Variables

Assignment

```
int i;
// ...
i = 42;
```


Variable *assignment* takes a value and stores it in an L-value. As seen previously, the left-hand side of an assignment must be an L-value: there must be a place to store the value being assigned.

Problems with variables

Uninitialized variables

```
int i;
// ...
if (i < 42)
```

```
uninitialized-variables.cpp:7:6: warning: variable 'i' is uninitialized when used here
        if (i < 42)
            ^
uninitialized-variables.cpp:3:7: note: initialize the variable 'i' to silence this war
        int i;
             ^
              = 0
1 warning generated.
```


Fortunately, we have some help to keep us from relying on uninitialized memory. When we compile software, we can tell the compiler to give us warnings about potentially unsafe constructs such as the use of uninitialized memory. With Clang and GCC, simply use the `-Wall` ("Warnings: all") flag at the command line (e.g., `clang++ -std=c++11 -Wall foo.cpp`).

Code example: [uninitialized-variables.cpp]

Problems with variables

More popular errors:

```
if (i = 42)
{
    // ...
}
```

```
while (someData[i]++)
{
    // ...
}
```


This slide shows another two popular errors. In the first case, it looks like we meant to check whether a value `i` is 42 or not, but in fact, we are assigning the value 42 to it. In fact, the assignment `i = 42` will evaluate to 42, which will evaluate to `true` because it is non-zero. Therefore, instead of executing some code if `i` is 42, we will set `i` to 42 and then ______________ execute the code inside the "then" clause.

In the second case, it looks like we meant to *iterate* over an array: to look at each of its elements in turn then take some action until we reached the end (in this case, until `someData[i] == 0`). However, rather than incrementing the loop index `i`, we are actually keeping `i` the same and changing the value at position `i` within `someData`!

Problems with variables

Here's what we meant:

```
if (i == 42)
{
    // ...
}
```

```
while (someData[i++])
{
    // ...
}
```

Oops!

These kinds of errors (____) can be prevented can be prevented through the use of *constants*.

Constants

Contracts

Remember *preconditions* and *postconditions*?

$$\langle x >= 0 \rangle \; x := x + 1 \; \langle x > 0 \rangle$$

Constants are contracts:

```
const int x = 42;
```

I promise not to modify x

Constants are a form of *contract*, which is a concept we should have seen before in the form of *preconditions* and *postconditions*.

A precondition is a logical expression that some code ____________________________________. The code may not execute correctly if its preconditions are not met. A postcondition is a statement of something will be true after the code runs ____________________________________. Together, the pre- and post-conditions provide a logical contract for some code: "if you give me some acceptable value of x, I will return you some acceptable value of y."

Similarly, the use of the C++ keyword `const` is a contract, a promise from the ____________________ to the ________________: I promise ______________________________. A constant is not necessarily a guarantee that ______________________________. That is sometimes the case, but __________________.

Constants as a safety belt

```
const int i = 42;
const char someData[] = { 1, 2, 3 };

if (i = 42)
{
    // ...
}

int j = 0;
while (someData[j]++)
{
    // ...
}
```


If we had used the `const` keyword librally in the code above, ____ from making these mistakes. We would not have been able to assign to `i` in the `if` condition, because we had promised not to modify `i`. We would not be able to modify the values in `someData`, because again, we had promised not to modify anything in `someData`.

Code example: [const-modification.cpp]

Constants as a safety belt

```
int i = 42;

const int& j = i;  // ok: r/o reference to r/w
int& k = j;  // not ok: r/w reference to r/o

k = 99;
```

```
const-modification.cpp:33:7: error: binding value of type 'const int' to reference to
        int& k = j;
             ^   ~
const-modification.cpp:26:12: note: variable 'i' declared const here
        const int i = 42;
        ~~~~~~~~~~^~~~~~
```


These promises also apply to references. A `const` reference to an L-value is a reference and a promise: you promise not to modify whatever memory you are referencing.

In order to keep this promise, any new references derived from a `const` reference must also be `const`.

It is an error to to derive a non-`const` (read/write) reference from something that is `const`: the compiler ______________________.

Constants as a safety belt

```
int i = 10;

void thread1() { i *= 5; }
void thread2() { i += 1; }
```

What is the value of i?

Another way that constants can be helpful is when we share data among multiple *threads* of execution. Concurrent programming is the topic of a Term 7 software course, but for now, it's enough to know that our computers can do increasing numbers of things at the same time, and that ____.

In the example shown here, if `thread1` and `thread2` execute at the same time, it's impossible to predict whether the final value of `i` will be 51 or 55. As programmers, this *non-determinism* is a bit of a problem: we don't want to write software whose behaviours ______________________. Put another way, we like ____________________.

Constants can help us write concurrent software correctly because _______________________ ________________. If neither thread can modify `i` then the value of `i` becomes easy to determine again. If we need to combine the results of the two threads in some way, we can be clear and explicit about this --- using one of the many techniques that will be explained in Term 7!

Constants as an optimization

```
const int DaysInWeek = 7;
const int WeeksInSemester = 12;
const int CourseDays = WeeksInSemester * DaysInWeek;

for (int i = 0; i < CourseDays; i++)
{
    // ...
}
```

can become:

```
// Behold: constant propagation!
for (int i = 0; i < 84; i++)
{
    // ...
}
```


Constants can actually make your software faster, too. In this code example, the programmer is using ____. This is good for programmer understanding, and it's also good for _______________. Your first instinct might be to write `i < 84` rather than use all of these named constants, but what if the University decides to change the number of weeks in a semester? If we have _________________ like 84 scattered around our code, it will be very difficult to answer the question, "which values depend on the number of weeks in a semester?" Using named constance, we can fix our code by modifying one value, `WeeksInSemester`. If we recompile our software, that new value will be picked up by all of our code without any manual intervention.

However, having to re-calculate 7×12 every time we run the program is a little bit redundant. Fortunately, if the compiler can see that these values really are constant, then it will perform *constant propagation*, effectively changing the loop to use the condition `i < 84` when it's compiled. This gives the ____________________________ with the ______________________________________.

Constants as an optimization

A simple loop:

```
const std::string& s = getName();

while (x > s.length())
{
    // do something to x
}
```


This is another example of a loop whose performance can be improved through the use of a constant variable. In this example, we need to check the length of a string every time we go through the loop.

If, however, we know that the string isn't going to change length...

Constants as an optimization

A more efficient loop:

```
const std::string& s = getName();
const size_t length = s.length();

while (x > length)
{
    // do something to x
}
```


... then we can pull out its length into a constant value, which we can check every time we go through the loop much faster than if we had to re-determine the string's length.

Depending on the exact details of your code, the compiler may be able to __________________ _____________________________, but it's not guaranteed. Writing the code this way, however, it is.

Constants

Useful for:

- Guarding against silly errors
- Guarding against pernicious errors
- Speeding up software execution

Ask yourself:

*Does it **need** to be mutable?*

In summary, then, constants can be useful for guarding against silly-but-easily-made mistakes (e.g., assigning to a variable instead of checking it for equality), difficult-to-do-correctly software patterns (e.g., shared mutable state) and can even help make our software faster.

For all of these reasons, when you create a new variable, it is good to ask yourself: "does this ____________ to be mutable (i.e., non-`const`)?" The answer is "no" more often than you'd think!

`const`-unfriendly APIs

A common occurrence:

```
void someBadOldAPI(string&);

int main(int argc, char *argv[])
{
  const string greeting = "Hello, world!";
  someBadOldAPI(greeting);
}
```


However, despite all of the advantages and uses of `const`, not everyone uses them all of the time. There may be times when you need to use an Application Programming Interface (API) that wasn't designed with thought to mutability and *immutability* (`const`-ness). For instance, in this code example, you must pass a non-`const string` reference into a function in order to, say, display it to the user. However, you only have a `const` reference, so what can you do?

In general, the only safe thing to do is to make a copy of the data into a new, non-`const` variable and pass the function a reference to that copy. However, if you know that the function isn't actually going to change the data (e.g., in this case, it's going to display it to the user and it could be marked `const` if the API authors had thought about it), then you can use a ________________________ to remedy the situation.

Explicit conversions

Recall:

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

In our previous introduction to C++ casts, we saw some casts that weren't elaborated on. One of these was `const_cast`.

All casts are a way of telling the compiler, "I know what I'm doing, it's ok that I'm breaking the rules." In the cast of `const_cast`, the programmer is telling the compiler that, "yes, passing this value would break my promise demand `const` promises of others, but I've checked that it's ok in this case."

Code example: `const-cast.cpp`

Constants

Can require:

- More care during design
- Less sloppiness during design!
- `const_cast`

The liberal use of `const`, then, can require a little more care in the design of software. Often this is because the use of `const` will expose sloppiness and danger in software design, so it's well worth the extra work. When interfacing with APIs that take less care, however, `const_cast` may help you to ________________________ without making the computer do lots of extra work ____________________________.

Summary

Explicit conversions

Variables

- L- vs R-values
- References

Constants