const

The use of the reserved word `const` in front of a declaration, makes the identifier a constant rather than a variable. It must be given a value at the time of declaration.

```c
const int a = 500;
const double pi = 3.141592;
```
These cannot be changed after the point of declaration. They cannot be left values or subject to such operators as ++ and --.

Global constants.

A const declared within a file and outside any function, in particular, prior to the function `main()`, is a global constant. It can be referenced (used) inside any function in the file.

```c
const double pi = 3.141592;
int main() {
    etc.
}
```

Global variables

Global variables are defined similarly to global constants, just without the use of `const` before the declaration. Just as with a global constant, a global variable can be used by any of the functions.

WARNING: The use of global variables is normally frowned upon. *You need to know why?*
Scope rules

Simply stated, the scope of an identifier is the region or place that it is defined and can be referenced. We will for now restrict our attention to block and global scope. Global has already been discussed in the previous slide.

Block scope

Blocks are segments enclosed in {}. Identifiers declared in these have block scope and are valid within the {}, but not outside the {}. Examples would include variables defined inside a function.

Nested blocks:

What about {... {...}...} (nested)? There are two blocks here resulting in two scopes, one within the other.

\{\text{int } a; \ldots \{\text{int } b; \ldots\}\} \}

The variable \(a\) is defined throughout the whole thing (including inside the nested block). \(b\) is only valid inside the inner-most block. Outside the outer-most block, neither is valid.
What about scope of parameters?

Parameters in the header of a function definition, have block scope within that function. Example:

```c
int gcd(int a, int b){
    a and b are defined throughout this definition but not outside the function.
}
```

Declaring a variable inside a `for` header

```c
for(int i=1; i<10; i++){...
```

By ANSI/ISO C++ standard, i should have block scope inside the `{ }` constituting the `for` statement. WARNING: A given compiler may not follow this standard. Try ours for example.

Argument Passing

- Pass-by-value (call-by-value)
- Pass-by-reference (call-by-reference)

We have already discussed passing an argument to a parameter by value. Simply stated, a new variable is created with block scope in the function which is called, and a copy of the corresponding argument is placed into the parameter.
A problem:
Suppose the following function is to swap two integers:
```c
void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
```
Why wouldn’t this work if called by `swap(a,b);`?

Solution:
Change the header from:
```c
void swap(int x, int y)
```
to
```c
void swap(int &x, int &y)
```
The presence of `&` in front of `x` and `y` indicates that the arguments are passed by reference. What does this mean?

Pass (Call)—by-reference
If a parameter is preceded by `&` in the function header, a new memory location will not be created within the function, and the value of the corresponding argument will not be copied. Rather, the parameter identifier merely “becomes another name for” (reference to) the corresponding argument.
Consequence of Pass-by-ref.

Since the parameter really just references the corresponding argument in the calling function, any change made to that parameter by the called function is applied to the argument.

Look at the swap function again

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

If called by `swap(a,b);` then x and y reference a and b respectively. Thus a is moved into temp; b is moved into a; and temp is moved into b. The end result is that a and b are swapped back in their own scope.

Prototypes using pass-by-reference

```c
int swap(int &, int &);
```

If the parameter is to be pass-by-reference, then follow the type by &.

How are parameters passed in the following?
```c
int foo(double, double&, int&);
```

The first argument is a double passed by value; second is a double passed by reference, and third is an integer passed by reference.
Function Overloading

Would the following prototypes be allowed in the same program file; i.e., can you have different functions with the same name? Discussion and answer will lead to concept of overloading.

```c
int MAX(int, int);
int MAX(int, int, int);
double MAX(double, double);
```

Overloading (cont.)

Would the following be allowed in the same program?

```c
int max(int, int);
int max(int & , int &);
```

Not really. A call by `max(a,b);` would result in a compiling error – "Ambiguous call to an overloaded function." The compiler cannot determine which one was intended to be called.

Questions/Answers

- What is overloading?
- How does one implement it?
- What type(s) of attempted overloading may lead to ambiguity?