Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 3

- Basic building blocks of C/C++ programs
- Number systems
- Simple types

Part 2: Procedural Programming

- Common C/C++ language features
- Standard C libraries
- Memory allocation
- Abstract data types
- Tools: compiler, makefiles, debugger

C vs. C++

Similar syntax and semantics

C can be considered a C++ subset (well, almost)
That is: most C programs are valid C++ programs

Major C++ additions: classes, inheritance, operator overloading, templates

```
// this is a C program
#include <stdio.h>


int main()
{
  printf("hello world\n");
  return 0;
}
```

```
// this is a C++ program
#include <iostream>
using namespace std;

int main()
{
  cout << "hello world" << endl;
  return 0;
}
```


Basic Building Blocks

```
// this is a comment
#include <iostream> // preprocessor command

int foo(int x)      // function declaration
{                   // block
  return x+1;       // return expression value
}

int main()          // this is where all C/C++ programs start
{
  int i = 0;                 // variable declaration + init.
  while (i < 10) {           // loop
    i = i+1;                 // expression + assignment
    std::cout << foo(i) << " ";  // operators + function call
  }
  if (i >= 10) i = 1;        // condition
  else         i = 0;
  return i;                  // return result, exit
}
```

Identifiers

Used for variable, function, member, and label names

- Identifiers are case-sensitive
- Start with _ or letter
- Remaining part all letters, digits, or _s
- Exceptions are C/C++ keywords such as
  `if else static for do while ...`

Valid identifiers:

`sumOfValues x0 FooBar foobar _x_y_z`

Invalid identifiers:

`0x $y .name while @abc foo# ^_^ ;-)`

Comments

It is important to comment your code – for others and yourself! C/C++ comments:

- `// this is a single line comment`
- ```
  /* this is a
     multi-
     line
     comment
  */
  ```
- Multi-line comments cannot be nested; not allowed:
  `/* /* */ */`

Where to put comments?

Good comments are very important. Put them

- at the beginning of files describing their purpose,
- on top of function definitions discussing parameters, function effects, and return values,
- on top of class definitions describing their purpose,
- in front of non-trivial parts, meaning anything you wouldn't instantly understand when looking at the code a month later

No need to write novels or to comment each program statement

Digression: Number Systems

- Common base: 10 (decimal system)
- Other bases: 1 (unary), 2 (binary), 3 (ternary), 8 (octal), 16 (hexadecimal) ...
- Binary number system: digits 0 and 1 (binary digit = "bit")
- Used in today's computers (low voltage, high voltage)
- Byte = sequence of 8 bits (contents of a memory cell)
- Integers are represented as sequence of bits
  - One byte stores one of $2^8 = 256$ different values
  - $00000000_2 = 0_{10}$ $00000001_2 = 1_{10}$
  - $00000010_2 = 2_{10}$ $00000011_2 = 3_{10}$ ...
  - $11111110_2 = 254_{10}$ $11111111_2 = 255_{10}$
  ```

Binary Arithmetic

- Instead of digits 0..9, we now only have 0,1
- Arithmetic is done analogously. E.g. $1_2 + 1_2 = 10_2$, $10_2 + 1_2 = 11_2$, $11_2 + 1_2 = 100_2$, ...

```
 1010111     87
+   1001    + 9
---1111--   -1--
 1100000     96
```

- Weight for each bit is power of 2, rather than power of 10

Integer Representation

k-bit unsigned number: range $0 \ldots 2^k - 1$

k-bit signed number: range $-2^{k-1} \ldots 2^{k-1} - 1$

Example: $k = 16$

	unsigned	signed
$0000\ 0000\ 0000\ 0000_2 =$	0_{10}	0_{10}
...		
$0111\ 1111\ 1111\ 1111_2 =$	32767_{10}	32767_{10}
$1000\ 0000\ 0000\ 0000_2 =$	32768_{10}	-32768_{10}
...		
$1111\ 1111\ 1111\ 1111_2 =$	65535_{10}	-1_{10}

Octal and Hexadecimal Numbers

Main purpose: short description of long bit sequences

Octal: base 8, digits $0 \ldots 7$

- $123_8 = 1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 83_{10}$
- converting to/from binary numbers easy: each octal digit represents group of 3 bits
- $100\ 110\ 001_2 = 461_8$

Hexadecimal: base 16, digits $0 \ldots 9$, $a \ldots f$ ("nibble")

- $3af_{16} = 3 \cdot 16^2 + 10 \cdot 16^1 + 15 \cdot 16^0 = 809_{10}$
- converting to/from binary numbers easy: each nibble represents group of 4 bits
- $1001\ 1101\ 0111_2 = 9d7_{16}$

Variable Declarations (1)

Imperative Programming : "How to create something"

- e.g. C++, Ada, Pascal, Fortran
- Program: change of program state (variables)

[as opposed to Declarative Programming : "What something is like" e.g. Prolog, Lisp, Haskell]

```
int lower, upper, step;
char c;          // all values undefined!
float f = 0;     // initialized with 0
int i = c + 1; // undefined! g++ complains?
const float PI = 3.1415926535;
PI = 0;          // compiler complains!(const)
```

Variable Declarations (2)

- In C/C++ variables need to be declared prior to usage
- Declarations define the type of data to be stored in a variable
- Syntax:

  <var-decl> ::= <type> <var>,<var>,.. or

  <var-decl> ::= <type> <var>=<exp>,..

  with <type> ::= <ident> and <var> ::= <ident>
- **Value of uninitialized variable is undefined!**
- **const-qualifier** makes variables read-only

Simple Types (1)

Integer Types

- finite range of integral numbers in $\mathbb{Z} = \{0, \pm1, \pm2, ...\}$
- multi-purpose: memory = sequence of integers

Floating-Point Types

- finite subset of rational numbers in $\mathbb{Q} = \{p/q \mid p, q \in \mathbb{Z}, q > 0\}$
- can express very small to very big numbers
- suitable for scientific computations
- inexact! rounding errors!
- fundamental algebraic laws no longer valid! e.g. $a + (b + c) \neq (a + b) + c$ for suitable a, b, c

Simple Types (2)

- **bool**: false, true; 1 byte (8 bits)
- **char**: ASCII character; 1 byte integer
- **short**: -32,768..+32,767; 2 byte integer (16 bits)
- **int**: -2,147,483,648..+2,147,483,647; 4 byte integer (32 bits)
- **float**: $\approx -10^{38}.. - 10^{-38}, 0, +10^{-38}.. + 10^{38}$

  4 byte floating-point value (7 digits)
- **double**: $\approx -10^{308}.. - 10^{-308}, 0, +10^{-308}.. + 10^{308}$

  8 byte floating-point value (15 digits)
- **long double**: $\approx -10^{4932}.. - 10^{-4932}, 0, +10^{-4932}.. + 10^{4932}$

  12 byte floating-point value (19 digits)

Examples

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

int main() {
  bool flag = false;      // 1 byte Boolean variable
  int numOfBeans = 0;     // 4 byte signed int. var.
  unsigned short bits16 = 0; // 2 byte unsigned int.
  float PI = 3.14159265;     // 4 byte fp variable

  cout << "flag="       << flag       << " "
       << "numOfBeans=" << numOfBeans << " "
       << "bits16="     << bits16     << " "
       << "PI="         << PI         << endl;
}

Output: flag=0 numOfBeans=0 bits16=0 PI=3.14159
```

Integer Type Qualifiers: signed, unsigned

Specifies whether a variable is signed or unsigned

No qualifier → signed

• signed char:	-128..127	1 byte
• unsigned char:	0..255	1 byte
• short:	-32768..32767	2 bytes
• unsigned short:	0..65535	2 bytes
• unsigned int:	$0..2^{32}-1$	4 bytes

No overflows in C++ arithmetic. Int +/- wraps around!

```
unsigned char foo = 255;
unsigned char bar = foo+1;
// value of bar is 0 !
```


Value Range of Variables

- **IMPORTANT**: make sure variables have the right type to avoid underflows and overflows
- In C/C++, integer expressions are **not checked** for overflows/underflows!
- Floating-point overflows/underflows are indicated by special values (+Inf, −Inf, NaN)