Basics of Programming in C^{++}

Hello World Program

```
#include <iostream>
   // Contains declarations of std::cout and std::in
int main( unsigned int nrpars, char* pars [] )
   // nrpars = the number of parameters with which
   // the program is called plus one.
   // pars [] are the parameters. pars [0] is the name
   // of the program.
{
   std::cout << "hello world\n";
      // std::cout is the name of the standard output
      // stream. In C++, names are structured. std
      // is the namespace, cout is the name.
   return 0;
      // Returning a 0 means normal return.
}
```

Comparison with *C*

C^{++} inherited a lot from *C*.

Interaction with the operating system is the same.

The program starts at a function called **main**, which should return **int**, and have parameters **unsigned int, char* pars** [ ] .

C^{++} inherited most of the pointer/array data type from *C*.

The Primitive Datatypes

C^{++} has the following primitive data types:

char : Holds a character.

int : Holds an integer.

unsigned int: Holds a natural number. Should be preferred over **int**.

bool: A boolean truth-value.

double: A floating point number.

There also exists **float**, but nobody seems to use it.

Ways of Addressing Data

In C^{++} there are three types of variables:

1. Direct variables (local variables, parameters)
2. References. A reference is a synomym for another variable.
3. Pointers. Direct representation of address in memory. Are needed for defining low level data structures.

Direct Variables

Local variables and parameters are the nicest way to store data in a program.

They have a functional semantics, and you should use them whenever possible.

Computing $N!$ with Functional Program

```
unsigned int fact( unsigned int n )
   // n is a parameter.
{
   if( n == 0 )
      return 1;
   else
      return n * fact( n - 1 );
}
```

Computing $N!$ with imperative program

```
unsigned int fact( unsigned int n )
   // n is a paramter.
{
   unsigned int res = 1;
      // res is a local variable.
   while( n > 0 )
   {
      res = res * n;
      -- n;
   }
}
```

Formalization

Imperative programs with direct variables can be expressed by equations:

$\mathbf{fact}(n) = \mathbf{fact1}(n, 1)$
if $n > 0$, then $\mathbf{fact1}(n, r) = \mathbf{fact2}(n, r.n)$
$\mathbf{fact2}(n, r) = \mathbf{fact1}(n - 1, r)$

References

A reference refers to another variable. A reference is initialized by specifying the other variable. They can be recognized by `&` .

```
int x;
int& y = x;

x = 4;
y = 3;
   // What is value of x?

const int& z = x;
   // assigning z = 3 is not possible, but z can
   // still be changed through x.
```

Dangers of References

```
void frameproblem( int& i1, int& i2 )
{
   std::cout << "please type two integers: ";
   std::cin >> i1;
   std::cin >> i2;

   std::cout << "you typed ";
   std::cout << i1 << ", " << i2 << "\n";
}
```

When to Use References

There are two reasons for using a reference:

1. References enable a procedure to have output parameters. Most of the time, it is possible to define a function, but not always.

   There may be more more than one output parameter, sometimes it is not natural to use a function.

2. Input parameters that are so big that copying is inefficient.

Safe Usage of References

Use **const** references wherever possible.

Inside a procedure, do not make assumptions about distinctness of references.

If the parameters have different types, you can assume they are disinct:

```
void noproblem1( double& x, unsigned int &i )
   // Safe to assume that x and i are distinct.

double force( double& m, double& a )
   // m is mass, a is acceleration.
   // This is almost the same as m,a having different
   // types. It is safe to assume they are distinct.
```

Danger of References

In the following example, one would like to avoid copying the matrix:

```
matrixmultiply( const matrix& m1, const matrix& m2,
                matrix& prod )
{
   for( unsigned int i = 0; i < 3; ++ i )
       for( unsigned int j = 0; j < 3; ++ j )
          prod [i] [j] = .... ;
}
```

This code will fail if the user writes

```
matrixmultiply( m, other_matrix, m );
```

Pointers and Arrays

Pointers and arrays are addresses of variables.

Arrays and pointers should not be used in normal programming, but they are necessary for definition of certain data structures.

- Needed for storing data structures that can be unlimited in size: **strings**, **vectors**, **bigints**.
- Needed for building complex data structures, like lists or trees.

Pointers and Arrays

```
unsigned int *p;
   // Declares, but does not initialize a pointer.
std::cout << *p << "\n";
   // If p points to something, then *p retrieves this
   // something.
p = p + 1;
   // If p is part of an array of integers, then this
   // makes p point to the next array of integers.
++ p;
   // Same as p = p + 1;
```

```
(*p) ++ ;
   // This increases the object that p points to by 1.
std::cout << * ( p + 2 );
   // If p is part of an array of integers, then the
   // value stored 2 positions behind p is printed.
std::cout << p[2] << "\n";
   // Alternative syntax for *(p+2).
```

Allocation and Deallocation

```
int* p = 0;
   // Null pointer. 0 is the value that is guaranteed
   // to point to nothing.

p = new int(4);
   // Creates new int on heap, initializes it with 4.
std::cout << "this is the number four:" << *p << "\n";
*p = 3;
std::cout << "this is the number three:" << *p << "\n";

delete p;
   // Returns the memory to the system.
```

```
// An example of a situation where the size is not
// known in advance:

std::cout << "How many numbers do you need? ";
std::cin >> n;

int* p = new int[n];

for( unsigned int i = 0; i < n; ++ n )
{
   std::cout << "please type " << n << "-th number";
   std::cin >> p[i];
}
```

```
std::cout << "here are the numbers that you typed: ";
for( unsigned int i = 0; i < n; ++ i )
{
   std::cout << i << ": " << p[i] << "\n";
       // p[i] is an abbreviation for *(p+i).
}

delete p;
   // Only pointers that are constructed by
   // new can be deleted.
```

Dangers of Pointers (1)

The frame problem (as with references).

Pointers can be unitialized, point to a position outside of the program's memory, or to nonsense.

Writing to an ill-defined pointer has unpredictable effect.

Deleting data belonging to a pointer that was not constructed by **new** has unpredictable effect. Deleting same pointer twice has unpredictable effect.

Forgetting to delete data causes memory leak.

Dangers of Pointers (2)

In my view, the frame problem with pointers is less dangerous than the frame problem with references:

- Pointers do not look like usual variables.
- Pointers are not used for parameter passing.
- Pointers are used more rarely, only in low level data structures.

Detecting Memory Leaks

Most memory leaks can be detected using **top** command.

Write a procedure of the following form:

```
for( unsigned int k = 0; k < 1000000; ++ k )
{
   suspected_code( );
}
```

Safe Usage of Pointers

You should use pointers only inside the definition of data types, that are big in size or complex in structure.

You should always build your data structures in such a way that the pointers are invisible from the outside, and that the data structures can be used in direct variables.

Comparison with Java and the like

Because in Java (and Python and Ruby) variables implicitly are references, it is impossible to write programs that have direct mathematical semantics in such languages.

For example, if one has a List of Objects in Java, and takes out one element, then it is possible to change the list, by changing the object that was taken out.

In C^{++}, one can build a lot of mess, but if you use the language in the proper way, it is possible to define data structures with precise, functional semantics.

Pure Object-Oriented Programming

The notion 'object' is a word that refers to physical reality. The first object-oriented languages were designed with the purpose of modelling physical entities.

The notion of reference makes sense in the physical world, but no sense in the mathematical universe.

C^{++} is not a purely object-oriented programming language, because not everything is an object, and the programming modal is not state-oriented, but (in my view) it took the good things from object-oriented programming, and left out the rest.

(The good thing is to allow type construction through characteristic functions, instead of only by tuples)

Some Examples

I will show more examples of allocation:

- Single linked lists.
- Strings.
- Binary search trees.

You can obtain functional semantics if you ensure that each allocated entity is reached in only one way.