Supplement
From C to C++

History of C++

B.C.   Early languages such as FORTRAN, COBOL, ALGOL, PL/I and others.

1970   Brian Kernigham and Dennis Ritchie invent C. The language they used for inspiration was called “B”

1980   Bjarne Stroustrup creates “C with Classes.”

1995   The ANSI Committee releases their draft of the C++ Standard.
1998   An official C++ standard is adopted.
(This is the day that C++ started to become obsolete.)

Quality

Quality is designed in, not tested in.

— Dave Packard

Maintaining Maintenance

The average number of lines of code in a typical application has skyrocketed from 23,000 in 1980 to 1.2 million in 1990, according to a recent survey of managers attending the 1990 Annual Meeting and Conference of the Software Maintenance Association. At the same time, system age has risen from 4.75 to 9.40 years. Fortunately, the number of people devoted to maintaining them has made a comparable jump from 0.41 to 19.4.

What's worse, 74% of the managers surveyed at the 1990 Annual Meeting and Conference of the Software Maintenance Association reported that they "have systems in their department that have to be maintained by specific individuals because no one else understands them."

— Software Maintenance News, February 1991

Comments

A program serves two masters.

- Code tells the computer what to do.
- Comments describe what the program does to the poor programmer who has to maintain it.

There are two types of comments in C++.

```
// Comments that begin with double-slash
// and go to the end of line

/* Comments that start with slash/star */
/*and go to star/slash */

/*
 * The second version can be used
 * for multi-line comments
 */
```

Hello World

```
#include <iostream>
int main()
{
    std::cout << "Hello World\n";
    return (0);
}
```

What’s missing from this program?

[illegible]

Hello Again

```
/********************************************************







 ********************************************************/
#include <iostream>
int main(){
    // Tell the world hello
    std::cout << "Hello World\n";
    return (0);
}
```

Beginning Comments

- Heading
- Author
- Purpose
- Usage
- References
- File Formats
- Restrictions
- Revision History
- Error Handling
- Notes
- Anything else that’s useful

Oualline's Law Of Documentation

90% of the time the documentation is lost.

Out of the remaining 10%, 9% of the time the revision of the documentation is different from the revision of the program and therefore completely useless.

The 1% of the time you actually have documentation and the correct revision of the documentation, it will be written in Japanese.

Boxing with VI

Edit the file *.exrc* and add:

```
 :abbr #b /****************************************
 :abbr #e ****************************************/
```

To create a top box, type
`#b<return>`

To create a box bottom
`#e<return>`

Text-Setting

```
/********************************************************
 ********************************************************



 ********************************************************
 ********************************************************/

/*------------> Another, less important warning<--------*/

/*>>>>>>>>>>>> Major section header <<<<<<<<<<<<<<<< */

/********************************************************


 ********************************************************/

/*------------------------------------------------------*\

\*------------------------------------------------------*/
```

More Text Setting

```
/*
 * This is the beginning of a section
 * ^^^^ ^^ ^^^ ^^^^^^^^^ ^^ ^ ^^^^^^^
 *
 * In the paragraph that follows we explain what
 * the section does and how it works.
 */

/*
 * A medium level comment explaining the next
 * dozen or so lines of code. Even though we dont' ha ve
 * the bold typeface we can **emphasize** words.
 */

/* A simple comment explaining the next line */
```

[illegible]

Variables

Use long names (but not too long).

```
int account
```

Always comment your variable declarations

```
                                    // names in the list
```

Units are important.

Is length, mm, cm, miles, light-years or microns? The answer’s important.

The following comes from a real program written by Steve Oualline:

```
/*****************************************************************



 *****************************************************************/
```

KISS (Keep it Simple, Stupid)

Which is more valuable?

1) A clear, well written, easy to read, but broken program

2) A clever complex working program.

[illegible]

Precedence Rules

ANSI Standard Rules

Practical Precedence Rules

Put parentheses around everything else.

Question: Which *if* does the *else* belong to?

```
if (count < 10)     // if #1
    if ((count % 4) == 2)   // if #2
        std::cout << "Condition:White\n";
  else   // (Indentation is wrong)
      std::cout << "Condition:Tan\n";
```

a. It belongs to **if** #1.
b. It belongs to **if** #2.
c. You don’t have to worry about this situation if you never write code like this.

Side Effects

A single statement should perform a single function.

- Don’t put assignment statements inside other statements

Don’t use ++ or -- inside other statements

What does the following code fragment print?

```
i = 2;
j = square(++i);
std::cout << "i is " << i << '\n';
```

Answer

Answer

The answer depends on how `square` is defined.

```
  int square(int arg)
  {
      return (arg * arg);
  }
```

We get a 3.

```
  #define square(x) ((x) * (x))
```

We get a 4 (and j contains the wrong answer).

Switch Statements

- End every case with either “break” or “/* fall through */”
- Every switch needs a default, even if it is “/* Do nothing */”

Switch Example

```
switch (command) {

        do_reset();
        // Fall Through
    case 'x':
        do_exit();
        break;
    default:
        // Do Nothing
        break;
}
```

Rules of Thumb

- Functions should be about 2 or 3 pages long

  About the time you start running into the right margin consider breaking your function into several smaller, simpler functions.

  C++ statements are like a sentence. They should be single subject and not go on forever.

Most important

- Program in the clearest and simplest manner possible.

Electronic Archeology

The art of going through someone else’s code to discover amazing things (like how and why the code works).

Contrary to popular belief, most C++ programs are not written by

commented in Swahili. They just look that way.

Ode to a maintenance programmer

Once more I travel that lone dark road

into someone else’s impossible code

Through “if” and “switch” and “do” and “while”

that twist and turn for mile and mile

Clever code full of traps and tricks

and you must discover how it ticks

And then I emerge to ask a new,

“What the heck does this program do?”

Archeological Tools

- Editor (browser)
- Cross referencer
- grep
- indention tools
- pretty printers
- call graphs
- debuggers

Techniques

- Mark up the program (several colored pens are useful)

Go through and comment the code

Change the short variables to long ones

Add comments

```
  machine

  correction?
```

What's New In C++

- New `bool` type
- A new string class
- New I/O System
- New variable types (const, reference, etc.)
- Overloaded procedures
- `inline` procedures
- Overloaded operators
- Classes
- Exceptions
- Templates

Boolean (`bool`) type

Boolean varaible can have one of two values:

```
true
false
```

Example:

```
bool flag;
flag = true;
```

Note: The `bool` type is relatively new to C++ and some legacy macros exist to implement a bool type. These macros use `BOOL` or `Bool` as a data type and `TRUE` and `FALSE` as the values. (These legacy types should be avoided.)

C++ Strings

Bring in the string package using the statement:

```
    #include <string>
```

Declaring a string

```
    std::string my_name;    // The name of the user
```

Assigning the string a value:

```
    my_name = "Oualline";
```

Using the “+” operator to concatenate strings:

```
    first_name = "Steve"; last_name = "Oualline";
    full_name = first_name + " " + last_name;
```

More on Strings

Extract a substring:

```
     result = str.substr(first, last);
     //     01234567890123
     str = "This is a test";
     sub = str.substr(5,6);

     // sub == “12 3”
```

Finding the length of a string

```
     string.length()
```

Wide strings contain wide characters.   Example:

```
    std::wstring funny_name;
    // If you see nothing between the "" below then you
    // don't have Chinese fonts installed
```

```
    funny_name = L"              ";
```

Accessing characters in a string

You can treat strings like arrays, but this is not safe:

```
// Gets the sixth character
ch = str[5];
// Will not check to see if
// the string has 6 characters
```

Better (and much safer)

```
// Gets the sixth character
// Aborts program if
// there is no such character
ch = str.at(5);
```

Reading Data

The standard class `std::cout` is used with `<<` for writing data.

The standard class `std::cin` is used with `>>` for reading data.

```
std::cin >> price >> number_on_hand;
```

Numbers are separated by whitespace (spaces, tabs, or newlines).

For example, if our input is:

```
32 6
```

Then `price` gets 32 and `number_on_hand` gets 6.

Doubling a number

```
int main()
{








}
```

Sample run

```
Enter a value: 12
Twice 12 is 24
```

Question: Why is `width` undefined?

```
main()
{




}
```

Reading Strings

The combination of `std::cin` and `>>` works fine for integers, floating point numbers and characters. It does not work well for strings.

To read a string use the `getline` function.

```
std::getline(std::cin, string);
```

For example:

```
std::string name;    // The name of a person
std::getline(std::cin, name);
```

New I/O System

C++ uses a new I/O system called streamed based I/O. We'll study the details of this I/O system later, but for now we'll learn the basics.

C++ uses `std::cin`, `std::cout`, and `std::cerr` for input, output and error output. The operators << and >> are used for input and output.

The stream `std:clog` is used for log information.

Using `std::cout`

Simple output

```
std::cout << "This is a test\n";
```

Outputting Numbers

```
float f = 1.2;
int i = 34;
std::cout << "This is an integer " << i <<
     " and this is a float " << f << '\n';
```

Notice that C++ automatically knows what variable types are being used and formats the data accordingly. (Unlike the C `printf` statement.)

(For example, what happens if we do the following in C:

```
printf("%d", 3.5);
```

)

Using `std::cin`

Examples:

```
int i;
float f;
char str[100];
std::cin >> i;
std::cin >> i >> f;

// Note: Space or newline ends the string
std::cin >> str;
```

New Variable Types

Constant declarations

Since `MAX_USERS` is a constant:

is illegal.

Constant declarations replace the old C style **#define** declarations. The previous declaration could have been written in classic C as:

```
#define MAX_USERS 100
/* The most users at one time */
```

Constant declaration must be initialized.

const Pointers

There are several flavors of constant pointers. It’s important to know what the *const* apples to.

```
    const char* first_ptr = "Forty-Two";
    first_ptr = "Fifty six";                  // Legal or Illegal
    *first_ptr = 'X';                         // Legal or Illegal

    char* const second_ptr = "Forty-Two";
    second_ptr = "Fifty six";                 // Legal or Illegal
    *second_ptr = 'X';                        // Legal or Illegal

    const char* const third_ptr = "Forty-Two";
    third_ptr = "Fifty six";                  // Legal or Illegal
    *third_ptr = 'X';                         // Legal or Illegal
```

Reference Parameters

Reference parameters allow the programmer to define a new name for an existing variable. For example:

```
int an_integer; // A random integer
// A reference to an_integer
int &ref_integer = an_integer;
```

Any changes made to ref_integer will change an_integer. These two variables are the same thing.

For example:

```
an_integer = 5; // Is the same as
ref_integer = 5;
```

References

Another reference example:

```
int total[100];
int &first_total = total[0];
```

Constants and Functions

Constants can be used in declaring function parameters:

```
int add_two(const int first,
            const int second) {
    return (first + second);
}
```

The value of these parameters can not be changed inside this function.

Reference Parameters

Reference may be used in parameter declarations

```
void inc_counter(int &counter){
    ++counter;
}
```

For example:

```
main()
{
   int a_count = 0;     // Random counter
   inc_counter(a_count);
   std::cout << a_count << '\n';
   return (0);
}
```

More Reference Parameters

```
:
```

When C++ sees the statement:

```
    inc_counter(a_count);
```

internally it generates the code:

```
    int &counter = a_counter;
```

Now any changes made to `counter` result in changes to `a_counter`. Since `counter` is a reference to `a_counter` these variables are the same thing.

Reference and Return Values

Let’s define a procedure to find the biggest element in an array:

```
int &biggest(int array[],
             const unsigned int array_size) {
    int big_index;  // Index of the biggest element
    int index;      // Index of the current element

    big_index = 0;
    for (index = 1; index < array_size; index++) {
        if (array[big_index] < array[index])
            big_index = index;
    }
    return (array[big_index]);
}
```

Reference Returns (II)

The function `biggest` returns a reference to the biggest element of an array. We can use this to print the biggest element of an array:

```
    int array[] = {1, 99, 2, 3};
    std::cout << "Biggest element is " <<
               biggest(array, 4) << "\n";
```

In this case `biggest(array, 4)` is a reference to `array[1]`. We can put it anywhere we can put `array[1]` *including the left side of an assignment*.

For example, to zero the biggest element we can write:

```
     biggest(array, 4) = 0;
```

Constant Reference Returns

Suppose we want to return the biggest element, but prohibit the caller from changing it. Then we use a constant reference return:

```
const int &biggest(int array[],
     const unsigned int array_size)
{
    // Usual junk
}
```

Dangling References

The following program illustrates a “dangling reference.”

```
const int &min(const int &i1,
 const int &i2){
    if (i1 < i2)
        return (i1);
    return (i2);
}

int main(){
   int &i = min(1+2, 3+4);

   return (0);
}
```

What's happening

```
create integer tmp1, assign it the value 1+2
create integer tmp2, assign it the value 3+4
bind parameter i1 so it refers to tmp1
bind parameter i2 so it refers to tmp2
call the function "min"
bind main's variable i so it refers to
        the return value (i1 - a reference to tmp1)
// At this point i is a reference to tmp1
destroy tmp1
destroy tmp2

// At this point i still refers to tmp1
```

Overloaded Procedures

In C no two procedures could have the same name. C++ allows you to define "overload" procedures as long as their parameter list is different.

For example:

```
int max(int i1, int i2)
    { return (i1 < i2) ? i2 : i1; }
float max(float f1, float f2)
    { return (f1 < f2) ? f2 : f1; }
```

In C you frequently see things like:

```
int max_int(int i1, int i2);
float max_float(float f1, float f2);
```

In C++ these can be replaced by one function `max`.

inline Procedures

The `max` function we’ve just defined is very short. The overhead to setup the parameters, make the call, and return from the call takes more code than the function itself.

The `inline` keyword tells C++ that the functions are to be expanded inline.

```
    inline int max(int i1, int i2)
        { return (i1 < i2) ? i2 : i1; }
```

So

```
     result = max(large, big);
```

does not generate any function call overhead.

Default Parameters

Default parameter specification:

```
void draw_it(const rectangle &data,
             const float scale = 1.0)
```

The function `draw_it` can be called as:

```
    // Draw a double sized rectangle
    draw_it(a_rectangle, 2.0);
```

or

```
  draw_it(a_rectangle);// Draw a normal sized rect.
```

The second case is the same as:

```
  // Draw a double sized rectangle
  draw_it(a_rectangle, 2.0);
```

Unused Parameter

Suppose we have a function that takes a single parameter and never uses it:

```
void do_it(int it)
{
    // Do nothing
}
```

C++ will issue a warning about the unused parameter. To avoid this warning, do not put in the name of the parameter:

```
void do_it(int)
```

Note: To program more clearly the parameter is often “put in” as a comment:

```
void do_it(int /* it */)
```

Call by Value Parameters

Declaration: `function(int var)`

Can change inside function: Yes

Changes made inside function reflected in caller: No

Notes: Not efficient for passing structures or classes.

Reference Parameters

Declaration: `function(int &var)`

Can change inside function: Yes

Changes made inside function reflected in caller: Yes

Notes: Efficient way of passing structures

Constant Reference Parameters

Declaration: `function(const int &var)`

Can change inside function: No

Changes made inside function reflected in caller: N.A.

Notes:Efficient way of passing structures

Array Parameters

Declaration:`function(int var[])`

Can change inside function: Yes

Changes made inside function reflected in caller: Yes

Note: Array parameters are always passed by reference

Address Parameters

Declaration: `function(int *var)`

Can change inside function: Yes

Changes made inside function reflected in caller: See notes.

Note: Changes to the pointer itself are not reflected in the caller. Changes to the data pointed to can be made.

```
var = new_value; // Illegal
*var = 1; // Legal
```

Parameter Type Summary

Type	**Declaration**
Call by value	`function(int var)`
	Value is passed into the function, and can be changed inside the function, but the changes are not passed to the caller.
Constant call by value	`function(const int var)`
	Value is passed into the function and cannot be changed.
Reference	`function(int &var)`
	Reference is passed to the function. Any changes made to the parameter are reflected in the caller.
Constant Reference	`function(const int &var)`
	Value cannot be changed in the function. This form of a parameter is more efficient then "constant call by value" for complex data types.
array	`function(int array[])`
	Value is passed in and may be modified. C++ automatically turns arrays into reference parameters.
Call by address	`function(int *var)`
	Passes a pointer to an item. Pointers will be covered later.

new and *delete* operators

The **new** operator creates a new variable from space in an area of memory called the heap.

```
    item *item_ptr;
    item_ptr = new item;
```

It can allocate an array of items:

```
    item_array_ptr = new item[10];
```

The **delete** operator returns an area of memory to the heap. (It should not be used after the **delete**.)

```
   delete pointer;
   // Where pointer is a pointer to a simple object
   pointer = NULL;
```

The **delete** operator also works for arrays as well:

```
   delete []array_pointer;
   // Where pointer is a pointer to a array
   array_pointer = NULL;
```