Basic Types in C++ (and strings)

- not in all details
  - just the differences to Java

- Integer values
  - counting, indexing, real world integer values, models

- Real numbers
  - floating point (not continuous !!, limits in precision)
  - physical values

- Characters
  - human readable decodification of data
  - (strings)

- Boolean
  - conditions
basic (built-in) types

- main difference: missing standards (size, sequence of bytes)
  - only ANSI C++ definition
    - `char <= short <= int <= long`
    - `sizeof(<name>)` operator

- symbolic constants for implementation dependent values
  - `<climits>`
  - `<cfloat>`
    - `INT_MIN, INT_MAX, SHRT_MIN; ...`

- `bool` (compatible with integer):
  - `0 = false`, everything else (interpreted as 1) = `true`

- `char` (ASCII), `wchar_t` (wide character for unicode)

- `int, short [int], long [int]`

- each (arithmetic, char) type can be `signed` or `unsigned`
**implicit conversions**

- **boolean** (C++ only)

```
byte -> short -> int -> long -> float -> double
```

- **char**

- **[long]**

- **String**

- **Java only**

- **bool**

- **byte** has no implicit conversion to String (for output reasons)

- **<< operator** is heavily overloaded to compensate for this, but must also be defined for user defined types (all your classes!)

- **explicit type casts** like Java:
  - `(int) 3.14159 == 3`
String literals

- Strings are no built in type, but have literal representations
  "this is a string constant"

- Strings are equivalent to arrays of char (ending with '\0')

- Arrays of char are equivalent to pointer (later topic)

C++ also has class string (comparable to Java String)
#include <string>

Create:

```cpp
string s = "copy of this String literal.",
t(s),
u("literal"),
v(=, 40);
```

Without '0' at the end!!

Copy constructor

constructor( char[ ] )

constructor( char c, int anz )

"========================================"

Manipulate

- Concat – Ops: + and s += " appended" (like java)
- Compare: s1 == s2 (Java: s1.equals( s2 ) )
- Find substring: int pos = s.find( "this" ); if ( pos != string::npos ) ... 
- Replace: s.replace( pos, 4, "another" );
#include <iostream> .. <string> .. <ctime>
using namespace std;
int main()
{
    time_t sek;
    time( &sek ); // get date & time (long)
    string t = ctime( &sek ); // convert to string
    string dayOfWeek( t, 0, 3 ),
        month( t, 4, 3 ),
        day( t, 8, 2 ),
        year( t, t.size() - 5, 4 );
    cout << dayOfWeek + " " << day + ". " << month + ". " << year + "\n";
} // main()
declaration of objects

local variables, global variables, parameters, attributes

`const double pi = 3.141593;`

`volatile clock_t ticks;`

`register int counter;`

`double pow(double base, double exponent); // better prototype`

`int rand(void); // same as int rand();`

`string s("create string object");`

`point u, v(2, 3);`

**C++ pattern:** Use `const` as much as possible (arguments, …)
Java -> C++: basics

- C++ / Java basic differences & philosophy
- Separation of declaration & definition (*.h & *.cpp)
  - edit / compile / link / test cycle, makefiles
- IO
  - streams, file streams, manipulators
- namespaces
  - std for cout
- basic types

expressions & statements are practically the same in Java and C++

( but sequence of evaluation in expressions not standardized)

```cpp
int i = 2; cout << i-- * i++; // g++: 2, CC: 4
```
Pointer & References

- references of objects
  - call by reference

- pointers
  - usage patterns
  - pointers & arrays

- storage classes
  - local objects => auto (stack)
  - global objects => extern (heap)
  - objects "pointed to" & static objects

null list . at (3) = ...
Declaration of variables

Similar to Java:

variable declaration

```
short int  i = 256S;
int       j = 42, k;
unsigned long int  l = 123456uL;
float     x = 42.1;
double    y = 123.456d;
bool      b = true;
```
Reference: $T &$

Reference = alias name for an existing object

Syntax:

- Declaration: Typename& varname;
- "varname is reference to variable of Typename"

```cpp
float x = 42.1f;
...
float& rx = x;
rx /= 421.0;
cout << "x = " << x;
const double pi = 3.14159;
const double & rpi = pi;
rpi = 2.345;
```

Explicit reference semantics as Java (implicit) class types

01.04.15 oo programming with C++
Use case for references

- reference declarations are used in C++ to implement **call by reference** parameter passing
- normal declaration: **call by value**
  - value of argument (expression) is copied into local parameter

```cpp
const int STEP = 5;
void nextStep(int var) {
    var += STEP;
} // nextStep()

int main() { ...
    int a = 42;
    nextStep(a);
    cout << "a=" << a << endl;
} // main()
```

- pass the value (copy)
- a not changed!
Use case for references

Reference declarations are used in C++ to implement call by reference parameter passing: call by reference.

Reference to the argument is passed to the formal parameter.

```cpp
const int STEP = 5;
void nextStep( int& var ) {
    var += STEP;
} // nextStep( )

int main( ) { ...
    int a = 42;
    nextStep( a );
    cout << "a=" << a << endl;
} // main( )
```

Pass the reference:

Status: main

a changed!

Status: nextStep

Pass the reference

Status: main

47

...
Java has **reference semantics** for objects and **value semantics** for basic (built-in) types

- wrapper classes for basic types for full compatibility
- the only parameter concept: **call by value**
- … but the value can be a reference to an object

```java
public static void main (...) {
    Rectangle s = new Rectangle( 2.1, 1.5, 3 );
    f( s );
} // main()

public static void f ( Rectangle r ) {
    r.setLineStrength( 0.5 );
    r = new Rectangle ( 2.5, 2.0, 1 );
} // f()
```

`f()` can not change the object (the reference), but its state!
Java has **reference semantics** for objects and **value semantics** for basic (built-in) types

- the only parameter concept: **call by value**
- … but the value can be a reference to an object

```java
public static void main (...) {
    Rectangle s = new Rectangle( 2.1, 1.5, 3 );
    f( s );
} // main( )

public static void f ( Rectangle r ) {
    r.setLineStrength( 0.5 );
    r = new Rectangle ( 2.5, 2.0, 1 );
} // f( )
```
Call by **value vs. reference** (C++)

- size of arguments with large objects => better runtime

```c++
void display( someClass o, ... ) {
    int i = sizeof( o ); ...
    cout << s + i << endl;
} // display()

int main( ) {
    someClass obj( ... ); ...
    display( obj, ... );
} // main()
```

- **Call by value**:
  - Arguments are copied and placed on the stack.
  - Example:
    ```c++
    display( obj, ... );
    ```

- **Call by reference**:
  - Arguments are passed by reference and do not have to be copied.
  - Example:
    ```c++
    display( &obj, ... );
    ```

Temporary objects created when calling the function and deleted when it is finished.
Call by value vs. reference (C++)

- size of arguments with large objects => better runtime

```cpp
void display( someClass& o, ... ) {
    int i = sizeof( o ); ...
    cout << s + i << endl;
} // display( )

int main( ) {
    someClass obj( ... ); ...
    display( obj, ... );
} // main( )
```

References are passed as addresses to the storage for which the alias name is defined.
Call by value vs. reference (C++)

- size of arguments with large objects => better runtime
- side effects: returning more than one result
- !! controversial !!

```c++
void swap(int& x, int& y) {
    int help = x;
    x = y; y = help;
} // swap()

int main() {
    for (...) {
        if (a[i] > a[j])
            swap(a[i], a[j]);
    } // main()
```
References as return values

- if objects are returned => save runtime to pass the reference

```cpp
string& msg() {
    static string s = "this is the message";
    return s;
} // msg()

int main() {
    string x = "!!!";
    msg() += x;
    cout << msg();
} // main()
```

- stored in **global store**, stays valid until end of program

- **cannot return reference to a local object!**

- but: object is created only **once**!!

- most cases need newly created objects per function call!

- solution: **Pointer**
double& max(double &, double &);

int main( ) {
    double x = 1.7, y = 42.3;
    x += ++max( x, y );
    max( x, y ) += 5.0;
    cout << "x = " << x << " y = " << y << endl;
} // main( )

double& max( double & a, double & b ) {
    return a > b ? a : b;
} // max( )

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>max( x, y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7</td>
<td>42.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>++ ref to y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+= 43.3</td>
</tr>
<tr>
<td>45.0</td>
<td></td>
<td>ref to x</td>
</tr>
<tr>
<td>50.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x = 50, y = 43.3
many operators are functions returning references to objects

cout << "x = " << x << " , y = " << y << endl;

```cpp
ostream& operator << ( ostream& o, double x ) {
    o.put( x );
    return o;
}
```

// operator <<

Operator `<<` takes two arguments and returns the output stream it writes to.
Reference Semantics

Java:
- automatically for all objects (class types)

C++
- must be explicitly defined by the programmer
  - Type & name
- practical uses
  - call by reference function arguments (large objects)
  - out-parameters to return more than one value (swap)
  - reference to objects to work within a pipeline (cout)
Pointer: T *

an expression which evaluates to an address and a type
declaration: Typename* variablename;

<table>
<thead>
<tr>
<th>variable</th>
<th>content (type)</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>42 (int)</td>
<td>0x415004</td>
</tr>
<tr>
<td>ip</td>
<td>0x415004 (int*)</td>
<td>0x415008</td>
</tr>
<tr>
<td>jp</td>
<td>0 (int*)</td>
<td>0x41500c</td>
</tr>
</tbody>
</table>

Address-Operator &

default value: NULL

int i = 42,  
*ip = &i;

int* jp; // no allocation!
Pointers and references are somehow equivalent (but not the same!!!)

- References are constant addresses of existing objects
- These addresses can not be changed (by the programmer)

Pointers are variables of addresses of (possible) objects
- These variables can be changed (can point to different objects)

```c++
int i1 = 0, i2 = 100,
int *p1 = &i1, *p2 = &i2;
*p1 = 42;  // as i1=42
*p2 = *p1; // as i2=i1
p1 = p2;
*p1 = 12345;
int* p3;
```

```plaintext
p1  i1  42
p2  i2  12345
p3

p3 == NULL special value 0x00000000
```
**Allocation and Deletion**

```c
int* ip = new int(333);
...
delete ip;

ip = &otherInt;
```

allocate store & assign value

variable ip holds new address (points to another int)

---

*leakage!*

store stays allocated

can not be reached

no automatic garbage collection

---

```c
??? difference between int* ip and int *ip; ???
```
Operators

- Dereferencing operator * (=> object pointed by)
- Reference-operator & (address of the object)
- declaration of a reference T&
- declaration of a pointer T*

```
T* px;
T x;

&px
&x, px
*px, x
```

pointer to a pointer to an object of type T
values not to be changed: declared as `const`

which of the following are legal?

```cpp
int i = 0;
const int ci = -1;
i=1;
```

- Assignment of read-only variable `ci`

```cpp
int* p1;
p1 = &i;
*p1 = 11;
p1 = &ci;
*p1 = 22;
```

- Depending on compiler and compiler options:
  - Only warning: discards `const`

```cpp
cout << "ci=", *p1 << endl;
```

- `ci=-1, *p1 = 22`
const and pointer (2)

```c
int i = 0; const int ci = -1;

const int* p2;
p2 = &i;
*p2 = 111;  // assignment of read-only location
p2 = &ci;
*p2 = 222;  // assignment of read-only location

int* const p3;
int* const p4 = &i;
*p4 = 1111;  // assignment of read-only variable `p4'
p4 = &ci;
```

uninitialized const `int * const p3'

assignment of read-only location

assignment of read-only location

assignment of read-only variable `p4'
const and pointer (3)

```c
int i = 0; const int ci = -1;

const int* const p5; // uninitialized const
const int* const p6 = &i;
*p6 = 111111; // assignment of read-only location
p6 = &ci; // assignment of read-only variable `p6'
const int* const p7 = &ci;
*p7 = 2222222; // assignment of read-only location
p7 = &i; // assignment of read-only variable `p7'
```