EDAF50 – C++ Programming


Sven Gestegård Robertz
Computer Science, LTH

2021
Outline

1. Bit operations
   - bit-fields
   - `<bitset>`
2. Multi-dimensional arrays
3. C-style strings
   - The C standard library string functions
   - C-strings – details and warnings
4. Types
   - Integer types
5. Syntax
   - Most vexing parse
   - Parentheses
   - The comma operator
6. Object slicing
Bitwise operators

Bitwise and: \( a \& b \)

Bitwise or: \( a | b \)

Bitwise xor: \( a ^ b \)

Bitwise complement: \( \sim a \)

shift left: \( a \ll 5 \)

shift right: \( a \gg 5 \)

>> on signed types is implementation defined

Common operations:

Set 5th bit
\[
\begin{align*}
a &= a | (1 \ll 4); \\
a &= (1 \ll 4);
\end{align*}
\]

Clear 5th bit
\[
\begin{align*}
a &= a \& \sim(1 \ll 4); \\
a &= \sim(1 \ll 4);
\end{align*}
\]

Toggle 5th bit
\[
\begin{align*}
a &= a \^ (1 \ll 4); \\
a &= (1 \ll 4);
\end{align*}
\]
Bitwise operators

Example:

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

```c
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`

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

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

    unsigned long quizA = 0;
    quizA |= 1UL << 27;
    status = quizA & (1 UL << 27);
    cout << "student 27: " ;
    cout << status << endl ;
    quizA &= ~(1 UL << 27);
    status = quizA & (1 UL << 27);
    cout << "student 27: ";
    cout << status << endl ;
}
```

```cpp
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 ;
}
```
multidimensional arrays

▶ Does not (really) exist in C++
  ▶ are arrays of arrays
  ▶ Look like in Java

▶ Java: array of references to arrays

▶ C++: two alternatives
  ▶ Array of arrays
  ▶ Array of pointers (to the first element of an array)
Multi-dimensional arrays

Initializing a matrix with an initializer list:

3 rows, 4 columns

```c
int a[3][4] = {
    {0, 1, 2, 3} ,  /* initializer list for row 0 */
    {4, 5, 6, 7} ,  /* initializer list for row 1 */
    {8, 9, 10, 11} /* initializer list for row 2 */
};
```

Instead of nested lists one can write the initialization as a single list:

```c
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- Multi-dimensional arrays are stored like an array in memory.
- The dimension *closest to the name* is the size of the array
- The remaining dimensions belong to the element type
An array $T$ array[3] is represented in memory by a sequence of three elements of type $T$:  

| $T$ | $T$ | $T$ |

An int[4] is represented as 

| int | int | int | int |

Thus, int[3][4] is represented as three int[4] objects: 

| int | int | int | int | int | int | int | int | int | int | int | int |
Multi-dimensional arrays

Examples

```c
int m[2][3]; // A 2x3-matrix

m[1][0] = 5;

int* e = m; // Error! cannot convert 'int [2][3]' to 'int*
int* p = &m[0][0];
*p = 2;

int* q = m[1]; // OK: int[3] decays to int*
q[2] = 7;
```

![Diagram of multi-dimensional array and pointers]
Multi-dimensional arrays

Parameters of type multi-dimensional arrays

// One way of declaring the parameter
void printmatr(int (*a)[4], int n);

// Another option
void printmatr(int a[][4], int n) {
    for (int i=0; i<n; ++i) {
        for (const auto& x : a[i]) { The elements of a are int[4]
            cout << x << " ";
        }
        cout << endl;
    }
}
Multi-dimensional arrays

Initialization and function call

```c
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
int b[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

printmatr(a,3);
cout << "------------------" << endl;
printmatr(b,3);
```

```
1  2  3  4
5  6  7  8
9 10 11 12
------------------
1  2  3  4
5  6  7  8
9 10 11 12
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcpy(dest, src)</code></td>
<td>Copies <code>src</code> to <code>dest</code></td>
</tr>
<tr>
<td><code>strncpy(dest, src, n)</code></td>
<td>Copies at most <code>n</code> chars</td>
</tr>
<tr>
<td></td>
<td>NB! <code>dest</code> is not null-terminated when truncating</td>
</tr>
<tr>
<td><code>strcat(s, t)</code></td>
<td>Appends a copy of <code>t</code> to the end of <code>s</code></td>
</tr>
<tr>
<td><code>strncat(s, t, n)</code></td>
<td>Appends at most <code>n</code> chars</td>
</tr>
<tr>
<td><code>strlen(s)</code></td>
<td>Gives the length of <code>s</code></td>
</tr>
<tr>
<td><code>strlen(s, n)</code></td>
<td>Gives the length of <code>s</code>, max <code>n</code> chars</td>
</tr>
<tr>
<td><code>strcmp(s, t)</code></td>
<td>Compare <code>s</code> and <code>t</code></td>
</tr>
<tr>
<td><code>strncmp(s, t, n)</code></td>
<td>... at most <code>n</code> chars</td>
</tr>
<tr>
<td></td>
<td>// <code>s &lt; t</code>, <code>s == t</code>, <code>s &gt; t</code> returns <code>&lt;0</code>, <code>=0</code>, <code>&gt;0</code> respectively</td>
</tr>
</tbody>
</table>

(even more) unsafe, avoid when possible!
String input
Example mistake

The read string does not fit in \texttt{x}

The statements

\begin{verbatim}
char z[] {"zzzz"};
char y[] {"yyyy"};
char x[5];

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

cout << x << " : " << y << " : " << z << endl;
\end{verbatim}

Give the output (on my computer):

\begin{verbatim}
aaaaaaaaaaaaaaaaaaaaaa : aa : zzzz
\end{verbatim}

- C-strings don’t do bounds checking
- the input to \texttt{x} has overwritten (part of) \texttt{y}
- getline() is safer
The statements

```c
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

```c
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

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

- the bytes of `data` is interpreted as `char`.

- representation in memory

<table>
<thead>
<tr>
<th>'a'</th>
<th>'b'</th>
<th>...</th>
<th>'p'</th>
<th>70</th>
<th>69</th>
<th>76</th>
<th>33</th>
<th>65</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>(16 bytes)</td>
<td>data[0]</td>
<td>data[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hexadecimal representation:

\[558646598_{10} = 214c4546_{16}\]

\[65_{10} = 41_{16}\]

Byte order: *little-endian*
### Integer types

#### Signed integers

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range (at least)</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>8 bits</td>
<td>$[-127, 127]^*$</td>
</tr>
<tr>
<td>short</td>
<td>at least 16 bits</td>
<td>$[-2^{15} + 1, 2^{15} - 1]$</td>
</tr>
<tr>
<td>int</td>
<td>at least 16 bits, usually 32</td>
<td>$[-2^{15} + 1, 2^{15} - 1]$</td>
</tr>
<tr>
<td>long</td>
<td>at least 32 bits</td>
<td>$[-2^{31} + 1, 2^{31} - 1]$</td>
</tr>
<tr>
<td>long long</td>
<td>at least 64 bits</td>
<td>$[-2^{63} + 1, 2^{63} - 1]$</td>
</tr>
</tbody>
</table>

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

overflow of a **signed** integer is *undefined*
#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>`

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CHAR_BIT</code></td>
<td>Number of bits in a <code>char</code> object (byte) (&gt;=8)</td>
</tr>
<tr>
<td><code>SCHAR_MIN</code></td>
<td>Minimum value for an object of type <code>signed char</code></td>
</tr>
<tr>
<td><code>SCHAR_MAX</code></td>
<td>Maximum value for an object of type <code>signed char</code></td>
</tr>
<tr>
<td><code>UCHAR_MAX</code></td>
<td>Maximum value for an object of type <code>unsigned char</code></td>
</tr>
<tr>
<td><code>CHAR_MIN</code></td>
<td>Minimum value for an object of type <code>char</code></td>
</tr>
<tr>
<td></td>
<td>(either <code>SCHAR_MIN</code> or 0)</td>
</tr>
<tr>
<td><code>CHAR_MAX</code></td>
<td>Maximum value for an object of type <code>char</code></td>
</tr>
<tr>
<td></td>
<td>(either <code>SCHAR_MAX</code> or <code>UCHAR_MAX</code>)</td>
</tr>
<tr>
<td><code>SHRT_MIN</code></td>
<td>Minimum value for an object of type <code>short int</code></td>
</tr>
<tr>
<td><code>SHRT_MAX</code></td>
<td>Maximum value for an object of type <code>short int</code></td>
</tr>
<tr>
<td><code>USHRT_MAX</code></td>
<td>Maximum value for an object of type <code>unsigned short int</code></td>
</tr>
<tr>
<td><code>INT_MIN</code></td>
<td>Minimum value for an object of type <code>int</code></td>
</tr>
<tr>
<td><code>INT_MAX</code></td>
<td>Maximum value for an object of type <code>int</code></td>
</tr>
<tr>
<td><code>UINT_MAX</code></td>
<td>Maximum value for an object of type <code>unsigned int</code></td>
</tr>
<tr>
<td><code>LONG_MIN</code></td>
<td>Minimum value for an object of type <code>long int</code></td>
</tr>
<tr>
<td><code>LONG_MAX</code></td>
<td>Maximum value for an object of type <code>long int</code></td>
</tr>
<tr>
<td><code>ULONGLONG_MAX</code></td>
<td>Maximum value for an object of type <code>unsigned long long</code></td>
</tr>
<tr>
<td><code>LLONG_MIN</code></td>
<td>Minimum value for an object of type <code>long long int</code></td>
</tr>
<tr>
<td><code>LLONG_MAX</code></td>
<td>Maximum value for an object of type <code>long long int</code></td>
</tr>
<tr>
<td><code>ULLONG_MAX</code></td>
<td>Maximum value for an object of type <code>unsigned long long</code></td>
</tr>
</tbody>
</table>
#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
Typedefs for specific sizes are in `<cstdint>` (<stdint.h>)

- integer types with exact width:
  ```
  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
Most vexing parse
Example 1

```c
struct Foo {
    int x;
};

text 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()'
```
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 (*)())'
```cpp
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
```
Parentheses matter in declarations of pointers to arrays and functions

- **int *a[10]** declares `a` as an array of `int*`
- **int (*a)[10]** declares `a` as a pointer to `int[10]`
- **int (*f)(int)** declares `f` as a pointer to function `int → int`

**BUT** may be used anywhere

```c
struct Foo;

Foo test;
Foo(f); // Foo f;

int x;
int(y); // int y;
int(z){17}; // int z{17};
int(q){}; // int q{};
```
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:

```cpp
string s;
while (cin >> s, s.length() > 5) { // better: use &&
   // do something
}

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!
Example
A class hierarchy

```cpp
struct Foo{
    virtual void print() const { cout << "Foo" << endl; }
};

struct Bar : Foo{
    void print() const override { cout << "Bar" << endl; }
};

struct Qux : Bar{
    void print() const override { cout << "Qux" << endl; }
};
```
Polymorph class example, *object slicing*

What is printed?

```cpp
void print1(const Foo* f) {
    f->print();
}
void print2(const Foo& f) {
    f.print();
}
void print3(Foo f) {
    f.print();
}

void test() {
    Foo* a = new Bar;
    Bar& b = *new Qux;
    Bar  c = *new Qux;

    print1(a);        Bar
    print1(&b);       Qux
    print1(&c);       Bar

    print2(*a);       Bar
    print2(b);        Qux
    print2(c);        Bar

    print3(*a);       Foo
    print3(b);        Foo
    print3(c);        Foo
}
```
References to sections in Lippman

C-style strings 3.5.4
Multi-dimensional arrays 3.6
Bitwise operations 4.8
The comma operator 4.10
Bit-fields 19.8.1