

## global variables

---

Variables declared outside any function are available to all functions  
They are called *external* variables or *global* variables

```
int g = 12;

void f(void) {
    printf("The value of g is %d\n", g); // prints 12
    g = 42;
}

int main(void) {
    f();
    printf("The value of g is %d\n", g); // prints 42
    return 0;
}
```



## *global variables*

---

- Avoid global variables - **NOT** needed in COMP1511
- make concurrency (threads) problematic
- creating hidden dependencies between parts of program
- make code reuse harder
- pollute the namespace - create a valid name everywhere you might accidentally use
- generally reduce readability
- global variable can be useful for "meta"-purposes  
e.g turning on-off debug logging through your program



## More C Operators

---

C provides some additional operators, which allow for shorter statements which can make your code a little more readable, or a lot less readable.

- pre/post-increment: `++i`, `i++` same as `i = i + 1`
- pre/post-decrement: `--i`, `i--` same as `i = i - 1`
- compound assignment operators:
  - ▶ `a += b` same as `a = a + b`
  - ▶ `a -= 5` same as `a = a - 5`
  - ▶ `a *= -10` same as `a = a * -10`
  - ▶ `a /= 2` same as `a = a / 2`
  - ▶ `a %= b` same as `a = a % b`



## Increment and Decrement Operators In Expressions

---

`++` and `--` can be used in in expressions

**NOT** recommended in COMP1511

They can be used *after* the variable:

```
k = 7;  
n = k--; // assign k to n, then decrement k by 1  
printf("%d %d", k, n) // k=6, n=7
```

They can be used *before* the variable:

```
k = 7;  
n = --k; // decrement k by 1, then assign k to n  
printf("%d %d", k, n) // k=6, n=6
```



## The *for* loop

---

There is also a construct called the *for* Loop:

```
for (expr1; expr2; expr3) {  
    statements;  
}
```

- *expr1* is evaluated before the loop starts.
- *expr2* is evaluated at the beginning of each loop; if it is non-zero, the loop is repeated.
- *expr3* is evaluated at the end of each loop.



## Example of *for* loop

---

```
for (x = 1; x <= 10; x++) {  
    printf("%d\n", x * x);  
}
```

Can declare variable if used only within for loop:

```
for (int x = 1; x <= 10; x++) {  
    printf("%d\n", x * x);  
}
```



## *for* loops and *while* loops

---

These two are equivalent:

```
for (expr1; expr2; expr3) {  
    statements;  
}
```

```
expr1;  
while (expr2) {  
    statements;  
    expr3;  
}
```



## Counting Down to Zero

---

Any of the 3 expressions in the *for* loop may be omitted  
';' must still be present. For example:

```
printf("Enter starting number for Countdown: ");  
scanf("%d", &n); // initial value entered by user  
for (; n >= 0; n--) {  
    printf("%d\n", n );  
}  
printf("Blast Off!\n");
```



## for Loop expressions

---

Although **NOT recommended**, the comma operator `,` can be used to squeeze multiple statements into *expr1* and *expr3*. For example,

```
for (int x=0, y=2; x < MAX; x++, y++) {  
    ...  
}
```



## *break* and *continue*

---

- *break* causes a loop to terminate; no more iterations are performed, and execution moves to whatever comes after the loop.
- *continue* causes the *current* iteration of the loop to terminate; execution moves to the next iteration.
  - ▶ with *while* and *do* loops, the conditional expression is tested before moving to the next iteration
  - ▶ with *for* loops, *expr3* is executed, then *expr2* is tested before moving to the next iteration
- *break* and *continue* used sparingly can make code more readable
- overuse of *break* and *continue* can make code incomprehensible



## *break and continue*

---

Here is a typical use of *break*:

```
for (int i = 0; i < LIMIT; i++) {  
  
    // lots of complex things happens here  
  
    if (/* need to stop loop immediately */) {  
        break; // exit loop immediately  
    }  
  
    // lots more complex things happens here  
}
```



## *break and continue Statement*

---

Here is a typical use of *continue*:

```
for (int i = 0; i < LIMIT; i++) {  
  
    // lots of complex things happens here  
  
    if (/* this is not what is wanted */) {  
        continue; // got next loop iteration  
    }  
  
    // lots more complex things happens here  
}
```



## Exiting A Program

---

- In main **return** will terminate program
- **stdlib.h** provides a function useful outside main::

```
void exit(int status);
```

- status passed to exit same a return value of main
- **stdlib.h** defines **EXIT\_SUCCESS** and **EXIT\_FAILURE**
- **EXIT\_SUCCESS** program executed successfully
- **EXIT\_FAILURE** program stopped due to an error
- **EXIT\_SUCCESS** == 0 on unix-like and almost all other systems



## Implicit Type Conversions

---

Recall that C supports 'hybrid' arithmetic operations involving certain types, in a way that mirrors our expectations. For example:

```
3 + 5.8
```

An integer is added to a double, giving a double result. However, at the machine level floating point addition requires two double arguments and is a distinct operation from integer addition.



## Implicit Type Conversions

---

Recall that C supports 'hybrid' arithmetic operations involving certain types, in a way that mirrors our expectations. For example:

```
3 + 5.8
```

An integer is added to a double, giving a double result. However, at the machine level floating point addition requires two double arguments and is a distinct operation from integer addition.

### Implicit Conversions

The compiler steps in and performs an automatic conversion, known as a *cast*, from integer to double.

```
double d = 3;  // 3 is converted to double  
int i = 5;  
d = d + i;     // i is converted to double
```



## Implicit Type Conversions

---

Implicit conversions are generally performed when considered 'safe', e.g., numeric types are converted to other numeric types with larger capacity. But sometimes unsafe implicit conversions are also performed, a common criticism of C. Consider:

```
int i = 1000;  
char c1 = 100; // statically checked, OK  
char c2 = 1000; // statically checked, warning  
char c3 = i;    // no warning
```

### NB

You should be mindful of implicit conversions, often they make coding easier, but sometimes they can mask programming errors!



## Explicit Type Conversions

---

C allows us to perform our own, explicit type casts, using the syntax *(type)*. For example:

```
double d1 = 1 / 2;  
double d2 = 1 / (double) 2;
```

Will the values of d1 and d2 be different?



## Explicit Type Conversions

---

C allows us to perform our own, explicit type casts, using the syntax *(type)*. For example:

```
double d1 = 1 / 2;  
double d2 = 1 / (double) 2;
```

Will the values of d1 and d2 be different? Yes!

It is good programming style to identify potentially unsafe implicit conversions and make them explicit:

```
#include <limits.h>  
#include <assert.h>  
...  
assert(i >= CHAR_MIN && i <= CHAR_MAX);  
char c = (char) i; // for some int i
```



## Explicit Type Conversions

---

### NB

When using explicit casts the compiler will often assume that you know what you are doing and not issue warnings even when a cast is very likely unsafe!

For example:

```
int i = 1000;
char c = (char) i;
int *ip = (int *) i;
int nums[] = {0};
printf("%c\n", (char) i);
printf("%s\n", (char *) &i);
printf("%s\n", (char *) nums);
```

Casts are used here to view one type as another, often dangerous!