# EDAF30 – Programming in C++

## *11.Low-level details. Loose ends.*

Sven Gestegård Robertz
*Computer Science, LTH*

2021

# Outline

# Integer types

- Signed integers

| Type | Size | Range (at least) |
|---|---|---|
| `signed char` | 8 bits | $[-127, 127]^*$ |
| `short` | at least 16 bits | $[-2^{15}+1,\ 2^{15}-1]$ |
| `int` | at least 16 bits, usually 32 | $[-2^{15}+1,\ 2^{15}-1]$ |
| `long` | at least 32 bits | $[-2^{31}+1,\ 2^{31}-1]$ |
| `long long` | at least 64 bits | $[-2^{63}+1,\ 2^{63}-1]$ |

$^*$typically $[-128, 127]$, etc.

- Unsigned integers
  - same size as corresponding **`signed`** type
  - **`unsigned char`**: $[0, 255]$ , **`unsigned short`**: $[0,\ 2^{16}-1]$. etc.
- special case
  - **`char`** (can be *represented* as **`signed char`** *or* **`unsigned char`**)
  - Use **`char`** only for characters
  - Use **`signed char`** or **`unsigned char`** for integer values
- Sizes according to the standard:

  **`char`** $\leq$ **`short`** $\leq$ **`int`** $\leq$ **`long`** $\leq$ **`long long`**

# Integer types
## Overflow

- overflow of an **`unsigned`** n-bit integer is defined as *the value modulo* $2^n$
- overflow of a **`signed`** integer is *undefined*

# Integer types

## Example with sizeof

```
#include <iostream>
using namespace std;
int main () {
   cout << "sizeof(char)= \t" << sizeof(char)<<endl;
   cout << "sizeof(short)= \t" << sizeof(short) <<endl;
   cout << "sizeof(int) = \t" << sizeof(int) <<endl;
   cout << "sizeof(long)= \t" << sizeof(long)<<endl;
}
```

```
sizeof(char)=   1
sizeof(short)=  2
sizeof(int) =   4
sizeof(long)=   8
```

# Integer types
## Sizes are specified in <climits>

| | |
|---|---|
| CHAR_BIT | Number of bits in a **char** object (byte) (>=8) |
| SCHAR_MIN | Minimum value for an object of type **signed char** |
| SCHAR_MAX | Maximum value for an object of type **signed char** |
| UCHAR_MAX | Maximum value for an object of type **unsigned char** |
| CHAR_MIN | Minimum value for an object of type **char** (either SCHAR_MIN or 0) |
| CHAR_MAX | Maximum value for an object of type **char** (either SCHAR_MAX or UCHAR_MAX) |
| SHRT_MIN | Minimum value for an object of type **short int** |
| SHRT_MAX | Maximum value for an object of type **short int** |
| USHRT_MAX | Maximum value for an object of type **unsigned short int** |
| INT_MIN | Minimum value for an object of type **int** |
| INT_MAX | Maximum value for an object of type **int** |
| UINT_MAX | Maximum value for an object of type **unsigned int** |
| LONG_MIN | Minimum value for an object of type **long int** |
| LONG_MAX | Maximum value for an object of type **long int** |
| ULONG_MAX | Maximum value for an object of type **unsigned long int** |
| LLONG_MIN | Minimum value for an object of type **long long int** |
| LLONG_MAX | Maximum value for an object of type **long long int** |
| ULLONG_MAX | Maximum value for an object of type **unsigned long long** |

# Integer types

## Sizes are specified in `<climits>`

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << CHAR_MIN << ", " << CHAR_MAX << ", ";
    std::cout << UCHAR_MAX << std::endl;
    std::cout << SHRT_MIN << ", " << SHRT_MAX << ", ";
    std::cout << USHRT_MAX << std::endl;
    std::cout << INT_MIN << ", " << INT_MAX << ", ";
    std::cout << UINT_MAX << std::endl;
    std::cout << LONG_MIN << ", " << LONG_MAX << ", ";
    std::cout << ULONG_MAX << std::endl;
    std::cout << LLONG_MIN << ", " << LLONG_MAX << ", ";
    std::cout << ULLONG_MAX << std::endl;

}
```

```
128, 127, 255
-32768, 32767, 65535
-2147483648, 2147483647, 4294967295
-9223372036854775808, 9223372036854775807, 18446744073709551615
-9223372036854775808, 9223372036854775807, 18446744073709551615
```

# Integer types
## Sizes are implementation defined

Typedefs for specific sizes are in `<cstdint>` (`<stdint.h>`)

- integer types with exact with:

  `int8_t int16_t int32_t int64_t`

- fastest signed integer type with at least the width

  `int_fast8_t int_fast16_t int_fast32_t int_fast64_t`

- smallest signed integer type with at least the width

  `int_least8_t int_least16_t int_least32_t int_least64_t`

- signed integer type capable of holding a pointer:

  `intptr_t`

- unsigned integer type capable of holding a pointer:

  `uintptr_t`

The corresponding unsigned typedefs are named `uint_..._t`

# union

The size of a normal **struct** (**class**) is *the sum* of its members

```
struct DataS {
    int nr;
    double v;
    char txt[6];
};
```

All members in a **struct** are laid out after each other in memory.

The size of a **union** is equal to *the size of the largest member*

```
union DataU {
    int nr;
    double v;
    char txt[6];
};
```

All members in a **union** have *the same address*: only one member can be used at any time.

## Example use of DataU

```
union DataU {
    int nr;
    double v;
    char txt[6];
};
```

```
DataU a;

a.nr = 57;
cout << a.nr << endl;          57

a.v = 12.345;
cout << a.v << endl;           12.345

strncpy(a.txt, "Tjo",6);
cout << a.txt << endl;         Tjo
```

*The programmer is responsible for only using the “right” member*

## union

# Example of wrong use

```
using std::cout;
using std::endl;

union Foo{
  int i;
  float f;
  double d;
  char c[10];
};
int main()
{
  Foo f;

  f.i = 12;
  cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;

  strncpy(f.c, "Hej, du", 10);
  cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;
}
```

```
12, 1.68156e-44, 5.92879e-323, ^L
745170248, 3.33096e-12, 1.90387e-306, Hej, du
```

## union

### encapsulate a union in a class to reduce the risk of mistakes

```
struct Bar{
    enum {undef, i, f, d, c} kind;
    Foo u;
};
void print(Bar b) {
    switch(b.kind){
    case Bar::i:
        cout << b.u.i << endl;
        break;
    case Bar::f:
        cout << b.u.f << endl;
        break;
    case Bar::d:
        cout << b.u.d << endl;
        break;
    case Bar::c:
        cout << b.u.c << endl;
        break;
    default:
        cout << "???" << endl;
        break;
    }
}
```

```
void test_kind()
{
    Bar b{};

    b.kind = Bar::i;
    b.u.i = 17;

    print(b);

    Bar b2{};
    print(b2);
}
```

```
17
???
```

## anonymous **union** – removes one level

An alternative to the previous example:

```
struct FooS{
  enum {undef, k_i, k_f, k_d, k_c} kind;
  union{
    int i;
    float f;
    double d;
    char c[10];
  };
};

FooS test;

test.kind = FooS::k_c;
strncpy(test.c, "Testing", 10);
if(test.kind == FooS::k_c)
    cout << test.c << endl;
```

```
Testing
```

*Exposing the tag to the users is brittle.*

# Tagged union
## A class with anonymous union and access functions

```
struct FooS{
  enum {undef, k_i, k_f, k_d, k_c} kind;
  union{
    int i;
    float f;
    double d;
    char c[10];
  };
  FooS()            :kind{undef} {}
  FooS(int ii)      :kind{k_i},i{ii} {}
  FooS(float fi)    :kind{k_f},f{fi} {}
  FooS(double di)   :kind{k_d},d{di} {}
  FooS(const char* ci) :kind{k_c} {strncpy(c,ci,10);}
  int    get_i() {assert(kind==k_i); return i;}
  float  get_f() {assert(kind==k_f); return f;}
  double get_d() {assert(kind==k_d); return d;}
  char*  get_c() {assert(kind==k_c); return c;}
  FooS& operator=(int ii)     {kind=k_i; i = ii; return *this;}
  FooS& operator=(float fi)   {kind=k_f; f = fi; return *this;}
  FooS& operator=(double di)  {kind=k_d; d = di; return *this;}
  FooS& operator=(const char* ci){kind=k_c; strncpy(c,ci,10);
                                 return *this;}
};
```

# Bitwise operators

Bitwise and: `a & b`

Bitwise or: `a | b`

Bitwise xor: `a ^ b`

Bitwise complement: `~a`

shift left: `a << 5`

shift right: `a >> 5`

*>> on signed types is implementation defined*

Common operations:

**set 5th bit**

```
a = a | (1 << 4);
a |= (1 << 4);
```

**clear 5th bit**

```
a = a & ~(1 << 4);
a &= ~(1 << 4);
```

**toggle 5th bit**

```
a = a ^ (1 << 4);
a ^= (1 << 4);
```

# Bitwise operators
## Example:

Low-level operations: Bitwise operators
All variables are unsigned 16 bit integers

```
a = 77;           // a = 0000 0000 0100 1101
b = 22;           // b = 0000 0000 0001 0110
c = ~a;           // c = 1111 1111 1011 0010
d = a & b;        // d = 0000 0000 0000 0100
e = a | b;        // e = 0000 0000 0101 1111
f = a ^ b;        // f = 0000 0000 0101 1011
g = a << 3;       // g = 0000 0010 0110 1000
h = c >> 5;       // h = 0000 0111 1111 1101
i = a & 0x000f;   // i = 0000 0000 0000 1101
j = a | 0xf000;   // j = 1111 0000 0100 1101
k = a ^ (1 << 4); // k = 0000 0000 0101 1101
```

# Bit-fields

## Can be used to save memory

Specify explicit size in bits with `var : bit_width`

```
struct Car {// record in a car database
    static constexpr int reg_sz = 6;
    char reg_nr[reg_sz];  NB! not null-terminated.
    unsigned int model_year : 12;
    bool tax_paid : 1;
    bool inspected : 1;
};
```

**sizeof**(Car) = 8 *on my computer*

# Bit-fields
## Example

### Access of members

```
Car c;

strncpy(c.reg_nr, "ABC123", Car::reg_sz);
c.model_year = 2011;
c.tax_paid = true;
c.inspected = true;

cout << "Year: " << c.model_year << '\n';
if (c.tax_paid && c.inspected)
    cout << std::string(c.reg_nr, c.reg_nr+Car::reg_sz) << " is OK\n";
```

# Bit-fields
## Warnings

Bit-fields can be useful in special cases, but
they are *not portable*

- the layout of the object is *implementation defined*
- the compiler can add *padding*
- bit-field members *have no address*
  - cannot use the address-of operator &
- always specify **`signed`** or **`unsigned`**
- access can be slower than a "normal" struct
- integer variables and bitwise operations is an alternative

# std::bitset (<bitset>)

- efficient class for storing a set of bits
  - compact
  - fast
- has convenient functions
  - `test`, **`operator`**`[]`
  - `set`, `reset`, `flip`
  - any, `all`, none, count
  - conversion to/from `string`
  - I/O operators
- cf. `std::vector<`**`bool`**`>`
  - `std::bitset` has fixed size
  - a `std::vector` can grow
  - but does not quite behave like a normal `std::vector<T>`

bitset

# Example: store 50 flags in 8 bytes

```
void test_bitop(){
  bool status;
  cout << std::boolalpha;

  unsigned long quizA = 0;

  quizA |= 1UL << 27 ;
  status = quizA & (1UL << 27);
  cout << "student 27: " ;
  cout << status << endl;

  quizA &= ~(1UL << 27);
  status = quizA & (1UL << 27);
  cout << "student 27: ";
  cout << status << endl;
}
```

```
student 27: true
student 27: false
```

```
void test_bitset(){
  bool status;
  cout << std::boolalpha;

  std::bitset<50> quizB;

  quizB.set(27);
  status = quizB[27];
  cout << "student 27: ";
  cout << status << endl;

  quizB.reset(27);
  status = quizB[27];
  cout << "student 27: ";
  cout << status << endl;
}
```

```
student 27: true
student 27: false
```

# The comma operator
## (Introduction and warning)

The comma operator expression `expression1, expression2`

- First evaluates `expression1`, then `expression2`
- the expression has the value of `expression2`
- NB! The comma separating function parameters or arguments is *not* the comma operator
- Examples:

```
string s;
while(cin >> s, s.length() > 5) {
  //do something
}
```

*This ignores the value of* `cin >> s`
*use* `&&` *instead of* `,`

```
std::vector<int> v(10);

vector<int>::size_type cnt = v.size();
for(vector<int>::size_type i=0; i < v.size(); ++i, --cnt) {
   v[i]=cnt;
}
```

`v now contains 10 9 8 7 6 5 4 3 2 1`

*Do not use the comma operator!*

# Template metaprogramming

- Write code that is executed *by the compiler, at compile-time*
- Common in the standard library
  - As optimization: move computations from run-time to compile-time
  - As utilities: e.g., `type_traits`, `iterator_traits`
- Metafunction: a class template containing the result
- Standard library conventions:
  - Type results: type member named `type`
  - Value results: value member named `value`

# Template metaprogramming
## Example of compile-time computation

```
template <int N>
struct Factorial{
    static constexpr int value = N * Factorial<N-1>::value;
};

template <>
struct Factorial<0>{
    static constexpr int value = 1;
};

void example()
{
    Show<int, Factorial<5>::value>{};
}
```

Result of the *meta-function call* as a compiler error:

```
error: invalid use of incomplete type 'struct Show<int, 120>'
     Show< int, Factorial<5>::value >{};
```

# Template metaprogramming
## Example of templates for getting values as compiler errors

- Trick: use a template that doesn't compile to get information about the template parameters through a compiler error.
- Can be useful for debugging templates.
- To get the type parameter `T`:

```
template <typename T>
struct ShowType;
```

- To get a value (`N`) of type `T`:

```
template <typename T, T N>
struct Show;
```

## less than for a Person class

```
#include <tuple>

struct Person
{
    std::string fname;
    std::string lname;

    /* Order Persons by lname, fname */
    bool operator<(const Person& p) const {
        return std::tie(lname, fname) < std::tie(p.lname, p.fname);
    }
};
```

*Tuple has an* **operator<**.

# std::tuple
## get of type

```
auto t = std::make_tuple<17, 42.1, "Hello">;

auto i = std::get<int>(t);
auto d = std::get<double>(t);
```

# Most vexing parse
## Example 1

```
struct Foo {
    int x;
};

int main()
{
#ifdef ERROR1
    Foo f(); // function declaration
#else
    Foo f{}; // Variable declaration C++11
    // Foo f; //C++98 (but not initialized)
#endif
    cout << f.x << endl; // Error

    Foo g = Foo(); // OK     // C++11: auto g = Foo();
    cout << g.x << endl;
```

```
error: request for member 'x' in 'f', which is of
non-class type 'Foo()'
```

# Most vexing parse
## Example 2

```
struct Foo {
    int x;
};

struct Bar {
    int x;
    Bar(Foo f) :x{f.x} {}
};

int main()
{
#ifdef ERROR2
    Bar b(Foo()); // function declaration
#else
    Bar b{Foo()}; // Variable declaration (C++11)
    // Bar b((Foo()));  // C++98 : extra parentheses --> expression
#endif
    cout << b.x << endl; // Error!
```

```
error: request for member 'x' in 'b', which is of
non-class type 'Bar(Foo (*)())'
```

# Most vexing parse
## Example: actual function

```
struct Foo {
    Foo(int i=0) :x{i} {}
    int x;
};

struct Bar {
    int x;
    Bar(Foo f) :x{f.x} {}
};

Bar b(Foo()); // forward declaration

Foo make_foo()
{
   return Foo(17);
}

Bar b(Foo(*f)())
{
    return Bar(f());
}
```

```
void test()
{
    Bar tmp = b(make_foo);
    cout <<  tmp.x << endl;
}
```

17

# C-strings – library functions

## functions in <cstring>

```
strcpy(dest,src)      // Copies src to dest
strncpy(dest,src,n)   // Copies at most n chars
                      NB! dest is not null-terminated when truncating

strcat(s,t)           // Appends a copy of t to the end of s
strncat(s,t,n)        // Appends at most n chars

strlen(s)             // Gives the length of s
strlen(s,n)           // Gives the length of s, max n chars

strcmp(s,t)           // Compare s and t
strncmp(s,t,n)        // ... at most n chars
// s<t, s==t, s>t returns <0, =0, >0 respectively
```

(even more) unsafe, avoid when possible!

# String input
## Example mistake

### The read string does not fit in `x`

The statements

```
char z[] {"zzzz"};
char y[] {"yyyy"};
char x[5];

stringstream sin{"aaaaaaaaaaaaaaaaa bbbbbb"};
sin >> x;

cout << x << " : " << y << " : " << z << endl;
```

Give the output (on my computer):

```
aaaaaaaaaaaaaaaaa : aa : zzzz
```

- C-strings don't do bounds checking
- the input to `x` has overwritten (part of) `y`
- `getline()` is safer

# Copying strings
## Failing example

The statements

```
char s[20];

strncpy(s, "abc", 4);
cout << s << endl;

strncpy(s, "kalle anka", 20);
cout << s << endl;

strncpy(s, "def", 3);
cout << s << endl;
```

produce the output

```
abc
kalle anka
defle anka
```

The statements

```
int data[] {558646598, 65, 66};
char x[16];
char t[30] {"test"};

strncpy(x,"abcdefghijklmnop",16);
strcpy(t,x);
cout << t << endl;
```

produce the output

```
abcdefghijklmnopFEL!A
```

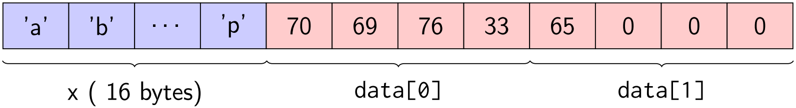
**Note**

- `strncpy` does not terminate the string with a `\0` when truncating.
- `strcpy` copies until it finds a `\0` in `src`.

# Copying strings
## Failing example: explanation

```
int data[]{558646598, 65, 66};
char x[16];
```

- the bytes of `data` is interpreted as **char**.
- representaton in memory




Hexadecimal representation:

$$558646598_{10} = 214c4546_{16}$$
$$65_{10} = 41_{16}$$

Byte order: *little-endian*

| hex | ASCII) | dec |
|---|---|---|
| 46 | F | 70 |
| 45 | E | 69 |
| 4c | L | 76 |
| 21 | ! | 33 |
| 41 | A | 65 |
| 0 | \0 | 0 |
| 0 | \0 | 0 |
| 0 | \0 | 0 |
| ... | ... | ... |

# Suggested reading

References to sections in Lippman

| | |
|---|---|
| C-style strings | 3.5.4–3.5.5 |
| Bitwise operations | 4.8 |
| The comma operator | 4.10 |
| Union | 19.6 |
| Bit-fields | 19.8.1 |