Computer Programming: Skills & Concepts (INF-1-CP1)
Functions

12th October, 2010

Summary of Lecture 9

- Developing `triangle` using the `descartes` graphics routines.

Functions

- Certain computations may have to be done repeatedly with different input
- For instance: compute the minimum or maximum of two numbers
- Functions enable compact handling of this

"Triangle numbers"

```
#include <stdlib.h>

int main(void)
{
  int i,  sum = 0,  n;
  printf("The integer n, please: ");
  scanf("%d",&n);
  for (i = 1;  i <= n;  ++i) {
    sum += i;     /* sum = sum + i */
  }
  printf("sum = %d\n", sum);
  return EXIT_SUCCESS;
}
```

Equivalent formulation

```c
#include <stdlib.h>
#include <stdio.h>

int SumTo(int n)
/* computes 1 + 2+ ... + n */
{
  int i, sum = 0;
  for (i = 1;  i <= n;  ++i) {
    sum += i;
  }
  return sum;
}

int main(void)
{
  int  n;
  printf("The integer n, please: ");
  scanf("%d",&n);
  printf("sum = %d\n", SumTo(n));
  return EXIT_SUCCESS;
}
```

Use of the function

A function must be defined before use.
After the declaration, `SumTo(`*expr*`)` is an expression of type `int` whenever *expr* is an expression of type `int`.

```c
printf("sum = %d\n", SumTo(n));
```

In the example, `printf()` expects an integer expression, so we're fine.

Function definition

The initial line

```c
int SumTo(int n)
```

is the *header*. It tells the compiler that `sum` is a function taking one argument of type `int` and returning a value of type `int`.
The part in braces

```c
{
   ...
}
```

is the *body*. It specifies how the function is to be computed. It is like a little program in itself: it opens with some *declarations*, followed by some *statements*.

A closer look at the header of a function

▶ `int` gives the type of the result. The keyword `void` indicates that the function does not produce a result.

▶ `SumTo` is the name of the function: how we will refer to it in the remainder of the program.

▶ The part in parentheses, in this case `(int n)`, specifies the *formal parameters* and their types. In this case there is one parameter of type `int`. The keyword `void` indicates that the function has no parameters.

All of this is required by the compiler so it can check that the function is always used correctly.

A closer look at the body of a function

- `{ }` braces enclose the body of the function definition;
- `int i, sum = 0;` variables local to the function are declared here. We may choose to initialise some of them;
- `for ...` the code to be executed when the function is called;
- `return sum;` the return statement terminates the function call and specifies the value returned.

Things to note

- Local variables overshadow existing global ones.
- `return` statements may appear anywhere in the body. When executed the function body is left.
- parameters may be used as local variables, e.g., we could have written the loop as

```
sum = 0;
while (n >= 1) {
    sum += n;
    --n;
}
```
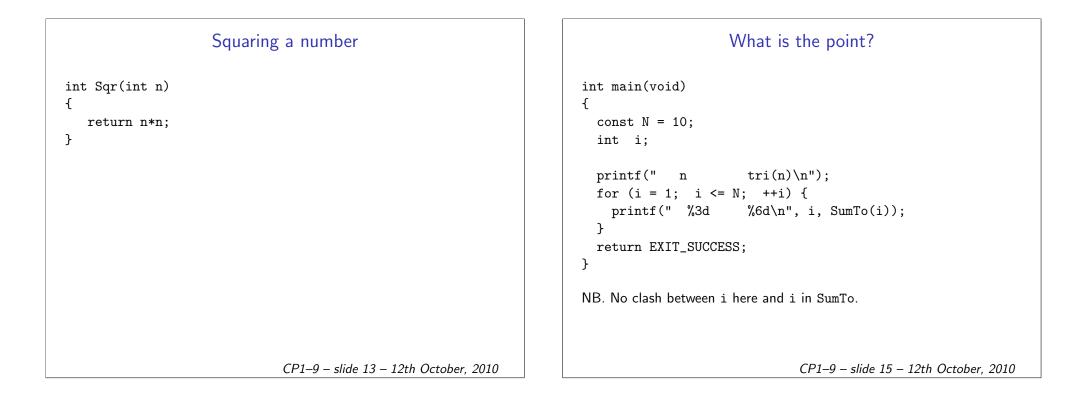
Local Variables

Variables defined *within a function* are its local variables.

- they are only valid within the function
- they are destroyed when the function finishes
- what happens when the function is called a second time?

Maximum, minimum

```
float max(float x, float y)
{
  if (x < y)
    return y;
  else
    return x;
}
```

Squaring a number

```c
int Sqr(int n)
{
   return n*n;
}
```

Length of a line segment

```c
float Length(lineSeg_t l)
{
   return sqrt(
     Sqr(XCoord(InitialPoint(l)) - XCoord(FinalPoint(l)))
     + Sqr(YCoord(InitialPoint(l)) - YCoord(FinalPoint(l)))
   );
}
```

What is the point?

```c
int main(void)
{
  const N = 10;
  int  i;

  printf("   n        tri(n)\n");
  for (i = 1;  i <= N;  ++i) {
    printf("  %3d     %6d\n", i, SumTo(i));
  }
  return EXIT_SUCCESS;
}
```

NB. No clash between `i` here and `i` in SumTo.

```c
if (ToRight(p, v1, v2)
    && ToRight(p, v2, v3)
    && ToRight(p, v3, v1)) {
  printf("Interior!\n");
} else if (!ToRight(p, v1, v3)
           && !ToRight(p, v3, v2)
           && !ToRight(p, v2, v1)) {
  printf("Boundary!\n");
} else {
  printf("Exterior!\n");
}
```

Saves writing similar code six times!

Scope

Scope refers to the conventions where a variable is valid

- ▶ global variables are defined before the `main` function and are valid everywhere
- ▶ local variables are defined within a function and are only valid there
- ▶ `main` is also a function: its variables are only valid there
- ▶ the scope of local variables overshadows the scope of global variables with the same name

Scope. (Spot the difference!)

```c
int a = 0, i = 0, n = 0;

void f(int n)
{ int i;  i = i + 1;  n = n + 1;  a = a + 1;}

int main(void)
{
  printf("Checkpoint A:  i = %d, n = %d and a = %d\n",
         i, n, a);
  f(n);
  printf("Checkpoint B:  i = %d, n = %d and a = %d\n",
         i, n, a);
  return EXIT_SUCCESS;
}
```

Scope

```c
int a = 0;

void f(int n)
{ int i;  i = i + 1;  n = n + 1;  a = a + 1;}

int main(void)
{
  int i = 0, n = 0;
  printf("Checkpoint A:  i = %d, n = %d and a = %d\n",
         i, n, a);
  f(n);
  printf("Checkpoint B:  i = %d, n = %d and a = %d\n",
         i, n, a);
  return EXIT_SUCCESS;
}
```