Lecture 5
C++ for real-time systems

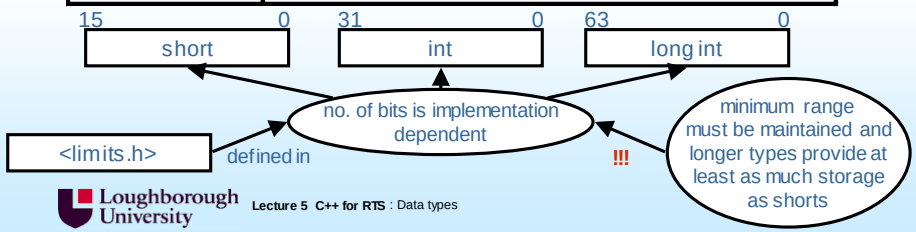
Objectives

This lecture aims to detail the aspects of software implementations in C/C++ that are target specific

- data types
- floating point (mis)behaviour
- portability (quick re-cap)
- assembly language interfacing

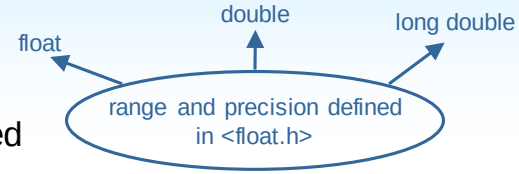
Integer data types

type	numeric range
char	integer character code range 0 to +127
signed char	signed integer range -128 to +127
unsigned char	unsigned integer range 0 to 255
short int	signed integer minimum range –32768 to 32767
unsigned short int	unsigned integer minimum range 0 to 65535
int	signed integer minimum range –32768 to 32767
unsigned int	unsigned integer minimum range 0 to 65535
long int	signed integer minimum range –2147483648 to 2147483647
unsigned long int	unsigned integer minimum range 0 to 4294967291



Floating point types

Real values are stored with a precision determined by the number of bits occupied by the mantissa



Each type has a value `EPSILON` which is the smallest number such that `1.0 + EPSILON != 1.0`

The range of each type is also defined, for example

- `FLT_MAX` is the largest floating point number
- `DBL_MIN` is the smallest double length floating point number

Numerical limits in C++

The standard C++ library provides a template *numeric_limits* defined in <limits>

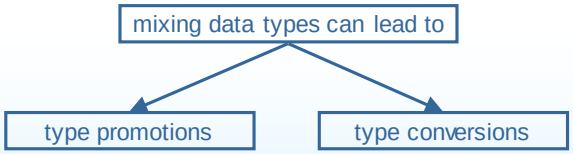
The template class contains public static members

```
#include <iostream>
#include <limits>
using namespace std;

void main(void)
{
  cout << "The minimum value for int is" <<
          numeric_limits <int>::min() << endl;
  cout << "The number of bits to represent an int is" <<
          numeric_limits <int>::digits << endl;
  cout << "The epsilon for float is" <<
          numeric_limits <float>::epsilon() << endl;
}
```

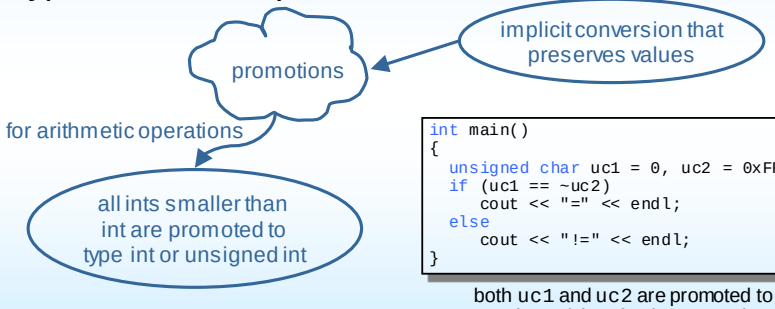

Mixing data types

Integer and floating point types can be mixed freely in assignments and expressions



Type promotions

The act of widening a narrower version of an integer type is known as *promotion*



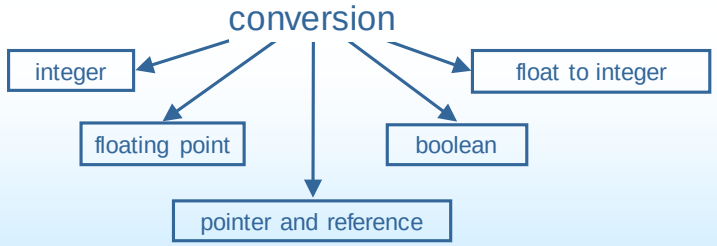

```
int main()
{
  unsigned char uc1 = 0, uc2 = 0xFF;
  if (uc1 == ~uc2)
     cout << "=" << endl;
  else
     cout << "!=" << endl;
}
```

both `uc1` and `uc2` are promoted to `unsigned int` for the comparison

Type conversions

C/C++ allows a wide range of type conversions

Can give rise to unexpected truncations when porting code



Integer conversion

an integer can be converted to other integer types

If the destination type is wider than source…
- ► **higher order bits are padded**

If the destination type is narrower than source…
- ► **higher order bits are discarded**

When converting signed `int` to `unsigned int`…
- ► **sign bit treated as information**

When converting `unsigned int` to signed `int`…
- ► **most significant bit treated as sign bit**

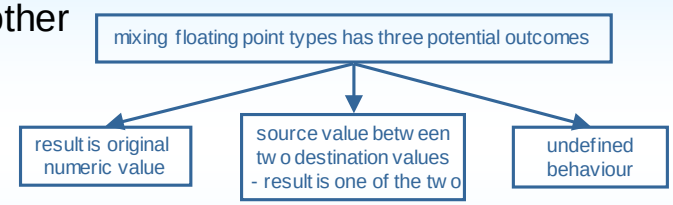
A Boolean value can be implicitly converted to its integer equivalent

Integer conversion examples

```
int main()                  // assume 16 bit short
{
  signed char     sc;
  unsigned char   uc;
  unsigned short  us;
  short           ss;
                            // decimal                binary
  uc = 255;                 //   255                  11111111
  us = uc;                  //   255           0000000011111111
  us = 0xFF00;              // 65280           1111111100000000
  uc = us;                  //     0                  00000000
  sc = -1;                  //    -1                  11111111
  uc = sc;                  //   255                  11111111
  uc = 128;                 //   128                  10000000
  sc = uc;                  //  -128                  10000000
}
```


Floating point conversions

One floating point type can be converted to another




```
#include <float.h>
int main()
{
  float   f = FLT_MAX;    //largest float
  double  d = f;          //OK d == f
  float  f2 = d;          //OK
  double d2 = DBL_MAX;    //largest double
  float  f3 = d2;         //undefined if FLT_MAX < DBL_MAX
}
```


Pointer and reference conversions

Any pointer to a variable can be implicitly converted to a void*

In C++, a pointer (reference) to a derived class can be implicitly converted to a pointer (reference) to an accessible base class

Boolean conversions


```
void main(void)
{
  int i = 0;
  int *ip = &i;
  float j = 32.0;
  bool b1, b2, b3;

  b1=ip;          //true if ip !=0
  b2=j;           //true
  b3=i;           //false
}
```


Floating to integer conversion

Floating point conversion to an integer value results in the fractional part being discarded

```
int i = 3.994;  //i=3
```

Converting from `int` to `float` is as accurate as the machine hardware allows

```
float f =(float)1234567890;   //OK as long as machine can
                              //represent this exactly as float
f = 1/2;
```


Usual arithmetic conversions (implicit casting)

- if either operand is of type `long double` the other is converted to `long double`
- if either operand is `double`, the other is converted to `double`
- if either operand is `float`, the other is converted to `float`
- any operand of a type shorter or equal to `int` in size is converted to `int` (i.e. `char`, `signed char`, `unsigned char`, `short`, or `unsigned short`)
- an enumerated type is converted to the first of `int`, `unsigned int`, `long`, or `unsigned long` that accommodates the range of the enumerators
- if either operand is `unsigned long`, the other is converted to `unsigned long`
- if one operand is `long` and the other is an `unsigned int`, then both operands are converted to `unsigned long`
- if either operand is `long`, the other is converted to `long`

Floating point misbehaviour

Some floating point truncation problems

- patriot missile time calculation results in it missing a Scud missile in the Gulf War leading to the loss of 28 lives
- Vancouver stock exchange index calculation truncated rather than rounded values, resulting in 50% fall in value

Most floating point errors are due to code fragments such as…

```
float a, b;
//assign values to a and b
if (a == b) { ...
}
else { ...
}
```

Result is not predictable in advance
Result may differ from machine to machine

Floating point misbehaviour

Comparisons - use `FLT_EPSILON`

```
enum {NO, YES};

int flpcmp(float a, char z[], float b)
{
  if (strcmp(z, "==")==0) {
    if (fabs(a-b) <= (max(fabs(a), fabs(b))*FLT_EPSILON)) {
      return YES;
    }
    else {
      return NO;
    }
  }
  return NO;
}
```

```
void main(void)
{
  float a, b;
  // assign values to a and b
  if(flpcmb (a, "==", b)) { ...
  }
}
```


Floating point misbehaviour

Loop counters - use `FLT_EPSILON`

```
void main(void)
{
  float a, b;
  // assign values to a and b
  for(a = 0.0; a < b; a + = 0.15) { ...
  }
}
```

should be replaced by

```
for(a = 0.0; flpcmp(a, "<", b); a += 0.15) { ...
}
```


Portability

Already discussed in lecture 1

Important aspects are…

- compilers have libraries of different quality
- C/C++ is open to interpretation in its implementation
- target platforms can have different CPUs, memory, peripherals, peripheral mappings, OSs

Assembly language interfacing

most C compilers support the **asm** keyword

allows inline assembly language statements

```
class io_port
{
 private:
  int data;
 public:
  // member list
};
io_port serial_port;

__asm
{
  mov ebx, OFFSET serial_port
  mov ecx, [ebx] serial_port.data; //[ebx].data if unique name
}
```

many C compilers have a -s option that instructs the compiler to generate an assembly language file

Obtaining language standards

The full range of unspecified, undefined and implementation defined features of C or C++ can be obtained from the relevant standard

C11 standard ISO/IEC 9899:2011

- can be purchased from http://www.iso.org - CHF238 for the PDF
- final draft of standard (very close to final version) is available at..
  http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

C++11 standard ISO/IEC 14882:2011

- can be purchased from http://www.ansi.org - $30 for the PDF
- final draft of standard is available at..
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

Summary

When dealing with data types be careful with

- rounding errors
- truncation
- resolution

Non-standard language features are needed to support low level programming