CS481 - Object-Oriented Programming

References

- Like a constant pointer
  - Automatically dereferences
- Acts as alias for an object
  - Works for built-in types as well
- Often used with functions
  - Argument or return value

1

Pointer vs Reference

```
// pointer
int x;
int* xp = &x;

*xp = 10;
cout << x;
```

```
// reference
int x;
int& y = x;

y = 10;
cout << x;
```

Note: “constant pointer” and not “pointer to const”

2

Function Arguments

```
// pointer
void f(int* px)
{
  *px = 25;
}

int y;
f(&y);
```

```
// reference
void f(int& x)
{
  x = 25;
}

int y;
f(y);
```

3

Swap Example

(References)

- Used in exchange sorts:

```
void swap(int& x, int& y)
{   int temp;
    temp = x;
    x = y;
    y = temp;
}
```

4

Swap Example

(References)

- Used in exchange sorts:

```
void swap(int& x, int& y)
{   int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Caller :
swap(a, b);

5

Swap Example

(Pointers)

- Used in exchange sorts:

```
void swap(int *x, int *y)
{   int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

6

© Copyright 1996, 1998 by Mark J. Sebern & Jeffrey Blessing

© 1998, Jeffrey Blessing

CS481 - Object-Oriented Programming

Swap Example

(Pointers)

- Used in exchange sorts:

```
void swap(int *x, int *y)
{   int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Caller : swap(&a, &b);

7

const Reference

Efficiency of “pass by pointer”; safety of “pass by value”

```
void g(const Complex& c)
{
  cout << c.Real();
}
...
Complex c1; // in “main”
g(c1);
```

8

Reference Rules

- Must be initialized on creation
  - If reference variable, in declaration
  - If function argument, when invoked
- Cannot change to reference a different object
- Cannot have `NULL` reference

9

Reference Arguments

- If `const` reference, no problem
  - Acts as efficient “pass by value”
- If non-`const` reference:
  - Caller’s argument can be modified
  - But no explicit “`&var`” as in C
  - Be sure to document behavior!

10

Copy Constructor

- A special form of constructor
  - Overloaded constructor function
  - Signature: one arg, a class reference
- Automatically used
  - Call generated by compiler
  - When passing or returning by value

11

Passing by Value

```
float f(int x)
{
  float y = x + 0.5;
  return y;
}
```

Compiler must copy values on call and return

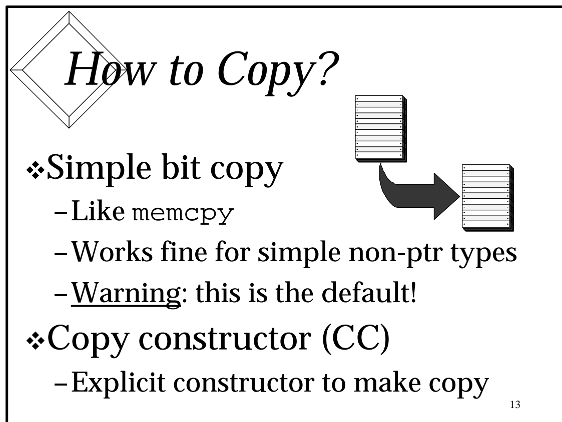
```
int aa = 2;
float bb;

bb = f(aa)-3;
```

12

© Copyright 1996, 1998 by Mark J. Sebern & Jeffrey Blessing CS481-Sum96-08-2

CS481 - Object-Oriented Programming



Bit Copy Problems

```
class MyClass
{
  char* str;...
};
```

Class with pointer to dynamic memory

```
MyClass::MyClass()
{
 str = (char*) malloc(100);
}
```

14

Bit Copy Problems

```
class MyClass
{
  char* str;...
};
```

Constructor allocates memory, keeps pointer

```
MyClass::MyClass()
{
 str = (char*) malloc(100);
}
```

15

Bit Copy Example

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
     (MyClass B)
{
  B.str[2]='X';
}
```

16

Bit Copy Example

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
     (MyClass B)
{
  B.str[2]='X';
}
```

A: str:

17

Bit Copy Example

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
     (MyClass B)
{
  B.str[2]='X';
}
```

A: str: 1000_{16}

1000_{16}

18

© Copyright 1996, 1998 by Mark J. Sebern & Jeffrey Blessing

© 1998, Jeffrey Blessing

CS481 - Object-Oriented Programming

Bit Copy Example

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
   (MyClass B)
{
  B.str[2]='X';
}
```

A: str: 1000_{16}

1000_{16} Hello\0

19

Bit Copy Example

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
   (MyClass B)
{
  B.str[2]='X';
}
```

A: str: 1000_{16}

1000_{16} Hello\0

B: str: 1000_{16}

20

Bit Copy Example

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
   (MyClass B)
{
  B.str[2]='X';
}
```

A: str: 1000_{16}

1000_{16} HeXlo\0

B: str: 1000_{16}

21

Implementing a CC

```
class MyClass {...
public:
  MyClass(const MyClass& mc);...
};
```

```
MyClass::MyClass
          (const MyClass& mc)
{
 str = (char*) malloc(100);
 memcpy(str, mc.str, 100);
}
```

22

Implementing a CC

const reference arg

```
class MyClass {...
public:
 MyClass(const MyClass& mc);...
};
```

```
MyClass::MyClass
          (const MyClass& mc)
{
 str = (char*) malloc(100);
 memcpy(str, mc.str, 100);
}
```

23

CC Copy Example

```
MyClass::MyClass
          (const MyClass& mc)
{
 str = (char*) malloc(100);
 memcpy(str, mc.str, 100);
}
```

A: str: 1000_{16}

1000_{16} Hello\0

24

© Copyright 1996, 1998 by Mark J. Sebern & Jeffrey Blessing

© 1998, Jeffrey Blessing

CS481 - Object-Oriented Programming

CC Copy Example

```
MyClass::MyClass
          (const MyClass& mc)
{
 str = (char*) malloc(100);
 memcpy(str, mc.str, 100);
}
```

A: str: 1000_{16} → 1000_{16} Hello\0

B: str:

25

CC Copy Example

```
MyClass::MyClass
          (const MyClass& mc)
{
 str = (char*) malloc(100);
 memcpy(str, mc.str, 100);
}
```

A: str: 1000_{16} → 1000_{16} Hello\0

B: str: 2000_{16} → 2000_{16}

26

CC Copy Example

```
MyClass::MyClass
          (const MyClass& mc)
{
 str = (char*) malloc(100);
 memcpy(str, mc.str, 100);
}
```

A: str: 1000_{16} → 1000_{16} Hello\0

B: str: 2000_{16} → 2000_{16} Hello\0

27

MyClass Destructor

- The problems associated with "Shallow Copy" are illustrated below:
  - First, lets apply good design rules and implement a destructor for MyClass, since it dynamically allocates memory:

```
MyClass::~MyClass()
{
    free(str);
}
```

28

When B is Destroyed!

```
MyClass A;
A.setStr
  ("Hello");
f(A);
...
```

```
void f
    (MyClass B)
{
 B.str[2]='X';
}
```

A: str: 1000_{16} → 1000_{16} HeXlo\0

B: str: 1000_{16} → 1000_{16} HeXlo\0

29

When B is Destroyed!

```
MyClass A;
A.SetStr
  ("Hello");
f(A);
...
```

```
void f
    (MyClass B)
{
 B.str[2]='X';
}
```

A: str: 1000_{16} → Dangling Pointer !!

30

© Copyright 1996, 1998 by Mark J. Sebern & Jeffrey Blessing CS481-Sum96-08-5

"Default" CC

- Compiler-generated
  - If you don't declare one
- Memberwise copy
  - For built-in types, bit copy
  - For user-defined types, calls members copy constructor (or "default")

31

CC Summary

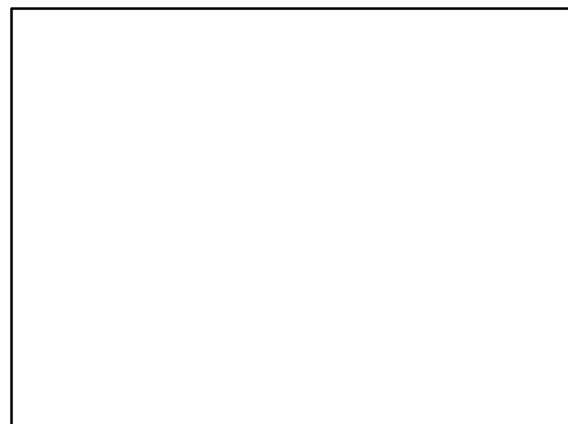
- Copy constructor needed to pass/return by value
- Memberwise default (bit copy?)
- Always create CC for class?
  - Whenever default not appropriate
    - E.g., initialization, dynamic memory

32

Pointers to Members

- In C, pointer to anything?
- But pointer to a class member?
  - Independent of which object?
    - Would be "offset" into object data
- C++: "`->*`" and "`.*`" syntax
  - Pretty sophisticated; enough for now

33



© Copyright 1996, 1998 by Mark J. Sebern & Jeffrey Blessing

© 1998, Jeffrey Blessing