**global variables**

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

```c
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;
}
```

- 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

**static functions**

- functions are shared between files by default
- this is undesirable in large programs because name clashes become likely
- name clashes also make code reuse difficult
- `static` keyword makes function visible only within file, in other words `static` limits function's scope
- if a function doesn't need to be visible declare it static, e.g.:
  ```c
  static double helper_function(int x, double y);
  ```
- allows files to be *de facto* modules in C
- similarly `static` makes global variables visible only within file
- beware `static` different meaning for local (function) variables

**Static Function Variables**

- when a function is called its variables are created
- when a function returns its variables are destroyed
- `static` changes lifetime of a function (local) variable
- value preserved between function calls
- static variables make concurrency difficult and programs harder to reader and understand
- rarely good reason to use static variables
- do **NOT** use in COMP1511
- note very different meaning to using `static` outside functions
- poor language design
Static Variables

For example, here is a function that counts how many times its has been called:

```c
void count(void) {
    static int call_count = 0;
    call_count++;
    printf("I have been called %d times\n", call_count);
}
```

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 expressions.

**NOT** recommended in COMP1511.

They can be used after the variable:

```c
k = 7;
++k; // assign k to n, then increment k by 1
printf("%d %d", k, n) // k=7, n=7
```

They can be used before the variable:

```c
k = 7;
--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:

```c
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

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

Can declare variable if used only within for loop:

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

---

*for* loops and *while* loops

These two are equivalent:

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

```c
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:

```c
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!\verb|\n|");
```

---

*for* Loop expressions

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

```c
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 Statement

Here is a typical use of **continue**:

```c
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::
  ```c
  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 Conversions

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

\[
\text{double } d = 3; \quad // \text{3 is converted to double}
\]

\[
\text{int } i = 5;
\]

\[
d = d + i; \quad // \text{i is converted to double}
\]

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 \((\text{type})\). For example:

\[
\text{double } d1 = 1 / 2;
\]

\[
\text{double } d2 = 1 / (\text{double}) 2;
\]

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:

\[
\#include <limits.h>
\]

\[
\#include <assert.h>
\]

\[
... \quad \text{assert}(i \geq \text{CHAR\_MIN} \&\& i \leq \text{CHAR\_MAX});
\]

\[
\text{char } c = (\text{char}) i; \quad // \text{for some int } i
\]

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:

\[
\text{int } i = 1000;
\]

\[
\text{char } c = (\text{char}) i;
\]

\[
\text{int } *ip = (\text{int} *) i;
\]

\[
\text{int} \text{ nums[]} = \{0\};
\]

\[
\text{printf}("%c\n", (\text{char}) i);
\]

\[
\text{printf}("%s\n", (\text{char} *) &i);
\]

\[
\text{printf}("%s\n", (\text{char} *) \text{nums});
\]

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

We can use the keyword typedef to give a name to a type:

```c
typedef double real;
```

This means variables can be declared as numeric but they will actually be of type double.

Do not overuse typedef - it can make programs harder to read, e.g.:

```c
typedef int andrew;
```

```c
andrew main(void) {
  andrew i,j;
  ....
```

Using typedef to make programs portable

Suppose have a program that does floating-point calculations. If we use a typedef'ed name for all variable, e.g.:

```c
typedef double real;
```

```c
real matrix[1000][1000][1000];
```

```c
real my_atanh(real x) {
  real u = (1.0 - x)/(1.0 + x);
  return -0.5 * log(u);
}
```

If we move to a platform with little RAM, we can save memory (and lose precision) by changing the typedef:

```c
typedef float real;
```

Structs

- We have seen simple types e.g. int, char, double
  - variables of these types hold single values
- We have seen a compound type: arrays
  - array variables hold multiple values
  - arrays are homogenous - every array element is the same type
  - array element selected using integer index
  - array size can be determined at runtime
- Another compound type: structs
  - structs hold multiple values (fields)
  - struct are heterogeneous - fields can be different type
  - struct field selected using name
  - struct fields fixed

```c
#define MAX_NAME 64
#define N_LABS 10

struct student {
  int zid;
  char name[MAX_NAME];
  double lab_marks[N_LABS];
  double assignment1_mark;
  double assignment2_mark;
}
```

We can declare an array to hold the details of all students:

```c
struct student comp1511_students[900];
```
combining structs and typedef

Common to use typedef to give name to a struct type.

```c
struct student {
    int zid;
    char name[64];
    double lab_marks[N_LABS] double assignment1_mark;
    double assignment2_mark;
};

typedef struct student student_t;

student_details_t comp1511_students[900];
```

Programmer often use convention to separate type names e.g. _t suffix.

Assigning structs

Unlike arrays, it is possible to copy all components of a structure in a single assignment:

```c
struct student_details student1, student2;
...
student2 = student1;
```

It is not possible to compare all components with a single comparison:

```c
if (student1 == student2) // NOT allowed!
```

If you want to compare two structures, you need to write a function to compare them component-by-component and decide whether they are “the same”.

 structs and functions

A structure can be passed as a parameter to a function:

```c
void print_student(student_t student) {
    printf("%s %d\n", d.name, d.zid);
}
```

Unlike arrays, a copy will be made of the entire structure, and only this copy will be passed to the function.

Unlike arrays, a function can return a struct:

```c
student_t read_student_from_file(char filename[]) {
    ....
}
```

Pointers to structs

If a function needs to modify a structs field or if we want to avoid the inefficiency of copying the entire struct, we can instead pass a pointer to the struct as a parameter:

```c
int scan_zid(student *s) {
    return scanf("%d", &((*s).zid));
}
```

The “arrow” operator is more readable:

```c
int scan_zid(student *s) {
    return scanf("%d", &(*s.zid));
}
```

If s is a pointer to a struct s->field is equivalent to (*s).field
Nested Structures

One structure can be nested inside another

typedef struct date Date;
typedef struct time Time;
typedef struct speeding Speeding;

struct date {
    int day, month, year;
};
struct time {
    int hour, minute;
};
struct speeding {
    Date date;
    Time time;
    double speed;
    char plate[MAX_PLATE];
};