
Sven Gestegård Robertz

Computer Science, LTH

2020
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**: `~a`

*Common operations:*

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example Code</th>
</tr>
</thead>
</table>
| **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);`  |

Shift operations:

- **Shift left**: `a << 5`
- **Shift right**: `a >> 5`

> **>> on signed types is implementation defined**
Low-level operations: Bitwise operators
All variables are unsigned 16 bit integers

```plaintext
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
    char reg_nr[6]; /* NB! not null-terminated. */
    unsigned int model_year : 12;
    unsigned int tax_paid : 1;
    unsigned int inspected : 1;
};

sizeof(Car) = 8 on my computer
```
Access of members

```cpp
Car c;

strncpy(c.reg_nr, "ABC123", 6);
c.model_year = 2011;
c.tax_paid = true;
c.inspected = true;

cout << "Year: " << c.model_year << endl;
if (c.tax_paid && c.inspected)
    cout << std::string(c.reg_nr, c.reg_nr+6) << " 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*
  - use *unsigned* for members of size 1

- 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, \texttt{operator[]} 
  - set, reset, flip
  - any, all, none, count
  - conversion to/from string
  - I/O operators
- cf. std::vector<\texttt{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 & (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
```

```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;
}

student 27: true
student 27: false
```
multi-dimensional 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 | 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;
```
// 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

```cpp
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
```
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!
The read string does not fit in \( x \)

The statements

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

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

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

Give the output (on my computer):

```
aaaaaaaaaaaaaaaaaaaaa : aa : zzzz
```

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

```cpp
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

```cpp
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

```
abcdefghijklmnop
FEL!A
```

Note

- `strncpy` does not terminate the string with a `\0` when truncating.
- `strcpy` copies until it finds a `\0` in `src`. 
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>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>69</td>
<td>76</td>
<td>33</td>
</tr>
<tr>
<td>65</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

x (16 bytes)
data[0]data[1]

Hexadecimal representation:

558646598_{10} = 214c4546_{16}

65_{10} = 41_{16}

Byte order: *little-endian*
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 \( \text{the value modulo } 2^n \)

overflow of a **signed** integer is **undefined**
# Integer types


types:

- **sizeof**

```cpp
#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>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td>Number of bits in a <code>char</code> object (byte) (&gt;=8)</td>
</tr>
<tr>
<td>SCHAR_MIN</td>
<td>Minimum value for an object of type <code>signed char</code></td>
</tr>
<tr>
<td>SCHAR_MAX</td>
<td>Maximum value for an object of type <code>signed char</code></td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>Maximum value for an object of type <code>unsigned char</code></td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>Minimum value for an object of type <code>char</code> (either SCHAR_MIN or 0)</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>Maximum value for an object of type <code>char</code> (either SCHAR_MAX or UCHAR_MAX)</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>Minimum value for an object of type <code>short int</code></td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>Maximum value for an object of type <code>short int</code></td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>Maximum value for an object of type <code>unsigned short int</code></td>
</tr>
<tr>
<td>INT_MIN</td>
<td>Minimum value for an object of type <code>int</code></td>
</tr>
<tr>
<td>INT_MAX</td>
<td>Maximum value for an object of type <code>int</code></td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>Maximum value for an object of type <code>unsigned int</code></td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>Minimum value for an object of type <code>long int</code></td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>Maximum value for an object of type <code>long int</code></td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>Maximum value for an object of type <code>unsigned long int</code></td>
</tr>
<tr>
<td>ULLONG_MIN</td>
<td>Minimum value for an object of type <code>unsigned long long int</code></td>
</tr>
<tr>
<td>ULLONG_MAX</td>
<td>Maximum value for an object of type <code>unsigned long long int</code></td>
</tr>
</tbody>
</table>
# Integer types

Sizes are specified in `<climits>`

```cpp
#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 << LONGLONG_MIN << " , " << LONGLONG_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 widths:
  - `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`
```cpp
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;
}
```

 ErrorResponse: 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;
}
```

Syntax: Most vexing parse

1. Low-level details. Loose ends.
Declarations and parentheses

- 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:

```c++
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; }
};
```
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
    std::cout << std::endl;
    print2(*a); Bar
    print2(b); Qux
    print2(c); Bar
    std::cout << std::endl;
    print3(*a); Foo
    print3(b); Foo
    print3(c); Foo
    std::cout << std::endl;
}
```
Suggested reading

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