# QUIZ: Is anything wrong with this class?

```
class Foo{
  int i;
public:
  static void printi(){cout <<this->i <<endl;}
};
```

# Solution

```
class Foo{
  int i;
public:
  static void printi(){cout <<this->i <<endl;}
};
```

```
main.cpp:8:31: error: 'this' is unavailable for static member functions
   static void printi(){cout <<this->i <<endl;}
                              ^
```

# QUIZ: Is anything wrong with this class?

```
class Foo{
  int i;
public:
  static void printi(){cout <<i <<endl;}
};
```

# Solution

```
class Foo{
  int i;
public:
  static void printi(){cout <<i <<endl;}
};
```

A: static member functions can only access static data members!

# QUIZ: Is anything wrong with this class?

```
class Foo{
  int i;
public:
  int geti(){return i;}
  static void printi(){cout <<geti() <<endl;}
};
```

# Solution

```
class Foo{
  int i;
public:
  int geti(){return i;}
  static void printi(){cout <<geti() <<endl;}
};
```

A: static member functions can only call other static member functions!

# QUIZ: Is anything wrong with this class?

```
class Foo{
  static int i = 42;
public:
  static void printi(){cout <<i <<endl;}
};
```

# Solution

```
class Foo{
  static int i = 42;
public:
  static void printi(){cout <<i <<endl;}
};
```

```
main.cpp:6:18: error: ISO C++ forbids in-class initialization of non-const static member 'Foo::i'
   static int i = 42;
                  ^
```

A: static data members must be initialized outside of the class declaration!

# QUIZ: Is anything wrong with this class?

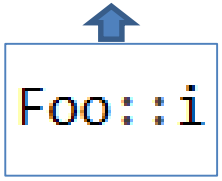
```
class Foo{
  static int i;
public:
  static void printi(){cout <<i <<endl;}
};

int i = 42;
```

# Solution

```
class Foo{
  static int i;
public:
  static void printi(){cout <<i <<endl;}
};

int i = 42;
```




The code compiles, but we're declaring and defining a global integer i, not the member i!
(Static objects are initialized by default with zero.)

# Thinking in C++, 2nd ed. Volume 1



## Ch. 11: References & the Copy-Constructor

# Pointers in C and C++

- Remember from chs. 3 and 8:
  - The “void pointer assignment” trick in C
- C++ is more strongly typed

examples

## C

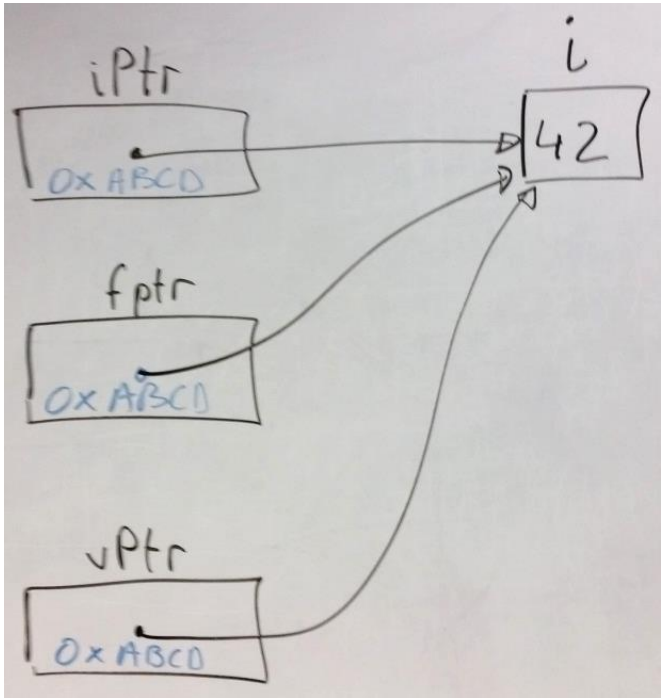
```
int i = 42;
int  *iPtr = &i;
float*fPtr;
void *vPtr;

vPtr = iPtr;
fPtr = vPtr;

printf("f = %f\n", *fPtr);
```

```
f = 0.000000
```

?




## C++

```
int i = 42;
int  *iPtr = &i;
float*fPtr;
void *vPtr;

vPtr = iPtr;
fPtr = vPtr;

cout <<*fPtr <<endl;
```

```
error C2440: '=' :
cannot convert from 'void *' to 'float *'
```

Actually, in most C compilers, the direct assignment

**fPtr = iPtr;**

works!

C

```
vPtr = iPtr;
fPtr = vPtr;
```

### 6.5.16.1 Simple assignment

**Constraints**

One of the following shall hold:[112)]

— the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type;

— the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of `void`, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

# QUIZ

Make it work with a C++ explicit cast!

**C++**

```
int i = 42;
int  *iPtr = &i;
float*fPtr;
void *vPtr;

vPtr = iPtr;
fPtr = vPtr;

cout <<*fPtr <<endl;
```

```
error C2440: '=' :
cannot convert from 'void *' to 'float *'
```

```
int i = 42;
int  *iPtr = &i;
float*fPtr;
void *vPtr;

vPtr = iPtr;
fPtr = static_cast<float *>(vPtr);

cout <<*fPtr <<endl;
```

```
5.88545e-044
```

?

It’s really a denormalized number in floating-point representation (IEEE 754).

# References

Review the coverage of references from:

Ch.3 – section **Introduction to C++ references**

Ch.8 – section **Passing and returning addresses**

- A *reference* (**&**) is like a constant pointer that is automatically dereferenced.
- One advantage of this “pointer” is that you never have to wonder whether it’s been initialized (the compiler enforces it) and how to dereference it (the compiler does it).

examples

```
// Ordinary free-standing reference:
int y;
int& r = y;
// When a reference is created, it must
// be initialized to a live object.
// However, you can also say:
const int& q = 12;  // (1)
// References are tied to someone else's storage:
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
} ///:~
```

There are certain rules when using references:

1. A reference must be initialized when it is created. (Pointers can be initialized at any time.)
2. Once a reference is initialized to an object, it cannot be changed to refer to another object. (Pointers can be pointed to another object at any time.)
3. You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

There are certain rules when using references:

1. A reference must be initialized when it is created. (Pointers can be initialized at any time.)
2. Once a reference is initialized to an object, it cannot be changed to refer to another object. (Pointers can be pointed to another object at any time.)
3. You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

The above rules make references:

- **safer** than pointers
- **less flexible** than pointers

# Pointers vs. references

Remember the philosophical difference between procedural and OO languages:

- C is close to the machine
- C++ is further from the machine, closer to the problem

……

However, C++ has both pointers and references! How to decide which one to use?

# Rules of thumb:

Use (const!) references in:

- **function arguments**
- **function return values**

Use pointers in:

- **function local variables**
- **class members**

# References in function arguments and return values

When a reference is used as a function argument, any modification to the reference *inside* the function will cause changes to the argument *outside* the function.

Of course, you could do the same thing by passing a pointer, but a reference has much cleaner syntax. (You can think of a reference as nothing more than a syntax convenience, if you want.)

examples

```
//: C11:Reference.cpp
// Simple C++ references

int* f(int* x) {
  (*x)++;
  return x; // Safe, x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe, outside this scope
}

int main() {
  int a = 0;
  f(&a); // Ugly (but explicit)
  g(a);  // Clean (but hidden)
} ///:~
```

Frequent mistake: Returning a reference (or pointer!) to an object whose scope is inside the function:

```
int& h() {
  int q;
//!  return q;  // Error
  static int x;
  return x; // Safe, x lives outside this scope
}
```

`return q` would be called a **leaked reference**.

# QUIZ

Is this program correct? Explain!

```
int* foo() {
  int a = 42;
  return &a;
}

int main() {
    int *iPtr;
    iPtr = foo();
    std::cout <<*iPtr  <<std::endl;
}
```

# QUIZ

Is this program correct? Explain!

```
int* foo() {
  int a = 42;
  return &a;
}

int main() {
    int *iPtr;
    iPtr = foo();
    std::cout <<*iPtr  <<std::endl;
}
```


Syntactically it's correct, but …

# QUIZ

Is this program correct? Explain!

```
int* foo() {
  int a = 42;
  return &a;
}

int main() {
    int *iPtr;
    iPtr = foo();
    std::cout <<*iPtr  <<std::endl;
}
```


```
warning C4172: returning address of local variable or temporary
```

Here's how it can go wrong:

```
int* foo() {
  int a = 42;
  return &a;
}

//this function doesn't do anything
//it's used just to overwrite stack
//space
void bar() {
  int a = 24;
}

int main() {
    int *iPtr;
    iPtr = foo();
    bar();
    std::cout <<*iPtr  <<std::endl;
}
```

```
24
```

We stopped before the subsection **Constant references**

This is all the material required for the 2nd exam.

The exam is during next week's lab (Monday).

Monday lecture is review/Q&A.