How do we implement **run-time constants** and **compile-time constants** **inside classes**?
Compile-time constants in classes

The **static** keyword inside a class means “**there’s only one instance, regardless of how many objects of the class are created**”

A **static const** of a **built-in type** can be treated as a compile-time constant.

**static const**s must be initialized when they are defined (all other data members must be initialized in the constructor or in other member functions)

User-defined types will be addressed soon!
QUIZ

What does the **volatile** keyword mean in C++?
volatile means “This data may change outside the knowledge of the compiler.”

Somehow, the environment is changing the data (possibly through multitasking, multithreading or interrupts), and volatile tells the compiler not to make any assumptions about that data, especially during optimization.
QUIZ

What is the c-v qualifier in C++?
What is the \texttt{c-v qualifier} in C++?

The syntax of \texttt{volatile} is identical to \texttt{const}, so discussions of the two are often treated together. The two are referred to in combination as the \texttt{c-v qualifier}. 
class Foo {
    int i;
public:
    Foo() { i=0; cout << "Construct!\n"; }
    ~Foo() { cout << "Destruct!\n"; }
};

void f() {
    static Foo f;
}

int main() {
    f();
    cout << "main ends here\n";
}

C:\Windows\system32\cmd.exe
Construct!
main ends here
Destruct!
Yes! Although the destructor is called when the object (f) ends its lifetime, for a static object the lifetime is the entire program!

class Foo {
    int i;
    public:
        Foo() { i=0; cout << "Construct!\n"; }
        ~Foo() { cout << "Destruct!\n"; }
    };

void f(){
    static Foo f;
}

int main(){
    f();
    cout << "main ends here\n";
}
Ch. 11: References & the Copy-Constructor
Pointers in C and C++

• The “void pointer assignment” trick in C
• C++ is more strongly typed
```c
int i = 42;
int *iPtr = &i;
float*fPtr;
void *vPtr;

vPtr = iPtr;
fPtr = vPtr;

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

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

vPtr = iPtr;
fPtr = vPtr;

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

Actually, in most C compilers, the direct assignment

```c
fPtr = iPtr;
```

works!
6.5.16.1 Simple assignment

Constraints

One of the following shall hold:\textsuperscript{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 \texttt{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!

```cpp
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 *'
```
It's really a denormalized number in floating-point representation (IEEE 754).
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).
// 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.
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.)
```cpp
//: C11:Reference.cpp
// Simple C++ references

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

int& g(int& x) {
    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:

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

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

Will this function compile?
If you know the function will respect the `const`ness of an object, making the argument a `const` reference will allow the function to be used in all situations.

This means that:

- for built-in types, the function will not modify the argument
- for user-defined types, the function will call only `const` member functions, and won’t modify any `public` data members.
Is this program correct? Explain!

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

int main() {
    int *iPtr;
    iPtr = foo();
    std::cout << *iPtr << std::endl;
}
```
Is this program correct? Explain!

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

int main() {
    int *iPtr;
    iPtr = foo();
    std::cout << *iPtr << std::endl;
}
```
Is this program correct? Explain!

```cpp
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:

```cpp
//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;
}
```
This can happen with references as well, as explained last time:

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

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

This is also known as *leaked pointers/references*. 
Quiz

One of the function calls is incorrect. Which one and why?

```cpp
void f(int&) {}
void g(const int&) {}

int main() {
    f(1);
    g(1);
}
```
QUIZ

Why would we want to pass a pointer-to-pointer in C?
In C++, we can pass a pointer reference (reference to a pointer) instead!

```cpp
//: C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
    int* i = 0;
    cout << "i = " << i << endl;
    increment(i);
    cout << "i = " << i << endl;
} ///:~
```

Make sure you grok this pointer arithmetic!
The copy-constructor

Passing & returning by value
a.k.a.

A little bit of assembly language and computer architecture
The definition of function f is elsewhere.

We’re showing the (simplified) assembly code that implements this line:

```assembly
push b
push a
call f()
add sp, 4
mov g, register a
```

Arguments pushed on stack from right to left.

Beginning address of f() - code is loaded in Program Counter.

When f() - code returns, execution continues here.
In cdecl convention, the caller is responsible for “cleaning up”, i.e. restoring the stack.

Do not confuse with the variable \texttt{a}. Functions return values in a specific register, known to the compiler, e.g. in an Intel CPU this is EAX.

```c
int f(int x, char c);
int g = f(a, b);
push b
push a
call f()
add sp,4
mov g, \texttt{register a}
```
int f(int x, char c);
int g = f(a, b);
push b
push a
call f()
add sp, 4
mov g, register a

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>before</td>
</tr>
<tr>
<td></td>
<td>f() is called</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>
The complete picture: **Function-call stack frame**

```c
int f(int x, char c);
int g = f(a, b);
```

```
push b
push a
call f()
add sp, 4
mov g, register a
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack before</td>
</tr>
<tr>
<td></td>
<td>f() is called</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Local address</td>
</tr>
<tr>
<td></td>
<td>Local variables</td>
</tr>
<tr>
<td></td>
<td>Function arguments</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Local variables</td>
</tr>
</tbody>
</table>
How does this change when passing a (large) user-defined object by value?

```c
int f(int x, char c);
int g = f(a, b);
```

```
push b
push a
call f()
add sp, 4
mov g, register a
```
//: C11: Passing Big Structures.cpp

struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} //://:
The question, however is: How to **return** the result? (There aren’t enough registers!)

```cpp
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} ///:~
```
One possible solution is to also push the return values on the stack:

```cpp
// C11: PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100;  // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
    ///:~
```
Historical note:
In K&R C, it was not possible to return structs, only pointers to structs.
Since C89, C functions can return structs, although many compilers don’t actually place the struct on the stack – they do return values optimization instead (where only a pointer is returned, e.g. in register eax).

This means that the caller is responsible for setting up space for return values, contrary to C custom!
However, the solution chosen in C++ is this: the address of `B2` is pushed before making the call (even though it’s not an argument).

```c++
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
} //://~
```
To comprehend why pushing return values on the stack is not easy, we need to understand the constraints on the compiler when it’s making a function call.

```cpp
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} //://~
Re-entrancy

• Interrupts and ISRs = Interrupt Service Routines (functions, really!)

• Recursive functions
Re-entrancy

Consider the following scenario:

• Function pushes return values on stack and returns normally

• Before the calling function has a chance to process the return values, an interrupt occurs, and the ISR overwrites them!

ISRIs are “smart” enough to save all registers on the stack, and restore them upon exit, but they have their limits.
Re-entrancy

Would placing the return values on the heap work?

Why not? 😊
Conclusion:

- Push the address of the return value’s destination on the stack as a *hidden* argument
- Let the function copy the return value into the destination before returning.

```cpp
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} //:~
We stopped before the subsection

Bitcopy vs. initialization

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

The exam is this Friday (Oct.30) during lab.

Material: chs. 5, 6, 7, 8, and 11 (partial).

Friday lecture is review.
Ch.11 exercises for review:

End-of-chapter 1 through 9.

What else you can do for review:

• Finish the homework for Ch.8 (due Friday!)

• Re-solve all quizzes from chs. 5, 6, 7, 8, 11. The instructor’s notes are on our webpage.