# Class 4: C++ Part II

## Contents

## Contents



## Explicit type conversion ("casting")

Syntax in C or C++: ( *typename* ) *expr*

Alternate C++ syntax: *typename* ( *expr* )

Numeric types are also converted implicitly when needed, particularly when it's a "promotion":

| Implicit type conversion | Explicit type conversion |
|---|---|
| `int i=5;`<br>`double x;`<br>`x= i;` | `double x=3.1415;`<br>`int i;`<br>`i= (int)x;  // C style`<br>`i= int(x);  // C++ style` |

## Usefulness of arrays and references:

Suppose in our bouncing ball program we had more than one ball to track.

- We could use a vector or an array for `x` and another one for `v`.
- We might want to simplify the main loop by making a separate `MoveBall()` function. The main loop in the code might look like this:

```
while(1) {
  for (int i=0; i<n; i++) {
    MoveBall(x[i], v[i], dt);  // alters x, v (pass by ref.)
  }
  DrawAllBalls(x,v,n);  // you can pass an array or vector
}
```

## An algorithm for DrawAllBalls()

**Algorithm goal:** Print a line showing where balls are.

Algorithm outline:

- Loop over positions in line:
  - Figure out whether ' ' or '*' should be printed
    - Loop over balls
    - Set character to be printed to '*' if any ball is at current position
  - Print character
- Print newline.

## Many things demonstrated by DrawAllBalls()

Implementation:

```
// Print a line showing where balls are:
//  at each position, check whether a ball is there,
//  and print a '*' or ' ' depending
void DrawAllBalls( vector<double> & x, vector<double> & v, int n)
{
  int icolumn; // column index
  int iball;   // ball index
```

```
  char c;      // character
  for (icolumn=0; icolumn<80; icolumn++) {
    c= ' ';  // space in this column, unless we find a ball
    // start of ball-search
    for (iball=0; iball<n; iball++) {
      if ( (int)(x[iball]) == icolumn ) {
        c='*';  // found a ball
        break;  // early exit from ball-search loop
      }
    }
    // end of ball-search
    cout << c;  // print ' ' or '*'
  }
  cout << '\n';
}
```

## A more CPU-efficient algorithm

The previous algorithm had two nested loops: one over 80 columns, the other over n balls. The ball position check is done as many as 80*n times. Can we be more efficient? Yes, we can.

Algorithm outline:

- Create a string 80 characters long, initialized with ' '
- Loop over balls:
  - For each ball, set corresponding position in string to '*'
- Print string and newline.

Now have only nball loop iterations, not 80*nball.

## New DrawAllBalls() algorithm implementation

```
// Print a line showing where balls are
void DrawAllBalls(vector<double>& x, vector<double>& v, int n)
{
  string s;
  s.resize(80,' ');      // set to 80 spaces
  int iball;             // ball index
  for (iball=0; iball<n; iball++) {  // start of ball loop
      int ix= (int)(x[iball]);
      s[ix] = '*';
  }
  // end of ball loop
  cout << s << '\n';
}
```

## Moral of the story

Sometimes you can find a more efficient algorithm in terms of CPU by doing something a little different. Often this involves using just a little more memory.

## Structs: motivation

Suppose our bouncing balls had more parameters: two dimensions, radius, color, name of video game character, etc. Then the main loop might look like this:

```
while(1) {
  for (int i=0; i<n; i++) {
    MoveBall(x[i],y[i],u[i],v[i],r[i],color[i],name[i],dt);
  }
  DrawAllBalls(x,y,u,v,r,color,name);
}
```

It would be nice if it could look like this:

```
while(1) {
  for (int i=0; i<n; i++) {
    MoveBall( balls[i], dt );
  }
  DrawAllBalls( balls );
}
```

## Structs in C/C++

C and C++ let us define our own *structures* which collect data, possibly of different types, in contiguous memory locations.

| **Declaration Syntax** | **Memory allocation** | |
|---|---|---|
| `struct MyBall_s {` | `Relative address` | `contents` |
| `  double x;` | `0 through 7` | `x` |
| `  double y;` | `8 through 15` | `y` |
| `  double u;` | `16 through 23` | `u` |
| `  double v;` | `24 through 31` | `v` |
| `  double r;` | `32 through 39` | `r` |
| `  int color;` | `40 through 43` | `color` |
| `  char symbol;` | `44` | `symbol` |
| `};` | | |
| | Total size: 45 bytes | |
| `struct MyBall_s ball;` | | |

## Example usage

```
void MoveBall( struct MyBall_s & ball, double dt )
{
```

```
  ball.x= ball.u * dt;
  ball.y= ball.v * dt;
  Bounce(ball.x, ball.u);
  Bounce(ball.x, ball.v);
}

void Bounce( double &x, double &v ) {
{
  // bounce on wall at 79 or 0.
  // Warning: explicit 79 is bad practice!
  if ( (x>=79.0 && v>0) || (x<=0.0 && v<0) )
    v= -v;
}
```

## Exercise (after break)

Modify bouncing ball to have

- Multiple balls
- Two dimensions
- Different positions of wall in x and y
- Show on screen by reprinting repeatedly

Class: further define specification, write algorithm. Instructor: code it, with class catching errors.

## Some good programming practices

- Avoid arbitrary constants (like 79) in your code.
  - This is especially important if you use it more than once!
  - Better to define a variable for it.
- Document what your code does at multiple levels:
  - Big picture: problem solved or task performed, input, and output
  - Algorithm: how it does it, in language-independent terms
  - Code: the variables and critical steps (comments in code)
- Keep copies of your older versions of code, especially if they work!
  - *Revision control systems* can help with this, but for small projects, just make backup copies manually with some systematic naming or subdirectory structure. (`v1/ball.cc`, `v2/ball.cc`, etc.)

## Assignment

- Come up with further ideas for improving either "bouncing ball" or "adventure" (pick one).
- Figure out algorithm for doing them, implement them in code.
- Document the features. (E.g., in "ReleaseNote.txt")
- Document the algorithm. (E.g., in "DeveloperNotes.txt")
- Document the code internally with comments.
- Keep your old version for reference, make new version.