Lecture 4

Code branching

is about choosing the right path

- Branching is deciding what actions to take
- The program chooses to follow one branch or another
- *if()*
- *? :* operator
- *switch()*

if()

If today is Monday I will study C programming

- Based on a concept of *TRUE* or *FALSE*
- *TRUE* is a statement that evaluates to a nonzero value
- *FALSE* evaluates to zero
- Use of relational operators:
  - $>$ greater than - $5 > 4$ is TRUE
  - $<$ less than - $4 < 5$ is TRUE
  - $>=$ greater than or equal - $4 >= 4$ is TRUE
  - $<=$ less than or equal - $3 <= 4$ is TRUE
  - $==$ equal to - $5 == 5$ is TRUE
  - $!=$ not equal to - $5 != 4$ is TRUE
- exapmles ...

Do not use = to test equality, use == !!!

AND and OR

&& and ||

- Used for more complex logical statements
- && - logical AND
- || - logical OR

Basic if syntax

```
if (statement that evaluates to TRUE or FALSE)
  instruction

if (statement that evaluates to TRUE or FALSE)
{
  multi
  line
  instruction
}
```

examples ...

What else?

```
if (statment that evaluates to TRUE or FALSE)
  instruction
else
  another set of instructions

if (statment that evaluates to TRUE or FALSE)
{
  multi
  line
  instruction
}
else
  another set of instructions - could be multiline
```

examples ...

else if()

```
if (statment that evaluates to TRUE or FALSE)
  instruction
else if()
  another set of instructions
else if()
 ...

if (statment that evaluates to TRUE or FALSE)
{
  multi
  line
  instruction
}
else if()
  another set of instructions - could be multiline
```

examples ...

Inline if

?:

- It is like an *if else*
- Might be used within expressions
- The only ternary operator in C

```
if condition is true ? then X return value : otherwise Y value;

int a=5;
int b=1, c=2;
int d = b > c ? a + b : a + c;
```

examples ...

switch()

- Much like nested if else
- Might be more efficient

```
switch( expression )
{
  case expr1:
    instructions;
    break;
  case expr2:
    instructions;
    break;
  default:
    instructions;
}
```

- key word *break*
- key word *default*
- examples...

Functions

Enclose actions into separate procedures

- It makes sense to split the program into easy to maintain pieces
- We use the concept of **functions** to encapsulate actions that our program might perform
- Much as mathematical functions, functions in **C** have arguments.
- Functions might **return** a value (or not) to the *caller*

Syntax of a function:

```
type_of_function function_name (list_of_arguments)
{
  // body of a function
  return statment ends the function
}
```

Functions

Enclose actions into separate procedures

- Functions are compiled separately.
- Functions can be stored in different files.
- Once compiled functions are linked into the executable during Linking process.
- Nots: Linker errors are nasty!

Functions

Enclose actions into separate procedures

- Functions can be **declared** and **defined** before the call is made,
- or the **definition** can be postponed with **declaration** only visible.
- Functions are identified by type, name and arguments.
- Consequently there can be more than one function with the same name.

```
int fun1 (int a) { return a+5; }
int fun2 (int a, int b);
int main(){
  int b = fun1(5);
  int c = fun1(5, 7);
}
int fun2 (int a, int b)
{
  return a+b;
}
```

Note: Identifiers of arguments apply only within a function body!