bag2
```

lots of Garbage

"data used capacity used"

"-9"

"4"

"1"

"-9"

"1"

"30"

"MyBag2"

"MyBag1"

"MyBag1"

"MyBag2"

"MyBag2"

"MyBag1"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag2"

"MyBag1"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

"MyBag1"

"MyBag2"

\*\*\*\*
The (1) part is retained. (2)

new value-type [bigger] to allocate a bigger array to replace (2).

When a bag is full, the insert and operator+ methods run

(2) a dynamically-allocated dynamically-siz ed array to actually hold the items.

(1) a statically-sized structure, and

Each bag2 consists of

Each bag1 consists of a statically-sized data structure only. Its capacity is

limited to 30 items. Memory is wasted when the bag has fewer items.
Remember: The assignment operator "="

```
MyBag1 = MyBag2;
```

```
MyBag2.insert(4);
MyBag2.insert(9);
```

```c
#include "bag2.h"
```
Something bad happens!

OthBag1.erase(-9);
OthBag2.erase(-9);
OthBag1.insert(7);
OthBag2.insert(7);

Ok, each bag was copied. Suppose these operations next:

It's OK.
more important for CS1310.

C-strings, linked structures, efficiency and recursion are
custom types to behave just like int, double, char, etc.

Study 4.3-4.4 for details. Advanced C++ features help you create

pointers.

}\;
...

base(const base &sc);

void operator=(const base &sc);

public:

class base

assignment operator and copy constructor

How do you declare and implement your own

base2 This(base2 myBase2);

otherBase2 = myBase2;

code you write runs for copy operations. Like:

(Unlike C, C++ gives you programmers the capability to program a class so
university at Albany Computer Science Dept.
Reminder: A Dynamic Variable is useless without a (separate) pointer variable to hold the address returned by new!

<table>
<thead>
<tr>
<th>Dynamic Array</th>
<th>Automatic Array</th>
<th>Structure/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Struct.</td>
<td>Automatic Struct.</td>
<td></td>
</tr>
<tr>
<td>{ new [9] }; }</td>
<td>{ new [9]; }</td>
<td></td>
</tr>
<tr>
<td>(\text{fun(sz)})</td>
<td>(\text{fun(sz)})</td>
<td></td>
</tr>
<tr>
<td>#include &quot;bage2.h&quot;</td>
<td>#include &quot;bage2.h&quot;</td>
<td></td>
</tr>
<tr>
<td>Dynamic Integer</td>
<td>Automatic Integer</td>
<td></td>
</tr>
<tr>
<td>{ new [9] }; }</td>
<td>{ new [9]; }</td>
<td></td>
</tr>
<tr>
<td>(\text{fun(sz)})</td>
<td>(\text{fun(sz)})</td>
<td></td>
</tr>
<tr>
<td>Dynamic Variable</td>
<td>Automatic Variable</td>
<td></td>
</tr>
<tr>
<td>(\text{fun}())</td>
<td>(\text{fun}())</td>
<td></td>
</tr>
</tbody>
</table>

More Orthogonality

University at Albany Computer Science Dept.
structures. That is the subject of Chapter 5 and beyond.

Unlike the auto. variables, dynamic vars. have no names. Unlike the auto. variables, AUTOBAGs data is a pointer to an array.

Unlike the auto. variables, dynamic vars. have no names.

It gets complicated! University at Albany Computer Science Dept.

```c
#include "bag2.h" //Ch.4 improved bag

main()
{
    bag AutoBag(4); //bag2 variable
    bag2 *PBag;      //Pointer variable!
    PBag = new bag2(4);
}
```

Automatic variables belonging to this call to main() 
"heap" or "free store" of dynamic variables

```
bag2 bag = new bag2(4); //constructor
//Pointer variable!
bag2 *PBag = bag; //bag2 varriable
```

`} /* bag2 AutoBag (4) ; */`
2. Illegal Pointers.

RECYCLED memory!

1. DANCING Pointers: Pointers to Garbage or

TWO (2) BIG Pitfalls of Pointers:

will create a DANCING POINTER (value in Pbase).

delete Pbase;

Now,
This is a "CRASH": Computer tried to read memory at the illegal address 0x0.

Segmentation Fault

```cpp
    cout >> *MyP
    0x0 0x0e00e04
    cout >> "" >> 4MyInt >> "" >> 4MyInt >> " endl;
    cout >> 4MyInt >> " endl;
    cout >> 4MyInt >> " endl;
    cout >> 4MyInt >> " endl;
```

After Assignment:

```
  4006  0x0 NULL
  MyInt
  MyP
  MyP = NULL;
```

Before Assignment:

```
  MyInt
  MyP
```

University at Albany Computer Science Dept.
Or is it the value of a pointer variable; that is an address?

So, when you or others say "pointer", think hard: Is it a pointer variable?

But most everyone, we and DSO, say for short, "PIVAR is a pointer" (illegal value),

C/C++ int variable, or else it might have the NULL value, or else some pointer type variable. The variable named PIVAR might store an address of a int *PIVAR; What is PIVAR?? Is it a "pointer"??

whose type is int. This variable stores a C/C++ integer.

Most say "It's an integer!" but, really, IVAR is (the name of) an variable

int IVAR; What is IVAR???

A linguistic pitfall—try not to fall into it!
Perhaps we should always use the word "address" for "pointer value".

done.

type which determines what values it can hold and what operations can be performed. Each variable has a

Technically, "pointer" and "int" describe C/C++ types.
(B) A single pointer locates the array.

Fastest way to control sequential processing of the array from beginning to end. Advantages of way 2 if you can live without the terminating value: (A) It's the

EXCEEDDED

FOR BOTH WAYS, THE ALLOCATED SPACE MUST NOT BE

value. Example: C-string pages 183-186

2. End of used prefix is marked by the next entity containing a terminating

1. Data structure = array + "used" count.

?? Where does the used data prefix end? Two ways to tell:

array is "idle".

The used data is stored in a contiguous prefix of the array. (The rest of the

Partially filled array: dynamically allocated or not.

University at Albany Computer Science Dept.
Advantages of Way I:

A) You can tell the number of items right away

B) Appending a new item to the end is last (also constant time).

C) Item values are not restricted. No special terminal-time value is required.
char type

"Null-Terminated Strings" or "C-Strings"
Unix and other system interface libraries use C-strings.

#include <cstring> // Library has very useful functions.
#include <iostream> // Facilities "know about" C-strings.