Deep Copy

Slides

1. Table of Contents
2. Assignment of Structs
3. Dynamic Content
4. Shallow Copying
5. Assignment Operator
6. Deep Assignment Copy
7. Assignment Problems
8. Assignment Problems 2
9. this Pointer
10. … Improved Deep Copy
11. Passing an Object
12. Passing Objects by Value
13. Passing Value Objects Results
14. Copy Constructors
15. Initialization
16. Moral

©1995-2001 Barnette ND, McQuain WD

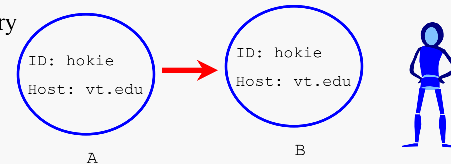
Assignment of Structs

A default member field assignment operation is provided for struct variables:

```
struct EmailAcct {
    string ID, Host;
};

void main() {
    EmailAcct A;
    A.ID   = "hokie" ;
    A.Host = "vt.edu";
    EmailAcct B;

    B = A;   // copies the field members of A into B
}
```

The default assignment operation simply copies values of the field members from the “source” struct into the corresponding field members of the “target” struct .

This is satisfactory in many cases:



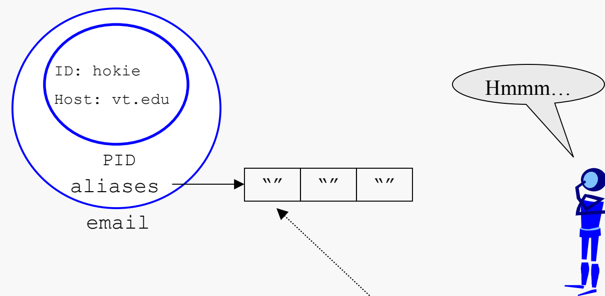
However, if an struct contains a pointer to dynamically allocated memory, the result of the default assignment operation is usually not desirable…

©1995-2001 Barnette ND, McQuain WD

Dynamic Content

Consider the `EmailAddr` struct below:

```
const int MAXALIAS = 3;

struct EmailAddr {
     EmailAcct PID;
     string* aliases;
};

     EmailAddr email;
     email.PID.ID    = "hokie";
     email.PID.Host = "vt.edu";
     email.aliases  = new string[MAXALIAS];
```



The aliases array is not a field member of the struct email.

Shallow Copying

Now, suppose we declare another `EmailAddr` struct variable and assign the original one to it:

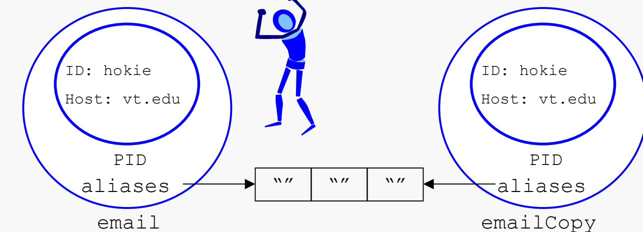
```
EmailAddr emailCopy;

emailCopy = email;
```

`emailCopy` does **not** get a new copy of the `aliases` array.

It just gets a copy of the `aliases` pointer from `email`.

So both `EmailAddr` structs share the same dynamic data.

Here's what results:



This is almost certainly **NOT** what was desired when the code above was written.

This is known as making a "**shallow copy**" of the source struct.

There is no good solution for this problem of assigning structs that contain dynamic memory.

The same problem occurs with **C++ class objects**, however the language provides an elegant solution.

Assignment Operator

When a class object contains a pointer to dynamically allocated data, we generally will want the assignment operation to create a complete duplicate of the "source" object. This is known as making a "**deep copy**".

In order to do this, you must provide your own implementation of the assignment operator for the class in question:

```
class EmailAcct {
private:
  string ID, Host;
public:
  EmailAcct( );
  EmailAcct(string ID2, string Host2);
  // .  .  .
  Print(ostream& out);
};

const int MAXALIAS = 3;

class EmailAddr {
private:
  EmailAcct PID;
  string* aliases;
public:
  EmailAddr( );
  EmailAddr(string ID2, string Host2);
 ~EmailAddr();
  EmailAddr& operator=(const EmailAddr& Eaddr);
  // .  .  .
 EmailAcct getPID();
 string*   getAliases();
};
```


©1995-2001 Barnette ND, McQuain WD

Deep Assignment Copy

In your own implementation of the overloaded assignment operator you must include code to handle the "deep" copy logic.

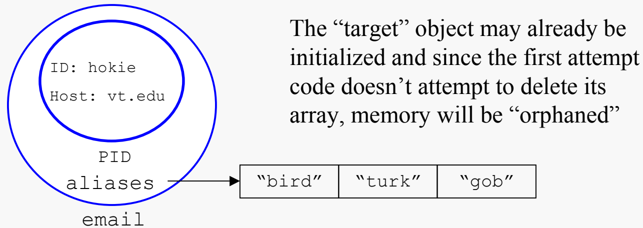
Here's a first attempt:

```
EmailAddr& EmailAddr ::operator=(const EmailAddr& Eaddr) {

  PID     = Eaddr.PID;    // assign non-dynamic members
  aliases = NULL;         // don't copy pointer
  aliases = new string[MAXALIAS];  // allocate array copy

  for (int i=0; i < MAXALIAS; i++) // copy array cells
      aliases[i] = Eaddr.aliases[i];

  //return ?
}
```

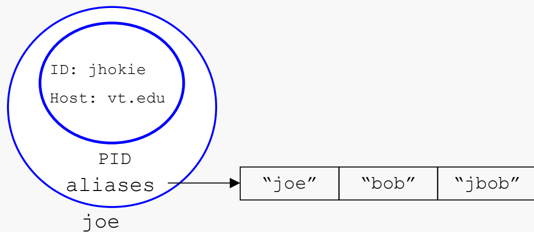
The above code contains some insidious logic problems.

©1995-2001 Barnette ND, McQuain WD

Assigning existing objects:



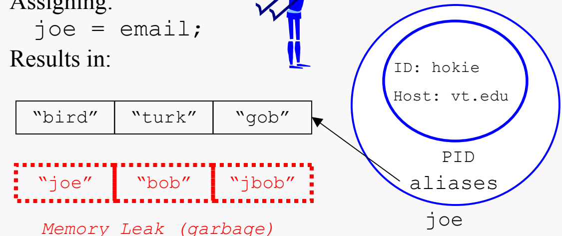
The "target" object may already be initialized and since the first attempt code doesn't attempt to delete its array, memory will be "orphaned"



Assigning:

```
joe = email;
```

Results in:



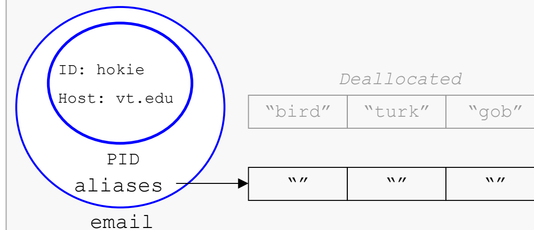
©1995-2001 Barnette ND, McQuain WD

Second attempt:

```
EmailAddr& EmailAddr::operator=(const EmailAddr& Eaddr) {

  PID     = Eaddr.PID;    // assign non-dynamic members
  delete [] aliases;      // delete existing array
  aliases = new string[MAXALIAS];  // allocate array copy

  for (int i=0; i < MAXALIAS; i++) // copy array cells
      aliases[i] = Eaddr.aliases[i];

  //return ?
}
```

Self Assignment:

```
email = email;
```



Results in a logic error. The implicit object and `Eaddr` are referencing the same object. Delete deallocates the same array that is then accessed in the `for` loop.

©1995-2001 Barnette ND, McQuain WD

this Pointer

Special Object Pointer: "**this**"
Every object contains a language supplied implicitly defined hidden pointer to itself termed "**this**" which contains the address of the object.

- Used when an object needs to refer to itself as whole, not just individual data members).
- The "**this**" pointer is not explicitly part of the object, (i. e. not counted in the sizeof() the object).
- Every member function receives the **this** pointer as an implicit parameter
- It is used implicitly to access an object's members whenever a member is directly referenced.
- It can however be used explicitly to indirectly access an object's members.
- The type of the "**this**" pointer is dependent upon the type of the object to which it refers.
  - For a non-const member function of class X, the type of the "**this**" pointer is:
    X * const this; // a const pointer
    // **this** is never explicitly defined or assigned
  - For a const member function of class X, the type of the "**this**" pointer is:
    const X * const this;
    //a const pointer to a const object

©1995-2001 Barnette ND, McQuain WD

…Improved Deep Copy

Here's a somewhat improved version:

```
EmailAddr& EmailAddr::operator=(const EmailAddr& Eaddr) {
 if (this != & Eaddr) {   // self-assignment?
  PID     = Eaddr.PID;    // assign non-dynamic members
  delete [] aliases;      // delete existing array
  aliases = new string[MAXALIAS];  // allocate array copy

  for (int i=0; i < MAXALIAS; i++) // copy array cells
      aliases[i] = Eaddr.aliases[i];
 }
  return( *this );
}
```

By returning a reference to an object, a member function allows chaining of the the operations. e.g.,

```
EmailAddr joe, bob;
bob = joe = email;
```

```
//Not the following
joe.=(email);
bob.=(joe);
```

Note: in the above example in all `EmailAddr` objects, allocated `aliases` array never changes size, once allocated during execution. Thus the `aliases` array would not need to be deleted and re-allocated. Its cells could be used to hold the copied strings.

However, if any two objects of the `EmailAddr` class contained different sized arrays the above approach would need to be implemented.

©1995-2001 Barnette ND, McQuain WD

Passing an Object

When an object is used as an actual parameter in a function call, the distinction between shallow and deep copying can cause seemingly mysterious problems.

```
void PrintAddrs(EmailAddr mail, ostream& Out) {

   Out << "Email address: " << endl;
   mail.getPID().Print(Out);
   Out << "Email Aliases: " << endl;
   string* eAliases = mail.getAliases();
   for (int i=0; i < MAXALIAS; i++)
          Out << eAliases[i] << endl;
}
```

Note that the `EmailAddr` parameter `mail` is **not** passed by constant reference, but by value. However, that will cause a new problem.

When an object is passed by value, the actual parameter must be copied to the formal parameter (which is a local variable in the called function).

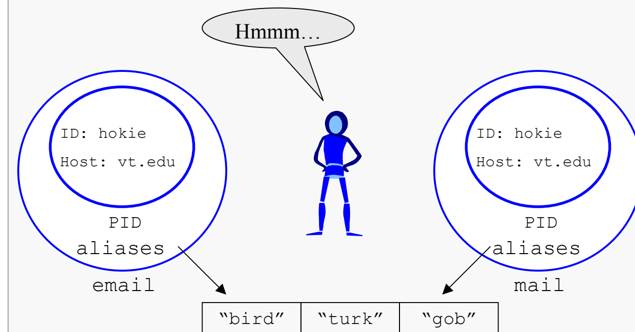
This copying is managed by using a special class constructor, called a *copy constructor*. By default this involves a member by member shallow copy. That means that if the actual parameter involves dynamically allocated data, then the formal parameter will share that data rather than have its own copy of it.

©1995-2001 Barnette ND, McQuain WD

Passing Objects by Value

In this case:

```
void PrintAddrs(EmailAddr mail, ostream& Out) {

   Out << "Email address: " << endl;
   mail.getPID().Print(Out);
   Out << "Email Aliases: " << endl;
   string* eAliases = mail.getAliases();
   for (int i=0; i < MAXALIAS; i++)
          Out << eAliases[i] << endl;
}
```

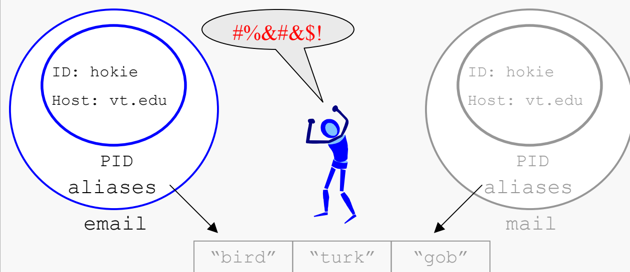


What happens when `PrintAddrs(email);` is called?
First, a local variable `mail` is created and the data members of `email` are copied into `mail`, resulting in the situation shown above.

©1995-2001 Barnette ND, McQuain WD

Passing Value Objects Results

When `PrintAddrs()` terminates, the lifetime of `mail` comes to an end and its destructor is automatically invoked:



Destructing `mail` causes the deallocation of the aliases array to which `mail.aliases` points.

But of course, that's the same array that `email` has created. So, when execution returns to `main()`, `email` will have lost its array, but `email.aliases` will still point to that deallocated memory, (dangling pointer).

Havoc will ensue.

Copy Constructors

There are several solutions to this problem:

- always pass objects by reference
- force a deep copy to be made when pass by value is used

The first option is undesirable since it raises the risk of undesired modification of the actual parameter.

The second option can be achieved by providing a user-defined copy constructor for the class, and implementing a deep copy.

When a user-defined copy constructor is available, it is used when an actual parameter is copied to a formal parameter.

```
EmailAddr::EmailAddr(const EmailAddr& Source) {
  PID     = Source.PID;   // assign non-dynamic members
  aliases = new string[MAXALIAS];  // allocate array copy

  for (int i=0; i < MAXALIAS; i++) // copy array cells
      aliases[i] = Source.aliases[i];
}
```

The copy constructor takes an object of the relevant type as a parameter (constant reference must be used). Implement a deep copy in the body of the copy constructor and the problem described on the previous slides is solved.

Initialization

When an object is declared, it may be initialized with the value of an existing object (of the same type):

```
void main3() {
   EmailAddr email;  // default construction
   // code to store data into email
   // . . .

   EmailAddr UserEmail = email;  // initialization
}
```

Technically initialization is different from assignment since here we know that the “target” object does not yet store any defined values.

Although it looks like an assignment, the initialization shown here is accomplished by the copy constructor.

If there is no user-defined copy constructor, the default (shallow) copy constructor manages the initialization.

If there is a user-defined copy constructor, it will manage the copying as the user wishes.

Copy constructors also execute when an object is returned by value from a function:

```
object x = getObject(obj);
```


©1995-2001 Barnette ND, McQuain WD

Moral

When implementing a class that involves dynamic allocation, if there is any chance that:

- objects of that type will be passed as parameters, or
- objects of that type will be used in initializations, or
- objects of that type will be returned by value

then your implementation should include a copy constructor that provides a proper deep copy.

If there is any chance that:

- objects of that type will be used in assignments

then your implementation should include an overloaded assignment operator that provides a proper deep copy.

This provides relatively cheap insurance against some very nasty behavior.

©1995-2001 Barnette ND, McQuain WD